The Mercury Language Reference Manual

Version rotd-2018-10-05

Fergus Henderson
Thomas Conway
Zoltan Somogyi
David Jeffery
Peter Schachte
Simon Taylor
Chris Speirs
Tyson Dowd
Ralph Becket
Mark Brown
Peter Wang

Table of Contents

1 Introduction ... 1

2 Syntax ... 2

2.1 Syntax overview ... 2
2.2 Character set ... 2
2.3 Whitespace ... 2
2.4 Tokens ... 2
2.5 Terms ... 5
2.6 Builtin operators ... 6
2.7 Items ... 9
2.8 Declarations ... 9
2.9 Facts ... 10
2.10 Rules ... 10
2.11 Goals ... 10
2.12 State variables ... 16
2.13 DCG-rules ... 19
2.14 DCG-goals ... 19
2.15 Data-terms ... 22
2.15.1 Data-functors ... 22
2.15.2 Record syntax ... 22
2.15.3 Unification expressions ... 24
2.15.4 Conditional expressions ... 24
2.15.5 Lambda expressions ... 24
2.15.6 Higher-order function applications ... 25
2.15.7 Explicit type qualification ... 25
2.16 Variable scoping ... 26
2.17 Implicit quantification ... 26
2.18 Elimination of double negation ... 27

3 Types ... 28

3.1 Builtin types ... 28
3.1.1 Primitive types ... 28
3.1.1.1 Signed integer types ... 28
3.1.1.2 Unsigned integer types ... 28
3.1.1.3 Floating-point type ... 29
3.1.1.4 Character type ... 29
3.1.1.5 String type ... 29
3.1.2 Other builtin types ... 29
3.1.2.1 Predicate and function types ... 29
3.1.2.2 Tuple types ... 29
3.1.2.3 The universal type ... 30
3.1.2.4 The “state-of-the-world” type ... 30

3.2 User-defined types ... 30
3.2.1 Discriminated unions ... 30
3.2.2 Equivalence types ... 32
3.2.3 Abstract types ... 32
3.3 Predicate and function type declarations ... 33
3.4 Field access functions ... 35
3.4.1 Field selection ... 35
3.4.2 Field update ... 35
3.4.3 User-supplied field access function declarations ... 36
3.4.4 Field access examples ... 36
3.5 The standard ordering ... 37

4 Modes ... 39
4.1 Insts, modes, and mode definitions ... 39
4.2 Predicate and function mode declarations ... 41
4.3 Constrained polymorphic modes ... 44
4.4 Different clauses for different modes ... 45

5 Unique modes ... 47
5.1 Destructive update ... 47
5.2 Backtrackable destructive update ... 47
5.3 Limitations of the current implementation ... 48

6 Determinism ... 49
6.1 Determinism categories ... 49
6.2 Determinism checking and inference ... 50
6.3 Replacing compile-time checking with run-time checking ... 54
6.4 Interfacing nondeterministic code with the real world ... 55
6.5 Committed choice nondeterminism ... 55

7 User-defined equality and comparison ... 57

8 Higher-order programming ... 60
8.1 Creating higher-order terms ... 60
8.2 Calling higher-order terms ... 62
8.3 Higher-order insts and modes ... 63
8.3.1 Builtin higher-order insts and modes ... 63
8.3.2 Default insts for functions ... 64
8.3.3 Combined higher-order types and insts ... 64

9 Modules 67
9.1 The module system 67
9.2 An example module. 68
9.3 Sub-modules 69
9.3.1 Nested sub-modules 69
9.3.2 Separate sub-modules 70
9.3.3 Visibility rules 70
9.3.4 Implementation bugs and limitations 71
9.4 Module initialisation 71
9.5 Module finalisation 72
9.6 Module-local mutable variables 72

10 Type classes 75
10.1 Typeclass declarations 75
10.2 Instance declarations 76
10.3 Abstract typeclass declarations 79
10.4 Abstract instance declarations 79
10.5 Type class constraints on predicates and functions 80
10.6 Type class constraints on type class declarations 80
10.7 Type class constraints on instance declarations 81
10.8 Functional dependencies 82

11 Existential types 85
11.1 Existentially typed predicates and functions 85
11.1.1 Syntax for explicit type quantifiers 85
11.1.2 Semantics of type quantifiers 85
11.1.3 Examples of correct code using type quantifiers 86
11.1.4 Examples of incorrect code using type quantifiers 87
11.2 Existential class constraints 88
11.3 Existentially typed data types 88
11.4 Some idioms using existentially quantified types 90

12 Exception handling 93

13 Semantics 96

14 Foreign language interface 98

14.1 Calling foreign code from Mercury 98
14.1.1 pragma foreign_proc 98
14.1.2 Foreign code attributes 100
14.2 Calling Mercury from foreign code 102
14.3 Data passing conventions 103
14.3.1 C data passing conventions 103
14.3.2 C# data passing conventions 104
14.3.3 Java data passing conventions 106
14.3.4 Erlang data passing conventions 108
14.4 Using foreign types from Mercury 109
14.5 Using foreign enumerations in Mercury code 110
14.6 Using Mercury enumerations in foreign code 112
14.7 Adding foreign declarations 113
14.8 Declaring Mercury exports to other modules 114
14.9 Adding foreign definitions 115
14.10 Language specific bindings 115
14.10.1 Interfacing with C 116
14.10.1.1 Using pragma foreign_type for C 116
14.10.1.2 Using pragma foreign_enum for C 117
14.10.1.3 Using pragma foreign_export_enum for C 117
14.10.1.4 Using pragma foreign_proc for C 117
14.10.1.5 Using pragma foreign_export for C 118
14.10.1.6 Using pragma foreign_decl for C 119
14.10.1.7 Using pragma foreign_code for C 120
14.10.1.8 Memory management for C 120
14.10.1.9 Linking with C object files 120
14.10.2 Interfacing with C# 121
14.10.2.1 Using pragma foreign_type for C# 121
14.10.2.2 Using pragma foreign_enum for C# 121
14.10.2.3 Using pragma foreign_export_enum for C# 121
14.10.2.4 Using pragma foreign_proc for C# 121
14.10.2.5 Using pragma foreign_export for C# 122
14.10.2.6 Using pragma foreign_decl for C# 122
14.10.2.7 Using pragma foreign_code for C# 123
14.10.3 Interfacing with Java 123
14.10.3.1 Using pragma foreign_type for Java 123
14.10.3.2 Using pragma foreign_enum for Java 124
14.10.3.3 Using pragma foreign_export_enum for Java 124
14.10.3.4 Using pragma foreign_proc for Java 124
14.10.3.5 Using pragma foreign_export for Java 124
14.10.3.6 Using pragma foreign_decl for Java 125
14.10.3.7 Using pragma foreign_code for Java 125
14.10.4 Interfacing with Erlang 126
14.10.4.1 Using pragma foreign_type for Erlang 126
14.10.4.2 Using pragma foreign_export_enum for Erlang 126
14.10.4.3 Using pragma foreign_proc for Erlang 126
14.10.4.4 Using pragma foreign_export for Erlang 127

14.10.4.5 Using pragma foreign_decl for Erlang 127
14.10.4.6 Using pragma foreign_code for Erlang 128

15 Impurity declarations 129
15.1 Choosing the right level of purity 129
15.2 Purity ordering 130
15.3 Semantics 130
15.4 Declaring impure functions and predicates 130
15.5 Marking a goal as impure 131
15.6 Promising that a predicate is pure 131
15.7 An example using impurity 132
15.8 Using impurity with higher-order code 133
15.8.1 Purity annotations on higher-order types 133
15.8.2 Purity annotations on lambda expressions 133
15.8.3 Purity annotations on higher-order calls 134

16 Solver types 135
16.1 The '`any`' inst 135
16.2 Abstract solver type declarations 135
16.3 Solver type definitions 135
16.4 Implementing solver types 137
16.5 Solver types and negated contexts 137

17 Trace goals 139

18 Pragmas 142
18.1 Inlining 142
18.2 Type specialization 142
18.2.1 Syntax and semantics of type specialization pragmas ... 142
18.2.2 When to use type specialization 143
18.2.3 Implementation specific details 143
18.3 Obsolescence 143
18.4 No determinism warnings 143
18.5 No dead predicate warnings 144
18.6 Source file name 144

19 Implementation-dependent extensions 146
19.1 Fact tables 146
19.2 Tabled evaluation 146
19.3 Termination analysis 150
19.4 Feature sets 151
19.5 Trailing 152
19.5.1 Choice points 153
19.5.2 Value trailing 153
19.5.3 Function trailing 153
19.5.4 Delayed goals and floundering 155
19.5.5 Avoiding redundant trailing 155

20 Bibliography 159
[1] 159
[2] 159
[3] 159
[4] 159
[5] 159

1 Introduction

Mercury is a general-purpose programming language, originally designed and implemented by a small group of researchers at the University of Melbourne, Australia. Mercury is based on the paradigm of purely declarative programming, and was designed to be useful for the development of large and robust "real-world" applications. It improves on existing logic programming languages by providing increased productivity, reliability and efficiency, and by avoiding the need for non-logical program constructs. Mercury provides the traditional logic programming syntax, but also allows the syntactic convenience of user-defined functions, smoothly integrating logic and functional programming into a single paradigm.

Mercury requires programmers to supply type, mode and determinism declarations for the predicates and functions they write. The compiler checks these declarations, and rejects the program if it cannot prove that every predicate or function satisfies its declarations. This improves reliability, since many kinds of errors simply cannot happen in successfully compiled Mercury programs. It also improves productivity, since the compiler pinpoints many errors that would otherwise require manual debugging to locate. The fact that declarations are checked by the compiler makes them much more useful than comments to anyone who has to maintain the program. The compiler also exploits the guaranteed correctness of the declarations for significantly improving the efficiency of the code it generates.

To facilitate programming-in-the-large, to allow separate compilation, and to support encapsulation, Mercury has a simple module system. Mercury's standard library has a variety of pre-defined modules for common programming tasks — see the Mercury Library Reference Manual.

2 Syntax

2.1 Syntax overview

Mercury's syntax is similar to the syntax of Prolog, with some additional declarations for types, modes, determinism, the module system, and pragmas, and with the distinction that function symbols may stand also for invocations of user-defined functions as well as for data constructors.

A Mercury program consists of a set of modules. Each module is a file containing a sequence of items (declarations and clauses). Each item is a term followed by a period. Each term is composed of a sequence of tokens, and each token is composed of a sequence of characters. Like Prolog, Mercury has the Definite Clause Grammar (DCG) notation for clauses.

2.2 Character set

Mercury program source files must be written using the UTF-8 encoding of the Unicode character set.

2.3 Whitespace

In Mercury program source files, whitespace is defined to be the following characters:

Unicode name	**Unicode code point**	**Notes**
SPACE	U+0020	
CHARACTER TABULATION	U+0009	Horizontal-tab
LINE FEED	U+000A	
LINE TABULATION	U+000B	Vertical-tab
FORM FEED	U+000C	
CARRIAGE RETURN	U+000D	

2.4 Tokens

Tokens in Mercury are the same as in ISO Prolog. The only differences are the '`#line`' token, which is used as a line number directive (see below) and the backquote ('`') token.

The different tokens are as follows. Tokens may be separated by whitespace.

line number directive

A line number directive consists of the character '`#`', a positive integer specifying the line number, and then a newline. A '`#line`' directive's only role is to specifying the line number; it is otherwise ignored by the syntax. Line number directives may occur anywhere a token may occur. They are used in conjunction with the '`pragma source_file`' declaration to indicate that the Mercury code following was generated by another tool; they serve to associate each line in the Mercury code with the source file name and line number of the original source from which the Mercury code was derived, so that the Mercury compiler can issue more informative error messages using the original source code locations. A '`#line`' directive specifies the line number for the immediately following line.

Line numbers for lines after that are incremented as usual, so the second line after a '`#100`' directive would be considered to be line number 101.

string A string is a sequence of characters enclosed in double quotes (`"`).

Within a string, two adjacent double quotes stand for a single double quote. For example, the string ' `""""` ' is a string of length one, containing a single double quote: the outermost pair of double quotes encloses the string, and the innermost pair stand for a single double quote.

Strings may also contain backslash escapes. '`\a`' stands for "alert" (a beep character), '`\b`' for backspace, '`\r`' for carriage-return, '`\f`' for form-feed, '`\t`' for tab, '`\n`' for newline, '`\v`' for vertical-tab. An escaped backslash, single-quote, or double-quote stands for itself.

The sequence '`\x`' introduces a hexadecimal escape; it must be followed by a sequence of hexadecimal digits and then a closing backslash. It is replaced with the character whose character code is identified by the hexadecimal number. Similarly, a backslash followed by an octal digit is the beginning of an octal escape; as with hexadecimal escapes, the sequence of octal digits must be terminated with a closing backslash.

The sequence '`\u`' or '`\U`' can be used to escape Unicode characters. '`\u`' must be followed by the Unicode character code expressed as four hexadecimal digits. '`\U`' must be followed by the Unicode character code expressed as eight hexadecimal digits. The highest allowed value is '`\U0010FFFF`'.

A backslash followed immediately by a newline is deleted; thus an escaped newline can be used to continue a string over more than one source line. (String literals may also contain embedded newlines.)

name A name is either an unquoted name or a quoted name. An unquoted name is a lowercase letter followed by zero or more letters, underscores, and digits. A quoted name is any sequence of zero or more characters enclosed in single quotes (`'`). Within a quoted name, two adjacent single quotes stand for a single single quote. Quoted names can also contain backslash escapes of the same form as for strings.

variable A variable is an uppercase letter or underscore followed by zero or more letters, underscores, and digits. A variable token consisting of single underscore is treated specially: each instance of '`_`' denotes a distinct variable. (In addition, variables starting with an underscore are presumed to be "don't-care" variables; the compiler will issue a warning if a variable that does not start with an underscore occurs only once, or if a variable starting with an underscore occurs more than once in the same scope.)

integer An integer is either a decimal, binary, octal, hexadecimal, or character-code literal. A decimal literal is any sequence of decimal digits. A binary literal is '`0b`' followed by any sequence of binary digits. An octal literal is '`0o`' followed by any sequence of octal digits. A hexadecimal literal is '`0x`' followed by any sequence of hexadecimal digits. A character-code literal is '`0'`' followed by any single character.

Decimal, binary, octal and hexadecimal literals may be optionally terminated by a suffix that indicates whether the literal represents a signed or unsigned integer and what the size of that integer is. These suffixes are:

Suffix	**Signedness**	**Size**
`i` or no suffix	Signed	Implementation-defined
`i8`	Signed	8-bit
`i16`	Signed	16-bit
`i32`	Signed	32-bit
`i64`	Signed	64-bit
`u`	Unsigned	Implementation-defined
`u8`	Unsigned	8-bit
`u16`	Unsigned	16-bit
`u32`	Unsigned	32-bit
`u64`	Unsigned	64-bit

For decimal, binary, octal and hexadecimal literals, an arbitrary number of underscores (‘`_`’) may be inserted between the digits. An arbitrary number of underscores may also be inserted between the radix prefix (i.e. ‘`0b`’, ‘`0o`’ and ‘`0x`’) and the initial digit. Similarly, an arbitrary number of underscores may be inserted between the final digit and the signedness suffix. The purpose of the underscores is to improve readability and they do not affect the numeric value of the literal.

float A floating point literal consists of a sequence of decimal digits, a decimal point (‘`.`’) and a sequence of digits (the fraction part), and the letter ‘`E`’ (or ‘`e`’), an optional sign (‘`+`’ or ‘`-`’), and then another sequence of decimal digits (the exponent). The fraction part or the exponent (but not both) may be omitted.

An arbitrary number of underscores (‘`_`’) may be inserted between the digits in a floating point literal. Underscores may *not* occur adjacent to any non-digit characters (i.e. ‘`.`’, ‘`e`’, ‘`E`’, ‘`+`’ or ‘`-`’) in a floating point literal. The purpose of the underscores is to improve readability and they do not affect the numeric value of the literal.

implementation_defined_literal
An implementation-defined literal consists of a dollar sign (‘`$`’) followed by an unquoted name.

open_ct A left parenthesis, ‘`(`’, that is not preceded by whitespace.

open A left parenthesis, ‘`(`’, that is preceded by whitespace.

close A right parenthesis, ‘`)`’.

open_list A left square bracket, ‘`[`’.

close_list A right square bracket, ‘`]`’.

open_curly
A left curly bracket, ‘`{`’.

close_curly
A right curly bracket, ‘`}`’.

ht_sep A "head-tail separator", i.e. a vertical bar, '`|`'.

comma A comma, '`,`'.

end A full stop (period), '`.`'.

eof The end of file.

2.5 Terms

Syntactically, terms in Mercury are exactly the same as in ISO Prolog, except that as extensions we permit higher-order terms and the introduction of infix operators by the use of grave accents (backquotes), as described below, and we support an extended set of builtin operators. See Section 2.6 [Builtin operators], page 6. Also, the constructor for list terms in Mercury is `[|]/2`, not `./2` as in Prolog.

Note, however, that the meaning of some terms in Mercury is different to that in Prolog. See Section 2.15 [Data-terms], page 22.

A term is either a variable or a functor.

A functor is an integer, a float, a string, a name, a compound term, or a higher-order term.

A compound term is a simple compound term, a list term, a tuple term, an operator term, or a parenthesized term.

A simple compound term is a name followed without any intervening whitespace by an open parenthesis (i.e. an open_ct token), a sequence of argument terms separated by commas, and a close parenthesis.

A list term is an open square bracket (i.e. an open_list token) followed by a sequence of argument terms separated by commas, optionally followed by a vertical bar (i.e. a ht_sep token) followed by a term, followed by a close square bracket (i.e. a close_list token). An empty list term is an open_list token followed by a close_list token. List terms are parsed as follows:

```
parse('[' ']') = [].
parse('[' List) = parse_list(List).
parse_list(Head ',' Tail) = '[|]'(parse_term(Head), parse_list(Tail)).
parse_list(Head '|' Tail ']') = '[|]'(parse_term(Head), parse_term(Tail)).
parse_list(Head ']') = '[|]'(parse_term(Head), []).
```

The following terms are all equivalent:

```
[1, 2, 3]
[1, 2, 3 | []]
[1, 2 | [3]]
[1 | [2, 3]]
'[|]'(1, '[|]'(2, '[|]'(3, [])))
```

A tuple term is a left curly bracket (i.e. an open_curly token) followed by a sequence of argument terms separated by commas, and a right curly bracket. For example, `{1, '2', "three"}` is a valid tuple term.

An operator term is a term specified using operator notation, as in Prolog. Operators can also be formed by enclosing a name, a module qualified name (see Section 9.1 [The

module system], page 67), or a variable between grave accents (backquotes). Any name or variable may be used as an operator in this way. If *fun* is a variable or name, then a term of the form `X ` fun` Y` is equivalent to `fun(X, Y)`. The operator is left associative and binds more tightly than every operator other than '^' (see Section 2.6 [Builtin operators], page 6).

A parenthesized term is just an open parenthesis followed by a term and a close parenthesis.

A higher-order term is a "closure" term, which can be any term other than a name or an operator term, followed without any intervening whitespace by an open parenthesis (i.e. an open_ct token), a sequence of argument terms separated by commas, and a close parenthesis. A higher-order term is equivalent to a simple compound term whose functor is the empty name, and whose arguments are the closure term followed by the argument terms of the higher-order term. That is, a term such as `Term(Arg1, ..., ArgN)` is parsed as `''(Term, Arg1, ..., ArgN)`. Note that the closure term can be a parenthesized term; for example, `(Term ^ FieldName)(Arg1, Arg2)` is a higher-order term, and so it gets parsed as if it were `''((Term ^ FieldName), Arg1, Arg2)`.

2.6 Builtin operators

The following table lists all of Mercury's builtin operators. Operators with a low "Priority" bind more tightly than those with a high "Priority". For example, given that `+` has priority 500 and `*` has priority 400, the term `2 * X + Y` would parse as `(2 * X) + Y`.

The "Specifier" field indicates what structure terms constructed with an operator are allowed to take. "f" represents the operator and "x" and "y" represent arguments. "x" represents an argument whose priority must be strictly lower than that of the operator. "y" represents an argument whose priority is lower or equal to that of the operator. For example, "yfx" indicates a left-associative infix operator, while "xfy" indicates a right-associative infix operator.

Operator	Specifier	Priority
.	yfx	10
!	fx	40
!.	fx	40
!:	fx	40
@	xfx	90
^	xfy	99
^	fx	100
event	fx	100
:	yfx	120
`op`	yfx	120 [1]
**	xfy	200
-	fx	200
\	fx	200
*	yfx	400
/	yfx	400

[1] Operator term (see Section 2.5 [Terms], page 5).

//	yfx	400
<<	yfx	400
>>	yfx	400
div	yfx	400
mod	xfx	400
rem	xfx	400
for	xfx	500
+	fx	500
+	yfx	500
++	xfy	500
-	yfx	500
--	yfx	500
/\	yfx	500
\/	yfx	500
..	xfx	550
:=	xfx	650
=^	xfx	650
<	xfx	700
=	xfx	700
=..	xfx	700
=:=	xfx	700
=<	xfx	700
==	xfx	700
=\=	xfx	700
>	xfx	700
>=	xfx	700
@<	xfx	700
@=<	xfx	700
@>	xfx	700
@>=	xfx	700
\=	xfx	700
\==	xfx	700
~=	xfx	700
is	xfx	701
and	xfy	720
or	xfy	740
func	fx	800
impure	fy	800
pred	fx	800
semipure	fy	800
\+	fy	900
not	fy	900
when	xfx	900
~	fy	900
<=	xfy	920
<=>	xfy	920
=>	xfy	920

```
all                                fxy             950
arbitrary                          fxy             950
atomic                             fxy             950
disable_warning                    fxy             950
disable_warnings                   fxy             950
promise_equivalent_solutions       fxy             950
promise_equivalent_solution_sets fxy               950
promise_exclusive                  fy              950
promise_exclusive_exhaustive       fy              950
promise_exhaustive                 fy              950
promise_impure                     fx              950
promise_pure                       fx              950
promise_semipure                   fx              950
require_complete_switch            fxy             950
require_switch_arms_det            fxy             950
require_switch_arms_semidet        fxy             950
require_switch_arms_multi          fxy             950
require_switch_arms_nondet         fxy             950
require_switch_arms_cc_multi       fxy             950
require_switch_arms_cc_nondet      fxy             950
require_switch_arms_erroneous      fxy             950
require_switch_arms_failure        fxy             950
require_det                        fx              950
require_semidet                    fx              950
require_multi                      fx              950
require_nondet                     fx              950
require_cc_multi                   fx              950
require_cc_nondet                  fx              950
require_erroneous                  fx              950
require_failure                    fx              950
trace                              fxy             950
try                                fxy             950
some                               fxy             950
,                                  xfy             1000
&                                  xfy             1025
->                                 xfy             1050
;                                  xfy             1100
or_else                            xfy             1100
then                               xfx             1150
if                                 fx              1160
else                               xfy             1170
::                                 xfx             1175
==>                                xfx             1175
where                              xfx             1175
--->                               xfy             1179
catch                              xfy             1180
type                               fx              1180
```

solver	fy	1181
catch_any	xfy	1190
end_module	fx	1199
import_module	fx	1199
include_module	fx	1199
initialise	fx	1199
initialize	fx	1199
finalise	fx	1199
finalize	fx	1199
inst	fx	1199
instance	fx	1199
mode	fx	1199
module	fx	1199
pragma	fx	1199
promise	fx	1199
rule	fx	1199
typeclass	fx	1199
use_module	fx	1199
-->	xfx	1200
:-	fx	1200
:-	xfx	1200
?-	fx	1200

2.7 Items

Each item in a Mercury module is either a declaration or a clause. If the top-level functor of the term is '`:-/1`', the item is a declaration, otherwise it is a clause. There are three types of clauses. If the top-level functor of the item is '`:-/2`', the item is a rule. If the top-level functor is '`-->/2`', the item is a DCG rule. Otherwise, the item is a fact. There are two types of rules and facts. If the top-level functor of the head of a rule is '`=/2`', the rule is a function rule, otherwise it is a predicate rule. If the top-level functor of the head of a fact is '`=/2`', the fact is a function fact, otherwise it is a predicate fact.

2.8 Declarations

The allowed declarations are:

```
:- type
:- solver type
:- pred
:- func
:- inst
:- mode
:- typeclass
:- instance
:- pragma
:- promise
:- initialise
```

```
:- finalise
:- mutable
:- module
:- interface
:- implementation
:- import_module
:- use_module
:- include_module
:- end_module
```

The '`type`', '`solver type`', '`pred`', '`func`', '`typeclass`' and '`instance`' declarations are used for the type system, the '`inst`' and '`mode`' declarations are for the mode system, the '`pragma`' declarations are for the foreign language interface, and for compiler hints about inlining, and the remainder are for the module system. They are described in more detail in their respective chapters.

2.9 Facts

A function fact is an item of the form '*`Head = Result`*'. A predicate fact is an item of the form '*`Head`*', where the top-level functor of *Head* is not `:-/1`, `:-/2`, `-->/2`, or `=/2`. In both cases, the *Head* term must not be a variable. The top-level functor of the *Head* determines which predicate or function the fact belongs to; the predicate or function must have been declared in a preceding '`pred`' or '`func`' declaration in this module. The *Result* (if any) and the arguments of the *Head* must be valid data-terms (optionally annotated with a mode qualifier; see Section 4.4 [Different clauses for different modes], page 45).

A fact is equivalent to a rule whose body is '`true`'.

2.10 Rules

A function rule is an item of the form '*`Head = Result :- Body`*'. A predicate rule is an item of the form '*`Head :- Body`*' where the top-level functor of '`Head`' is not `=/2`. In both cases, the *Head* term must not be a variable. The top-level functor of the *Head* determines which predicate or function the clause belongs to; the predicate or function must have been declared in a preceding '`pred`' or '`func`' declaration in this module. The *Result* and the arguments of the *Head* must be valid data-terms (optionally annotated with a mode qualifier; see Section 4.4 [Different clauses for different modes], page 45). The *Body* must be a valid goal.

2.11 Goals

A goal is a term of one of the following forms:

`some` *`Vars Goal`*

An existential quantification. *Vars* must be a list of variables. *Goal* must be a valid goal.

Each existential quantification introduces a new scope. The variables in *Vars* are local to the goal *Goal*: for each variable named in *Vars*, any occurrences of variables with that name in *Goal* are considered to name a different variable

than any variables with the same name that occur outside of the existential quantification.

Operationally, existential quantification has no effect, so apart from its effect on variable scoping, '`some Vars Goal`' is the same as '*Goal*'.

Mercury's rules for implicit quantification (see Section 2.17 [Implicit quantification], page 26) mean that variables are often implicitly existentially quantified. There is usually no need to write existential quantifiers explicitly.

`all` *Vars Goal*
: A universal quantification. *Vars* must be a list of variables. *Goal* must be a valid goal. This is an abbreviation for '`not (some` *Vars* `not` *Goal*`)`'.

Goal1`,` *Goal2*
: A conjunction. *Goal1* and *Goal2* must be valid goals.

Goal1 `&` *Goal2*
: A parallel conjunction. This has the same declarative semantics as the normal conjunction. Operationally, implementations may execute *Goal1* & *Goal2* in parallel. The order in which parallel conjuncts begin execution is not fixed. It is an error for *Goal1* or *Goal2* to have a determinism other than `det` or `cc_multi`. See Section 6.1 [Determinism categories], page 49.

Goal1 `;` *Goal2*
: where *Goal1* is not of the form '*Goal1a* `->` *Goal1b*': a disjunction. *Goal1* and *Goal2* must be valid goals.

`true`
: The empty conjunction. Always succeeds.

`fail`
: The empty disjunction. Always fails.

`not` *Goal*
`\+` *Goal*
: A negation. The two different syntaxes have identical semantics. *Goal* must be a valid goal. Both forms are equivalent to '`if` *Goal* `then fail else true`'.

Goal1 `=>` *Goal2*
: An implication. This is an abbreviation for '`not (`*Goal1*`, not` *Goal2*`)`'.

Goal1 `<=` *Goal2*
: A reverse implication. This is an abbreviation for '`not (`*Goal2*`, not` *Goal1*`)`'.

Goal1 `<=>` *Goal2*
: A logical equivalence. This is an abbreviation for '`(`*Goal1* `=>` *Goal2*`), (`*Goal1* `<=` *Goal2*').

`if` *CondGoal* `then` *ThenGoal* `else` *ElseGoal*
CondGoal `->` *ThenGoal* `;` *ElseGoal*
: An if-then-else. The two different syntaxes have identical semantics. *CondGoal*, *ThenGoal*, and *ElseGoal* must be valid goals. Note that the "else" part is *not* optional.

The declarative semantics of an if-then-else is given by `(` *CondGoal*`,` *ThenGoal* `; not(`*CondGoal*`),` *ElseGoal*`)`, but the operational semantics are different, and it is treated differently for the purposes of determinism inference (see Chapter 6

[Determinism], page 49). Operationally, it executes the *CondGoal*, and if that succeeds, then execution continues with the *ThenGoal*; otherwise, i.e. if *CondGoal* fails, it executes the *ElseGoal*. Note that *CondGoal* can be nondeterministic — unlike Prolog, Mercury's if-then-else does not commit to the first solution of the condition if the condition succeeds.

If *CondGoal* is an explicit existential quantification, `some` *`Vars QuantifiedCondGoal`*, then the variables *Vars* are existentially quantified over the conjunction of the goals *QuantifiedCondGoal* and *ThenGoal*. Explicit existential quantifications that occur as subgoals of *CondGoal* do *not* affect the scope of variables in the "then" part. For example, in

```
( if some [V] C then T else E )
```

the variable V is quantified over the conjunction of the goals C and T because the top-level goal of the condition is an explicit existential quantification, but in

```
( if true, some [V] C then T else E )
```

the variable V is only quantified over C because the top-level goal of the condition is not an explicit existential quantification.

`Term1 = Term2`
: A unification. *Term1* and *Term2* must be valid data-terms.

`Term1 \= Term2`
: An inequality. *Term1* and *Term2* must be valid data-terms. This is an abbreviation for '`not (`*`Term1 = Term2`*`)`'.

`call(Closure)`
`call(Closure1, Arg1)`
`call(Closure2, Arg1, Arg2)`
`call(Closure3, Arg1, Arg2, Arg3)`
`...`
: A higher-order predicate call. The closure and arguments must be valid data-terms. '`call(Closure)`' just calls the specified closure. The other forms append the specified arguments onto the argument list of the closure before calling it. See Chapter 8 [Higher-order], page 60.

`Var`
`Var(Arg1)`
`Var(Arg2)`
`Var(Arg2, Arg3)`
`...`
: A higher-order predicate call. *Var* must be a variable. The semantics are exactly the same as for the corresponding higher-order call using the `call/N` syntax, i.e. '`call(Var)`', '`call(Var, Arg1)`', etc.

`promise_pure` *`Goal`*
: A purity cast. *Goal* must be a valid goal. This goal promises that *Goal* implements a pure interface, even though it may include impure and semipure components.

`promise_semipure` *`Goal`*
: A purity cast. *Goal* must be a valid goal. This goal promises that *Goal* implements a semipure interface, even though it may include impure components.

`promise_impure` *Goal*

A purity cast. *Goal* must be a valid goal. This goal instructs the compiler to treat *Goal* as though it were impure, regardless of its actual purity.

`promise_equivalent_solutions` *Vars Goal*

A determinism cast. *Vars* must be a list of variables. *Goal* must be a valid goal. This goal promises that *Vars* is the set of variables bound by *Goal*, and that while *Goal* may have more than one solution, all of these solutions are equivalent with respect to the equality theories of the variables in *Vars*. It is an error for *Vars* to include a variable not bound by *Goal* or for *Goal* to bind a non-local variable that is not listed in *Vars* (non-local variables with inst `any` are assumed to be further constrained by *Goal* and must also be included in *Vars*). If *Goal* has determinism `multi` or `cc_multi` then `promise_equivalent_solutions` *Vars Goal* has determinism `det`. If *Goal* has determinism `nondet` or `cc_nondet` then `promise_equivalent_solutions` *Vars Goal* has determinism `semidet`.

`promise_equivalent_solution_sets` *Vars Goal*

A determinism cast, of the kind performed by `promise_equivalent_solutions`, on any goals of the form `arbitrary` *ArbVars ArbGoal* inside *Goal*, of which there should be at least one. *Vars* and *ArbVars* must be lists of variables, and *Goal* and *ArbGoal* must be valid goals. *Vars* must be the set of variables bound by *Goal*, and *ArbVars* must be the set of variables bound by *ArbGoal*, It is an error for *Vars* to include a variable not bound by *Goal* or for *Goal* to bind a non-local variable that is not listed in *Vars*, and similarly for *ArbVars* and *ArbGoal*. The intersection of *Vars* and the *ArbVars* list of any `arbitrary` *ArbVars ArbGoal* goal included inside *Goal* must be empty.

The overall *promise_equivalent_solution_sets* goal promises that the set of solutions computed for *Vars* by *Goal* is not influenced by which of the possible solutions for *ArbVars* is computed by each *ArbGoal*; while different choices of solutions for some of the *ArbGoal*s may lead to syntactically different solutions for *Vars* for *Goal*, all of these solutions are equivalent with respect to the equality theories of the variables in *Vars*. If an *ArbGoal* has determinism `multi` or `cc_multi` then `arbitrary` *ArbVars ArbGoal* has determinism `det`. If *ArbGoal* has determinism `nondet` or `cc_nondet` then `arbitrary` *ArbVars ArbGoal* has determinism `semidet`. *Goal* itself may have any determinism.

There is no requirement that given one of the *ArbGoal*s, all its solutions must be equivalent with respect to the equality theories of the corresponding *ArbVars*; in fact, in typical usage, this won't be the case. The different solutions of the nested *arbitrary* goals are not required to be equivalent in any context except the *promise_equivalent_solution_sets* goal they are nested inside.

Goals of the form `arbitrary` *ArbVars ArbGoal* are not allowed to occur outside `promise_equivalent_solution_sets` *Vars Goal* goals.

`require_det` *Goal*
`require_semidet` *Goal*
`require_multi` *Goal*
`require_nondet` *Goal*
`require_cc_multi` *Goal*
`require_cc_nondet` *Goal*
`require_erroneous` *Goal*
`require_failure` *Goal*

A determinism check, typically used to enhance the robustness of code. *Goal* must be a valid goal. If *Goal* is det, then `require_det` *Goal* is equivalent to just *Goal*. If *Goal* is not det, then the compiler is required to generate an error message.

The `require_det` keyword may be replaced with `require_semidet`, `require_multi`, `require_nondet`, `require_cc_multi`, `require_cc_nondet`, `require_erroneous` or `require_failure`, each of which requires *Goal* to have the named determinism.

`require_complete_switch [`*Var*`]` *Goal*

A switch completeness check, typically used to enhance the robustness of code. If *Goal* is a switch on *Var* and the switch is *complete*, i.e. the switch has an arm for every function symbol that *Var* could be bound to at this point in the code, then `require_complete_switch [`*Var*`]` *Goal* is equivalent to *Goal*. If *Goal* is a switch on *Var* but is *not* complete, or *Goal* is not a switch on *Var* at all, then the compiler is required to generate an error message.

`require_switch_arms_det [`*Var*`]` *Goal*
`require_switch_arms_semidet [`*Var*`]` *Goal*
`require_switch_arms_multi [`*Var*`]` *Goal*
`require_switch_arms_nondet [`*Var*`]` *Goal*
`require_switch_arms_cc_multi [`*Var*`]` *Goal*
`require_switch_arms_cc_nondet [`*Var*`]` *Goal*
`require_switch_arms_erroneous [`*Var*`]` *Goal*
`require_switch_arms_failure [`*Var*`]` *Goal*

`require_switch_arms_det` is a determinism check, typically used to enhance the robustness of code. *Goal* must be a valid goal. If *Goal* is a switch on *Var*, and all arms of the switch would be allowable in a det context, `require_switch_arms_det [`*Var*`]` *Goal* is equivalent to *Goal*. If *Goal* is not a switch on *Var*, or if it is a switch on *Var* but some of its arms would *not* be allowable in a det context, then the compiler is required to generate an error message.

The `require_switch_arms_det` keyword may be replaced with `require_switch_arms_semidet`, `require_switch_arms_multi`, `require_switch_arms_nondet`, `require_switch_arms_cc_multi`, `require_switch_arms_cc_nondet`, `require_switch_arms_erroneous` or `require_switch_arms_failure`, each of which requires the arms of the switch on *Var* to have a determinism that is *at least as tight* as the named determinism. The determinism match need not be exact; the requirement is that the arms' determinisms should make all the promises about the minimum and maximum number of solutions as the named determinism does. For

example, it is ok to have a det switch arm in a `require_switch_arms_semidet` scope, even though it would not be ok to have a det goal in a `require_semidet` scope.

`disable_warnings [`*`Warning`*`] `*`Goal`*
`disable_warning [`*`Warning`*`] `*`Goal`*
: The Mercury compiler can generate warnings about several kinds of constructs that whose legal Mercury semantics is likely to differ from the semantics intended by the programmer. While such warnings are useful most of the time, they are a distraction in cases where the programmer's intention *does* match the legal semantics. Programmers can disable all warnings of a particular kind for an entire module by compiling that module with the appropriate compiler option, but in many cases this is not a good idea, since some of the warnings it disables may *not* have been mistaken. This is what these goals are for. The goal `disable_warnings [`*`Warning`*`] `*`Goal`* is equivalent to *`Goal`* in all respects, with one exception: the Mercury compiler will not generate warnings of any of the categories whose names appear in `[`*`Warning`*`]`.

  At the moment, the Mercury compiler supports the disabling of the following warning category:

  `singleton_vars`
  : Disable the generation of singleton variable warnings.

  The keyword starting this scope may be written either as `disable_warnings` or as `disable_warning`. This is intended to make the code read more naturally regardless of whether the list contains the name of more than one disabled warning category.

`trace `*`Params Goal`*
: A trace goal, typically used for debugging or logging. *Goal* must be a valid goal; *Params* must be a valid list of trace parameters. Some trace parameters specify compile time or run time conditions; if any of these conditions are false, *Goal* will not be executed. Since in some program invocations *Goal* may be replaced by '`true`' in this way, *Goal* may not bind or change the instantiation state of any variables it shares with the surrounding context. The things it may do are thus restricted to side effects; good programming style requires these side effects to not have any affect on the execution of the program itself, but to be confined to the provision of extra information for the user of the program. See Chapter 17 [Trace goals], page 139 for the details.

`try `*`Params Goal`*` ... catch `*`Term -> CGoal`*` ...`
: A try goal. Exceptions thrown during the execution of *Goal* may be caught and handled. A summary of the try goal syntax is:

```
try Params Goal
then ThenGoal
else ElseGoal
catch Term -> CatchGoal
...
catch_any CatchAnyVar -> CatchAnyGoal
```

See Chapter 12 [Exception handling], page 93 for the full details.

`event` *Goal*
: An event goal. *Goal* must be a predicate call. Event goals are an extension used by the Melbourne Mercury implementation to support user defined events in the Mercury debugger, '`mdb`'. See the "Debugging" chapter of the Mercury User's Guide for further details.

Call
: Any goal which does not match any of the above forms must be a predicate call. The top-level functor of the term determines the predicate called; the predicate must be declared in a `pred` declaration in the module or in the interface of an imported module. The arguments must be valid data-terms.

2.12 State variables

Clauses may use '`state variables`' as a shorthand for naming intermediate values in a sequence. That is, where in the plain syntax one might write

```
main(IO0, IO) :-
    io.write_string("The answer is ", IO0, IO1),
    io.write_int(calculate_answer(...), IO1, IO2),
    io.nl(IO3, IO).
```

using state variable syntax one could write

```
main(!IO) :-
    io.write_string("The answer is ", !IO),
    io.write_int(calculate_answer(...), !IO),
    io.nl(!IO).
```

A state variable is written '`!.`*X*' or '`!:`*X*', denoting the "current" or "next" value of the sequence labelled *X*. An argument '`!`*X*' is shorthand for two state variable arguments '`!.`*X*`, !:`*X*'; that is, '`p(..., !`*X*`, ...)`' is parsed as '`p(..., !.`*X*`, !:`*X*`, ...)`'.

Within each clause, a transformation converts state variables into sequences of ordinary logic variables. The syntactic conversion is described in terms of the notional '`transform`' function defined next.

The transformation is applied once for each state variable *X* with some fresh variables which we shall call *ThisX* and *NextX*.

The expression '`substitute(`*Term*`, `*X*`, `*ThisX*`, `*NextX*`)`' stands for a copy of *Term* with free occurrences of '`!.`*X*' replaced with *ThisX* and free occurrences of '`!:`*X*' replaced with *NextX* (a free occurrence is one not bound by the head of a clause or lambda or by explicit quantification.)

State variables obey special scope rules. A state variable *X* must be explicitly introduced either in the head of the clause or lambda (in which case it may appear as either or both of '`!.`*X*' or '`!:`*X*') or in an explicit quantification (in which case it must appear as '`!`*X*'.) A state variable *X* in the enclosing scope of a lambda or if-then-else expression may only be referred to as '`!.`*X*' (unless the enclosing *X* is masked by a more local state variable of the same name.)

For instance, the following clause employing a lambda expression

```
p(A, B, !S) :-
    F = (pred(C::in, D::out) is det :-
        q(C, D, !S)
    ),
    ( F(A, E) ->
        B = E
    ;
        B = A
    ).
```

is illegal because it implicitly refers to '`!:S`' inside the lambda expression. However

```
p(A, B, !S) :-
    F = (pred(C::in, D::out, !.S::in, !:S::out) is det :-
        q(C, D, !S)
    ),
    ( F(A, E, !S) ->
        B = E
    ;
        B = A
    ).
```

is acceptable because the state variable S accessed inside the lambda expression is locally scoped to the lambda expression (shadowing the state variable of the same name outside the lambda expression), and the lambda expression may refer to the next version of a local state variable.

There are three restrictions concerning state variables in lambdas: first, '`!X`' is not a legitimate function result, since it stands for two arguments, rather than one; second, '`!X`' may not appear as a parameter term in the head of a lambda since there is no syntax for specifying the modes of the two implied parameters; third, '`!X`' may not appear as an argument in a function application since this would not make sense given the usual interpretation of state variables and functions.

`Head :- Body`

```
transform((Head :- Body), X, ThisX, NextX) =
substitute(Head, X, ThisX, NextX) :- transform(Body, X, ThisX, NextX)
```

`Head --> Body`

```
transform((Head --> Body), X, ThisX, NextX) =
substitute(Head, X, ThisX, NextX) :- transform(Body, X, ThisX, NextX)
```

`Goal1, Goal2`

```
transform((Goal1, Goal2), X, ThisX, NextX) =
transform(Goal1, X, ThisX, TmpX), transform(Goal2, X, TmpX, NextX)
```

for some fresh variable *TmpX*.

`Goal1 ; Goal2`

```
transform((Goal1 ; Goal2), X, ThisX, NextX) =
transform(Goal1, X, ThisX, NextX) ; transform(Goal2, X, ThisX, NextX)
```

`not Goal`

`\+ Goal` A negation. The two different syntaxes have identical semantics.

```
transform((not Goal), X, ThisX, NextX) =
not transform(Goal1, X, ThisX, DummyX), NextX = ThisX
```

for some fresh variable *DummyX*.

`if Goal1 then Goal2 else Goal3`
`Goal1 -> Goal2 ; Goal3`
An if-then-else. The two different syntaxes have identical semantics.

```
transform((if Goal1 then Goal2 else Goal3), X, ThisX, NextX) =
if transform(Goal1, X, ThisX, TmpX) then transform(Goal2, X, TmpX,  NextX)
                                    else transform(Goal3, X, ThisX, NextX)
```

for some fresh variable *TmpX*.

`Goal1 => Goal2`
`Goal2 <= Goal1`
An implication. The two different syntaxes have identical semantics.

```
transform((Goal1 => Goal2), X, ThisX, NextX) =
transform(Goal1, X, ThisX, TmpX) => transform(Goal2, X, TmpX,  NextX),
NextX = ThisX
```

for some fresh variable *TmpX*.

`all Vars Goal`

```
transform((all Vars Goal), X, ThisX, NextX) =
all Vars transform(Goal, X, ThisX, DummyX), NextX = ThisX
```

for some fresh variable *DummyX*.

`some Vars Goal`

```
transform((some Vars Goal), X, ThisX, NextX) =
some Vars transform(Goal, X, ThisX, NextX)
```

`!X ^ field_list := Term`
A field update. Replaces a field in the argument. *Term* must be a valid data-term. *field_list* must be a valid field list. See Section 2.15.2 [Record syntax], page 22.

```
transform((!X ^ field_list := Term), X, ThisX, NextX) =
NextX = ThisX ^ field_list := Term
```

`Call_or_Unification`
If '`!:X`' does not appear in *Call_or_Unification* then

```
transform(Call_or_Unification, X, ThisX, NextX) =
substitute(Call_or_Unification, X, ThisX, NextX), NextX = ThisX
```

If '`!:X`' does appear in *Call_or_Unification* then

```
transform(Call_or_Unification, X, ThisX, NextX) =
substitute(Call_or_Unification, X, ThisX, NextX)
```

This transformation can lead to the introduction of chains of unifications for variables that do not otherwise play a role in the definition. Such chains are removed transparently.

The following code fragments illustrate appropriate use of state variable syntax.

Threading the I/O state

```
main(!IO) :-
    io.write_string("The 100th prime is ", !IO),
    X = prime(100),
    io.write_int(X, !IO),
    io.nl(!IO).
```

Handling accumulators (1)

```
foldl2(_, [], !A, !B).
foldl2(P, [X | Xs], !A, !B) :-
    P(X, !A, !B),
    foldl2(P, Xs, !A, !B).
```

Handling accumulators (2)

```
iterate_while2(P, F, !A, !B) :-
    ( if P(!.A, !.B) then
        F(!A, !B),
        iterate_while2(P, F, !A, !B)
    else
        true
    ).
```

2.13 DCG-rules

(DCG notation is intended for writing parsers and sequence generators in a particular style; in the past it has also been used to thread an implicit state variable, typically the I/O state, through code. As a matter of style, we recommend that in future DCG notation be reserved for writing parsers and sequence generators and that state variable syntax be used for passing state threads.)

DCG-rules in Mercury have identical syntax and semantics to DCG-rules in Prolog.

A DCG-rule is an item of the form '`Head --> Body`'. The *Head* term must not be a variable. A DCG-rule is an abbreviation for an ordinary rule with two additional implicit arguments appended to the arguments of *Head*. These arguments are fresh variables which we shall call *V_in* and *V_out*. The *Body* must be a valid DCG-goal, and is an abbreviation for an ordinary goal. The next section defines a mathematical function '`DCG-transform(V_in, V_out, DCG-goal)`' which specifies the semantics of how DCG goals are transformed into ordinary goals. (The '`DCG-transform`' function is purely for the purposes of exposition, to define the semantics — it is not part of the language.)

2.14 DCG-goals

A DCG-goal is a term of one of the following forms:

`some Vars DCG-goal`

A DCG existential quantification. *Vars* must be a list of variables. *DCG-goal* must be a valid DCG-goal.

Semantics:

```
transform(V_in, V_out, some Vars DCG_goal) =
some Vars transform(V_in, V_out, DCG_goal)
```

`all` *`Vars DCG-goal`*

A DCG universal quantification. *Vars* must be a list of variables. *DCG-goal* must be a valid DCG-goal.

Semantics:

```
transform(V_in, V_out, all Vars DCG_goal) =
all Vars transform(V_in, V_out, DCG_goal)
```

`DCG-goal1``,` *`DCG-goal2`*

A DCG sequence. Intuitively, this means "parse *DCG-goal1* and then parse *DCG-goal2*" or "do *DCG-goal1* and then do *DCG-goal2*". (Note that the only way this construct actually forces the desired sequencing is by the modes of the implicit DCG arguments.) *DCG-goal1* and *DCG-goal2* must be valid DCG-goals.

Semantics:

```
transform(V_in, V_out, (DCG-goal1, DCG-goal2)) =
(transform(V_in, V_new, DCG_goal1),
 transform(V_new, V_out, DCG_goal2))
```

where V_new is a fresh variable.

`DCG-goal1` `;` *`DCG-goal2`*

A disjunction. *DCG-goal1* and *DCG-goal2* must be valid goals. *DCG-goal1* must not be of the form '`DCG-goal1a -> DCG-goal1b`'. (If it is, then the goal is an if-then-else, not a disjunction.)

Semantics:

```
transform(V_in, V_out, (DCG_goal1 ; DCG_goal2)) =
( transform(V_in, V_out, DCG_goal1)
; transform(V_in, V_out, DCG_goal2) )
```

`{` *`Goal`* `}` A brace-enclosed ordinary goal. *Goal* must be a valid goal.

Semantics:

```
transform(V_in, V_out, { Goal }) = (Goal, V_out = V_in)
```

`[`*`Term`*`, ...]`

A DCG input match. Unifies the implicit DCG input variable V_in, which must have type '`list(_)`', with a list whose initial elements are the terms specified and whose tail is the implicit DCG output variable V_out. The terms must be valid data-terms.

Semantics:

```
transform(V_in, V_out, [Term1, ...]) = (V_in = [Term, ... | V_Out])
```

`[]` The null DCG goal (an empty DCG input match). Equivalent to '`{ true }`'.

Semantics:

```
transform(V_in, V_out, []) = (V_out = V_in)
```

`not` *`DCG-goal`*
`\+` *`DCG-goal`*

A DCG negation. The two different syntaxes have identical semantics. *Goal* must be a valid goal.

Semantics:

```
transform(V_in, V_out, not DCG_goal) =
(not transform(V_in, V_new, DCG_goal), V_out = V_in)
```

where V_new is a fresh variable.

`if` *CondGoal* `then` *ThenGoal* `else` *ElseGoal*
CondGoal `->` *ThenGoal* `;` *ElseGoal*
: A DCG if-then-else. The two different syntaxes have identical semantics. *CondGoal*, *ThenGoal*, and *ElseGoal* must be valid DCG-goals.

  Semantics:

```
transform(V_in, V_out, if CondGoal then ThenGoal else ElseGoal) =
if transform(V_in, V_cond, CondGoal) then
        transform(V_cond, V_out, ThenGoal)
else
        transform(V_in, V_out, ElseGoal)
```

`=(`*Term*`)`
: A DCG unification. Unifies *Term* with the implicit DCG argument. *Term* must be a valid data-term.

  Semantics:

```
transform(V_in, V_out, =(Term)) = (Term = V_in, V_out = V_in)
```

`:=(`*Term*`)`
: A DCG output unification. Unifies *Term* with the implicit DCG output argument, ignoring the input DCG argument. *Term* must be a valid data-term.

  Semantics:

```
transform(V_in, V_out, :=(Term)) = (V_out = Term)
```

Term `=^` *field_list*
: A DCG field selection. Unifies *Term* with the result of applying the field selection *field_list* to the implicit DCG argument. *Term* must be a valid data-term. *field_list* must be a valid field list. See Section 2.15.2 [Record syntax], page 22.

  Semantics:

```
transform(V_in, V_out, Term =^ field_list) =
        (Term = V_in ^ field_list, V_out = V_in)
```

`^` *field_list* `:=` *Term*
: A DCG field update. Replaces a field in the implicit DCG argument. *Term* must be a valid data-term. *field_list* must be a valid field list. See Section 2.15.2 [Record syntax], page 22.

  Semantics:

```
transform(V_in, V_out, ^ field_list := Term) =
        (V_out = V_in ^ field_list := Term)
```

DCG-call
: Any term which does not match any of the above forms must be a DCG predicate call. If the term is a variable *Var*, it is treated as if it were '`call(`*Var*`)`'. Then, the two implicit DCG arguments are appended to the specified arguments.

  Semantics:

```
transform(V_in, V_out, p(A1, ..., AN)) =
p(A1, ..., AN, V_in, V_out)
```

2.15 Data-terms

Syntactically, a data-term is just a term.

There are a couple of differences from Prolog. The first one is that double-quoted strings are atomic in Mercury, they are not abbreviations for lists of character codes. The second is that Mercury provides several extensions to Prolog's term syntax: Mercury terms may contain record field selection and field update expressions, conditional (if-then-else) expressions, function applications, higher-order function applications, lambda expressions, and explicit type qualifications.

A data-term is either a variable, a data-functor, or a special data-term. A special data-term is a conditional expression, a record syntax expression, a unification expression, a lambda expression, a higher-order function application, or an explicit type qualification.

2.15.1 Data-functors

A data-functor is an integer, a float, a string, a character literal (any single-character name), a name, an implementation-defined literal, or a compound data-term. A compound data-term is a compound term which does not match the form of a special data-term (see Section 2.15 [Data-terms], page 22), and whose arguments are data-terms. If a data-functor is a name or a compound data-term, its top-level functor must name a function, predicate, or data constructor declared in the program or in the interface of an imported module.

Implementation-defined literals are symbolic names whose value represents a property of the compilation environment or the context in which it appears. The implementation replaces these symbolic names with actual constants during compilation. Implementation-defined literals can only appear within clauses. The following literals must be supported by all Mercury implementations:

'`$file`' a string that gives the name of the file that contains the module being compiled. If the name of the file cannot be determined then it is replaced by an arbitrary string.

'`$line`' the line number (integer) of the goal in which the literal appears or -1 if it cannot be determined.

'`$module`' a string representation of the fully-qualified module name.

'`$pred`' a string containing the fully-qualified predicate or function name and arity.

The Melbourne Mercury implementation additionally supports the following extension:

'`$grade`' the grade (string) in which the module is compiled.

2.15.2 Record syntax

Record syntax provides a convenient way to select or update fields of data constructors, independent of the definition of the constructor. Record syntax expressions are transformed into sequences of calls to field selection or update functions (see Section 3.4 [Field access functions], page 35).

A field specifier is a name or a compound data-term. A field list is a list of field specifiers separated by `^`. `field`, `field1 ^ field2` and `field1(A) ^ field2(B, C)` are all valid field lists.

If the top-level functor of a field specifier is '*`field/N`*', there must be a visible selection function '*`field/(N + 1)`*'. If the field specifier occurs in a field update expression, there must also be a visible update function named '*`'field :='/(N + 2)`*'.

Record syntax expressions have one of the following forms. There are also record syntax DCG goals (see Section 2.14 [DCG-goals], page 19), which provide similar functionality to record syntax expressions, except that they act on the DCG arguments of a DCG clause.

`Term ^ field_list`

A field selection. For each field specifier in *field_list*, apply the corresponding selection function in turn.

Term must be a valid data-term. *field_list* must be a valid field list.

A field selection is transformed using the following rules:

```
transform(Term ^ Field(Arg1, ...)) = Field(Arg1, ..., Term).
transform(Term ^ Field(Arg1, ...) ^ Rest) =
                transform(Field(Arg1, ..., Term) ^ Rest).
```

Examples:

`Term ^ field` is equivalent to `field(Term)`.

`Term ^ field(Arg)` is equivalent to `field(Arg, Term)`.

`Term ^ field1(Arg1) ^ field2(Arg2, Arg3)` is equivalent to `field2(Arg2, Arg3, field1(Arg1, Term))`.

`Term ^ field_list := FieldValue`

A field update, returning a copy of *Term* with the value of the field specified by *field_list* replaced with *FieldValue*.

Term must be a valid data-term. *field_list* must be a valid field list.

A field update is transformed using the following rules:

```
transform(Term ^ Field(Arg1, ...) := FieldValue) =
                'Field :='(Arg1, ..., Term, FieldValue)).

transform(Term0 ^ Field(Arg1, ...) ^ Rest := FieldValue) = Term :-
        OldFieldValue = Field(Arg1, ..., Term0),
        NewFieldValue = transform(OldFieldValue ^ Rest := FieldValue),
        Term = 'Field :='(Arg1, ..., Term0, NewFieldValue).
```

Examples:

`Term ^ field := FieldValue` is equivalent to `'field :='(Term, FieldValue)`.

`Term ^ field(Arg) := FieldValue` is equivalent to `'field :='(Arg, Term, FieldValue)`.

`Term ^ field1(Arg1) ^ field2(Arg2) := FieldValue` is equivalent to the code

```
OldField1 = field1(Arg1, Term),
NewField1 = 'field2 :='(Arg2, OldField1, FieldValue),
Result = 'field1 :='(Arg1, Term, NewField1)
```

2.15.3 Unification expressions

A unification expression is an expression of the form

```
X @ Y
```

where *X* and *Y* are data-terms.

The meaning of a unification expression is that the arguments are unified, and the expression is equivalent to the unified value.

The strict sequential operational semantics (see Chapter 13 [Semantics], page 96) of an expression `X @ Y` is that the expression is replaced by a fresh variable `Z`, and immediately after `Z` is evaluated, the conjunction `Z = X, Z = Y` is evaluated.

For example

```
p(X @ f(_, _), X).
```

is equivalent to

```
p(H1, H2) :-
    H1 = X,
    H1 = f(_, _),
    H2 = X.
```

Unification expressions are most useful when writing switches (see Section 6.2 [Determinism checking and inference], page 50). The arguments of a unification expression are examined when checking for switches. The arguments of an equivalent user-defined function would not be.

2.15.4 Conditional expressions

A conditional expression is an expression of either of the two following forms

```
(if Goal then Expression1 else Expression2)
(Goal -> Expression1 ; Expression2)
```

Goal is a goal; *Expression1* and *Expression2* are both data-terms. The semantics of a conditional expression is that if *Goal* is true, then the expression has the meaning of *Expression1*, else the expression has the meaning of *Expression2*.

If *Goal* takes the form `some [X, Y, Z] ...` then the scope of *X*, *Y*, and *Z* includes *Expression1*.

2.15.5 Lambda expressions

A lambda expression is a compound term of one of the following forms

```
pred(Arg1::Mode1, Arg2::Mode2, ...) is Det :- Goal
pred(Arg1::Mode1, Arg2::Mode2, ..., DCGMode0, DCGMode1) is Det --> DCGGoal
func(Arg1::Mode1, Arg2::Mode2, ...) = (Result::Mode) is Det :- Goal
func(Arg1, Arg2, ...) = (Result) is Det :- Goal
func(Arg1, Arg2, ...) = Result :- Goal
```

where *Arg1*, *Arg2*, ... are zero or more data-terms, *Result* is a data-term, *Mode1*, *Mode2*, ... are zero or more modes (see Chapter 4 [Modes], page 39), *DCGMode0* and *DCGMode1* are modes (see Chapter 4 [Modes], page 39), *Det* is a determinism (see Chapter 6 [Determinism], page 49), *Goal* is a goal (see Section 2.11 [Goals], page 10), and *DCGGoal* is a DCG Goal (see Section 2.14 [DCG-goals], page 19). The '`:- Goal`' part is optional; if it is

not specified, then '`:- true`' is assumed. A lambda expression denotes a higher-order predicate or function term whose value is the predicate or function of the specified arguments determined by the specified goal. See Chapter 8 [Higher-order], page 60.

A lambda expression introduces a new scope: any variables occurring in the arguments *Arg1*, *Arg2*, ... are locally quantified, i.e. any occurrences of variables with that name in the lambda expression are considered to name a different variable than any variables with the same name that occur outside of the lambda expression. For variables which occur in *Result* or *Goal*, but not in the arguments, the usual Mercury rules for implicit quantification apply (see Section 2.17 [Implicit quantification], page 26).

The form of lambda expression using '`-->`' as its top level functor is a syntactic abbreviation: an expression of the form

```
pred(Var1::Mode1, Var2::Mode2, ..., DCGMode0, DCGMode1) is Det --> DCGGoal
```

is equivalent to

```
pred(Var1::Mode1, Var2::Mode2, ...,
    DCGVar0::DCGMode0, DCGVar1::DCGMode1) is Det :- Goal
```

where *DCGVar0* and *DCGVar1* are fresh variables, and *Goal* is the result of '`DCG-transform(DCGVar0, DCGVar1, DCGGoal)`' where DCG-transform is the function specified in Section 2.14 [DCG-goals], page 19.

2.15.6 Higher-order function applications

A higher-order function application is a compound term of one of the following two forms

```
apply(Func, Arg1, Arg2, ..., ArgN)
FuncVar(Arg1, Arg2, ..., ArgN)
```

where *N* >= 0, *Func* is a term of type '`func(T1, T2, ..., Tn) = T`', *FuncVar* is a variable of that type, and *Arg1*, *Arg2*, ..., *ArgN* are terms of types '`T1`', '`T2`', ..., '`Tn`'. The type of the higher-order function application term is *T*. It denotes the result of applying the specified function to the specified arguments. See Chapter 8 [Higher-order], page 60.

2.15.7 Explicit type qualification

Explicit type qualifications are occasionally useful to resolve ambiguities that can arise from overloading or polymorphic types.

An explicit type qualification expression is a term of the form

```
Term : Type
```

Term must be a valid data-term. *Type* must be a valid type (see Chapter 3 [Types], page 28).

An explicit type qualification expression constrains the specified term to have the specified type. Apart from that, the meaning of an explicit type qualification expression is just the same as the specified *Term*.

Currently we also support the following alternative syntax for type qualification:

```
with_type(Term, Type)
```

or equivalently, as it is more commonly written,

```
Term `with_type` Type
```

2.16 Variable scoping

There are three sorts of variables in Mercury: ordinary variables, type variables, and inst variables.

Variables occurring in types are called type variables. Variables occurring in insts or modes are called inst variables. Variables that occur in data-terms, and that are not inst variables or type variables, are called ordinary variables.

(Type variables can occur in data-terms in the right-hand [*Type*] operand of an explicit type qualification. Inst variables can occur in data-terms in the right-hand [*Mode*] operand of an explicit mode qualification. Apart from that, all other variables in data-terms are ordinary variables.)

The three different variable sorts occupy different namespaces: there is no semantic relationship between two variables of different sorts (e.g. a type variable and an ordinary variable) even if they happen to share the same name. (However, as a matter of programming style, it is generally a bad idea to use the same name for variables of different sorts in the same clause.)

The scope of ordinary variables is the clause or declaration in which they occur, unless they are quantified, either explicitly (see Section 2.11 [Goals], page 10) or implicitly (see Section 2.17 [Implicit quantification], page 26).

The scope of type variables in a predicate or function's type declaration extends over any explicit type qualifications (see Section 2.15.7 [Explicit type qualification], page 25) in the clauses for that predicate or function, and over '`pragma type_spec`' (see Section 18.2 [Type specialization], page 142) declarations for that predicate or function, so that explicit type qualifications and '`pragma type_spec`' declarations can refer to those type variables. The scope of any type variables in an explicit type qualification which do not occur in the predicate or function's type declaration is the clause in which they occur.

The scope of inst variables is the clause or declaration in which they occur.

2.17 Implicit quantification

The rule for implicit quantification in Mercury is not the same as the usual one in mathematical logic. In Mercury, variables that do not occur in the head of a clause are implicitly existentially quantified around their closest enclosing scope (in a sense to be made precise in the following paragraphs). This allows most existential quantifiers to be omitted, and leads to more concise code.

An occurrence of a variable is *in a negated context* if it is in a negation, in a universal quantification, in the condition of an if-then-else, in an inequality, or in a lambda expression.

Two goals are *parallel* if they are different disjuncts of the same disjunction, or if one is the "else" part of an if-then-else and the other goal is either the "then" part or the condition of the if-then-else, or if they are the goals of disjoint (distinct and non-overlapping) lambda expressions.

If a variable occurs in a negated context and does not occur outside of that negated context other than in parallel goals (and in the case of a variable in the condition of an if-then-else, other than in the "then" part of the if-then-else), then that variable is implicitly existentially quantified inside the negation.

2.18 Elimination of double negation

The treatment of inequality, universal quantification, implication, and logical equivalence as abbreviations can cause the introduction of double negations which could make otherwise well-formed code mode-incorrect. To avoid this problem, the language specifies that after syntax analysis and implicit quantification, and before mode analysis is performed, the implementation must delete any double negations and must replace any negations of conjunctions of negations with disjunctions. (Both of these transformations preserve the logical meaning and type-correctness of the code, and they preserve or improve mode-correctness: they never transform code fragments that would be well-moded into ones that would be ill-moded.)

3 Types

The type system is based on many-sorted logic, and supports polymorphism, type classes (see Chapter 10 [Type classes], page 75), and existentially quantified types (see Chapter 11 [Existential types], page 85).

3.1 Builtin types

This section describes the special types that are built into the Mercury implementation, or are defined in the standard library.

3.1.1 Primitive types

There is a special syntax for constants of all primitive types except `char`. (For `char`, the standard syntax suffixes.)

3.1.1.1 Signed integer types

There are five primitive signed integer types: `int`, `int8`, `int16`, `int32` and `int64`.

Except for `int`, the width in bits of each of these is given by the numeric suffix in its name.

The width in bits of `int` is implementation defined, but must be at least 32-bits.

All signed integer types use two's-complement representation. Their width must be equal to the width of the corresponding unsigned type.

Values of the type `int8` must be in the range -128 ($-(2^{8-1})$) to 127 ($2^{8-1}-1$), both inclusive.

Values of the type `int16` must be in the range -32768 ($-(2^{16-1})$) to 32767 ($2^{16-1}-1$), both inclusive.

Values of the type `int32` must be in the range -2147483648 ($-(2^{32-1})$) to 2147483647 ($2^{32-1}-1$), both inclusive.

Values of the type `int64` must be in the range -9223372036854775808 ($-(2^{64-1})$) to 9223372036854775807 ($2^{64-1}-1$), both inclusive.

Values of the type `int` must be in the range to $-(2^{N-1})$ to $2^{N-1}-1$, both inclusive; N being the width of `int` in bits.

3.1.1.2 Unsigned integer types

There are five primitive unsigned integer types: `uint`, `uint8`, `uint16`, `uint32` and `uint64`.

Except for `uint`, the width in bits of each of these types is given by the numeric suffix in its name.

The width in bits of `uint` is implementation defined, but must be at least 32-bits. It must be equal to the width of the type `int`.

Values of the type `uint8` must be in the range 0 (2^0-1) to 255 (2^8-1), both inclusive.

Values of the type `uint16` must be in the range 0 (2^0-1) to 65535 (2^16-1), both inclusive.

Values of the type `uint32` must be in the range 0 (2^0-1) to 4294967295 (2^32-1), both inclusive.

Values of the type `uint64` must be in the range 0 ($2^0 - 1$) to 18446744073709551615 ($2^64 - 1$), both inclusive.

Values of the type *uint* must be in the range 0 ($2^0 - 1$) to $2^N - 1$, both inclusive; N being the width of `uint` in bits.

3.1.1.3 Floating-point type

There is one floating-point type: `float`.

It is represented using either the 32-bit single-precision IEEE 754 format or the 64-bit double-precision IEEE 754 format.

The choice between the two formats is implementation dependent.

In the Melbourne Mercury implementation, `float`s are represented using the 32-bit single-precision IEEE 754 format in grades that have `.spf` grade component, and using the 64-bit double-precision IEEE 754 format in every other grade.

3.1.1.4 Character type

There is one character type: `char`.

Values of this type represent Unicode code points.

3.1.1.5 String type

There is one string type: `string`.

A string is a sequence of characters encoded using either the UTF-8 or UTF-16 encoding of Unicode.

The choice between the two encodings is implementation dependent.

In the Melbourne Mercury implementation, `string`s are represented using UTF-8 when generating code for C or Erlang, and using UTF-16 when generating code for C# or Java.

3.1.2 Other builtin types

3.1.2.1 Predicate and function types

The predicate types are `pred`, `pred(T)`, `pred(T1, T2)`, ...

The function types are `(func) = T`, `func(T1) = T`, `func(T1, T2) = T`, ...

Higher-order predicate and function types are used to pass closures to other predicates and functions. See Chapter 8 [Higher-order], page 60.

3.1.2.2 Tuple types

The tuple types are `{}`, `{T}`, `{T1, T2}`, ...

A tuple type is equivalent to a discriminated union type (see Section 3.2.1 [Discriminated unions], page 30) with declaration

```
:- type {Arg1, Arg2, ..., ArgN}
        --->    { {Arg1, Arg2, ..., ArgN} }.
```

3.1.2.3 The universal type

The type `univ` is defined in the standard library module `univ`, along with the predicates `type_to_univ/2` and `univ_to_type/2`. With those predicates, values of any type can be converted to the universal type and back again. The universal type is useful for situations where you need heterogeneous collections.

3.1.2.4 The "state-of-the-world" type

The type `io.state` is defined in the standard library module `io`, and represents the state of the world. Predicates which perform I/O are passed the only reference to the current state of the world, and produce a unique reference to the new state of the world. In this way, we can give a declarative semantics to code that performs I/O.

3.2 User-defined types

New types can be introduced with '`:- type`' declarations. There are several categories of derived types:

3.2.1 Discriminated unions

These encompass both enumeration and record types in other languages. A derived type is defined using '`:- type` *`type`* `--->` *`body`*'. (Note there are *three* dashes in that arrow. It should not be confused with the two-dash arrow used for DCGs or the one-dash arrow used for if-then-else.) If the *type* term is a functor of arity zero (i.e. one having zero arguments), it names a monomorphic type. Otherwise, it names a polymorphic type; the arguments of the functor must be distinct type variables. The *body* term is defined as a sequence of constructor definitions separated by semi-colons.

Ordinarily, each constructor definition must be a functor whose arguments (if any) are types. Ordinary discriminated union definitions must be *transparent*: all type variables occurring in the *body* must also occur in the *type*.

However, constructor definitions can optionally be existentially typed. In that case, the functor will be preceded by an existential type quantifier and can optionally be followed by an existential type class constraint. For details, see Chapter 11 [Existential types], page 85. Existentially typed discriminated union definitions need not be transparent.

The arguments of constructor definitions may be labelled. These labels cause the compiler to generate functions which can be used to conveniently select and update fields of a term in a manner independent of the definition of the type (see Section 3.4 [Field access functions], page 35). A labelled argument has the form *`fieldname`* `::` *`Type`*. It is an error for two fields in the same module to have the same label.

Here are some examples of discriminated union definitions:

```
:- type fruit
        --->    apple
        ;       orange
        ;       banana
        ;       pear.

:- type strange
        --->    foo(int)
```

```
        ;       bar(string).

:- type employee
        --->    employee(
                        name        :: string,
                        age         :: int,
                        department  :: string
                ).

:- type tree
        --->    empty
        ;       leaf(int)
        ;       branch(tree, tree).

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

:- type pair(T1, T2)
        --->    T1 - T2.
```

If the body of a discriminated union type definition contains a term whose top-level functor is `';'/2`, the semi-colon is normally assumed to be a separator. This makes it difficult to define a type whose constructors include `';'/2`. To allow this, curly braces can be used to quote the semi-colon. It is then also necessary to quote curly braces. The following example illustrates this:

```
:- type tricky
        --->    { int ; int }
        ;       { { int } }.
```

This defines a type with two constructors, `';'/2` and `'{}'/1`, whose argument types are all `int`. We recommend against using constructors named `'{}'` because of the possibility of confusion with the builtin tuple types.

Each discriminated union type definition introduces a distinct type. Mercury considers two discriminated union types that have the same bodies to be distinct types (name equivalence). Having two different definitions of a type with the same name and arity in the same module is an error.

Constructors may be overloaded among different types: there may be any number of constructors with a given name and arity, so long as they all have different types. However, there must not be more than one constructor with the same name, arity, and result type in the same module. (There is no particularly good reason for this restriction; in the future we may allow several such functors as long as they have different argument types.) Note that excessive overloading of constructors can slow down type checking and can make the program confusing for human readers, so overloading should not be over-used.

Note that user defined types may not have names that have meanings in Mercury. (Most of these are documented in later sections.)

The list of reserved type names is

```
int
int8
int16
int32
int64
uint
uint8
uint16
uint32
uint64
float
character
string
{}
=
pred
func
pure
semipure
impure
''
```

3.2.2 Equivalence types

These are type abbreviations. They are defined using '`==`' as follows. They may be polymorphic.

```
:- type money == int.
:- type assoc_list(KeyType, ValueType)
        == list(pair(KeyType, ValueType)).
```

Equivalence type definitions must be transparent. Unlike discriminated union type definitions, equivalence type definitions must not be cyclic; that is, the type on the left hand side of the '`==`' ('`assoc_list`' and '`money`' in the examples above) must not occur on the right hand side of the '`==`'.

Mercury treats an equivalence type as an abbreviation for the type on the right hand side of the definition; the two are equivalent in all respects in scopes where the equivalence type is visible.

3.2.3 Abstract types

These are types whose implementation is hidden. The type declarations

```
:- type t1.
:- type t2(T1, T2).
```

declare types `t1/0` and `t2/2` to be abstract types. Such declarations are only useful in the interface section of a module. This means that the type names will be exported, but the constructors (functors) for these types will not be exported. The implementation section of a module must give a definition for all of the abstract types named in the interface section of the module. Abstract types may be defined as either discriminated union types or as equivalence types.

3.3 Predicate and function type declarations

The argument types of each predicate must be explicitly declared with a '`:- pred`' declaration. The argument types and return type of each function must be explicitly declared with a '`:- func`' declaration. For example:

```
:- pred is_all_uppercase(string).

:- func strlen(string) = int.
```

Predicates and functions can be polymorphic; that is, their declarations can include type variables. For example:

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

:- func length(list(T)) = int.
```

A predicate or function can be declared to have a given higher-order type (see Chapter 8 [Higher-order], page 60) by using an explicit type qualification in the type declaration. This is useful where several predicates or functions need to have the same type signature, which often occurs for type class method implementations (see Chapter 10 [Type classes], page 75), and for predicates to be passed as higher-order terms.

For example,

```
:- type foldl_pred(T, U) == pred(T, U, U).
:- type foldl_func(T, U) == (func(T, U) = U).

:- pred p(int) : foldl_pred(T, U).
:- func f(int) : foldl_func(T, U).
```

is equivalent to

```
:- pred p(int, T, U, U).
:- pred f(int, T, U) = U.
```

Type variables in predicate and function declarations are implicitly universally quantified by default; that is, the predicate or function may be called with arguments and (in the case of functions) return value whose actual types are any instance of the types specified in the declaration. For example, the function '`length/1`' declared above could be called with the argument having type '`list(int)`', or '`list(float)`', or '`list(list(int))`', etc.

Type variables in predicate and function declarations can also be existentially quantified; this is discussed in Chapter 11 [Existential types], page 85.

There must only be one predicate with a given name and arity in each module, and only one function with a given name and arity in each module. It is an error to declare the same predicate or function twice.

There must be at least one clause defined for each declared predicate or function, except for those defined using the foreign language interface (see Chapter 14 [Foreign language interface], page 98). However, Mercury implementations are permitted to provide a method of processing Mercury programs in which such errors are not reported until and unless the predicate or function is actually called. (The University of Melbourne Mercury implementation provides this with its '`--allow-stubs`' option. This can be useful during program development, since it allows you to execute parts of a program while the program's implementation is still incomplete.)

Note that a predicate defined using DCG notation (see Section 2.13 [DCG-rules], page 19) will appear to be defined with two fewer arguments than it is declared with. It will also appear to be called with two fewer arguments when called from predicates defined using DCG notation. However, when called from an ordinary predicate or function, it must have all the arguments it was declared with.

The compiler infers the types of data-terms, and in particular the types of variables and overloaded constructors, functions, and predicates. A *type assignment* is an assignment of a type to every variable and of a particular constructor, function, or predicate to every name in a clause. A type assignment is *valid* if it satisfies the following conditions.

Each constructor in a clause must have been declared in at least one visible type declaration. The type assigned to each constructor term must match one of the type declarations for that constructor, and the types assigned to the arguments of that constructor must match the argument types specified in that type declaration.

The type assigned to each function call term must match the return type of one of the '`:- func`' declarations for that function, and the types assigned to the arguments of that function must match the argument types specified in that type declaration.

The type assigned to each predicate argument must match the type specified in one of the '`:- pred`' declarations for that predicate. The type assigned to each head argument in a predicate clause must exactly match the argument type specified in the corresponding '`:- pred`' declaration.

The type assigned to each head argument in a function clause must exactly match the argument type specified in the corresponding '`:- func`' declaration, and the type assigned to the result term in a function clause must exactly match the result type specified in the corresponding '`:- func`' declaration.

The type assigned to each data-term with an explicit type qualification (see Section 2.15.7 [Explicit type qualification], page 25) must match the type specified by the type qualification expression[1].

(Here "match" means to be an instance of, i.e. to be identical to for some substitution of the type parameters, and "exactly match" means to be identical up to renaming of type parameters.)

One type assignment A is said to be *more general* than another type assignment B if there is a binding of the type parameters in A that makes it identical (up to renaming of parameters) to B. If there is more than one valid type assignment, the compiler must choose the most general one. If there are two valid type assignments which are not identical up to renaming and neither of which is more general than the other, then there is a type ambiguity, and compiler must report an error. A clause is *type-correct* if there is a unique (up to renaming) most general valid type assignment. Every clause in a Mercury program must be type-correct.

[1] The type of an explicitly type qualified term may be an instance of the type specified by the qualifier. This allows explicit type qualifications to constrain the types of two data-terms to be identical, without knowing the exact types of the data-terms. It also allows type qualifications to refer to the types of the results of existentially typed predicates or functions.

3.4 Field access functions

Fields of constructors of discriminated union types may be labelled (see Section 3.2.1 [Discriminated unions], page 30). These labels cause the compiler to generate functions which can be used to select and update fields of a term in a manner independent of the definition of the type.

The Mercury language includes syntactic sugar to make it more convenient to select and update fields inside nested terms (see Section 2.15.2 [Record syntax], page 22) and to select and update fields of the DCG arguments of a clause (see Section 2.14 [DCG-goals], page 19).

3.4.1 Field selection

```
field(Term)
```

Each field label '*field*' in a constructor causes generation of a field selection function '*field*/1', which takes a data-term of the same type as the constructor and returns the value of the labelled field, failing if the top-level constructor of the argument is not the constructor containing the field.

If the declaration of the field is in the interface section of the module, the corresponding field selection function is also exported from the module.

By default, this function has no declared modes — the modes are inferred at each call to the function. However, the type and modes of this function may be explicitly declared, in which case it will have only the declared modes.

To create a higher-order term from a field selection function, an explicit lambda expression must be used, unless a single mode declaration is supplied for the field selection function.

3.4.2 Field update

```
'field :='(Term, ValueTerm)
```

Each field label '*field*' in a constructor causes generation of a field update function ''*field* :='/2'. The first argument of this function is a data-term of the same type as the constructor. The second argument is a data-term of the same type as the labelled field. The return value is a copy of the first argument with the value of the labelled field replaced by the second argument. ''*field* :='/2' fails if the top-level constructor of the first argument is not the constructor containing the labelled field.

If the declaration of the field is in the interface section of the module, the corresponding field update function is also exported from the module.

By default, this function has no declared modes — the modes are inferred at each call to the function. However, the type and modes of this function may be explicitly declared, in which case it will have only the declared modes.

To create a higher-order term from a field update function, an explicit lambda expression must be used, unless a single mode declaration is supplied for the field update function.

Some fields cannot be updated using field update functions. For the constructor 'unsettable/2' below, neither field may be updated because the resulting term would not be well-typed. A future release may allow multiple fields to be updated by a single expression to avoid this problem.

```
:- type unsettable
        --->    some [T] unsettable(
                    unsettable1 :: T,
                    unsettable2 :: T
                ).
```

3.4.3 User-supplied field access function declarations

Type and mode declarations for compiler-generated field access functions for fields of constructors local to a module may be placed in the interface section of the module. This allows the implementation of a type to be hidden while still allowing client modules to use record syntax to manipulate values of the type. Supplying a type declaration and a single mode declaration also allows higher-order terms to be created from a field access function without using explicit lambda expressions.

Declarations for field access functions for fields occurring in the interface section of a module must also occur in the interface section.

Declarations and clauses for field access functions can also be supplied for fields which are not a part of any type. This is useful when the data structures of a program change so that a value which was previously stored as part of a type is now computed each time it is requested. It also allows record syntax to be used for type class methods.

User-declared field access functions may take extra arguments. For example, the Mercury standard library module `map` contains the following functions:

```
:- func elem(K, map(K, V)) = V is semidet.
:- func 'elem :='(K, map(K, V), V) = map(K, V).
```

Field access syntax may be used at the top-level of `func` and `mode` declarations and in the head of clauses. For instance:

```
:- func map(K, V) ^ elem(K) = V.
:- mode in        ^ in      = out is semidet.
Map ^ elem(Key) = map.lookup(Map, Key).

:- func (map(K, V) ^ elem(K) := V)  = V.
:- mode (in        ^ in      := in) = out is semidet.
(Map ^ elem(Key) := Value) = map.set(Map, Key, Value).
```

The Mercury standard library modules `array` and `bt_array` define similar functions.

3.4.4 Field access examples

The examples make use of the following type declarations:

```
:- type type1
        --->    type1(
                    field1 :: type2,
                    field2 :: string
                ).

:- type type2
        --->    type2(
                    field3 :: int,
```

```
        field4 :: int
    ).
```

The compiler generates some field access functions for ‘`field1`’. The functions generated for the other fields are similar.

```
:- func type1 ^ field1 = type2.
type1(Field1, _) ^ field1 = Field1.

:- func (type1 ^ field1 := type2) = type1.
(type1(_, Field2) ^ field1 := Field1) = type1(Field1, Field2).
```

Using these functions and the syntactic sugar described in Section 2.15.2 [Record syntax], page 22, programmers can write code such as

```
:- func type1 ^ increment_field3 = type1.

Term0 ^ increment_field3 =
    Term0 ^ field1 ^ field3 := Term0 ^ field1 ^ field3 + 1.
```

The compiler expands this into

```
increment_field3(Term0) = Term :-
    OldField3 = field3(field1(Term0)),
    OldField1 = field1(Term0),
    NewField1 = 'field3 :='(OldField1, OldField3 + 1),
    Term = 'field1 :='(Term0, NewField1).
```

The field access functions defined in the Mercury standard library module ‘`map`’ can be used as follows:

```
:- func update_field_in_map(map(int, type1), int, string)
    = map(int, type1) is semidet.

update_field_in_map(Map, Index, Value) =
    Map ^ elem(Index) ^ field2 := Value.
```

3.5 The standard ordering

For every Mercury type there exists a standard ordering; any two values of the same type can be compared under this ordering by using the `builtin.compare/3` predicate. The ordering is total, meaning that the corresponding binary relations are reflexive, transitive and anti-symmetric.

The existence of this ordering makes it possible to implement generic data structures such as sets and maps, without needing to know the specifics of the ordering. Furthermore, different platforms often have their own natural orderings which are not necessarily consistent with each other. As such, the standard ordering for most types is not fully defined.

For the primitive integer types, the standard ordering is the usual numerical ordering. Implementations should reject code containing overflowing integer literals.

For the primitive type `float`, the standard ordering approximates the usual numerical ordering. If the result of `builtin.compare/3` is `(<)` or `(>)` then this relation holds in the numerical ordering, but this is not necessarily the case for `(=)` due to lack of precision. In the standard ordering, “negative” and “positive” zero values are equal. Implementations

should replace overflowing literals with the infinity of the same sign; in the standard ordering positive infinity is greater than all finite values and negative infinity is less than all finite values. Implementations must throw an exception when comparing a "not a number" (NaN) value.

For the primitive type `char`, the standard ordering is the numerical ordering of the Unicode code point values.

For the primitive type `string`, the standard ordering is implementation dependent. The current implementation performs string comparison using the C `strcmp()` function, the Java `String.compareTo()` method, the C# `System.String.CompareOrdinal()` method, and the Erlang term comparison operators, when compiling to C, Java, C# and Erlang respectively.

For tuple types, corresponding arguments are compared, with the first argument being the most significant.

For discriminated union types, if both values have the same principal constructor then corresponding arguments are compared, with the first argument being the most significant. If the values have different principal constructors, the result of comparing them is not defined.

4 Modes

4.1 Insts, modes, and mode definitions

The *mode* of a predicate, or function, is a mapping from the initial state of instantiation of the arguments of the predicate, or the arguments and result of a function, to their final state of instantiation. To describe states of instantiation, we use information provided by the type system. Types can be viewed as regular trees with two kinds of nodes: or-nodes representing types and and-nodes representing constructors. The children of an or-node are the constructors that can be used to construct terms of that type; the children of an and-node are the types of the arguments of the constructors. We attach mode information to the or-nodes of type trees.

An *instantiatedness tree* is an assignment of an *instantiatedness* — either *free* or *bound* — to each or-node of a type tree, with the constraint that all descendants of a free node must be free.

A term is *approximated by* an instantiatedness tree if for every node in the instantiatedness tree,

- if the node is "free", then the corresponding node in the term (if any) is a free variable that does not share with any other variable (we call such variables *distinct*);
- if the node is "bound", then the corresponding node in the term (if any) is a function symbol.

When an instantiatedness tree tells us that a variable is bound, there may be several alternative function symbols to which it could be bound. The instantiatedness tree does not tell us which of these it is bound to; instead for each possible function symbol it tells us exactly which arguments of the function symbol will be free and which will be bound. The same principle applies recursively to these bound arguments.

Mercury's mode system allows users to declare names for instantiatedness trees using declarations such as

```
:- inst listskel == bound( [] ; [free | listskel] ).
```

This instantiatedness tree describes lists whose skeleton is known but whose elements are distinct variables. As such, it approximates the term `[A,B]` but not the term `[H|T]` (only part of the skeleton is known), the term `[A,2]` (not all elements are variables), or the term `[A,A]` (the elements are not distinct variables).

As a shorthand, the mode system provides `free` and `ground` as names for instantiatedness trees all of whose nodes are free and bound respectively (with the exception of solver type values which may be semantically ground, but be defined in terms of non-ground solver type values; see Chapter 16 [Solver types], page 135 for more detail). The shape of these trees is determined by the type of the variable to which they apply.

A more concise, alternative syntax exists for `bound` instantiatedness trees:

```
:- inst maybeskel
    --->    no
    ;       yes(ground).
```

which is equivalent to writing

```
:- inst maybeskel == bound(no ; yes(ground)).
```

As execution proceeds, variables may become more instantiated. A *mode mapping* is a mapping from an initial instantiatedness tree to a final instantiatedness tree, with the constraint that no node of the type tree is transformed from bound to free. Mercury allows the user to specify mode mappings directly by expressions such as `inst1 >> inst2`, or to give them a name using declarations such as

```
:- mode m == inst1 >> inst2.
```

Two standard shorthand modes are provided, corresponding to the standard notions of inputs and outputs:

```
:- mode in == ground >> ground.
:- mode out == free >> ground.
```

Prolog fans who want to use the symbols '`+`' and '`-`' can do so by simply defining them using a mode declaration:

```
:- mode (+) == in.
:- mode (-) == out.
```

These two modes are enough for most functions and predicates. Nevertheless, Mercury's mode system is sufficiently expressive to handle more complex data-flow patterns, including those involving partially instantiated data structures i.e. data structures with "free" *holes* in them. (In the current implementation, partially instantiated data structures are unsupported due to a lack of alias tracking in the mode system. For more information please see the '`LIMITATIONS`' file distributed with Mercury.)

For example, consider an interface to a database that associates data with keys, and provides read and write access to the items it stores. To represent accesses to the database over a network, you would need declarations such as

```
:- type operation
     --->   lookup(key, data)
     ;      set(key, data).
:- inst request
    --->    lookup(ground, free)
    ;       set(ground, ground).
:- mode create_request == free >> request.
:- mode satisfy_request == request >> ground.
```

'`inst`' and '`mode`' declarations can be parametric. For example, the following declaration

```
:- inst listskel(Inst)
    --->    []
    ;       [Inst | listskel(Inst)].
```

defines the inst '`listskel(Inst)`' to be a list skeleton whose elements have inst `Inst`; you can then use insts such as '`listskel(listskel(free))`', which represents the instantiation state of a list of lists of free variables. The standard library provides the parametric modes

```
:- mode in(Inst) == Inst >> Inst.
:- mode out(Inst) == free >> Inst.
```

so that for example the mode '`create_request`' defined above could have be defined as

```
:- mode create_request == out(request).
```

There must not be more than one inst definition with the same name and arity in the same module. Similarly, there must not be more than one mode definition with the same name and arity in the same module.

Note that user defined insts and modes may not have names that have meanings in Mercury. (Most of these are documented in later sections.)

The list of reserved inst names is

```
=<
any
bound
bound_unique
clobbered
clobbered_any
free
ground
is
mostly_clobbered
mostly_unique
mostly_unique_any
not_reached
unique
unique_any
```

The list of reserved mode names is

```
=
>>
any_func
any_pred
func
is
pred
```

4.2 Predicate and function mode declarations

A *predicate mode declaration* assigns a mode mapping to each argument of a predicate. A *function mode declaration* assigns a mode mapping to each argument of a function, and a mode mapping to the function result. Each mode of a predicate or function is called a *procedure*. For example, given the mode names defined by

```
:- mode out_listskel == free >> listskel.
:- mode in_listskel == listskel >> listskel.
```

the (type and) mode declarations of the function '`length`' and predicate '`append`' are as follows:

```
:- func length(list(T)) = int.
:- mode length(in_listskel) = out.
:- mode length(out_listskel) = in.

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

```
:- mode append(in, in, out).
:- mode append(out, out, in).
```

Note that functions may have more than one mode, just like predicates; functions can be reversible.

Alternately, the mode declarations for '`length`' could use the standard library modes '`in/1`' and '`out/1`':

```
:- func length(list(T)) = int.
:- mode length(in(listskel)) = out.
:- mode length(out(listskel)) = in.
```

As for type declarations, a predicate or function can be defined to have a given higher-order inst (see Section 8.3 [Higher-order insts and modes], page 63) by using '`with_inst`' in the mode declaration.

For example,

```
:- inst foldl_pred == (pred(in, in, out) is det).
:- inst foldl_func == (func(in, in) = out is det).

:- mode p(in) `with_inst` foldl_pred.
:- mode f(in) `with_inst` foldl_func.
```

is equivalent to

```
:- mode p(in, in, in, out) is det.
:- mode f(in, in, in) = out is det.
```

('`is det`' is explained in Chapter 6 [Determinism], page 49.)

If a predicate or function has only one mode, the '`pred`' and '`mode`' declaration can be combined:

```
:- func length(list(T)::in) = (int::out).
:- pred append(list(T)::in, list(T)::in, list(T)::out).

:- pred p `with_type` foldl_pred(T, U) `with_inst` foldl_pred.
```

It is an error for a predicate or function whose '`pred`' and '`mode`' declarations are so combined to have any other separate '`mode`' declarations.

If there is no mode declaration for a function, the compiler assumes a default mode for the function in which all the arguments have mode `in` and the result of the function has mode `out`. (However, there is no requirement that a function have such a mode; if there is any explicit mode declaration, it overrides the default.)

If a predicate or function type declaration occurs in the interface section of a module, then all mode declarations for that predicate or function must occur in the interface section of the *same* module. Likewise, if a predicate or function type declaration occurs in the implementation section of a module, then all mode declarations for that predicate or function must occur in the implementation section of the *same* module. Therefore, is an error for a predicate or function to have mode declarations in both the interface and implementation sections of a module.

A function or predicate mode declaration is an assertion by the programmer that for all possible argument terms and (if applicable) result term for the function or predicate

that are approximated (in our technical sense) by the initial instantiatedness trees of the mode declaration and all of whose free variables are distinct, if the function or predicate succeeds then the resulting binding of those argument terms and (if applicable) result term will in turn be approximated by the final instantiatedness trees of the mode declaration, with all free variables again being distinct. We refer to such assertions as *mode declaration constraints*. These assertions are checked by the compiler, which rejects programs if it cannot prove that their mode declaration constraints are satisfied.

Note that with the usual definition of '`append`', the mode

```
:- mode append(in_listskel, in_listskel, out_listskel).
```

would not be allowed, since it would create aliasing between the different arguments — on success of the predicate, the list elements would be free variables but they would not be distinct.

In Mercury it is always possible to call a procedure with an argument that is more bound than the initial inst specified for that argument in the procedure's mode declaration. In such cases, the compiler will insert additional unifications to ensure that the argument actually passed to the procedure will have the inst specified. For example, if the predicate `p/1` has mode '`p(out)`', you can still call '`p(X)`' if `X` is ground. The compiler will transform this code to '`p(Y), X = Y`' where `Y` is a fresh variable. It is almost as if the predicate `p/1` has another mode '`p(in)`'; we call such modes "implied modes".

To make this concept precise, we introduce the following definition. A term *satisfies* an instantiatedness tree if for every node in the instantiatedness tree,

- if the node is "free", then the corresponding node in the term (if any) is either a distinct free variable, or a function symbol.
- if the node is "bound", then the corresponding node in the term (if any) is a function symbol.

The *mode set* for a predicate or function is the set of mode declarations for the predicate or function. A mode set is an assertion by the programmer that the predicate should only be called with argument terms that satisfy the initial instantiatedness trees of one of the mode declarations in the set (i.e. the specified modes and the modes they imply are the only allowed modes for this predicate or function). We refer to the assertion associated with a mode set as the *mode set constraint*; these are also checked by the compiler.

A predicate or function p is *well-moded with respect to a given mode declaration* if given that the predicates and functions called by p all satisfy their mode declaration constraints, there exists an ordering of the conjuncts in each conjunction in the clauses of p such that

- p satisfies its mode declaration constraint, and
- p satisfies the mode set constraint of all of the predicates and functions it calls

We say that a predicate or function is well-moded if it is well-moded with respect to all the mode declarations in its mode set, and we say that a program is well-moded if all its predicates and functions are well-moded.

The mode analysis algorithm checks one procedure at a time. It abstractly interprets the definition of the predicate or function, keeping track of the instantiatedness of each variable, and selecting a mode for each call and unification in the definition. To ensure that the mode set constraints of called predicates and functions are satisfied, the compiler

may reorder the elements of conjunctions; it reports an error if no satisfactory order exists. Finally it checks that the resulting instantiatedness of the procedure's arguments is the same as the one given by the procedure's declaration.

The mode analysis algorithm annotates each call with the mode used.

4.3 Constrained polymorphic modes

Mode declarations for predicates and functions may also have inst parameters. However, such parameters must be constrained to be *compatible* with some other inst. In a predicate or function mode declaration, an inst of the form '*`InstParam =< Inst`*', where *InstParam* is a variable and *Inst* is an inst, states that *InstParam* is constrained to be *compatible* with *Inst*, that is, *InstParam* represents some inst that can be used anywhere where *Inst* is required. If an inst parameter occurs more than once in a declaration, it must have the same constraint on each occurrence.

For example, in the mode declaration

```
:- mode append(in(list_skel(I =< ground)), in(list_skel(I =< ground)),
    out(list_skel(I =< ground))) is det.
```

`I` is an inst parameter which is constrained to be ground. If '`append`' is called with the first two arguments having an inst of, say, '`list_skel(bound(f))`' then after '`append`' returns, all three arguments will have inst '`list_skel(bound(f))`'. If the mode of append had been simply

```
:- mode append(in(list_skel(ground)), in(list_skel(ground)),
    out(list_skel(ground))) is det.
```

then we would only have been able to infer an inst of '`list_skel(ground)`' for the third argument, not the more specific inst.

Note that attempting to call '`append`' when the first two arguments do not have ground insts (e.g. '`list_skel(bound(g(free)))`') is a mode error because it violates the constraint on the inst parameter.

To avoid having to repeat a constraint everywhere that an inst parameter occurs, it is possible to list the constraints after the rest of the mode declaration, following a '`<=`'. E.g. the above example could have been written as

```
:- (mode append(in(list_skel(I)), in(list_skel(I)),
    out(list_skel(I))) is det) <= I =< ground.
```

Note that in the current Mercury implementation this syntax requires parentheses around the '`mode(...) is Det`' part of the declaration.

Also, if the constraint on an inst parameter is '`ground`' then it is not necessary to give the constraint in the declaration. The example can be further shortened to

```
:- mode append(in(list_skel(I)), in(list_skel(I)), out(list_skel(I)))
    is det.
```

Constrained polymorphic modes are particularly useful when passing objects with higher-order types to polymorphic predicates since they allow the higher-order mode information to be retained (see Chapter 8 [Higher-order], page 60).

4.4 Different clauses for different modes

Because the compiler automatically reorders conjunctions to satisfy the modes, it is often possible for a single clause to satisfy different modes. However, occasionally reordering of conjunctions is not sufficient; you may want to write different code for different modes.

For example, the usual code for list append

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

works fine in most modes, but is not very satisfactory for the '`append(out, in, in)`' mode of append, because although every call in this mode only has at most one solution, the compiler's determinism inference will not be able to infer that. This means that using the usual code for append in this mode will be inefficient, and the overly conservative determinism inference may cause spurious determinism errors later.

For this mode, it is better to use a completely different algorithm:

```
append(Prefix, Suffix, List) :-
    list.length(List, ListLength),
    list.length(Suffix, SuffixLength),
    PrefixLength = ListLength - SuffixLength,
    list.split_list(PrefixLength, List, Prefix, Suffix).
```

However, that code doesn't work in the other modes of '`append`'.

To handle such cases, you can use mode annotations on clauses, which indicate that particular clauses should only be used for particular modes. To specify that a clause only applies to a given mode, each argument *Arg* of the clause head should be annotated with the corresponding argument mode *Mode*, using the '`::`' mode qualification operator, i.e. '*`Arg`* `::` *`Mode`*'.

For example, if '`append`' was declared as

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

then you could implement it as

```
append(L1::in,  L2::in,  L3::out) :- usual_append(L1, L2, L3).
append(L1::out, L2::out, L3::in)  :- usual_append(L1, L2, L3).
append(L1::in,  L2::out, L3::in)  :- usual_append(L1, L2, L3).
append(L1::out, L2::in,  L3::in)  :- other_append(L1, L2, L3).

usual_append([], Ys, Ys).
usual_append([X|Xs], Ys, [X|Zs]) :- usual_append(Xs, Ys, Zs).

other_append(Prefix, Suffix, List) :-
    list.length(List, ListLength),
    list.length(Suffix, SuffixLength),
    PrefixLength = ListLength - SuffixLength,
    list.split_list(PrefixLength, List, Prefix, Suffix).
```

This language feature can be used to write "impure" code that doesn't have any consistent declarative semantics. For example, you can easily use it to write something similar to Prolog's (in)famous '`var/1`' predicate:

```
:- mode var(in).
:- mode var(free>>free).
var(_::in) :- fail.
var(_::free>>free) :- true.
```

As you can see, in this case the two clauses are *not* equivalent.

Because of this possibility, predicates or functions which are defined using different code for different modes are by default assumed to be impure; the programmer must either (1) carefully ensure that the logical meaning of the clauses is the same for all modes, which can be declared to the compiler through a '`pragma promise_equivalent_clauses`' declaration, or a '`pragma promise_pure`' declaration, or (2) declare the predicate or function as impure. See Chapter 15 [Impurity], page 129.

In the example with '`append`' above, the two ways of implementing append do have the same declarative semantics, so we can safely use the first approach:

```
:- pragma promise_equivalent_clauses(append/3).
```

The pragma

```
:- pragma promise_pure(append/3).
```

would also promise that the clauses are equivalent, but on top of that would also promise that the code of each clause is pure. Sometimes, if some clauses contain impure code, that is a promise that the programmer wants to make, but this extra promise is unnecessary in this case.

In the example with '`var/1`' above, the two clauses have different semantics, so the predicate must be declared as impure:

```
:- impure pred var(T).
```

5 Unique modes

Mode declarations can also specify so-called "unique modes". Mercury's unique modes are similar to "linear types" in some functional programming languages such as Clean. They allow you to specify when there is only one reference to a particular value, and when there will be no more references to that value. If the compiler knows there will be no more references to a value, it can perform "compile-time garbage collection" by automatically inserting code to deallocate the storage associated with that value. Even more importantly, the compiler can also simply reuse the storage immediately, for example by destructively updating one element of an array rather than making a new copy of the entire array in order to change one element. Unique modes are also the mechanism Mercury uses to provide declarative I/O.

We have not yet implemented unique modes fully, and the details are still in a state of flux. So the following should be considered tentative.

5.1 Destructive update

In addition to the insts mentioned above (`free`, `ground`, and `bound(...)`), Mercury also provides "unique" insts `unique` and `unique(...)` which are like `ground` and `bound(...)` respectively, except that they carry the additional constraint that there can only be one reference to the corresponding value. There is also an inst `dead` which means that there are no references to the corresponding value, so the compiler is free to generate code that reuses that value. There are three standard modes for manipulating unique values:

```
% unique output
:- mode uo == free >> unique.

% unique input
:- mode ui == unique >> unique.

% destructive input
:- mode di == unique >> dead.
```

Mode `uo` is used to create a unique value. Mode `ui` is used to inspect a unique value without losing its uniqueness. Mode `di` is used to deallocate or reuse the memory occupied by a value that will not be used.

Note that a value is not considered `unique` if it might be needed on backtracking. This means that unique modes are generally only useful for code whose determinism is `det` or `cc_multi` (see Chapter 6 [Determinism], page 49).

Unlike `bound` instantiatedness trees, there is no alternative syntax for `unique` instantiatedness trees.

5.2 Backtrackable destructive update

> "Well it just so happens that your friend here is only *mostly* dead.
> There's a big difference between mostly dead and all dead...
> Now, mostly dead is slightly alive.
> Now, all dead — well, with all dead, there's usually only one thing that you can do."

"What's that?"

"Go through his clothes and look for loose change!"

— from the movie "The Princess Bride".

To allow for backtrackable destructive updates — that is, updates whose effect is undone on backtracking, perhaps by recording the overwritten values on a "trail" so that they can be restored after backtracking — Mercury also provides "mostly unique" modes. The insts `mostly_unique` and `mostly_dead` are equivalent to `unique` and `dead`, except that only references which will be encountered during forward execution are counted — it is OK for `mostly_unique` or `mostly_dead` values to be needed again on backtracking.

Mercury defines some standard modes for manipulating "mostly unique" values, just as it does for unique values:

```
% mostly unique output
:- mode muo == free >> mostly_unique.

% mostly unique input
:- mode mui == mostly_unique >> mostly_unique.

% mostly destructive input
:- mode mdi == mostly_unique >> mostly_dead.
```

5.3 Limitations of the current implementation

The implementation of the mode analysis algorithm is not quite complete; as a result, it is not possible to use nested unique modes, i.e. modes in which anything but the top level of a variable is unique. If you do, you will get unique mode errors when you try to get a unique field of a unique data structure. It is also not possible to use unique-input modes; only destructive-input and unique-output modes work.

The Mercury compiler does not (yet) reuse `dead` values. The only destructive update in the current implementation occurs in library modules, e.g. for I/O and arrays. We do however plan to implement structure reuse and compile-time garbage collection in the very near future.

6 Determinism

6.1 Determinism categories

For each mode of a predicate or function, we categorise that mode according to how many times it can succeed, and whether or not it can fail before producing its first solution.

If all possible calls to a particular mode of a predicate or function which return to the caller (calls which terminate, do not throw an exception and do not cause a fatal runtime error)

- have exactly one solution, then that mode is *deterministic* (`det`);
- either have no solutions or have one solution, then that mode is *semideterministic* (`semidet`);
- have at least one solution but may have more, then that mode is *multisolution* (`multi`);
- have zero or more solutions, then that mode is *nondeterministic* (`nondet`);
- fail without producing a solution, then that mode has a determinism of `failure`.

If no possible calls to a particular mode of a predicate or function can return to the caller, then that mode has a determinism of `erroneous`.

The determinism annotation `erroneous` is used on the library predicates '`require.error/1`' and '`exception.throw/1`', but apart from that determinism annotations `erroneous` and `failure` are generally not needed.

To summarize:

	Maximum number of solutions		
Can fail?	0	1	> 1
no	erroneous	det	multi
yes	failure	semidet	nondet

(Note: the "Can fail?" column here indicates only whether the procedure can fail before producing at least one solution; attempts to find a *second* solution to a particular call, e.g. for a procedure with determinism `multi`, are always allowed to fail.)

The determinism of each mode of a predicate or function is indicated by an annotation on the mode declaration. For example:

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

:- func length(list(T)) = int.
:- mode length(in) = out is det.
:- mode length(in(list_skel)) = out is det.
:- mode length(in) = in is semidet.
```

An annotation of `det` or `multi` is an assertion that for every value each of the inputs, there exists at least one value of the outputs for which the predicate is true, or (in the case of functions) for which the function term is equal to the result term. Conversely, an annotation of `det` or `semidet` is an assertion that for every value each of the inputs, there

exists at most one value of the outputs for which the predicate is true, or (in the case of functions) for which the function term is equal to the result term. These assertions are called the *mode-determinism assertions*; they can play a role in the semantics, because in certain circumstances they may allow an implementation to perform optimizations that would not otherwise be allowed, such as optimizing away a goal with no outputs even though it might infinitely loop.

If the mode of the predicate is given in the `:- pred` declaration rather than in a separate `:- mode` declaration, then the determinism annotation goes on the `:- pred` declaration (and similarly for functions). In particular, this is necessary if a predicate does not have any argument variables. If the determinism declaration is given on a `:- func` declaration without the mode, the function is assumed to have the default mode (see Chapter 4 [Modes], page 39 for more information on default modes of functions).

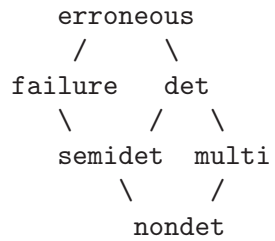
For example:

```
:- pred loop(int::in) is erroneous.
loop(X) :- loop(X).

:- pred p is det.
p.

:- pred q is failure.
q :- fail.
```

If there is no mode declaration for a function, then the default mode for that function is considered to have been declared as `det`. If you want to write a partial function, i.e. one whose determinism is `semidet`, then you must explicitly declare the mode and determinism.

In Mercury, a function is supposed to be a true mathematical function of its arguments; that is, the value of the function's result should be determined only by the values of its arguments. Hence, for any mode of a function that specifies that all the arguments are fully input (i.e. for which the initial inst of all the arguments is a ground inst), the determinism of that mode can only be `det`, `semidet`, `erroneous`, or `failure`.

The determinism categories form this lattice:



The higher up this lattice a determinism category is, the more the compiler knows about the number of solutions of procedures of that determinism.

6.2 Determinism checking and inference

The determinism of goals is inferred from the determinism of their component parts, according to the rules below. The inferred determinism of a procedure is just the inferred determinism of the procedure's body.

For procedures that are local to a module, the determinism annotations may be omitted; in that case, their determinism will be inferred. (To be precise, the determinism of procedures without a determinism annotation is defined as the least fixpoint of the transformation which, given an initial assignment of the determinism `det` to all such procedures, applies those rules to infer a new determinism assignment for those procedures.)

It is an error to omit the determinism annotation for procedures that are exported from their containing module.

If a determinism annotation is supplied for a procedure, the declared determinism is compared against the inferred determinism. If the declared determinism is greater than or not comparable to the inferred determinism (in the partial ordering above), it is an error. If the declared determinism is less than the inferred determinism, it is not an error, but the implementation may issue a warning.

The determinism category of each goal is inferred according to the following rules. These rules work with the two components of determinism category: whether the goal can fail without producing a solution, and the maximum number of solutions of the goal (0, 1, or more). If the inference process below reports that a goal can succeed more than once, but the goal generates no outputs that are visible from outside the goal, and the goal is not impure (see Chapter 15 [Impurity], page 129), then the final determinism of the goal will be based on the goal succeeding at most once, since the compiler will implicitly prune away any duplicate solutions.

Calls
: The determinism category of a call is the determinism declared or inferred for the called mode of the called procedure.

Unifications
: The determinism of a unification is either `det`, `semidet`, or `failure`, depending on its mode.

  A unification that assigns the value of one variable to another is deterministic. A unification that constructs a structure and assigns it to a variable is also deterministic. A unification that tests whether a variable has a given top function symbol is semideterministic, unless the compiler knows the top function symbol of that variable, in which case its determinism is either det or failure depending on whether the two function symbols are the same or not. A unification that tests two variables for equality is semideterministic, unless the compiler knows that the two variables are aliases for one another, in which case the unification is deterministic, or unless the compiler knows that the two variables have different function symbols in the same position, in which case the unification has a determinism of failure.

  The compiler knows the top function symbol of a variable if the previous part of the procedure definition contains a unification of the variable with a function symbol, or if the variable's type has only one function symbol.

Conjunctions
: The determinism of the empty conjunction (the goal '`true`') is `det`. The conjunction '`(A, B)`' can fail if either *A* can fail, or if *A* can succeed at least once, and *B* can fail. The conjunction can succeed at most zero times if either *A* or *B* can succeed at most zero times. The conjunction can succeed more than once

if either A or B can succeed more than once and both A and B can succeed at least once. (If e.g. A can succeed at most zero times, then even if B can succeed many times the maximum number of solutions of the conjunction is still zero.) Otherwise, i.e. if both A and B succeed at most once, the conjunction can succeed at most once.

Switches

A disjunction is a *switch* if each disjunct has near its start a unification that tests the same bound variable against a different function symbol. For example, consider the common pattern

```
(
    L = [], empty(Out)
;
    L = [H|T], nonempty(H, T, Out)
)
```

If `L` is input to the disjunction, then the disjunction is a switch on `L`.

If two variables are unified with each other, then whatever function symbol one variable is unified with, the other variable is considered to be unified with the same function symbol. In the following example, since `K` is unified with `L`, the second disjunct unifies `L` as well as `K` with cons, and thus the disjunction is recognized as a switch.

```
(
    L = [], empty(Out)
;
    K = L, K = [H|T], nonempty(H, T, Out)
)
```

A switch can fail if the various arms of the switch do not cover all the function symbols in the type of the switched-on variable, or if the code in some arms of the switch can fail, bearing in mind that in each arm of the switch, the unification that tests the switched-on variable against the function symbol of that arm is considered to be deterministic. A switch can succeed several times if some arms of the switch can succeed several times, possibly because there are multiple disjuncts that test the switched-on variable against the same function symbol. A switch can succeed at most zero times only if all arms of the switch can succeed at most zero times.

Only unifications may occur before the test of the switched-on variable in each disjunct. Tests of the switched-on variable may occur within existential quantification goals.

The following example is a switch.

```
(
    Out = 1, L = []
;
    some [H, T] (
        L = [H|T],
        nonempty(H, T, Out)
    )
)
```

The following example is not a switch because the call in the first disjunct occurs before the test of the switched-on variable.

```
(
    empty(Out), L = []
;
    L = [H|T], nonempty(H, T, Out)
)
```

The unification of the switched-on variable with a function symbol may occur inside a nested disjunction in a given disjunct, provided that unification is preceded only by other unifications, both inside the nested disjunction and before the nested disjunction. The following example is a switch on `X`, provided `X` is bound beforehand.

```
(
    X = f
    p(Out)
;
    Y = X,
    (
        Y = g,
        Intermediate = 42
    ;
        Z = Y,
        Z = h(Arg),
        q(Arg, Intermediate)
    ),
    r(Intermediate, Out)
)
```

Disjunctions

The determinism of the empty disjunction (the goal '`fail`') is `failure`. A disjunction '`(A ; B)`' that is not a switch can fail if both A and B can fail. It can succeed at most zero times if both A and B can succeed at most zero times. It can succeed at most once if one of A and B can succeed at most once and the other can succeed at most zero times. Otherwise, i.e. if either A or B can succeed more than once, or if both A and B can succeed at least once, it can succeed more than once.

If-then-else

If the condition of an if-then-else cannot fail, the if-then-else is equivalent to the conjunction of the condition and the "then" part, and its determinism is computed accordingly. Otherwise, an if-then-else can fail if either the "then" part or the "else" part can fail. It can succeed at most zero times if the "else" part can succeed at most zero times and if at least one of the condition and the "then" part can succeed at most zero times. It can succeed more than once if any one of the condition, the "then" part and the "else" part can succeed more than once.

Negations

If the determinism of the negated goal is `erroneous`, then the determinism of the negation is `erroneous`. If the determinism of the negated goal is `failure`, the determinism of the negation is `det`. If the determinism of the negated goal is `det` or `multi`, the determinism of the negation is `failure`. Otherwise, the determinism of the negation is `semidet`.

6.3 Replacing compile-time checking with run-time checking

Note that "perfect" determinism inference is an undecidable problem, because it requires solving the halting problem. (For instance, in the following example

```
:- pred p(T, T).
:- mode p(in, out) is det.

p(A, B) :-
    (
        something_complicated(A, B)
    ;
        B = A
    ).
```

'`p/2`' can have more than one solution only if '`something_complicated/2`' can succeed.) Sometimes, the rules specified by the Mercury language for determinism inference will infer a determinism that is not as precise as you would like. However, it is generally easy to overcome such problems. The way to do this is to replace the compiler's static checking with some manual run-time checking. For example, if you know that a particular goal should never fail, but the compiler infers that goal to be `semidet`, you can check at runtime that the goal does succeed, and if it fails, call the library predicate '`error/1`'.

```
:- pred q(T, T).
:- mode q(in, out) is det.

q(A, B) :-
    ( goal_that_should_never_fail(A, B0) ->
        B = B0
    ;
        error("goal_that_should_never_fail failed!")
    ).
```

The predicate `error/1` has determinism `erroneous`, which means the compiler knows that it will never succeed or fail, so the inferred determinism for the body of `q/2` is `det`. (Checking assumptions like this is good coding style anyway. The small amount of up-front work that Mercury requires is paid back in reduced debugging time.) Mercury's mode analysis knows that computations with determinism `erroneous` can never succeed, which is why it does not require the "else" part to generate a value for `B`. The introduction of the new variable `B0` is necessary because the condition of an if-then-else is a negated context, and can export the values it generates only to the "then" part of the if-then-else, not directly to the surrounding computation. (If the surrounding computations had direct access to values generated in conditions, they might access them even if the condition failed.)

6.4 Interfacing nondeterministic code with the real world

Normally, attempting to call a `nondet` or `multi` mode of a predicate from a predicate declared as `semidet` or `det` will cause a determinism error. So how can we call nondeterministic code from deterministic code? There are several alternative possibilities.

If you just want to see if a nondeterministic goal is satisfiable or not, without needing to know what variable bindings it produces, then there is no problem - determinism analysis considers `nondet` and `multi` goals with no non-local output variables to be `semidet` and `det` respectively.

If you want to use the values of output variables, then you need to ask yourself which one of possibly many solutions to a goal do you want? If you want all of them, you need to use the predicate '`solutions/2`' in the standard library module '`solutions`', which collects all of the solutions to a goal into a list — see Chapter 8 [Higher-order], page 60.

If you just want one solution and don't care which, the calling predicate should be declared `nondet` or `multi`. The nondeterminism should then be propagated up the call tree to the point at which it can be pruned. In Mercury, pruning can be achieved in several ways.

The first way is the one mentioned above: if a goal has no non-local output variables then the implementation will only attempt to satisfy the goal once. Any potential duplicate solutions will be implicitly pruned away.

The second way is to rely on the fact that the implementation will only seek a single solution to '`main/2`', so alternative solutions to '`main/2`' (and hence also to `nondet` or `multi` predicates called directly or indirectly from '`main/2`') are implicitly pruned away. This is one way to achieve "don't care" style nondeterminism in Mercury.

The other situation in which you may want pruning and committed choice style nondeterminism is when you know that all the solutions returned will be equivalent. For example, you might want to find the maximum element in a set by iterating over the elements in the set. Iterating over the elements in a set in an unspecified order is a nondeterministic operation, but no matter which order you remove them, the maximum value in the set should be the same.

If you know that there will only ever be at most one distinct solution under the equality theory of the output variables, then you can use a '`promise_equivalent_solutions`' determinism cast.

Note that specifying a user-defined equivalence relation as the equality predicate for user-defined types (see Chapter 7 [User-defined equality and comparison], page 57) means that '`promise_equivalent_solutions`' can be used to express more general forms of equivalence. For example, if you define a set type which represents sets as unsorted lists, you would want to define a user-defined equivalence relation for that type, which could sort the lists before comparing them. The '`promise_equivalent_solutions`' determinism cast could then be used for sets even though the lists used to represent the sets might not be in the same order in every solution.

6.5 Committed choice nondeterminism

In addition to the determinism annotations described earlier, there are "committed choice" versions of `multi` and `nondet`, called `cc_multi` and `cc_nondet`. These can be used instead

of `multi` or `nondet` if all calls to that mode of the predicate (or function) occur in a context in which only one solution is needed.

Such single-solution contexts are determined as follows.

- The body of any procedure declared `cc_multi` or `cc_nondet` is in a single-solution context. For example, the program entry point '`main/2`' may be declared `cc_multi`, and in that case the clauses for `main` are in a single-solution context.
- Any goal with no output variables is in a single-solution context.
- If a conjunction is in a single-solution context, then the right-most conjunct is in a single-solution context, and if the right-most conjunct cannot fail, then the rest of the conjunction is also in a single-solution context. ("Right-most" here refers to the order *after* mode reordering.)
- If an if-then-else is in a single-solution context, then the "then" part and the "else" part are in single-solution contexts, and if the "then" part cannot fail, then the condition of the if-then-else is also in a single-solution context.
- For other compound goals, i.e. disjunctions, negations, and (explicitly) existentially quantified goals, if the compound goal is in a single-solution context, then the immediate sub-goals of that compound goal are also in single-solution contexts.

The compiler will check that all calls to a committed-choice mode of a predicate (or function) do indeed occur in a single-solution context.

You can declare two different modes of a predicate (or function) which differ only in "cc-ness" (i.e. one being `multi` and the other `cc_multi`, or one being `nondet` and the other `cc_nondet`). In that case, the compiler will select the appropriate one for each call depending on whether the call comes from a single-solution context or not. Calls from single-solution contexts will call the committed choice version, while calls which are not from single-solution contexts will call the backtracking version.

There are several reasons to use committed choice determinism annotations. One reason is for efficiency: committed choice annotations allow the compiler to generate much more efficient code. Another reason is for doing I/O, which is allowed only in `det` or `cc_multi` predicates, not in `multi` predicates. Another is for dealing with types that use non-canonical representations (see Chapter 7 [User-defined equality and comparison], page 57). And there are a variety of other applications.

7 User-defined equality and comparison

When defining abstract data types, often it is convenient to use a non-canonical representation — that is, one for which a single abstract value may have more than one different possible concrete representations. For example, you may wish to implement an abstract type '`set`' by representing a set as an (unsorted) list.

```
:- module set_as_unsorted_list.
:- interface.
:- type set(T).

:- implementation.
:- import_module list.
:- type set(T)
    --->    set(list(T)).
```

In this example, the concrete representations '`set([1,2])`' and '`set([2,1])`' would both represent the same abstract value, namely the set containing the elements 1 and 2.

For types such as this, which do not have a canonical representation, the standard definition of equality is not the desired one; we want equality on sets to mean equality of the abstract values, not equality of their representations. To support such types, Mercury allows programmers to specify a user-defined equality predicate for user-defined types:

```
:- type set(T)
    --->    set(list(T))
            where equality is set_equals.
```

Here '`set_equals`' is the name of a user-defined predicate that is used for equality on the type '`set(T)`'. It could for example be defined in terms of a '`subset`' predicate.

```
:- pred set_equals(set(T)::in, set(T)::in) is semidet.
set_equals(S1, S2) :-
    subset(S1, S2),
    subset(S2, S1).
```

A comparison predicate can also be supplied.

```
:- type set(T)
    --->    set(list(T))
            where equality is set_equals, comparison is set_compare.

:- pred set_compare(builtin.comparison_result::uo,
    set(T)::in, set(T)::in) is det.

set_compare(Result, Set1, Set2) :-
    promise_equivalent_solutions [Result] (
        set_compare_2(Set1, Set2, Result)
    ).

:- pred set_compare_2(set(T)::in, set(T)::in,
    builtin.comparison_result::uo) is cc_multi.
```

```
set_compare_2(set(List1), set(List2), Result) :-
    builtin.compare(Result, list.sort(List1), list.sort(List2)).
```

If a comparison predicate is supplied and the unification predicate is omitted, a unification predicate is generated by the compiler in terms of the comparison predicate. For the '`set`' example, the generated predicate would be:

```
set_equals(S1, S2) :-
    set_compare((=), S1, S2).
```

If a unification predicate is supplied without a comparison predicate, the compiler will generate a comparison predicate which throws an exception of type '`exception.software_error`' when called.

A type declaration for a type '`foo(T1, ..., TN)`' may contain a '`where equality is` *`equalitypred`*' specification only if it declares a discriminated union type or a foreign type (see Section 14.4 [Using foreign types from Mercury], page 109) and the following conditions are satisfied:

- *equalitypred* must be the name of a predicate with signature

  ```
  :- pred equalitypred(foo(T1, ..., TN)::in,
                       foo(T1, ..., TN)::in) is semidet.
  ```

  It is legal for the type, mode and determinism to be more permissive: the type or the mode's initial insts may be more general (e.g. the type of the equality predicate could be just the polymorphic type '`pred(T, T)`') and the mode's final insts or the determinism may be more specific (e.g. the determinism of the equality predicate could be any of `det`, `failure` or `erroneous`).
- If the type is a discriminated union then its definition cannot be a single zero-arity constructor.
- The equality predicate must be "pure" (see Chapter 15 [Impurity], page 129).
- The equality predicate must be defined in the same module as the type.
- If the type is exported the equality predicate must also be exported.
- *equalitypred* should be an equivalence relation; that is, it must be symmetric, reflexive, and transitive. However, the compiler is not required to check this[1].

Types with user-defined equality can only be used in limited ways. Because there are multiple representations for the same abstract value, any attempt to examine the representation of such a value is a conceptually non-deterministic operation. In Mercury this is modelled using committed choice nondeterminism.

The semantics of specifying '`where equality is` *`equalitypred`*' on the type declaration for a type T are as follows:

- If the program contains any deconstruction unification or switch on a variable of type T that could fail, other than unifications with mode '`(in, in)`', then it is a compile-time error.

[1] If *equalitypred* is not an equivalence relation, then the program is inconsistent: its declarative semantics contains a contradiction, because the additional axioms for the user-defined equality contradict the standard equality axioms. That implies that the implementation may compute any answer at all (see Chapter 13 [Semantics], page 96), i.e. the behaviour of the program is undefined.

- If the program contains any deconstruction unification or switch on a variable of type T that cannot fail, then that operation has determinism `cc_multi`.
- Any attempts to examine the representation of a variable of type T using facilities of the standard library (e.g. '`argument`'/3 and '`functor/3`' in '`deconstruct`') that do not have determinism `cc_multi` or `cc_nondet` will result in a run-time error.
- In addition to the usual equality axioms, the declarative semantics of the program will contain the axiom '`X = Y <=> equalitypred(X, Y)`' for all X and Y of type '`T`'.
- Any '`(in, in)`' unifications for type T are computed using the specified predicate *equalitypred*.

A type declaration for a type '`foo(T1, ..., TN)`' may contain a '`where comparison is comparepred`' specification only if it declares a discriminated union type or a foreign type (see Section 14.4 [Using foreign types from Mercury], page 109) and the following conditions are satisfied:

- *comparepred* must be the name of a predicate with signature

  ```
  :- pred comparepred(builtin.comparison_result::uo,
                      foo(T1, ..., TN)::in, foo(T1, ..., TN)::in) is det.
  ```

  As with equality predicates, it is legal for the type, mode and determinism to be more permissive.
- If the type is a discriminated union then its definition cannot be a single zero-arity constructor.
- The comparison predicate must also be "pure" (see Chapter 15 [Impurity], page 129).
- The comparison predicate must be defined in the same module as the type.
- If the type is exported the comparison predicate must also be exported.
- The relation

  ```
  compare_eq(X, Y) :- comparepred((=), X, Y).
  ```

  must be an equivalence relation; that is, it must be symmetric, reflexive, and transitive. The compiler is not required to check this.
- The relations

  ```
  compare_leq(X, Y) :- comparepred(R, X, Y), (R = (=) ; R = (<)).
  compare_geq(X, Y) :- comparepred(R, X, Y), (R = (=) ; R = (>)).
  ```

  must be total order relations: that is they must be antisymmetric, reflexive and transitive. The compiler is not required to check this.

For each type for which the declaration has a '`where comparison is comparepred`' specification, any calls to the standard library predicate '`builtin.compare/3`' with arguments of that type are evaluated as if they were calls to *comparepred*.

A type declaration may contain a '`where equality is equalitypred, comparison is comparepred`' specification only if in addition to the conditions above, '`all [X, Y] (comparepred((=), X, Y) <=> equalitypred(X, Y))`'. The compiler is not required to check this.

8 Higher-order programming

Mercury supports higher-order functions and predicates with currying, closures, and lambda expressions. (To be pedantic, it would be more accurate to say that Mercury supports higher-order procedures: in Mercury, when you construct a higher-order term, you only get one mode of a predicate or function; if you want multiple modes, you must pass multiple higher-order procedures.)

8.1 Creating higher-order terms

To create a higher-order predicate or function term, you can use a lambda expression, or, if the predicate or function has only one mode and it is not a zero-arity function, you can just use its name. For example, if you have declared a predicate

```
:- pred sum(list(int), int).
:- mode sum(in, out) is det.
```

the following unifications have the same effect:

```
X = (pred(List::in, Length::out) is det :- sum(List, Length))
Y = sum
```

In the above example, the type of `X`, and `Y` is '`pred(list(int), int)`', which means a predicate of two arguments of types `list(int)` and `int` respectively.

Similarly, given

```
:- func scalar_product(int, list(int)) = list(int).
:- mode scalar_product(in, in) = out is det.
```

the following three unifications have the same effect:

```
X = (func(Num, List) = NewList :- NewList = scalar_product(Num, List))
Y = (func(Num::in, List::in) = (NewList::out) is det
        :- NewList = scalar_product(Num, List))
Z = scalar_product
```

In the above example, the type of `X`, `Y`, and `Z` is '`func(int, list(int)) = list(int)`', which means a function of two arguments, whose types are `int` and `list(int)`, with a return type of `int`. As with '`:- func`' declarations, if the modes and determinism of the function are omitted in a higher-order function term, then the modes default to `in` for the arguments, `out` for the function result, and the determinism defaults to `det`.

The Melbourne Mercury implementation currently requires that you use an explicit lambda expression to specify which mode you want, if the predicate or function has more than one mode (but see below for an exception to this rule).

You can also create higher-order function terms of non-zero arity and higher-order predicate terms by "currying", i.e. specifying the first few arguments to a predicate or function, but leaving the remaining arguments unspecified. For example, the unification

```
Sum123 = sum([1,2,3])
```

binds `Sum123` to a higher-order predicate term of type '`pred(int)`'. Similarly, the unification

```
Double = scalar_product(2)
```

binds `Double` to a higher-order function term of type '`func(list(int)) = list(int)`'.

As a special case, currying of a multi-moded predicate or function is allowed provided that the mode of the predicate or function can be determined from the insts of the higher-order curried arguments. For example, '`P = list.foldl(io.write)`' is allowed because the inst of '`io.write`' matches exactly one mode of '`list.foldl`'.

For higher-order predicate expressions that thread an accumulator pair, we have syntax that allows you to use DCG notation in the goal of the expression. For example,

```
Pred = (pred(Strings::in, Num::out, di, uo) is det -->
    io.write_string("The strings are: "),
    { list.length(Strings, Num) },
    io.write_strings(Strings),
    io.nl
)
```

is equivalent to

```
Pred = (pred(Strings::in, Num::out, IO0::di, IO::uo) is det :-
    io.write_string("The strings are: ", IO0, IO1),
    list.length(Strings, Num),
    io.write_strings(Strings, IO1, IO2),
    io.nl(IO2, IO)
)
```

Higher-order function terms of zero arity can only be created using an explicit lambda expression; you have to use e.g. '`(func) = foo`' rather than plain '`foo`', because the latter denotes the result of evaluating the function, rather than the function itself.

Note that when constructing a higher-order term, you cannot just use the name of a builtin language construct such as '`=`', '`\=`', '`call`', or '`apply`', and nor can such constructs be curried. Instead, you must either use an explicit lambda expression, or you must write a forwarding predicate or function. For example, instead of

```
list.filter(\=(2), [1, 2, 3], List)
```

you must write either

```
list.filter((pred(X::in) is semidet :- X \= 2), [1, 2, 3], List)
```

or

```
list.filter(not_equal(2), [1, 2, 3], List)
```

where you have defined '`not_equal`' using

```
:- pred not_equal(T::in, T::in) is semidet.
not_equal(X, Y) :- X \= Y.
```

Another case when this arises is when want to curry a higher-order term. Suppose, for example, that you have a higher-order predicate term '`OldPred`' of type '`pred(int, char, float)`', and you want to construct a new higher-order predicate term '`NewPred`' of type '`pred(char, float)`' from '`OldPred`' by supplying a value for just the first argument. The solution is the same: use an explicit lambda expression or a forwarding predicate. In either case, the body of the lambda expression or the forwarding predicate must contain a higher-order call with all the arguments supplied.

8.2 Calling higher-order terms

Once you have created a higher-order predicate term (sometimes known as a closure), the next thing you want to do is to call it. For predicates, you use the builtin goal call/N:

```
call(Closure)
call(Closure1, Arg1)
call(Closure2, Arg1, Arg2)
...
```
A higher-order predicate call. '`call(Closure)`' just calls the specified higher-order predicate term. The other forms append the specified arguments onto the argument list of the closure before calling it.

For example, the goal

```
call(Sum123, Result)
```

would bind `Result` to the sum of '`[1, 2, 3]`', i.e. to 6.

For functions, you use the builtin expression apply/N:

```
apply(Closure)
apply(Closure1, Arg1)
apply(Closure2, Arg1, Arg2)
...
```
A higher-order function application. Such a term denotes the result of invoking the specified higher-order function term with the specified arguments.

For example, given the definition of '`Double`' above, the goal

```
List = apply(Double, [1, 2, 3])
```

would be equivalent to

```
List = scalar_product(2, [1, 2, 3])
```

and so for a suitable implementation of the function '`scalar_product/2`' this would bind `List` to '`[2, 4, 6]`'.

One extremely useful higher-order predicate in the Mercury standard library is '`solutions/2`', which has the following declaration:

```
:- pred solutions(pred(T), list(T)).
:- mode solutions(pred(out) is nondet, out) is det.
```

The term which you pass to '`solutions/2`' is a higher-order predicate term. You can pass the name of a one-argument predicate, or you can pass a several-argument predicate with all but one of the arguments supplied (a closure). The declarative semantics of '`solutions/2`' can be defined as follows:

```
solutions(Pred, List) is true iff
    all [X] (call(Pred, X) <=> list.member(X, List))
    and List is sorted.
```

where '`call(Pred, X)`' invokes the higher-order predicate term `Pred` with argument `X`, and where '`list.member/2`' is the standard library predicate for list membership. In other words, '`solutions(Pred, List)`' finds all the values of `X` for which '`call(Pred, X)`' is true, collects these solutions in a list, sorts the list, and returns that list as its result. Here's an example: the standard library defines a predicate '`list.perm(List0, List)`'

```
:- pred list.perm(list(T), list(T)).
:- mode list.perm(in, out) is nondet.
```

which succeeds iff List is a permutation of List0. Hence the following call to solutions

```
solutions(list.perm([3,1,2]), L)
```

should return all the possible permutations of the list '`[3,1,2]`' in sorted order:

```
L = [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]].
```

See also '`unsorted_solutions/2`' and '`solutions_set/2`', which are defined in the standard library module '`solutions`' and documented in the Mercury Library Reference Manual.

8.3 Higher-order insts and modes

In Mercury, the mode and determinism of a higher-order predicate or function term are generally part of that term's *inst*, not its *type*. This allows a single higher-order predicate to work on argument predicates of different modes and determinism, which is particularly useful for library predicates such as '`list.map`' and '`list.foldl`'.

8.3.1 Builtin higher-order insts and modes

The language contains builtin '`inst`' values

```
(pred) is Determinism
pred(Mode) is Determinism
pred(Mode1, Mode2) is Determinism
...
(func) = Mode is Determinism
func(Mode1) = Mode is Determinism
func(Mode1, Mode2) = Mode is Determinism
...
```

These insts represent the instantiation state of variables bound to higher-order predicate and function terms with the appropriate mode and determinism. For example, '`pred(out) is det`' represents the instantiation state of being bound to a higher-order predicate term which is `det` and accepts one output argument; the term '`sum([1,2,3])`' from the example above is one such higher-order predicate term which matches this instantiation state.

As a convenience, the language also contains builtin '`mode`' values of the same name (and they are what we have been using in the examples up to now). These modes map from the corresponding '`inst`' to itself. It is as if they were defined by

```
:- mode (pred is Determinism) == in(pred is Determinism).
:- mode (pred(Inst) is Determinism) ==
    in(pred(Inst) is Determinism).
...
```

using the parametric inst '`in/1`' mentioned in Chapter 4 [Modes], page 39 which maps an inst to itself.

If you want to define a predicate which returns a higher-order predicate term, you would use a mode such as '`free >> pred(...) is ...`', or '`out(pred(...) is ... )`'. For example:

```
:- pred foo(pred(int)).
:- mode foo(free >> pred(out) is det) is det.

foo(sum([1,2,3])).
```

Note that in Mercury it is an error to attempt to unify two higher-order terms. This is because equivalence of higher-order terms is undecidable in the general case.

For example, given the definition of '`foo`' above, the goal

```
foo((pred(X::out) is det :- X = 6))
```

is illegal. If you really want to compare higher-order predicates for equivalence, you must program it yourself; for example, the above goal could legally be written as

```
P = (pred(X::out) is det :- X = 6),
foo(Q),
all [X] (call(P, X) <=> call(Q, X)).
```

Note that the compiler will only catch direct attempts at higher-order unifications; indirect attempts (via polymorphic predicates, for example '`(list.append([], [P], [Q])`' may result in an error at run-time rather than at compile-time.

Mercury also provides builtin '`inst`' values for use with solver types:

```
any_pred is Determinism
any_pred(Mode) is Determinism
any_pred(Mode1, Mode2) is Determinism
...
any_func = Mode is Determinism
any_func(Mode1) = Mode is Determinism
any_func(Mode1, Mode2) = Mode is Determinism
...
```

See Chapter 16 [Solver types], page 135 for more details.

8.3.2 Default insts for functions

In order to call a higher-order term, the compiler must know its higher-order inst. This can cause problems when higher-order terms are placed into a polymorphic collection type and then extracted, since the declared mode for the extraction will typically be `out` and the higher-order inst information will be lost. To partially alleviate this problem, and to make higher-order functional programming easier, if the term to be called has a function type, but no higher-order inst information is explicitly provided, we assume that it has the default higher-order function inst '`func(in, ..., in) = out is det`'.

As a consequence of this, a higher-order function term can *only* be passed where a term with no higher-order inst information is expected if it can be passed where a term with the default higher-order function inst is expected. Higher-order predicate terms can always be passed to such a place, but note that there is little value in doing so because there is no default higher-order inst for predicates therefore it will not be possible to call those terms.

8.3.3 Combined higher-order types and insts

A higher-order type may optionally specify an inst in the following manner:

```
(pred) is Determinism
pred(Type::Mode) is Determinism
pred(Type1::Mode1, Type2::Mode2) is Determinism
...
(func) = (Type::Mode) is Determinism
func(Type1::Mode1) = (Type::Mode) is Determinism
func(Type1::Mode1, Type2::Mode2) = (Type::Mode) is Determinism
...
```

When used as argument types of functors in type declarations, types of this form have two effects. First, for any unification that constructs a term using such an argument, there is an additional mode constraint that the argument must be approximated by the inst. In other words, to be mode correct a program must not construct any term where a functor has an argument that does not have the declared inst, if present.

The second effect is that when a unification deconstructs a ground term to extract an argument with such a declared inst, the extracted argument may then be used as if it had that inst.

For example, given this type declaration:

```
:- type job
    --->    job(pred(int::out, io::di, io::uo) is det).
```

the following goal is correct:

```
:- pred run(job::in, io::di, io::uo) is det.
run(Job, !IO) :-
    Job = job(Pred),
    Pred(Result, !IO),          % Pred has the necessary inst
    write_line(Result, !IO).
```

However, the following would be a mode error:

```
:- pred bad(job::out) is det.
bad(job(p)).                    % Error: p does not have required mode

:- pred p(int::in, io::di, io::out) is det.
...
```

As a new feature, combined higher-order types and insts are only permitted as direct arguments of functors in discriminated unions. So the following examples currently result in errors.

```
% Error: use on the RHS of equivalence types.
:- type p == (pred(io::di, io::uo) is det).
:- type f == (func(int::in) = (int::out) is semidet).

% Error: use inside a type constructor.
:- type jobs
    --->    jobs(list(pred(int::out, io::di, io::uo) is det)).

% Error: use in a pred/func declaration.
:- pred p((pred(io::di, io::uo) is det)::in, io::di, io::uo) is det.
:- func f(func(int::in) = (int::out) is semidet, int) = int.
```

Future versions of the language may allow these forms.

9 Modules

9.1 The module system

The Mercury module system is relatively simple and straightforward.

Each module must start with a '`:- module` *`ModuleName`*' declaration, specifying the name of the module.

An '`:- interface.`' declaration indicates the start of the module's interface section: this section specifies the entities that are exported by this module. Mercury provides support for abstract data types, by allowing the definition of a type to be kept hidden, with the interface only exporting the type name. The interface section may contain definitions of types, type classes, data constructors, instantiation states, and modes, and declarations for abstract data types, abstract type class instances, functions, predicates, and (sub-)modules. The interface section may not contain definitions for functions or predicates (i.e. clauses), or definitions of (sub-)modules.

An '`:- implementation.`' declaration indicates the start of the module's implementation section. Any entities declared in this section are local to the module (and its sub-modules) and cannot be used by other modules. The implementation section must contain definitions for all abstract data types, abstract instance declarations, functions, predicates, and sub-modules exported by the module, as well as for all local types, type class instances, functions, predicates, and sub-modules. The implementation section can be omitted if it is empty.

The module may optionally end with a '`:- end_module` *`ModuleName`*' declaration; the name specified in the '`end_module`' must be the same as that in the corresponding '`module`' declaration.

If a module wishes to make use of entities exported by other modules, then it must explicitly import those modules using one or more '`:- import_module` *`Modules`*' or '`:- use_module` *`Modules`*' declarations, in order to make those declarations visible. In both cases, *Modules* is a comma-separated list of fully-qualified module names. These declarations may occur either in the interface or the implementation section. If the imported entities are used in the interface section, then the corresponding `import_module` or `use_module` declaration must also be in the interface section. If the imported entities are only used in the implementation section, the `import_module` or `use_module` declaration should be in the implementation section.

The names of predicates, functions, constructors, constructor fields, types, modes, insts, type classes, and (sub-)modules can be explicitly module qualified using the '`.`' operator, e.g. '`module.name`' or '`module.submodule.name`'. This is useful both for readability and for resolving name conflicts. Uses of entities imported using `use_module` declarations *must* be explicitly fully module qualified.

Currently we also support '`__`' as an alternative module qualifier, so you can write `module__name` instead of `module.name`.

Certain optimizations require information or source code for predicates defined in other modules to be as effective as possible. At the moment, inlining and higher-order specialization are the only optimizations that the Mercury compiler can perform across module boundaries.

One module must export a predicate '`main/2`', which must be declared as either

```
:- pred main(io.state::di, io.state::uo) is det.
```

or

```
:- pred main(io.state::di, io.state::uo) is cc_multi.
```

(or any declaration equivalent to one of the two above).

Mercury has a standard library which includes over 100 modules, including modules for lists, stacks, queues, priority queues, sets, bags (multi-sets), maps (dictionaries), random number generation, input/output, and filename and directory handling. See the Mercury Library Reference Manual for a list of the available modules, and for the documentation of each module.

9.2 An example module.

For illustrative purposes, here is the definition of a simple module for managing queues:

```
:- module queue.
:- interface.

% Declare an abstract data type.

:- type queue(T).

% Declare some predicates which operate on the abstract data type.

:- pred empty_queue(queue(T)).
:- mode empty_queue(out) is det.
:- mode empty_queue(in) is semidet.

:- pred put(queue(T), T, queue(T)).
:- mode put(in, in, out) is det.

:- pred get(queue(T), T, queue(T)).
:- mode get(in, out, out) is semidet.

:- implementation.

% Queues are implemented as lists. We need the ‘list’ module
% for the declaration of the type list(T), with its constructors
% ’[]’/0 % and ’.’/2, and for the declaration of the predicate
% list.append/3.

:- import_module list.

% Define the queue ADT.

:- type queue(T) == list(T).

% Define the exported predicates.
```

```
empty_queue([]).

put(Queue0, Elem, Queue) :-
    list.append(Queue0, [Elem], Queue).

get([Elem | Queue], Elem, Queue).

:- end_module queue.
```

9.3 Sub-modules

As mentioned above, modules may contain sub-modules. There are two kinds of sub-modules, called nested sub-modules and separate sub-modules; the difference is that nested sub-modules are defined in the same source file as the containing module, whereas separate sub-modules are defined in separate source files. Implementations should support separate compilation of separate sub-modules.

A module may not contain more than one sub-module with the same name.

9.3.1 Nested sub-modules

Nested sub-modules within a module are delimited by matching '`:- module`' and '`:- end_module`' declarations. (Note that '`:- end_module`' for nested sub-modules are mandatory, not optional, even if the nested sub-module is the last thing in the source file. Also note that the module name in a '`:- module`' or '`:- end_module`' declaration need not be fully-qualified.) The sequence of items thus delimited is known as a sub-module item sequence.

The interface and implementation parts of a nested sub-module may be specified in two different sub-module declarations. If a sub-module item sequence includes an interface section, then it is a declaration of that sub-module; if it includes an implementation section, then it is a definition of that sub-module; and if includes both, then it is both declaration and definition.

It is an error to declare a sub-module twice, or to define it twice. It is an error to define a sub-module without declaring it. As mentioned earlier, it is an error to define a sub-module in the interface section of its parent module.

If a sub-module is declared but not explicitly defined, then there is an implicit definition with an empty implementation section for that sub-module. This empty implementation section will result in an error if the interface section of a sub-module contains any of the following:

- a declaration for a function or a predicate;
- an abstract declaration for a type, inst, mode or typeclass, i.e. a declaration that does not itself serve as a definition of that type, inst, mode or typeclass;
- an abstract declaration of a typeclass instance; or
- a (doubly, triply, etc) nested sub-module (which perforce has only an interface section, and no implementation section) and which contains any of the above.

9.3.2 Separate sub-modules

Separate sub-modules are declared using '`:- include_module` *`Modules`*' declarations. Each '`:- include_module`' declaration specifies a comma-separated list of sub-modules.

```
:- include_module Module1, Module2, ..., ModuleN.
```

Each of the named sub-modules in an '`:- include_module`' declaration must be defined in a separate source file. The mapping between module names and source file names is implementation-defined. (For a module named '`foo.bar.baz`', The University of Melbourne Mercury implementation requires the source to be located in a file named '`foo.bar.baz.m`', '`bar.baz.m`', or '`baz.m`'.) The separate source file must contain the declaration (interface) and definition (implementation) of the sub-module. It must start with a '`:- module`' declaration which matches that in the '`:- include_module`' declaration in the parent, followed by the interface and (if necessary) implementation sections, and it may optionally end with a '`:- end_module`' declaration. (Note: the module names in the '`:- module`', '`:- end_module`', and '`:- include_module`' declarations need not be fully-qualified. However, if the file name used for a particular module does not include all the module qualifiers, then the University of Melbourne Mercury implementation requires the module name in the '`:- module`' declaration for that module to be fully qualified.)

The semantics of separate sub-modules are identical to those of nested sub-modules. The procedure to transform a separate sub-module into a nested sub-module is as follows:

1. Replace the '`:- include_module` *`submodule`*' declaration with the interface section of the sub-module enclosed within '`:- module` *`submodule`*' and '`:- end_module` *`submodule`*' declarations.
2. Place the implementation section of the sub-module enclosed within '`:- module` *`submodule`*' and '`:- end_module` *`submodule`*' declarations in the implementation section of the parent module.

For example

```
:- module x.
:- interface.
:- include_module y.
:- end_module x.
```

is equivalent to

```
:- module x.
:- interface.
    :- module y.
    % interface section of module `y'
    :- end_module y.
:- implementation.
    :- module y.
    % implementation section of module `y'
    :- end_module y.
:- end_module x.
```

9.3.3 Visibility rules

Any declarations in the parent module, including those in the parent module's implementation section, are visible in the parent's sub-modules, including indirect sub-modules (i.e.

sub-sub-modules, etc.). Similarly, declarations in the interfaces of any modules imported using an '`:- import_module`' or a '`:- use_module`' in the parent module are visible in the parent's sub-modules, including indirect sub-modules.

Declarations in a child module are not visible in the parent module, or in "sibling" modules (other children of the same parent), or in other unrelated modules unless the child is explicitly imported using an '`:- import_module`' or '`:- use_module`' declaration. It is an error to import a module without importing all of its parent modules.

Note that a sub-module for which the '`:- module`' or '`:- include_module`' declaration occurs only in the implementation section of the parent module may only be imported or used by its parent module or by sub-modules of its parent module.

Note that as mentioned previously, all '`:- import_module`' and '`:- use_module`' declarations must use fully-qualified module names.

9.3.4 Implementation bugs and limitations

The current implementation of sub-modules has a couple of minor limitations.

- The compiler sometimes reports spurious errors if you define an equivalence type in a sub-module and export it as an abstract type.
- Using '`mmake`' to do parallel makes (e.g. '`mmake --jobs 2`') doesn't always work correctly if you're using nested sub-modules. (The work-around is to use separate sub-modules instead of nested sub-modules, i.e. to put the sub-modules in separate source files.)

9.4 Module initialisation

Modules that interact with foreign libraries or services may require special initialisation before use. Such modules may include any number of '`initialise`' directives in their implementation sections. An '`initialise`' directive has the following form:

```
:- initialise initpredname/arity.
```

where the predicate *initpredname* must be declared with one of the following signatures:

```
:- pred initpredname(io::di, io::uo) is Det.
:- impure pred initpredname is Det.
```

Det must be either `det` or `cc_multi`.

The effect of the '`initialise`' declaration is to ensure that '*`initpredname/arity`*' is invoked before the program's '`main/2`' predicate. Initialisation predicates within a module are executed in the order in which they are specified, although no order may be assumed between different modules or sub-modules. Initialisation predicates are only invoked after any initialisation required by the Mercury standard library.

If '*`initpredname/arity`*' terminates with an uncaught exception then the program will immediately abort execution. In this circumstance those predicates specified by other '`initialise`' directives that have not yet been executed will not be executed, '`main/2`' will not be executed and no predicate specified in a '`finalise`' directive will be executed.

'`initialize`' is also allowed as a synonym for '`initialise`'.

9.5 Module finalisation

Modules that require special finalisation at program termination may include any number of '`finalise`' directives in their implementation sections.

A '`finalise`' directive has the following form:

```
:- finalise finalpredname/arity.
```

where the predicate '`finalpredname/arity`' must be declared with one of the following signature:

```
:- pred finalpredname(io::di, io::uo) is Det.
:- impure pred finalpredname is Det
```

Det must be either `det` or `cc_multi`.

The effect of the '`finalise`' declaration is to ensure that '*`finalpredname/arity`*' is invoked after the program's '`main`' predicate. Finalisation predicates within a module are executed in the order in which they are specified, although no order may be assumed between different modules or sub-modules. Any finalisation required by the Mercury standard library will always occur after any finalisation predicates have been invoked.

If '*`finalpredname/arity`*' terminates with an uncaught exception then the program will immediately abort execution. No predicates specified by other '`finalise`' directives that have not yet been executed will be executed. If the program's '`main/2`' predicate terminates with an uncaught exception then no finalisation predicates will be executed.

'`finalize`' is also allowed as a synonym for '`finalise`'.

9.6 Module-local mutable variables

Certain special cases require a module to have one or more mutable (i.e. destructively updatable) variables, for example to hold the constraint store for a solver type.

A mutable variable is declared using the '`mutable`' directive:

```
:- mutable(varname, vartype, initial_value, varinst, [attribute, ...]).
```

This constructs a new mutable variable with access predicates that have the following signatures:

```
:- semipure pred get_varname(vartype::out(varinst)) is det.
:- impure   pred set_varname(vartype::in(varinst)) is det.
```

The initial value of *varname* is *initial_value*, which is set before the program's '`main/2`' predicate is executed.

The type *vartype* is not allowed to contain any type variables or have any type class constraints.

The inst *varinst* is not allowed to contain any inst variables. It is also not allowed to be equivalent to, or contain components that are equivalent to, the builtin insts `free`, `unique`, `mostly_unique`, `dead` (`clobbered`) or `mostly_dead` (`mostly_clobbered`).

The initial value of a mutable, *initial_value*, may be any Mercury expression with type *vartype* and inst *varinst* subject to the above restrictions. It may be impure or semipure.

The following *attributes* must be supported:

‘`trailed`’/‘`untrailed`’
: This attribute declares if the implementation should generate code so that the effects of ‘`set_`*varname*`/1`’ can be undone on backtracking. The default, in case none is specified, is ‘`trailed`’.

‘`attach_to_io_state`’
: This attribute causes the compiler to also construct access predicates that have the following signatures:

```
:- pred get_varname(vartype::out(varinst), io::di, io::uo) is det.
:- pred set_varname(vartype::in(varinst),  io::di, io::uo) is det.
```

‘`constant`’
: This attribute causes the compiler to construct only a ‘`get`’ access predicate, but not a ‘`set`’ access predicate. Since *varname* will always have the initial value given to it, the ‘`get`’ access predicate is pure; its signature will be:

```
:- pred get_varname(vartype::out(varinst)) is det.
```

The ‘`constant`’ attribute cannot be specified together with the ‘`attach_to_io_state`’ attribute (since they disagree on this signature). It also cannot be specified together with an explicit ‘`trailed`’ attribute.

The Melbourne Mercury compiler also supports the following attributes:

‘`foreign_name(`*Lang*`, `*Name*`)`’
: Allow foreign code to access the mutable variable in some implementation dependent manner. *Lang* must be a valid target language for this Mercury implementation. *Name* must be a valid identifier in that language. It is an error to specify more than one foreign name attribute for each language.

For the C backends this attribute allows foreign code to access the mutable variable as an external variable called *Name*. For the low-level C backend, e.g. the asm_fast grades, the type of this variable will be `MR_Word`. For the high-level C backend, e.g. the hlc grades, the type of this variable depends upon the Mercury type of the mutable. For mutables of a Mercury primitive type, the corresponding C type is given by the mapping in Section 14.3.1 [C data passing conventions], page 103. For mutables of any other type the corresponding C type will be `MR_Word`.

This attribute is not currently implemented for the non-C backends.

‘`thread_local`’
: This attribute allows a mutable to take on different values in each thread. When a child thread is spawned, it inherits all the values of thread-local mutables of the parent thread. Changing the value of a thread-local mutable does not affect its value in any other threads.

The ‘`thread_local`’ attribute cannot be specified together with either of the ‘`trailed`’ or ‘`constant`’ attributes.

It is an error for a ‘`mutable`’ directive to appear in the interface section of a module. The usual visibility rules for sub-modules apply to the mutable variable access predicates.

For the purposes of determining when mutables are assigned their initial values, the expression *initial_value* behaves as though it were a predicate specified in an ‘`initialise`’ directive.

```
:- initialise foo/2.
:- mutable(bar, int, 561, ground, [untrailed]).
:- initialise baz/2.
```

In the above example '`foo/2`' is invoked first, then '`bar`' is set with an initial value of 561 and the '`baz/2`' is invoked.

The effect of a mutable initial value expression terminating with an uncaught exception is also the same as though it were a predicate specified in an '`initialise`' directive.

10 Type classes

Mercury supports constrained polymorphism in the form of type classes. Type classes allow the programmer to write predicates and functions which operate on variables of any type (or sequence of types) for which a certain set of operations is defined.

10.1 Typeclass declarations

A *type class* is a name for a set of types (or a set of sequences of types) for which certain predicates and/or functions, called the *methods* of that type class, are defined. A '`typeclass`' declaration defines a new type class, and specifies the set of predicates and/or functions that must be defined on a type (or sequence of types) for it (them) to be considered to be an instance of that type class.

The `typeclass` declaration gives the name of the type class that it is defining, the names of the type variables which are parameters to the type class, and the operations (i.e. methods) which form the interface of the type class. For each method, all parameters of the typeclass must be determined by the type variables appearing in the type signature of the method. A variable is determined by a type signature if it appears in the type signature, but if functional dependencies are present then it may also be determined from the other variables (see Section 10.8 [Functional dependencies], page 82).

For example,

```
:- typeclass point(T) where [
        % coords(Point, X, Y):
        %       X and Y are the cartesian coordinates of Point
        pred coords(T, float, float),
        mode coords(in, out, out) is det,

        % translate(Point, X_Offset, Y_Offset) = NewPoint:
        %       NewPoint is Point translated X_Offset units in the X direction
        %       and Y_Offset units in the Y direction
        func translate(T, float, float) = T
].
```

declares the type class `point`, which represents points in two dimensional space.

`pred`, `func` and `mode` declarations are the only legal declarations inside a `typeclass` declaration. The mode and determinism of type class methods must be explicitly declared or (for functions) defaulted, not inferred. In other words, for each predicate declared in a type class, there must be at least one mode declaration, and each mode declaration in a type class must include an explicit determinism annotation. Functions with no explicit mode declaration get the usual default mode (see Chapter 4 [Modes], page 39): all arguments have mode `in`, the result has mode `out`, and the determinism is `det`.

The number of parameters to the type class (e.g. `T`) is not limited. For example, the following is allowed:

```
:- typeclass a(T1, T2) where [...].
```

The parameters must be distinct variables. Each `typeclass` declaration must have at least one parameter.

It is legal for a `typeclass` declaration to declare no methods, for example

```
:- typeclass foo(T) where [].
```

There must not be more than one type class declaration with the same name and arity in the same module.

10.2 Instance declarations

Once the interface of the type class has been defined in the `typeclass` declaration, we can use an `instance` declaration to define how a particular type (or sequence of types) satisfies the interface declared in the `typeclass` declaration.

An instance declaration has the form

```
:- instance classname(typename(typevar, ...), ...)
        where [methoddefinition, methoddefinition, ...].
```

An '`instance`' declaration gives a type for each parameter of the type class. Each of these types must be either a type with no arguments, or a polymorphic type whose arguments are all type variables. For example `int`, `list(T)`, `bintree(K, V)` and `bintree(T, T)` are allowed, but `T` and `list(int)` are not. The types in an instance declaration must not be abstract types which are elsewhere defined as equivalence types. A program may not contain more than one instance declaration for a particular type (or sequence of types, in the case of a multi-parameter type class) and typeclass. These restrictions ensure that there are no overlapping instance declarations, i.e. for each typeclass there is at most one instance declaration that may be applied to any type (or sequence of types).

Each *methoddefinition* entry in the '`where [...]`' part of an `instance` declaration defines the implementation of one of the class methods for this instance. There are two ways of defining methods. The first way is to define a method by giving the name of the predicate or function which implements that method. In this case, the *methoddefinition* must have one of the following forms:

```
pred(methodname/arity) is predname
func(methodname/arity) is funcname
```

The *predname* or *funcname* must name a predicate or function of the specified arity whose type, modes, determinism, and purity are at least as permissive as the declared type, modes, determinism, and purity of the class method with the specified *methodname* and *arity*, after the types of the arguments in the instance declaration have been substituted in place of the parameters in the type class declaration.

The second way of defining methods is by listing the clauses for the definition inside the instance declaration. A *methoddefinition* can be a clause. These clauses are just like the clauses used to define ordinary predicates or functions (see Section 2.7 [Items], page 9), and so they can be facts, rules, or DCG rules. The only difference is that in instance declarations, clauses are separated by commas rather than being terminated by periods, and so rules and DCG rules in instance declarations must normally be enclosed in parentheses. As with ordinary predicates, you can have more than one clause for each method. The clauses must satisfy the declared type, modes, determinism and purity for the method, after the types of the arguments in the instance declaration have been substituted in place of the parameters in the type class declaration.

These two ways are mutually exclusive: each method must be defined either by a single naming definition (using the '`pred(...) is predname`' or '`func(...) is funcname`' form), or by a set of one or more clauses, but not both.

Here's an example of an instance declaration and the different kinds of method definitions that it can contain:

```
:- typeclass foo(T) where [
    func method1(T, T) = int,
    func method2(T) = int,
    pred method3(T::in, int::out) is det,
    pred method4(T::in, io.state::di, io.state::uo) is det,
    func method5(bool, T) = T
].

:- instance foo(int) where [
    % method defined by naming the implementation
    func(method1/2) is (+),

    % method defined by a fact
    method2(X) = X + 1,

    % method defined by a rule
    (method3(X, Y) :- Y = X + 2),

    % method defined by a DCG rule
    (method4(X) --> io.print(X), io.nl),

    % method defined by multiple clauses
    method5(no, _) = 0,
    (method5(yes, X) = Y :- X + Y = 0)
].
```

Each '`instance`' declaration must define an implementation for every method declared in the corresponding '`typeclass`' declaration. It is an error to define more than one implementation for the same method within a single '`instance`' declaration.

Any call to a method must have argument types (and in the case of functions, return type) which are constrained to be a member of that method's type class, or which match one of the instance declarations visible at the point of the call. A method call will invoke the predicate or function specified for that method in the instance declaration that matches the types of the arguments to the call.

Note that even if a type class has no methods, an explicit instance declaration is required for a type to be considered an instance of that type class.

Here's an example of some code using an instance declaration:

```
:- type coordinate
    --->    coordinate(
                float,  % X coordinate
                float   % Y coordinate
            ).

:- instance point(coordinate) where [
```

```
    pred(coords/3) is coordinate_coords,
    func(translate/3) is coordinate_translate
].

:- pred coordinate_coords(coordinate, float, float).
:- mode coordinate_coords(in, out, out) is det.

coordinate_coords(coordinate(X, Y), X, Y).

:- func coordinate_translate(coordinate, float, float) = coordinate.

coordinate_translate(coordinate(X, Y), Dx, Dy) = coordinate(X + Dx, Y + Dy).
```

We have now made the `coordinate` type an instance of the `point` type class. If we introduce a new type, `coloured_coordinate` which represents a point in two dimensional space with a colour associated with it, it can also become an instance of the type class:

```
:- type rgb
    --->    rgb(
                int,
                int,
                int
            ).

:- type coloured_coordinate
    --->    coloured_coordinate(
                float,
                float,
                rgb
            ).

:- instance point(coloured_coordinate) where [
    pred(coords/3) is coloured_coordinate_coords,
    func(translate/3) is coloured_coordinate_translate
].

:- pred coloured_coordinate_coords(coloured_coordinate, float, float).
:- mode coloured_coordinate_coords(in, out, out) is det.

coloured_coordinate_coords(coloured_coordinate(X, Y, _), X, Y).

:- func coloured_coordinate_translate(coloured_coordinate, float, float)
    = coloured_coordinate.

coloured_coordinate_translate(coloured_coordinate(X, Y, Colour), Dx, Dy)
    = coloured_coordinate(X + Dx, Y + Dy, Colour).
```

If we call '`translate/3`' with the first argument having type '`coloured_coordinate`', this will invoke '`coloured_coordinate_translate`'. Likewise, if we call '`translate/3`' with the first argument having type '`coordinate`', this will invoke '`coordinate_translate`'.

Further instances of the type class could be made, e.g. a type that represents the point using polar coordinates.

10.3 Abstract typeclass declarations

Abstract typeclass declarations are typeclass declarations whose definitions are hidden. An abstract typeclass declaration has the same form as a typeclass declaration, but without the '`where[...]`' part. An abstract typeclass declaration defines a name for a set of (sequences of) types, but does not define what methods must be implemented for instances of the type class.

Like abstract type declarations, abstract typeclass declarations are only useful in the interface section of a module. Each abstract typeclass declaration must be accompanied by a corresponding non-abstract typeclass declaration that defines the methods for that type class.

Non-abstract instance declarations can only be made in scopes where the non-abstract typeclass declaration is visible.

10.4 Abstract instance declarations

Abstract instance declarations are instance declarations whose implementations are hidden. An abstract instance declaration has the same form as an instance declaration, but without the '`where [...]`' part. An abstract instance declaration declares that a sequence of types is an instance of a particular type class without defining how the type class methods are implemented for those types. Like abstract type declarations, abstract instance declarations are only useful in the interface section of a module. Each abstract instance declaration must be accompanied by a corresponding non-abstract instance declaration that defines how the type class methods are implemented.

Here's an example:

```
:- module hashable.
:- interface.
:- import_module int, string.

:- typeclass hashable(T) where [func hash(T) = int].
:- instance hashable(int).
:- instance hashable(string).

:- implementation.

:- instance hashable(int) where [func(hash/1) is hash_int].
:- instance hashable(string) where [func(hash/1) is hash_string].

:- func hash_int(int) = int.
hash_int(X) = X.
```

```
:- func hash_string(string) = int.
hash_string(S) = H :-
    % Use the standard library predicate string.hash/2.
    string.hash(S, H).

:- end_module hashable.
```

10.5 Type class constraints on predicates and functions

Mercury allows a type class constraint to appear as part of a predicate or function's type signature. This constrains the values that can be taken by type variables in the signature to belong to particular type classes.

A type class constraint has the form:

```
<= Typeclass(Type, ...), ...
```

where *Typeclass* is the name of a type class and *Type* is a type. Any variable that appears in *Type* must be determined by the predicate's or function's type signature. A variable is determined by a type signature if it appears in the type signature, but if functional dependencies are present then it may also be determined from other variables (see Section 10.8 [Functional dependencies], page 82). Each type class constraint in a predicate or function declaration must contain at least one variable.

For example

```
:- pred distance(P1, P2, float) <= (point(P1), point(P2)).
:- mode distance(in, in, out) is det.

distance(A, B, Distance) :-
    coords(A, Xa, Ya),
    coords(B, Xb, Yb),
    XDist = Xa - Xb,
    YDist = Ya - Yb,
    Distance = sqrt(XDist*XDist + YDist*YDist).
```

In the above example, the `distance` predicate is able to calculate the distance between any two points, regardless of their representation, as long as the `coords` operation has been defined. These constraints are checked at compile time.

10.6 Type class constraints on type class declarations

Type class constraints may also appear in `typeclass` declarations, meaning that one type class is a "superclass" of another.

The arguments of a constraint on a type class declaration must be either type variables or ground types. Each constraint must contain at least one variable argument and all variables that appear in the arguments must also be arguments to the type class in question.

For example, the following declares the '`ring`' type class, which describes types with a particular set of numerical operations defined:

```
:- typeclass ring(T) where [
    func zero = (T::out) is det,              % '+' identity
    func one = (T::out) is det,               % '*' identity
```

```
    func plus(T::in, T::in) = (T::out) is det, % '+'/2 (forward mode)
    func mult(T::in, T::in) = (T::out) is det, % '*'/2 (forward mode)
    func negative(T::in) = (T::out) is det     % '-'/1 (forward mode)
].
```

We can now add the following declaration:

```
:- typeclass euclidean(T) <= ring(T) where [
    func div(T::in, T::in) = (T::out) is det,
    func mod(T::in, T::in) = (T::out) is det
].
```

This introduces a new type class, `euclidean`, of which `ring` is a superclass. The operations defined by the `euclidean` type class are `div`, `mod`, as well as all those defined by the `ring` type class. Any type declared to be an instance of `euclidean` must also be declared to be an instance of `ring`.

Type class constraints on type class declarations gives rise to a superclass relation. This relation must be acyclic. That is, it is an error if a type class is its own (direct or indirect) superclass.

10.7 Type class constraints on instance declarations

Type class constraints may also be placed upon instance declarations. The arguments of such constraints must be either type variables or ground types. Each constraint must contain at least one variable argument and all variables that appear in the arguments must be type variables that appear in the types in the instance declaration.

For example, consider the following declaration of a type class of types that may be printed:

```
:- typeclass portrayable(T) where [
    pred portray(T::in, io.state::di, io.state::uo) is det
].
```

The programmer could declare instances such as

```
:- instance portrayable(int) where [
    pred(portray/3) is io.write_int
].

:- instance portrayable(char) where [
    pred(portray/3) is io.write_char
].
```

However, when it comes to writing the instance declaration for a type such as `list(T)`, we want to be able print out the list elements using the `portray/3` for the particular type of the list elements. This can be achieved by placing a type class constraint on the `instance` declaration, as in the following example:

```
:- instance portrayable(list(T)) <= portrayable(T) where [
    pred(portray/3) is portray_list
].

:- pred portray_list(list(T), io.state, io.state) <= portrayable(T).
```

```
:- mode portray_list(in, di, uo) is det.

portray_list([], !IO).
portray_list([X | Xs], !IO) :-
    portray(X, !IO),
    io.write_char(' ', !IO),
    portray_list(Xs, !IO).
```

For abstract instance declarations, the type class constraints on an abstract instance declaration must exactly match the type class constraints on the corresponding non-abstract instance declaration that defines that instance.

10.8 Functional dependencies

Type class constraints may include any number of functional dependencies. A *functional dependency* constraint takes the form `(`*`Domain`*` -> `*`Range`*`)`. The *Domain* and *Range* arguments are either single type variables, or conjunctions of type variables separated by commas.

```
:- typeclass Typeclass(Var, ...) <= ((D -> R), ...) ...

:- typeclass Typeclass(Var, ...) <= (D1, D2, ... -> R1, R2, ...), ...
```

Each type variable must appear in the parameter list of the typeclass. Abstract typeclass declarations must have exactly the same functional dependencies as in the implementation.

Mutually recursive functional dependencies are allowed, so the following examples are legal:

```
:- typeclass foo(A, B) <= ((A -> B), (B -> A)).
:- typeclass bar(A, B, C, D) <= ((A, B -> C), (B, C -> D), (D -> A, C)).
```

A functional dependency on a typeclass places an additional requirement on the set of instances which are allowed for that type class. The requirement is that all types bound to variables in the range of the functional dependency must be able to be uniquely determined by the types bound to variables in the domain of the functional dependency. If more than one functional dependency is present, then the requirement for each one must be satisfied.

For example, given the typeclass declaration

```
:- typeclass baz(A, B) <= (A -> B) where ...
```

it would be illegal to have both of the instances

```
:- instance baz(int, int) where ...
:- instance baz(int, string) where ...
```

although either one would be acceptable on its own.

The following instance would also be illegal

```
:- instance baz(string, list(T)) where ...
```

since the variable `T` may not always be bound to the same type. However, the instance

```
:- instance baz(list(S), list(T)) <= baz(S, T) where ...
```

is legal because the '`baz(S, T)`' constraint ensures that whatever `T` is bound to, it is always uniquely determined from the binding of `S`.

The extra requirements that result from the use of functional dependencies allow the bindings of some variables to be determined from the bindings of others. This in turn relaxes some of the requirements of typeclass constraints on predicate and function signatures, and on existentially typed data constructors.

Without any functional dependencies, all variables in constraints must appear in the signature of the predicate or function being declared. However, variables which are in the range of a functional dependency need not appear in the signature, since it is known that their bindings will be determined from the bindings of the variables in the domain.

More formally, the constraints on a predicate or function signature *induce* a set of functional dependencies on the variables appearing in those constraints. A functional dependency '`(A1, ... -> B1, ...)`' is induced from a constraint '*`Typeclass`*`(`*`Type1, ...`*`)`' if and only if the typeclass '*`Typeclass`*' has a functional dependency '`(D1, ... -> R1, ...)`', and for each typeclass parameter '`Di`' there exists an '`Aj`' every type variable appearing in the '*`Typek`*' corresponding to '`Di`', and each '`Bi`' appears in the '*`Typej`*' bound to the typeclass parameter '`Rk`' for some *k*.

For example, with the definition of `baz` above, the constraint `baz(map(X, Y), list(Z))` induces the constraint `(X, Y -> Z)`, since *X* and *Y* appear in the domain argument, and *Z* appears in the range argument.

The set of type variables determined from a signature is the *closure* of the set appearing in the signature under the functional dependencies induced from the constraints. The closure is defined as the smallest set of variables which includes all of the variables appearing in the signature, and is such that, for each induced functional dependency '*`Domain -> Range`*', if the closure includes all of the variables in *Domain* then it includes all of the variables in *Range*.

For example, the declaration

```
:- pred p(X, Y) <= baz(map(X, Y), list(Z)).
```

is acceptable since the closure of {*X*, *Y*} under the induced functional dependency must include *Z*. Moreover, the typeclass `baz/2` would be allowed to have a method that only uses the first parameter, *A*, since the second parameter, *B*, would always be determined from the first.

Note that, since all instances must satisfy the superclass constraints, the restrictions on instances obviously transfer from superclass to subclass. Again, this allows the requirements of typeclass constraints to be relaxed. Thus, the functional dependencies on the ancestors of constraints also induce functional dependencies on the variables, and the closure that we calculate takes these into account.

For example, in this code

```
:- typeclass quux(P, Q, R) <= baz(R, P) where ...

:- pred q(Q, R) <= quux(P, Q, R).
```

the signature of `q/2` is acceptable since the superclass constraint on `quux/2` induces the dependency '`R -> P`' on the type variables, hence *P* is in the closure of {*Q*, *R*}.

The presence of functional dependencies also allows "improvement" to occur during type inference. This can occur in two ways. First, if two constraints of a given class match on all of the domain arguments of a functional dependency on that class, then it can be

inferred that they also match on the range arguments. For example, given the constraints `baz(A, B1)` and `baz(A, B2)`, it will be inferred that `B1 = B2`.

Similarly, if a constraint of a given class is subsumed by a known instance of that class in the domain arguments, then its range arguments can be unified with the corresponding instance range arguments. For example, given the instance:

```
:- instance baz(list(T), string) where ...
```

then the constraint `baz(list(int), X)` can be improved with the inference that `X = string`.

11 Existential types

Existentially quantified type variables (or simply "existential types" for short) are useful tools for data abstraction. In combination with type classes, they allow you to write code in an "object oriented" style that is similar to the use of interfaces in Java or abstract base classes in C++.

Mercury supports existential type quantifiers on predicate and function declarations, and in data type definitions. You can put type class constraints on existentially quantified type variables.

11.1 Existentially typed predicates and functions

11.1.1 Syntax for explicit type quantifiers

Type variables in type declarations for polymorphic predicates or functions are normally universally quantified. However, it is also possible to existentially quantify such type variables, by using an explicit existential quantifier of the form '`some` *`Vars`*' before the '`pred`' or '`func`' declaration, where *Vars* is a list of variables.

For example:

```
% Here the type variables 'T' is existentially quantified
:- some [T] pred foo(T).

% Here the type variables 'T1' and 'T2' are existentially quantified.
:- some [T1, T2] func bar(int, list(T1), set(T2)) = pair(T1, T2).

% Here the type variable 'T2' is existentially quantified,
% but the type variables 'T1' and 'T3' are universally quantified.
:- some [T2] pred foo(T1, T2, T3).
```

Explicit universal quantifiers, of the form '`all` *`Vars`*', are also permitted on '`pred`' and '`func`' declarations, although they are not necessary, since universal quantification is the default. (If both universal and existential quantifiers are present, the universal quantifiers must precede the existential quantifiers.) For example:

```
% Here the type variable 'T2' is existentially quantified,
% but the type variables 'T1' and 'T3' are universally quantified.
:- all [T3] some [T2] pred foo(T1, T2, T3).
```

11.1.2 Semantics of type quantifiers

If a type variable in the type declaration for a polymorphic predicate or function is universally quantified, this means the caller will determine the value of the type variable, and the callee must be defined so that it will work for *all* types which are an instance of its declared type.

For an existentially quantified type variable, the situation is the converse: the *callee* must determine the value of the type variable, and all *callers* must be defined so as to work for all types which are an instance of the called procedure's declared type.

When type checking a predicate or function, if a variable has a type that occurs as a universally quantified type variable in the predicate or function declaration, or a type that

occurs as an existentially quantified type variable in the declaration of one of the predicates or functions that it calls, then its type is treated as an opaque type. This means that there are very few things which it is legal to do with such a variable — basically you can only pass it to another procedure expecting the same type, unify it with another value of the same type, put it in a polymorphic data structure, or pass it to a polymorphic procedure whose argument type is universally quantified. (Note, however, that the standard library includes some quite powerful procedures such as '`io.write`' which can be useful in this context.)

A non-variable type (i.e. a type that is not a type variable) is considered *more general* than an existentially quantified type variable. Type inference will therefore never infer an existentially quantified type for a predicate or function unless that predicate or function calls (directly or indirectly) a predicate or function which was explicitly declared to have an existentially quantified type.

Note that an existentially typed procedure is not allowed to have different types for its existentially typed arguments in different clauses (even mode-specific clauses) or in different subgoals of a single clause; however, the same effect can be achieved in other ways (see Section 11.4 [Some idioms using existentially quantified types], page 90).

For procedures involving calls to existentially-typed predicates or functions, the compiler's mode analysis must take account of the modes for type variables in all polymorphic calls. Universally quantified type variables have mode `in`, whereas existentially quantified type variables have mode `out`. As usual, the compiler's mode analysis will attempt to reorder the elements of conjunctions in order to satisfy the modes.

11.1.3 Examples of correct code using type quantifiers

Here are some examples of type-correct code using universal and existential types.

```
/* simple examples */

:- pred foo(T).
foo(_).
% ok

:- pred call_foo.
call_foo :- foo(42).
% ok (T = int)

:- some [T] pred e_foo(T).
e_foo(X) :- X = 42.
% ok (T = int)

:- pred call_e_foo.
call_e_foo :- e_foo(_).
% ok

/* examples using higher-order functions */

:- func bar(T, T, func(T) = int) = int.
bar(X, Y, F) = F(X) + F(Y).
```

```
% ok

:- func call_bar = int.
call_bar = bar(2, 3, (func(X) = X*X)).
% ok (T = int)
% returns 13 (= 2*2 + 3*3)

:- some [T] pred e_bar(T, T, func(T) = int).
:-          mode e_bar(out, out, out(func(in) = out is det)).
e_bar(2, 3, (func(X) = X * X)).
% ok (T = int)

:- func call_e_bar = int.
call_e_bar = F(X) + F(Y) :- e_bar(X, Y, F).
% ok
% returns 13 (= 2*2 + 3*3)
```

11.1.4 Examples of incorrect code using type quantifiers

Here are some examples of code using universal and existential types that contains type errors.

```
/* simple examples */

:- pred bad_foo(T).
bad_foo(42).
% type error

:- some [T] pred e_foo(T).
e_foo(42).
% ok

:- pred bad_call_e_foo.
bad_call_e_foo :- e_foo(42).
% type error

:- some [T] pred e_bar1(T).
e_bar1(42).
e_bar1(42).
e_bar1(43).
% ok (T = int)

:- some [T] pred bad_e_bar2(T).
bad_e_bar2(42).
bad_e_bar2("blah").
% type error (cannot unify types ‘int’ and ‘string’)
```

```
:- some [T] pred bad_e_bar3(T).
bad_e_bar3(X) :- e_foo(X).
bad_e_bar3(X) :- e_foo(X).
% type error (attempt to bind type variable 'T' twice)
```

11.2 Existential class constraints

Existentially quantified type variables are especially useful in combination with type class constraints.

Type class constraints can be either universal or existential. Universal type class constraints are written using '<=', as described in Section 10.5 [Type class constraints on predicates and functions], page 80; they signify a constraint that the *caller* must satisfy. Existential type class constraints are written in the same syntax as universal constraints, but using '=>' instead of '<='; they signify a constraint that the *callee* must satisfy. (If a declaration has both universal and existential constraints, then the existential constraints must precede the universal constraints.)

For example:

```
% Here 'c1(T2)' and 'c2(T2)' are existential constraints,
% and 'c3(T1)' is a universal constraint,
:- all [T1] some [T2] ((pred p(T1, T2) => (c1(T2), c2(T2))) <= c3(T1)).
```

Existential constraints must only constrain type variables that are explicitly existentially quantified. Likewise, universal constraints must only constrain type variables that are universally quantified, although in this case the quantification does not have to be explicit because universal quantification is the default (see Section 11.1.1 [Syntax for explicit type quantifiers], page 85).

11.3 Existentially typed data types

Type variables occurring in the body of a discriminated union type definition may be existentially quantified. Constructor definitions within discriminated union type definitions may be preceded by an existential type quantifier and followed by one or more existential type class constraints.

For example:

```
% A simple heterogeneous list type.
:- type list_of_any
    --->    nil_any
    ;       some [T] cons_any(T, list_of_any).

% A heterogeneous list type with a type class constraint.
:- typeclass showable(T) where [ func show(T) = string ].
:- type showable_list
    --->    nil
    ;       some [T] (cons(T, showable_list) => showable(T)).

% A different way of doing the same kind of thing, this
```

```
% time using the standard type list(T).
:- type showable
    --->    some [T] (s(T) => showable(T)).
:- type list_of_showable == list(showable).

% Here's an arbitrary example involving multiple
% type variables and multiple constraints.
:- typeclass foo(T1, T2) where [ /* ... */ ].
:- type bar(T)
    --->    f1
    ;       f2(T)
    ;       some [T]
            f4(T)
    ;       some [T1, T2]
            (f4(T1, T2, T) => showable(T1), showable(T2))
    ;       some [T1, T2]
            (f5(list(T1), T2) => fooable(T1, list(T2))).
```

Construction and deconstruction of existentially quantified data types are inverses: when constructing a value of an existentially quantified data type, the "existentially quantified" functor acts for purposes of type checking like a universally quantified function: the caller will determine the values of the type variables. Conversely, for deconstruction the functor acts like an existentially quantified function: the caller must be defined so as to work for all possible values of the existentially quantified type variables which satisfy the declared type class constraints.

In order to make this distinction clear to the compiler, whenever you want to construct a value using an existentially quantified functor, you must prepend '`new` ' onto the functor name. This tells the compiler to treat it as though it were universally quantified: the caller can bind that functor's existentially quantified type variables to any type which satisfies the declared type class constraints. Conversely, any occurrence without the '`new` ' prefix must be a deconstruction, and is therefore existentially quantified: the caller must not bind the existentially quantified type variables, but the caller is allowed to depend on those type variables satisfying the declared type class constraints, if any.

For example, the function '`make_list`' constructs a value of type '`list_of_showable`' containing a sequence of values of different types, all of which are instances of the '`showable`' class

```
:- instance showable(int).
:- instance showable(float).
:- instance showable(string).

:- func make_list = showable_list.
make_list = List :-
Int = 42,
Float = 1.0,
String = "blah",
List =  'new cons'(Int,
'new cons'(Float,
```

```
    'new cons'(String, nil))).
```

while the function '`process_list`' below applies the '`show`' method of the '`showable`' class to the values in such a list.

```
:- func process_list(list_of_showable) = list(string).
process_list(nil) = "".
process_list(cons(Head, Tail)) = [show(Head) | process_list(Tail)].
```

11.4 Some idioms using existentially quantified types

The standard library module '`univ`' provides an abstract type named '`univ`' which can hold values of any type. You can form heterogeneous containers (containers that can hold values of different types at the same time) by using data structures that contain `univ`s, e.g. '`list(univ)`'.

The interface to '`univ`' includes the following:

```
% 'univ' is a type which can hold any value.
:- type univ.

% The function univ/1 takes a value of any type and constructs
% a 'univ' containing that value (the type will be stored along
% with the value)
:- func univ(T) = univ.

% The function univ_value/1 takes a 'univ' argument and extracts
% the value contained in the 'univ' (together with its type).
% This is the inverse of the function univ/1.
:- some [T] func univ_value(univ) = T.
```

The '`univ`' type in the standard library is in fact a simple example of an existentially typed data type. It could be implemented as follows:

```
:- implementation.
:- type univ
    --->    some [T] mkuniv(T).
univ(X) = 'new mkuniv'(X).
univ_value(mkuniv(X)) = X.
```

An existentially typed procedure is not allowed to have different types for its existentially typed arguments in different clauses or in different subgoals of a single clause. For instance, both of the following examples are illegal:

```
:- some [T] pred bad_example(string, T).

bad_example("foo", 42).
bad_example("bar", "blah").
    % type error (cannot unify 'int' and 'string')

:- some [T] pred bad_example2(string, T).

bad_example2(Name, Value) :-
```

```
    ( Name = "foo", Value = 42
    ; Name = "bar", Value = "blah"
    ).
    % type error (cannot unify ‘int’ and ‘string’)
```

However, using ‘univ’, it is possible for an existentially typed function to return values of different types at each invocation.

```
:- some [T] pred good_example(string, T).

good_example(Name, univ_value(Univ)) :-
    ( Name = "foo", Univ = univ(42)
    ; Name = "bar", Univ = univ("blah")
    ).
```

Using ‘univ’ doesn’t work if you also want to use type class constraints. If you want to use type class constraints, then you must define your own existentially typed data type, analogous to ‘univ’, and use that:

```
:- type univ_showable
    --->    some [T] (mkshowable(T) => showable(T)).

:- some [T] pred harder_example(string, T) => showable(T).

harder_example(Name, Showable) :-
    ( Name = "bar", Univ = 'new mkshowable'(42)
    ; Name = "bar", Univ = 'new mkshowable'("blah")
    ),
    Univ = mkshowable(Showable).
```

The issue can also arise for mode-specific clauses (see Section 4.4 [Different clauses for different modes], page 45). For instance, the following example is illegal:

```
:- some [T] pred bad_example3(string, T).
:-         mode bad_example3(in(bound("foo")), out) is det.
:-          mode bad_example3(in(bound("bar")), out) is det.
:- pragma promise_pure(bad_example3/2).
bad_example3("foo"::in(bound("foo")), 42::out).
bad_example3("bar"::in(bound("bar")), "blah"::out).
    % type error (cannot unify ‘int’ and ‘string’)
```

The solution is similar, although in this case an intermediate predicate is required:

```
:- some [T] pred good_example3(string, T).
:-          mode good_example3(in(bound("foo")), out) is det.
:-          mode good_example3(in(bound("bar")), out) is det.
good_example3(Name, univ_value(Univ)) :-
good_example3_univ(Name, Univ).

:- pred good_example3_univ(string, univ).
:- mode good_example3_univ(in(bound("foo")), out) is det.
:- mode good_example3_univ(in(bound("bar")), out) is det.
:- pragma promise_pure(good_example3_univ/2).
```

```
good_example3_univ("foo"::in(bound("foo")), univ(42)::out).
good_example3_univ("bar"::in(bound("bar")), univ("blah")::out).
```

12 Exception handling

Mercury procedures may throw exceptions. Exceptions may be caught using the predicates defined in the '`exception`' library module, or using try goals.

A '`try`' goal has the following form:

```
try Params Goal
then ThenGoal
else ElseGoal
catch Term -> CatchGoal
...
catch_any CatchAnyVar -> CatchAnyGoal
```

Goal, *ThenGoal*, *ElseGoal*, *CatchGoal*, *CatchAnyGoal* must be valid goals.

Goal must have one of the following determinisms: `det`, `semidet`, `cc_multi`, or `cc_nondet`.

The non-local variables of *Goal* must not have an inst equivalent to `unique`, `mostly_unique` or `any`, unless they have the type '`io.state`'.

Params must be a valid list of zero or more try parameters.

The "then" part is mandatory. The "else" part is mandatory if *Goal* may fail; otherwise it must be omitted. There may be zero or more "catch" branches. The "catch_any" part is optional. *CatchAnyVar* must be a single variable.

The try parameter '`io`' takes a single argument, which must be the name of a state variable prefixed by '`!`'; for example, '`io(!IO)`'. The state variable must have the type '`io.state`', and be in scope of the try goal. The state variable is threaded through *Goal*, so it may perform I/O but cannot fail. If no '`io`' parameter exists, *Goal* may not perform I/O and may fail.

A try goal has determinism `cc_multi`.

On entering a try goal, *Goal* is executed. If it succeeds without throwing an exception, *ThenGoal* is executed. Any variables bound by *Goal* are visible in *ThenGoal* only. If *Goal* fails, then *ElseGoal* is executed.

If *Goal* throws an exception, the exception value is unified with each of the *Term*s in the "catch" branches in turn. On the first successful unification, the corresponding *CatchGoal* is executed (and other "catch" and "catch_any" branches ignored). Variables bound during the unification of the *Term* are in scope of the corresponding *CatchGoal*.

If the exception value does not unify with any of the terms in "catch" branches, and a "catch_any" branch is present, the exception is bound to *CatchAnyVar* and the *CatchAnyGoal* executed. *CatchAnyVar* is visible in the *CatchAnyGoal* only, and is existentially typed, i.e. it has type '`some [T] T`'.

Finally, if the thrown value did not unify with any "catch" term, and there is no "catch_any" branch, the exception is rethrown.

The declarative semantics of a try goal is:

```
(
    try [] Goal
    then Then
    else Else
    catch CP1 -> CG1
    catch CP2 -> CG2
    ...
    catch_any CAV -> CAG
)
<=>
(
    Goal, Then
;
    not Goal, Else
;
    some [Excp]
    ( if Excp = CP1 then
        CG1
    else if Excp = CP2 then
        CG2
    else if ...
        ...
    else
        Excp = CAV,
        CAG
    )
).
```

If no '`else`' branch is present, then '`Else = fail`'. If no '`catch_any`' branch is present, then '`CAG = fail`'.

An example of a try goal that performs I/O is:

```
:- pred p_carefully(io::di, io::uo) is cc_multi.

p_carefully(!IO) :-
    (try [io(!IO)] (
        io.write_string("Calling p\n", !IO),
        p(Output, !IO)
    )
    then
        io.write_string("p returned: ", !IO),
        io.write(Output, !IO),
        io.nl(!IO)
    catch S ->
        io.write_string("p threw a string: ", !IO),
        io.write_string(S, !IO),
        io.nl(!IO)
    catch 42 ->
```

```
        io.write_string("p threw 42\n", !IO)
    catch_any Other ->
        io.write_string("p threw something: ", !IO),
        io.write(Other, !IO),
        % Rethrow the value.
        throw(Other)
    ).
```

13 Semantics

A legal Mercury program is one that complies with the syntax, type, mode, determinism, and module system rules specified in earlier chapters. If a program does not comply with those rules, the compiler must report an error.

For each legal Mercury program, there is an associated predicate calculus theory whose language is specified by the type declarations in the program and whose axioms are the completion of the clauses for all predicates in the program, plus the usual equality axioms extended with the completion of the equations for all functions in the program, plus axioms corresponding to the mode-determinism assertions (see Chapter 6 [Determinism], page 49), plus axioms specifying the semantics of library predicates and functions. The declarative semantics of a legal Mercury program is specified by this theory.

Mercury implementations must be sound: the answers they compute must be true in every model of the theory. Mercury implementations are not required to be complete: they may fail to compute an answer in finite time, or they may exhaust the resource limitations of the execution environment, even though an answer is provable in the theory. However, there are certain minimum requirements that they must satisfy with respect to completeness.

There is an operational semantics of Mercury programs called the *strict sequential* operational semantics. In this semantics, the program is executed top-down, starting from '`main/2`' preceded by any module initialisation goals (as per Section 9.4 [Module initialisation], page 71), followed by any module finalisation goals (as per Section 9.5 [Module finalisation], page 72), and function calls within a goal, conjunctions and disjunctions are all executed in depth-first left-to-right order. Conjunctions and function calls are "minimally" reordered as required by the modes: the order is determined by selecting the first mode-correct sub-goal (conjunct or function call), executing that, then selecting the first of the remaining sub-goals which is now mode-correct, executing that, and so on. (There is no interleaving of different individual conjuncts or function calls, however; the sub-goals are reordered, not split and interleaved.) Function application is strict, not lazy.

Mercury implementations are required to provide a method of processing Mercury programs which is equivalent to the strict sequential operational semantics.

There is another operational semantics of Mercury programs called the *strict commutative* operational semantics. This semantics is equivalent to the strict sequential operational semantics except that there is no requirement that function calls, conjunctions and disjunctions be executed left-to-right; they may be executed in any order, and may even be interleaved. Furthermore, the order may even be different each time a particular goal is entered.

As well as providing the strict sequential operational semantics, Mercury implementations may optionally provide additional implementation-defined operational semantics, provided that any such implementation-defined operational semantics are at least as complete as the strict commutative operational semantics. An implementation-defined semantics is "at least as complete" as the strict commutative semantics if and only if the implementation-defined semantics guarantees to compute an answer in finite time for any program for which an answer would be computed in finite time for all possible executions under the strict commutative semantics (i.e. for all possible orderings of conjunctions and disjunctions).

Thus, to summarize, there are in fact a variety of different operational semantics for Mercury. In one of them, the strict sequential semantics, there is no nondeterminism —

the behaviour is always specified exactly. Programs are executed top-down using SLDNF (or something equivalent), mode analysis does "minimal" reordering (in a precisely defined sense), function calls, conjunctions and disjunctions are executed depth-first left-to-right, and function evaluation is strict. All implementations are required to support the strict sequential semantics, so that a program which works on one implementation using this semantics will be guaranteed to work on any other implementation. However, implementations are also allowed to support other operational semantics, which may have non-determinism, so long as they are sound with respect to the declarative semantics, and so long as they meet a minimum level of completeness (they must be at least as complete as the strict commutative semantics, in the sense that every program which terminates for all possible orderings must also terminate in any implementation-defined operational semantics).

This compromise allows Mercury to be used in several different ways. Programmers who care more about ease of programming and portability than about efficiency can use the strict sequential semantics, and can then be guaranteed that if their program works on one correct implementation, it will work on all correct implementations. Compiler implementors who want to write optimizing implementations that do lots of clever code reorderings and other high-level transformations or that want to offer parallelizing implementations which take maximum advantage of parallelism can define different semantic models. Programmers who care about efficiency more than portability can write code for these implementation-defined semantic models. Programmers who care about efficiency *and* portability can achieve this by writing code for the strict commutative semantics. Of course, this is not quite as easy as using the strict sequential semantics, since it is in general not sufficient to test your programs on just one implementation if you are to be sure that it will be able to use the maximally efficient operational semantics on any implementation. However, if you do write code which works for all possible executions under the strict commutative semantics (i.e. for all possible orderings of conjunctions and disjunctions), then you can be guaranteed that it will work correctly on every implementation, under every possible implementation-defined semantics.

The University of Melbourne Mercury implementation offers eight different semantics, which can be selected with different combinations of the '`--no-reorder-conj`', '`--no-reorder-disj`', and '`--no-fully-strict`' options. (The '`--no-fully-strict`' option allows the compiler to improve completeness by optimizing away infinite loops and goals with determinism `erroneous`.) The default semantics are the strict commutative semantics. Enabling '`--no-reorder-conj`' and '`--no-reorder-disj`' gives the strict sequential semantics.

Future implementations of Mercury may wish to offer other operational semantics. For example, they may wish to provide semantics in which function evaluation is lazy, rather than strict; semantics with a guaranteed fair search rule; and so forth.

14 Foreign language interface

This chapter documents the foreign language interface.

14.1 Calling foreign code from Mercury

Mercury procedures can be implemented using fragments of foreign language code using '`pragma foreign_proc`'.

14.1.1 pragma foreign_proc

A declaration of the form

```
:- pragma foreign_proc("Lang",
    Pred(Var1::Mode1, Var2::Mode2, ...),
    Attributes, Foreign_Code).
```

or

```
:- pragma foreign_proc("Lang",
    Func(Var1::Mode1, Var2::Mode2, ...) = (Var::Mode),
    Attributes, Foreign_Code).
```

means that any calls to the specified mode of *Pred* or *Func* will result in execution of the foreign code given in *Foreign_Code* written in language *Lang*, if *Lang* is selected as the foreign language code by this implementation. See the "Foreign Language Interface" chapter of the Mercury User's Guide, for more information about how the implementation selects the appropriate '`foreign_proc`' to use.

The foreign code fragment may refer to the specified variables (*Var1*, *Var2*, ..., and *Var*) directly by name. It is an error for a variable to occur more than once in the argument list. These variables will have foreign language types corresponding to their Mercury types, as determined by language and implementation specific rules.

All '`foreign_proc`' implementations are assumed to be impure. If they are actually pure or semipure, they must be explicitly promised as such by the user (either by using foreign language attributes specified below, or a '`promise_pure`' or '`promise_semipure`' pragma as specified in Chapter 15 [Impurity], page 129).

Additional restrictions on the foreign language interface code depend on the foreign language and compilation options. For more information, including the list of supported foreign languages and the strings used to identify them, see the language specific information in the "Foreign Language Interface" chapter of the Mercury User's Guide.

If there is a `pragma foreign_proc` declaration for any mode of a predicate or function, then there must be either a clause or a `pragma foreign_proc` declaration for every mode of that predicate or function.

Here's an example of code using '`pragma foreign_proc`': The following code defines a Mercury function '`sin/1`' which calls the C function '`sin()`' of the same name.

```
:- func sin(float) = float.
:- pragma foreign_proc("C",
    sin(X::in) = (Sin::out),
    [promise_pure, may_call_mercury],
"
    Sin = sin(X);
").
```

If the foreign language code does not recursively invoke Mercury code, as in the above example, then you can use '`will_not_call_mercury`' in place of '`may_call_mercury`' in the declarations above. This allows the compiler to use a slightly more efficient calling convention. (If you use this form, and the foreign code *does* invoke Mercury code, then the behaviour is undefined — your program may misbehave or crash.)

If there are both Mercury definitions and foreign_proc definitions for a procedure and/or foreign_proc definitions for different languages, it is implementation-defined which definition is used.

For pure and semipure procedures, the declarative semantics of the foreign_proc definitions must be the same as that of the Mercury code. The only thing that is allowed to differ is the efficiency (including the possibility of non-termination) and the order of solutions.

It is an error for a procedure with a '`pragma foreign_proc`' declaration to have a determinism of `multi` or `nondet`.

Since foreign_procs with the determinism `multi` or `nondet` cannot be defined directly, procedures with those determinisms that require foreign code in their implementation must be defined using a combination of Mercury clauses and (semi)deterministic foreign_procs. The following implementation for the standard library predicate '`string.append/3`' in the mode '`append(out, out, in) is multi`' illustrates this technique:

```
:- pred append(string, string, string).
:- mode append(out, out, in) is multi.

append(S1, S2, S3) :-
    S3Len = string.length(S3),
    append_2(0, S3Len, S1, S2, S3).

:- pred append_2(int::in, int::in, string::out, string::out, string::in) is multi.

append_2(NextS1Len, S3Len, S1, S2, S3) :-
    ( NextS1Len = S3Len ->
        append_3(NextS1Len, S3Len, S1, S2, S3)
    ;
        (
            append_3(NextS1Len, S3Len, S1, S2, S3)
        ;
            append_2(NextS1Len + 1, S3Len, S1, S2, S3)
        )
    ).
```

```
:- pred append_3(int::in, int::in, string::out, string::out, string::in) is det.

:- pragma foreign_proc("C",
    append_3(S1Len::in, S3Len::in, S1::out, S2::out, S3::in),
    [will_not_call_mercury, promise_pure],
"
    S1 = allocate_string(S1Len);   /* Allocate a new string of length S1Len */
    memcpy(S1, S3, S1Len);
    S1[S1Len] = '\\0';
    S2 = allocate_string(S2, S3Len - S1Len);
    strcpy(S2, S3Len + S1Len);
").
```

14.1.2 Foreign code attributes

As described above, '`pragma foreign_proc`' declarations may include a list of attributes describing properties of the given foreign function or code. All Mercury implementations must support the attributes listed below. They may also support additional attributes.

The attributes which must be supported by all implementations are as follows:

'`may_call_mercury`'/'`will_not_call_mercury`'
: This attribute declares whether or not execution inside this foreign language code may call back into Mercury or not. The default, in case neither is specified, is '`may_call_mercury`'. Specifying '`will_not_call_mercury`' may allow the compiler to generate more efficient code. If you specify '`will_not_call_mercury`', but the foreign language code *does* invoke Mercury code, then the behaviour is undefined.

'`promise_pure`'/'`promise_semipure`'
: This attribute promises that the purity of the given predicate or function definition is pure or semipure. It is equivalent to a corresponding '`pragma promise_pure`' or '`pragma promise_semipure`' declaration (see Chapter 15 [Impurity], page 129). If omitted, the clause specified by the '`foreign_proc`' is assumed to be impure.

'`thread_safe`'/'`not_thread_safe`'/'`maybe_thread_safe`'
: This attribute declares whether or not it is safe for multiple threads to execute this foreign language code concurrently. The default, in case none is specified, is '`not_thread_safe`'. If the foreign language code is declared '`thread_safe`', then the Mercury implementation is permitted to execute the code concurrently from multiple threads without taking any special precautions. If the foreign language code is declared '`not_thread_safe`', then the Mercury implementation must not invoke the code concurrently from multiple threads. If the Mercury implementation does use multithreading, then it must take appropriate steps to prevent this. (The multithreaded version of the University of Melbourne Mercury implementation protects '`not_thread_safe`' code using a mutex: C code that is not thread-safe has code inserted around it to obtain and release a mutex. All non-thread-safe foreign language code shares a single mutex.) If the

foreign language code is declared '`maybe_thread_safe`' then whether the code is considered '`thread_safe`' or '`not_thread_safe`' depends upon a compiler flag. This attribute is useful when the thread safety of the foreign code itself is conditional. The Melbourne Mercury compiler uses the '`--maybe-thread-safe`' option to set the value of the '`maybe_thread_safe`' attribute.

Additional attributes which are supported by the Melbourne Mercury compiler are as follows:

'`tabled_for_io`'
: This attribute should be attached to foreign procedures that do I/O. It tells the debugger to make calls to the foreign procedure idempotent. This allows the debugger to safely retry across such calls and also allows safe declarative debugging of code containing such calls. For more information, see the "I/O tabling" section of the Mercury User's Guide. If the foreign procedure contains gotos or static variables then the '`pragma no_inline`' directive should also be given. Note that currently I/O tabling will only be done for foreign procedures that take a pair of I/O state arguments. Impure foreign procedures that perform I/O will not be made idempotent, even if the tabled_for_io attribute is present. Note also that the tabled_for_io attribute will likely be replaced in a future release with a more general solution.

'`terminates`'/'`does_not_terminate`'
: This attribute specifies the termination properties of the given predicate or function definition. It is equivalent to the corresponding '`pragma terminates`' or '`pragma does_not_terminate`' declaration. If omitted, the termination property of the procedure is determined by the value of the '`may_call_mercury`'/'`will_not_call_mercury`' attribute. See Section 19.3 [Termination analysis], page 150 for more details.

'`will_not_throw_exception`'
: This attribute promises that the given predicate or function will not make calls back to Mercury that may result in an exception being thrown. It is an error to apply this attribute to procedures that have determinism `erroneous`. This attribute is ignored for code that is declared as not making calls back to Mercury via the '`will_not_call_mercury`' attribute. Note: predicates or functions that have polymorphic arguments but do not explicitly throw an exception, via a call to '`exception.throw/1`' or '`require.error/1`', may still throw exceptions because they may be called with arguments whose types have user-defined equality or comparison predicates. If these user-defined equality or comparison predicates throw exceptions then unifications or comparisons involving these types may also throw exceptions. As such, we recommend that only implementors of the Mercury system use this annotation for polymorphic predicates and functions.

'`will_not_modify_trail/may_modify_trail`'
: This attribute declares whether or not a foreign procedure modifies the trail (see Section 19.5 [Trailing], page 152). Specifying that a foreign procedure will not modify the trail may allow the compiler to generate more efficient code for that

procedure. In compilation grades that do not support trailing this attribute is ignored. The default, in case none is specified, is '`may_modify_trail`'.

'`will_not_call_mm_tabled/may_call_mm_tabled`'
: This attribute declares whether or not a foreign procedure makes calls back to Mercury procedures that are evaluated using minimal model tabling (see Section 19.2 [Tabled evaluation], page 146). Specifying that a foreign procedure will not call procedures evaluated using minimal model tabling may allow the compiler to generate more efficient code. In compilation grades that do not support minimal model tabling this attribute is ignored. These attributes may not be used with procedures that do not make calls back to Mercury, i.e. that have the '`will_not_call_mercury`' attribute. The default for foreign procedures that '`may_call_mercury`', in case none is specified, is '`may_call_mm_tabled`'.

'`affects_liveness/does_not_affect_liveness`'
: This attribute declares whether or not a foreign procedure uses and/or modifies any part of the Mercury virtual machine (registers, stack slots) through means other than its arguments. The '`affects_liveness`' attribute says that it does; The '`does_not_affect_liveness`' attribute says that it does not. In the absence of either attribute, the compiler assumes '`affects_liveness`', unless the code of the foreign_proc in question is empty.

'`may_duplicate/may_not_duplicate`'
: This attribute tells the compiler whether it is allowed to duplicate the foreign code fragment through optimizations such as inlining. The '`may_duplicate`' attribute says that it may; The '`may_not_duplicate`' attribute says that it may not. In the absence of either attribute, the compiler is allowed make its own judgement in the matter, based on factors such as the size of the code fragment.

14.2 Calling Mercury from foreign code

Mercury procedures may be exported so that they can be called by code written in a foreign language.

A declaration of the form:

```
:- pragma foreign_export("Lang",
    Pred(Mode1, Mode2, ...), "ForeignName").
```

or

```
:- pragma foreign_export("Lang",
    Func(Mode1, Mode2, ...) = Mode,
    "ForeignName").
```

exports a procedure for use by foreign language *Lang*. For each exported procedure the Mercury implementation will create an interface to the named Mercury procedure in the foreign language using the name *ForeignName*. The form of this interface is dependent upon the specified foreign language. For further details see the language specific information below.

It is an error to export a Mercury procedure that has a determinism of `multi` or `nondet`.

14.3 Data passing conventions

For each supported foreign language, we explain how to map a Mercury type to a type in that foreign language. We also map the Mercury parameter passing convention to the foreign language's parameter passing convention.

14.3.1 C data passing conventions

The Mercury primitive types are mapped to the following C types:

Mercury type	C type
`int`	`MR_Integer`
`int8`	`int8_t`
`int16`	`int16_t`
`int32`	`int32_t`
`int64`	`int64_t`
`uint`	`MR_Unsigned`
`uint8`	`uint8_t`
`uint16`	`uint16_t`
`uint32`	`uint32_t`
`uint64`	`uint64_t`
`float`	`MR_Float`
`char`	`MR_Char`
`string`	`MR_String`

In the current implementation, `MR_Integer` is a typedef for a signed integral type which is the same size as a pointer of type '`void *`'; `MR_Unsigned` is a typedef for an unsigned integral type which is the same size as a pointer of type '`void *`'; `MR_Float` is a typedef for `double` (unless the program and the Mercury library was compiled with '`--single-prec-float`', in which case it is a typedef for `float`); `MR_Char` is a typedef for a signed 32-bit integral type and `MR_String` is a typedef for '`char *`'.

Mercury variables of primitive type are passed to and from C as C variables of the corresponding C type.

For the Mercury standard library type '`bool.bool`', there is a corresponding C type, `MR_Bool`. C code can refer to the boolean data constructors '`yes`' and '`no`', as `MR_YES` and `MR_NO` respectively.

For the Mercury standard library type '`builtin.comparison_result`', there is a corresponding C type, `MR_Comparison_Result`. C code can refer to the data constructors of this type, '`(<)`', '`(=)`' and '`(>)`', as `MR_COMPARE_LESS`, `MR_COMPARE_EQUAL` and `MR_COMPARE_GREATER` respectively.

Mercury variables of a type for which there is a C '`pragma foreign_type`' declaration (see Section 14.4 [Using foreign types from Mercury], page 109) will be passed as the corresponding C type.

Mercury tuple types are passed as `MR_Tuple`, which in the current implementation is a typedef for a pointer of type '`void *`' if '`--high-level-code`' is enabled, and a typedef for `MR_Word` otherwise.

Mercury variables of any other type are passed as a `MR_Word`, which in the current implementation is a typedef for an unsigned type whose size is the same size as a pointer.

(Note: it would in fact be better for each Mercury type to map to a distinct abstract type in C, since that would be more type-safe, and thus we may change this in a future release. We advise programmers who are manipulating Mercury types in C code to use typedefs for each user-defined Mercury type, and to treat each such type as an abstract data type. This is good style and it will also minimize any compatibility problems if and when we do change this.)

Mercury lists can be manipulated by C code using the following macros, which are defined by the Mercury implementation.

```
MR_list_is_empty(list)     /* test if a list is empty */
MR_list_head(list)         /* get the head of a list */
MR_list_tail(list)         /* get the tail of a list */
MR_list_empty()            /* create an empty list */
MR_list_cons(head,tail)    /* construct a list with the given head and tail */
```

Note that the use of these macros is subject to some caveats (see Section 14.10.1.8 [Memory management for C], page 120).

The implementation provides the macro `MR_word_to_float` for converting a value of type `MR_Word` to one of type `MR_Float`, and the macro `MR_float_to_word` for converting a value of type `MR_Float` to one of type `MR_Word`. These macros must be used to perform these conversions since for some Mercury implementations '`sizeof(MR_Float)`' is greater than '`sizeof(MR_Word)`'.

The following fragment of C code illustrates the correct way to extract the head of a Mercury list of floats.

```
MR_Float f;
f = MR_word_to_float(MR_list_head(list));
```

Omitting the call to `MR_word_to_float` in the above example would yield incorrect results for implementations where '`sizeof(MR_Float)`' is greater than '`sizeof(MR_Word)`'.

Similarly, the implementation provides the macros `MR_word_to_int64` and `MR_word_to_uint64` for converting values of type `MR_Word` to ones of type `int64_t` or `uint64_t` respectively, and the macros `MR_int64_to_word` and `MR_uint64_to_word` for converting values of type `int64_t` or `uint64_t` respectively to ones of type `MR_Word`. These macros must be used to perform these conversions since for some Mercury implementations '`sizeof(int64_t)`' or '`sizeof(uint64_t)`' are greater than '`sizeof(MR_Word)`'.

14.3.2 C# data passing conventions

The Mercury primitive types are mapped to the following Common Language Infrastructure (CLI) and C# types:

Mercury type	**CLI type**	**C# type**
`int`	`System.Int32`	`int`
`int8`	`System.Int8`	`sbyte`
`int16`	`System.Int16`	`short`
`int32`	`System.Int32`	`int`
`int64`	`System.Int64`	`long`
`uint`	`System.UInt32`	`uint`

```
uint8          System.UInt8   byte
uint16         System.UInt16  ushort
uint32         System.UInt32  uint
uint64         System.UInt64  ulong
float          System.Double  double
char           System.Int32   int
string         System.String  string
```

Note that the Mercury type `char` is mapped like `int`; *not* to the CLI type `System.Char` because that only holds 16-bit numeric values.

For the Mercury standard library type '`bool.bool`', there is a corresponding C# type, `mr_bool.Bool_0`. C# code can refer to the boolean data constructors '`yes`' and '`no`', as `mr_bool.YES` and `mr_bool.NO` respectively.

For the Mercury standard library type '`builtin.comparison_result`', there is a corresponding C# type, `builtin.Comparison_result_0`. C# code can refer to the data constructors of this type, '`(<)`', '`(=)`' and '`(>)`', as `builtin.COMPARE_LESS`, `builtin.COMPARE_EQUAL` and `builtin.COMPARE_GREATER` respectively.

Mercury variables of a type for which there is a C# '`pragma foreign_type`' declaration (see Section 14.4 [Using foreign types from Mercury], page 109) will be passed as the corresponding C# type. Both reference and value types are supported.

Mercury tuple types are passed as '`object[]`' where the length of the array is the number of elements in the tuple.

Mercury variables whose type is a type variable will be passed as `System.Object`.

Mercury variables whose type is a Mercury discriminated union type will be passed as a CLI type whose type name is determined from the Mercury type name (ignoring any type parameters) followed by an underscore and then the type arity, expressed as a decimal integer. The first character of the type name will have its case inverted, and the name may be mangled to satisfy C# lexical rules.

For example, the following Mercury type corresponds to the C# class that follows (some implementation details elided):

```
:- type maybe(T)
    --->    yes(yes_field :: T)
    ;       no.

public static class Maybe_1 {
    public static class Yes_1 : Maybe_1 {
        public object yes_field;
        public Yes_1(object x) { ... }
    }
    public static class No_0 : Maybe_1 {
        public No_0() { ... }
    }
}
```

C# code generated by the Mercury compiler is placed in the '`mercury`' namespace. Mercury module qualifiers are converted into a C# class name by concatenating the components

with double underscore separators ('__'). For example the Mercury type 'foo.bar.baz/1' will be passed as the C# type 'mercury.foo__bar.Baz_1'.

Mercury array types are mapped to System.Array.

Mercury variables whose type is a Mercury equivalence type will be passed as the representation of the right hand side of the equivalence type.

This mapping is subject to change and you should try to avoid writing code that relies heavily upon a particular representation of Mercury terms.

Mercury arguments declared with input modes are passed by value to the C# function.

Arguments of type 'io.state' or 'store.store(_)' are not passed or returned at all. (The reason for this is that these types represent mutable state, and in C# modifications to mutable state are done via side effects, rather than argument passing.)

The handling of multiple output arguments is as follows.

If the Mercury procedure is deterministic and has no output arguments, then the return type of the C# function is 'void'; if it has one output argument, then the return value of the function is that output argument.

If the Mercury procedure is deterministic and has two or more output arguments, then the return type of the C# function is 'void'. At the position of each output argument, the C# function has an 'out' parameter.

If the Mercury procedure is semi-deterministic then the C# function returns a 'bool'. A 'true' return value denotes success and 'false' denotes failure. Output arguments are handled in the same way as multiple outputs for deterministic procedures, using 'out' parameters. On failure the values of the 'val' fields are undefined.

Mercury lists can be manipulated by C# code using the following methods, which are defined by the Mercury implementation.

```
bool      list.is_empty(List_1 list)     // test if a list is empty
object    list.det_head(List_1 list)     // get the head of a list
List_1    list.det_tail(List_1 list)     // get the tail of a list
List_1    list.empty_list()              // create an empty list
List_1    list.cons(object head, List_1 tail)
                                         // construct a list with
                                         //  the given head and tail
```

14.3.3 Java data passing conventions

The Mercury primitive types are mapped to the following Java types:

Mercury type	Java type
int	int
int8	byte
int16	short
int32	int
int64	long
uint	int
uint8	byte
uint16	short

```
uint32          int
uint64          long
float           double
char            int
string          java.lang.String
```

Note that since Java lacks unsigned integer types, Mercury's unsigned integer types correspond to signed integer types in Java.

Also, note that the Mercury type `char` is mapped like `int`; *not* to the Java type `char` because that only holds 16-bit numeric values.

For the Mercury standard library type '`bool.bool`', there is a corresponding Java type, `bool.Bool_0`. Java code can refer to the boolean data constructors '`yes`' and '`no`', as `bool.YES` and `bool.NO` respectively.

For the Mercury standard library type '`builtin.comparison_result`', there is a corresponding Java type, `builtin.Comparison_result_0`. Java code can refer to the data constructors of this type, '`(<)`', '`(=)`' and '`(>)`', as `builtin.COMPARE_LESS`, `builtin.COMPARE_EQUAL` and `builtin.COMPARE_GREATER` respectively.

Mercury variables of a type for which there is a Java '`pragma foreign_type`' declaration (see Section 14.4 [Using foreign types from Mercury], page 109) will be passed as the corresponding Java type.

Mercury tuple types are passed as `java.lang.Object[]` where the length of the array is the number of elements in the tuple.

Mercury variables whose types are universally quantified type variables will have generic types. Mercury variables whose types are existentially quantified type variables will be passed as `java.lang.Object`.

Mercury variables whose type is a Mercury discriminated union type will be passed as a Java type whose type name is determined from the Mercury type name (ignoring any type parameters) followed by an underscore and then the type arity, expressed as a decimal integer. The first character of the type name will have its case inverted, and the name may be mangled to satisfy Java lexical rules. Generics are used in the Java type for any type parameters.

For example, the following Mercury type corresponds to the Java class that follows (some implementation details elided):

```
:- type maybe(T)
    --->    yes(yes_field :: T)
    ;       no.

public static class Maybe_1<T> {
    public static class Yes_1<T> extends Maybe_1 {
        public T yes_field;
        public Yes_1(T x) { ... }
    }
    public static class No_0<T> extends Maybe_1 {
        public No_0() { ... }
    }
}
```

Java code generated by the Mercury compiler is placed in the 'jmercury' package. Mercury module qualifiers are converted into a Java class name by concatenating the components with double underscore separators ('__'). For example the Mercury type 'foo.bar.baz/1' will be passed as the Java type 'jmercury.foo__bar.Baz_1'.

Mercury array types are mapped to Java array types.

Mercury variables whose type is a Mercury equivalence type will be passed as the representation of the right hand side of the equivalence type.

This mapping is subject to change and you should try to avoid writing code that relies heavily upon a particular representation of Mercury terms.

Mercury arguments declared with input modes are passed by value to the corresponding Java function. If the Mercury procedure is a function whose result has an input mode, then the Mercury function result is appended to the list of input parameters, so that the Mercury function result becomes the last parameter to the corresponding Java function.

Arguments of type 'io.state' or 'store.store(_)' are not passed or returned at all. (The reason for this is that these types represent mutable state, and in Java modifications to mutable state are done via side effects, rather than argument passing.)

The handling of multiple output arguments is as follows.

If the Mercury procedure is deterministic and has no output arguments, then the return type of the Java function is void; if it has one output argument, then the return value of the function is that output argument.

If the Mercury procedure is deterministic and has two or more output arguments, then the return type of the Java function is void. At the position of each output argument, the Java function takes a value of the type 'jmercury.runtime.Ref<T>' where 'T' is the Java type corresponding to the type of the output argument. 'Ref' is a class with a single field 'val', which is assigned the output value when the function returns.

If the Mercury procedure is semi-deterministic then the Java function returns a 'boolean'. A 'true' return value denotes success and 'false' denotes failure. Output arguments are handled in the same way as multiple outputs for deterministic procedures, using the 'Ref' class. On failure the values of the 'val' fields are undefined.

Mercury lists can be manipulated by Java code using the following methods, which are defined by the Mercury implementation.

```
boolean   list.is_empty(List_1<E> list)     // test if a list is empty
E         list.det_head(List_1<E> list)     // get the head of a list
List_1<E> list.det_tail(List_1<E> list)     // get the tail of a list
List_1<E> list.empty_list()                 // create an empty list
<E, F extends E> List_1<E> list.cons(F head, List_1<E> tail)
                                            // construct a list with
                                            // the given head and tail
```

14.3.4 Erlang data passing conventions

The Mercury types int, float and char are mapped to Erlang integers, floats and integers respectively. A Mercury string is represented by an Erlang binary, not by a list of integers.

Mercury variables whose type is a Mercury discriminated union type will be passed as an Erlang tuple with the first element of the tuple being an Erlang atom named after the Mercury data constructor. For example, values of the type:

```
:- type maybe_int
    --->    yes(int)
    ;       no.
```

would be represented in Erlang as '`{yes, integer}`' and '`{no}`'.

Mercury variables whose type is a Mercury equivalence type will be passed as the representation of the right hand side of the equivalence type.

This mapping is subject to change and you should try to avoid writing code that relies heavily upon a particular representation of Mercury terms.

Arguments of dummy types, e.g. '`io.state`', are represented by the atom '`false`' when necessary. They are not passed to and from calls to monomorphic procedures.

Mercury arguments declared with input modes are passed by value to the corresponding Erlang function.

The result of an Erlang function depends on the determinism of the Mercury procedure that it was derived from. Procedures which succeed exactly once and have a single output variable return the single value directly. Procedures which succeed exactly once and have zero or two or more output variables return a tuple of those output variables in order.

Procedures which are semideterministic return, on success, a tuple of the variables with output modes (including when the number of output variables is one). On failure they return the Erlang atom '`fail`'.

Procedures which are nondeterministic take as a final argument a success continuation. This is an function which has an input variable for each variable of the Mercury procedure with an output mode. For each solution, the success continuation is called with the values of those output variables. When there are no more solutions the Erlang function returns with an undefined value.

14.4 Using foreign types from Mercury

Types defined in a foreign language can be accessed in Mercury using a declaration of the form

```
:- pragma foreign_type(Lang, MercuryTypeName, ForeignTypeDescriptor).
```

This defines *MercuryTypeName* as a synonym for type *ForeignTypeDescriptor* defined in the foreign language *Lang*. You must declare *MercuryTypeName* using a (possibly abstract) '`:- type`' declaration as usual. The '`pragma foreign_type`' must not have wider visibility than the type declaration (if the '`pragma foreign_type`' declaration is in the interface, the '`:- type`' declaration must be also).

ForeignTypeDescriptor defines how the Mercury type is mapped for a particular foreign language. Specific syntax is given in the language specific information below.

MercuryTypeName is treated as an abstract type at all times in Mercury code. However, if *MercuryTypeName* is one of the parameters of a foreign_proc for *Lang*, and the '`pragma foreign_type`' declaration is visible to the foreign_proc, it will be passed to that foreign_proc as specified by *ForeignTypeDescriptor*.

Multiple foreign language definitions may be given for the same type — the appropriate definition will be used for the appropriate language (see the language specific information below for details). All definitions must have the same visibility. A Mercury definition, which

must define a discriminated union type, may also be given. The constructors for the type will only be visible in Mercury clauses for predicates or functions with '`pragma foreign_proc`' clauses for all of the languages for which there are '`foreign_type`' declarations for the type.

You can also associate assertions about the properties of the foreign type with the '`foreign_type`' declaration, using the following syntax:

```
:- pragma foreign_type(Lang, MercuryTypeName, ForeignTypeDescriptor,
    [ForeignTypeAssertion, ...]).
```

Currently, three kinds of assertions are supported.

The '`can_pass_as_mercury_type`' assertion states that on the C backends, values of the given type can be passed to and from Mercury code without boxing, via simple casts, which is faster. This requires the type to be either an integer type or a pointer type, and requires it to be castable to '`MR_Word`' and back without loss of information (which means that its size may not be greater than the size of '`MR_Word`').

The '`word_aligned_pointer`' assertion implies '`can_pass_as_mercury_type`' and additionally states that values of the given type are pointer values clear in the tag bits. It allows the Mercury implementation to avoid boxing values of the given type when the type appears as the sole argument of a data constructor.

The '`stable`' assertion is meaningful only in the presence of the '`can_pass_as_mercury_type`' or '`word_aligned_pointer`' assertions. It states that either the C type is an integer type, or it is a pointer type pointing to memory that will never change. Together, these assertions are sufficient to allow tabling (see Section 19.2 [Tabled evaluation], page 146) and the '`compare_representation`' primitive to work on values of such types.

Violations of any of these assertions are very likely to result in the generated executable silently doing the wrong thing, giving no clue to where the problem might be. Since deciding whether a C type satisfies the conditions of these assertions requires knowledge of the internals of the Mercury implementation, we do not recommend the use of any of these assertions unless you are confident of your expertise in those internals.

As with discriminated union types, programmers can specify the unification and/or comparison predicates to use for values of the type using the following syntax (see Chapter 7 [User-defined equality and comparison], page 57):

```
:- pragma foreign_type(Lang, MercuryTypeName, ForeignTypeDescriptor)
        where equality is EqualityPred, comparison is ComparePred.
```

You can use Mercury foreign language interfacing declarations which specify language X to interface to types that are actually written in a different language Y provided that X and Y have compatible interface conventions. Support for this kind of compatibility is described in the language specific information below.

14.5 Using foreign enumerations in Mercury code

While a '`pragma foreign_type`' declaration imports a foreign *type* into Mercury, a '`pragma foreign_enum`' declaration imports *the values of the constants of an enumeration type* into Mercury.

While languages such as C have special syntax for defining enumeration types, in Mercury, an enumeration type is simply an ordinary discriminated union type whose function symbols all have arity zero.

Given an enumeration type such as

```
:- type unix_file_permissions
    --->    user_read
    ;       user_write
    ;       user_executable
    ;       group_read
    ;       group_write
    ;       group_executable
    ;       other_read
    ;       other_write
    ;       other_executable.
```

the values used to represent each constant are usually decided by the Mercury compiler. However, the values assigned this way may not match the values expected by foreign language code that uses values of the enumeration, and even if they happen to match, programmers probably would not want to *rely* on this coincidence.

This is why Mercury supports a mechanism that allows programmers to specify the representation of each constant in an enumeration type when generating code for a given target language. This mechanism is the '`pragma foreign_enum`' declaration, which looks like this:

```
:- pragma foreign_enum("C", unix_file_permissions/0,
[
    user_read        - "S_IRUSR",
    user_write       - "S_IWUSR",
    user_executable  - "S_IXUSR",
    group_read       - "S_IRGRP",
    group_write      - "S_IWGRP",
    group_executable - "S_IXGRP",
    other_read       - "S_IROTH",
    other_write      - "S_IWOTH",
    other_executable - "S_IXOTH"
]).
```

(Unix systems have a standard header file that defines each of '`S_IRUSR`', ..., '`S_IXOTH`' as macros that each expand to an integer constant; these constants happen *not* to be the ones that the Mercury compiler would assign to those constants.)

The general form of '`pragma foreign_enum`' declarations is

```
:- pragma foreign_enum("Lang", MercuryType, CtorValues).
```

where *CtorValues* is a list of pairs of the form:

```
[
    ctor_0 - "ForeignValue_0",
    ctor_1 - "ForeignValue_1",
    ...
    ctor_N - "ForeignValue_N"
]
```

The first element of each pair is a constant (function symbol of arity 0) of the type *MercuryType*, and the second is either a numeric or a symbolic name for the integer value in the language *Lang* that the programmer wants to be used to represent that constructor.

The mapping defined by this list of pairs must form a bijection, i.e. the list must map distinct constructors to distinct values, and vice versa. The Mercury compiler is not required to check this, because it cannot; even if two symbolic names (such as C macros) are distinct, they may expand to the same integer in the target language.

Mercury implementations may impose further foreign-language-specific restrictions on the form that values used to represent enumeration constructors may take. See the language specific information below for details.

It is an error for any given *MercuryType* to be the subject of more than one '`pragma foreign_enum`' declaration for any given foreign language, since that would amount to an attempt to specify two or more (probably) conflicting representations for each of the type's function symbols.

A '`pragma foreign_enum`' declaration must occur in the implementation section of the module that defines the type *MercuryType*.

Note that the default comparison for types that are the subject of a '`pragma foreign_enum`' declaration will be defined by the foreign values, rather than the order of the constructors in the type declaration (as would otherwise be the case).

14.6 Using Mercury enumerations in foreign code

A '`pragma foreign_enum`' declaration imports the values of the constants of an enumeration type into Mercury. However, sometimes one needs the reverse: the ability to *export* the values of the constants of an enumeration type (whether those values were assigned by '`foreign_enum`' pragmas or not) from Mercury to foreign language code in '`foreign_proc`' and '`foreign_code`' pragmas. This is what '`pragma foreign_export_enum`' declarations are for.

These pragmas have the following general form:

```
:- pragma foreign_export_enum("Lang", MercuryType,
    Attributes, Overrides).
```

When given such a pragma, the compiler will define a symbolic name in language *Lang* for each of the constructors of *MercuryType* (which must be an enumeration type). Each symbolic name allows code in that foreign language to create a value corresponding to that of the constructor it represents. (The exact mechanism used depends upon the foreign language; see the language specific information below for further details.)

For each foreign language, there is a default mapping between the name of a Mercury constructor and its symbolic name in the language *Lang*. This default mapping is not required to map every valid constructor name to a valid name in language *Lang*; where it does not, the programmer must specify a valid symbolic name. The programmer may also choose to map a constructor to a symbolic name that differs from the one supplied by the default mapping for language *Lang*. *Overrides* is a list whose elements are pairs of constructor names and strings. The latter specify the name that the implementation should use as the symbolic name in the foreign language. *Overrides* has the following form:

```
[cons_I - "symbol_I", ..., cons_J - "symbol_J"]
```

This can be used to provide either a valid symbolic name where the default mapping does not, or to override a valid symbolic name generated by the default mapping. This argument may be omitted if *Overrides* is empty.

The argument *Attributes* is a list of optional attributes. If empty, it may be omitted from the '`pragma foreign_export_enum`' declaration if the *Overrides* argument is also omitted. The following attributes must be supported by all Mercury implementations.

'`prefix(`*Prefix*`)`'
: Prefix each symbolic name, regardless of how it was generated, with the string *Prefix*. A '`pragma foreign_export_enum`' declaration may contain at most one '`prefix`' attribute.

'`uppercase`'
: Convert any alphabetic characters in a Mercury constructor name to uppercase when generating the symbolic name using the default mapping. Symbolic names specified by the programmer using *Overrides* are not affected by this attribute. If the '`prefix`' attribute is also specified, then the prefix is added to the symbolic name *after* the conversion to uppercase has been performed, i.e. the characters in the prefix are not affected by the '`uppercase`' attribute.

The implementation does not check the validity of a symbolic name in the foreign language until after the effects of any attributes have been applied. This means that attributes may cause an otherwise valid symbolic name to become invalid, or vice versa.

A Mercury module may contain '`pragma foreign_export_enum`' declarations that refer to imported types, subject to the usual visibility restrictions.

A Mercury module, or program, may contain more than one '`pragma foreign_export_enum`' declaration for a given Mercury type for a given language. This can be useful when a project is transitioning from using one naming scheme for Mercury constants in foreign code to another naming scheme.

It is an error if the mapping between constructors and symbolic names in a '`pragma foreign_export_enum`' declaration does not form a bijection. It is also an error if two separate '`pragma foreign_export_enum`' declarations for a given foreign language, *whether or not for the same type*, specify the same symbolic name, since in that case, the Mercury compiler would generate two conflicting definitions for that symbolic name. However, the Mercury implementation is not required to check either condition.

A '`pragma foreign_export_enum`' declaration may occur only in the implementation section of a module.

14.7 Adding foreign declarations

Foreign language declarations (such as type declarations, header file inclusions or macro definitions) can be included in the Mercury source file as part of a '`foreign_decl`' declaration of the form

```
:- pragma foreign_decl("Lang", DeclCode).
```

This declaration will have effects equivalent to including the specified *DeclCode* in an automatically generated source file of the specified programming language, in a place appropriate for declarations, and linking that source file with the Mercury program (after having compiled it with a compiler for the specified programming language, if appropriate).

Entities declared in '`pragma foreign_decl`' declarations are visible in '`pragma foreign_code`', '`pragma foreign_type`', '`pragma foreign_proc`', and '`pragma foreign_enum`' declarations that specify the same foreign language and occur in the same Mercury module.

By default, the contents of '`pragma foreign_decl`' declarations are also visible in the same kinds of declarations in other modules that import the module containing the '`pragma foreign_decl`' declaration. This is because they may be required to make sense of types defined using '`pragma foreign_type`' and/or predicates defined using '`pragma foreign_proc`' in the containing module, and these may be visible in other modules, especially in the presence of intermodule optimization,

If you do not want the contents of a '`pragma foreign_decl`' declaration to be visible in foreign language code in other modules, you can use the following variant of the declaration:

```
:- pragma foreign_decl("Lang", local, DeclCode).
```

Note: currently only the C and Erlang backends support this variant of the '`pragma foreign_decl`' declaration.

The Melbourne Mercury implementation additionally supports the forms

```
:- pragma foreign_decl("Lang", include_file("Path")).
:- pragma foreign_decl("Lang", local, include_file("Path")).
```

These have the same effects as the standard forms except that the contents of the file referenced by *Path* are included in place of the string literal in the last argument, without further interpretation. *Path* may be an absolute path to a file, or a path to a file relative to the directory that contains the source file of the module containing the declaration. The interpretation of the path is platform-dependent. If the filesystem uses a different character set or encoding from the Mercury source file (which must be UTF-8), the file may not be found.

'`mmc --make`' and '`mmake`' treat included files as dependencies of the module.

14.8 Declaring Mercury exports to other modules

The declarations for Mercury predicates or functions exported to a foreign language using a '`pragma foreign_export`' declaration are visible to foreign code in a '`pragma foreign_code`' or '`pragma foreign_proc`' declaration of the same module and also in those of any sub-modules. They are not visible to the foreign code in '`pragma foreign_code`' or '`pragma foreign_proc`' declarations in any other module. They can be made visible using a declaration of the form:

```
:- pragma foreign_import_module("Lang", ImportedModule).
```

where *ImportedModule* is the name of the module containing the '`pragma foreign_export`' declarations.

If *Lang* is `"C"` this is equivalent to

```
:- pragma foreign_decl("C", "#include ""ImportedModule.mh""").
```

where '*ImportedModule*`.mh`' is the automatically generated header file containing the C declarations for the predicates and functions exported to C.

'`pragma foreign_import_module`' should be used instead of the explicit `#include` because '`pragma foreign_import_module`' tells the implementation that

‘*ImportedModule*.mh’ must be built before the object file for the module containing the ‘pragma foreign_import_module’ declaration.

A cycle of ‘pragma foreign_import_module’, where the language is ‘"C#"’ or ‘"Java"’, is not permitted.

Note that the Melbourne Mercury implementation often implicitly inserts ‘pragma foreign_import_module’ declarations but programmers should *not* write code that depends upon this behaviour; ‘pragma foreign_import_module’ declarations should always be explicitly included if needed.

14.9 Adding foreign definitions

Definitions of foreign language entities (such as functions or global variables) may be included using a declaration of the form

```
:- pragma foreign_code("Lang", Code).
```

This declaration will have effects equivalent to including the specified *Code* in an automatically generated source file of the specified programming language, in a place appropriate for definitions, and linking that source file with the Mercury program (after having compiled it with a compiler for the specified programming language, if appropriate).

Entities declared in ‘pragma foreign_code’ declarations are visible in ‘pragma foreign_proc’ declarations that specify the same foreign language and occur in the same Mercury module.

The Melbourne Mercury implementation additionally supports the form

```
:- pragma foreign_code("Lang", include_file("Path")).
```

This has the same effect as the standard form except that the contents of the file referenced by *Path* are included in place of the string literal in the last argument, without further interpretation. *Path* may be an absolute path to a file, or a path to a file relative to the directory that contains the source file of the module containing the declaration. The interpretation of the path is platform-dependent. If the filesystem uses a different character set or encoding from the Mercury source file (which must be UTF-8), the file may not be found.

‘mmc --make’ and ‘mmake’ treat included files as dependencies of the module.

14.10 Language specific bindings

All Mercury implementations should support interfacing with C. The set of other languages supported is implementation-defined. A suitable compiler or assembler for the foreign language must be available on the system.

The University of Melbourne Mercury implementation supports interfacing with the following languages:

‘C’ Use the string "C" to set the foreign language to C.

‘C#’ Use the string "C#" to set the foreign language to C#.

‘Java’ Use the string "Java" to set the foreign language to Java.

‘Erlang’ Use the string "Erlang" to set the foreign language to Erlang.

14.10.1 Interfacing with C

14.10.1.1 Using pragma foreign_type for C

A C '`pragma foreign_type`' declaration has the form:

```
:- pragma foreign_type("C", MercuryTypeName, "CForeignType").
```

For example,

```
:- pragma foreign_type("C", long_double, "long double").
```

The *CForeignType* can be any C type name that obeys the following restrictions. Function types, array types, and incomplete types are not allowed. The type name must be such that when declaring a variable in C of that type, that no part of the type name is required after the variable name. (This rule prohibits, for example, function pointer types such as '`void (*)(void)`'. However, it would be OK to use a typedef name which was defined as a function pointer type.)

C preprocessor directives (such as '`#if`') may not be used in *CForeignType*. (You can however use a typedef name that refers to a type defined in a '`pragma foreign_decl`' declaration, and the '`pragma foreign_decl`' declaration may contain C preprocessor directives.)

If the *MercuryTypeName* is the type of a parameter of a procedure defined using '`pragma foreign_proc`', it will be passed to the foreign_proc's foreign language code as *CForeignType*.

Furthermore, any Mercury procedure exported with '`pragma foreign_export`' will use *CForeignType* as the type for any parameters whose Mercury type is *MercuryTypeName*.

The builtin Mercury type `c_pointer` may be used to pass C pointers between C functions which are called from Mercury. For example:

```
:- module pointer_example.
:- interface.

:- type complicated_c_structure.

% Initialise the abstract C structure that we pass around in Mercury.
:- pred initialise_complicated_structure(complicated_c_structure::uo) is det.

% Perform a calculation on the C structure.
:- pred do_calculation(int::in, complicated_c_structure::di,
        complicated_c_structure::uo) is det.

:- implementation.

% Our C structure is implemented as a c_pointer.
:- type complicated_c_structure
    --->    complicated_c_structure(c_pointer).

:- pragma foreign_decl("C",
   extern struct foo *init_struct(void);
   extern struct foo *perform_calculation(int, struct foo *);
");
```

```
:- pragma foreign_proc("C",
    initialise_complicated_structure(Structure::uo),
    [will_not_call_mercury, may_call_mercury],
"
    Structure = init_struct();
").

:- pragma foreign_proc("C",
    do_calculation(Value::in, Structure0::di, Structure::uo),
    [will_not_call_mercury, may_call_mercury],
"
    Structure = perform_calculation(Value, Structure0);
").
```

We strongly recommend the use of '`pragma foreign_type`' instead of `c_pointer` as the use of '`pragma foreign_type`' results in more type-safe code.

14.10.1.2 Using pragma foreign_enum for C

Foreign enumeration values in C must be constants of type `MR_Integer`. They may be specified as either integer literals or via preprocessor macros that expand to integer literals.

14.10.1.3 Using pragma foreign_export_enum for C

For C the symbolic names generated by a '`pragma foreign_export_enum`' must form valid C identifiers. These identifiers are used as the names of preprocessor macros. The body of each of these macros expands to a value that is identical to that of the constructor to which the symbolic name corresponds in the mapping established by the '`pragma foreign_export_enum`' declaration.

As noted in the Section 14.3.1 [C data passing conventions], page 103, the type of these values is `MR_Word`.

The default mapping used by '`pragma foreign_export_enum`' declarations for C is to use the Mercury constructor name as the base of the symbolic name. For example, the symbolic name for the Mercury constructor '`foo`' would be `foo`.

14.10.1.4 Using pragma foreign_proc for C

The input and output variables will have C types corresponding to their Mercury types, as determined by the rules specified in Section 14.3.1 [C data passing conventions], page 103.

The C code fragment may declare local variables, up to a total size of 10kB for the procedure. If a procedure requires more than this for its local variables, the code can be moved into a separate function (defined in a '`pragma foreign_code`' declaration, for example).

The C code fragment should not declare any labels or static variables unless there is also a '`pragma no_inline`' declaration or a '`may_not_duplicate`' foreign code attribute for the procedure. The reason for this is that otherwise the Mercury implementation may inline the procedure by duplicating the C code fragment for each call. If the C code fragment declared a static variable, inlining it in this way could result in the program having multiple

instances of the static variable, rather than a single shared instance. If the C code fragment declared a label, inlining it in this way could result in an error due to the same label being defined twice inside a single C function.

C code in a `pragma foreign_proc` declaration for any procedure whose determinism indicates that it can fail must assign a truth value to the macro `SUCCESS_INDICATOR`. For example:

```
:- pred string.contains_char(string, character).
:- mode string.contains_char(in, in) is semidet.

:- pragma foreign_proc("C",
    string.contains_char(Str::in, Ch::in),
    [will_not_call_mercury, promise_pure],
"
    SUCCESS_INDICATOR = (strchr(Str, Ch) != NULL);
").
```

`SUCCESS_INDICATOR` should not be used other than as the target of an assignment. (For example, it may be `#define`d to a register, so you should not try to take its address.) Procedures whose determinism indicates that they cannot fail should not access `SUCCESS_INDICATOR`.

Arguments whose mode is input will have their values set by the Mercury implementation on entry to the C code. If the procedure succeeds, the C code must set the values of all output arguments. If the procedure fails, the C code need only set `SUCCESS_INDICATOR` to false (zero).

The behaviour of a procedure defined using a '`pragma foreign_proc`' declaration whose body contains a `return` statement is undefined.

14.10.1.5 Using pragma foreign_export for C

A '`pragma foreign_export`' declaration for C has the form:

```
:- pragma foreign_export("C", MercuryMode, "C_Name").
```

For example,

```
:- pragma foreign_export("C", foo(in, in, out), "FOO").
```

For each Mercury module containing '`pragma foreign_export`' declarations for C, the Mercury implementation will automatically create a header file for that module which declares a C function *C_Name*() for each of the '`pragma foreign_export`' declarations. Each such C function is the C interface to the specified Mercury procedure.

The type signature of the C interface to a Mercury procedure is determined as follows. Mercury types are converted to C types according to the rules in Section 14.3.1 [C data passing conventions], page 103. Input arguments are passed by value. For output arguments, the caller must pass the address in which to store the result. If the Mercury procedure can fail, then its C interface function returns a truth value indicating success or failure. If the Mercury procedure is a Mercury function that cannot fail, and the function result has an output mode, then the C interface function will return the Mercury function result value. Otherwise the function result is appended as an extra argument. Arguments of type '`io.state`' or '`store.store(_)`' are not passed at all. (The reason for this is that these

types represent mutable state, and in C modifications to mutable state are done via side effects, rather than argument passing.)

Calling polymorphically typed Mercury procedures from C is a little bit more difficult than calling ordinary (monomorphically typed) Mercury procedures. The simplest method is to just create monomorphic forwarding procedures that call the polymorphic procedures, and export them, rather than exporting the polymorphic procedures.

If you do export a polymorphically typed Mercury procedure, the compiler will prepend one '`type_info`' argument to the parameter list of the C interface function for each distinct type variable in the Mercury procedure's type signature. The caller must arrange to pass in appropriate '`type_info`' values corresponding to the types of the other arguments passed. These '`type_info`' arguments can be obtained using the Mercury '`type_of`' function in the Mercury standard library module '`type_desc`'.

To use the C declarations produced see Section 14.10.1.6 [Using pragma foreign_decl for C], page 119.

Throwing an exception across the C interface is not supported. That is, if a Mercury procedure that is exported to C using '`pragma foreign_export`' throws an exception which is not caught within that procedure, then you will get undefined behaviour.

14.10.1.6 Using pragma foreign_decl for C

Any macros, function prototypes, or other C declarations that are used in '`foreign_code`', '`foreign_type`' or '`foreign_proc`' pragmas must be included using a '`foreign_decl`' declaration of the form

```
:- pragma foreign_decl("C", HeaderCode).
```

HeaderCode can be a C '`#include`' line, for example

```
:- pragma foreign_decl("C", "#include <math.h>")
```

or

```
:- pragma foreign_decl("C", "#include ""tcl.h""").
```

or it may contain any C declarations, for example

```
:- pragma foreign_decl("C", "
        extern int errno;
        #define SIZE 200
        struct Employee {
                char name[SIZE];
        };
        extern int bar;
        extern void foo(void);
").
```

Mercury automatically includes certain headers such as `<stdlib.h>`, but you should not rely on this, as the set of headers which Mercury automatically includes is subject to change.

If a Mercury predicate or function exported using a '`pragma foreign_export`' declaration is to be used within a '`:- pragma foreign_code`' or '`:- pragma foreign_proc`' declaration the header file for the module containing the '`pragma foreign_export`' declaration should be included using a '`pragma foreign_import_module`' declaration, for example

```
:- pragma foreign_import_module("C", exporting_module).
```

14.10.1.7 Using pragma foreign_code for C

Definitions of C functions or global variables may be included using a declaration of the form

```
:- pragma foreign_code("C", Code).
```

For example,

```
:- pragma foreign_code("C", "
        int bar = 42;
        void foo(void) {}
").
```

Such code is copied verbatim into the generated C file.

14.10.1.8 Memory management for C

Passing pointers to dynamically-allocated memory from Mercury to code written in other languages, or vice versa, is in general implementation-dependent.

The current Mercury implementation supports two different methods of memory management: conservative garbage collection, or no garbage collection. (With the latter method, heap storage is reclaimed only on backtracking.)

Conservative garbage collection makes inter-language calls simplest. When using conservative garbage collection, heap storage is reclaimed automatically. Pointers to dynamically-allocated memory can be passed to and from C without taking any special precautions.

When using no garbage collection, you must be careful not to retain pointers to memory on the Mercury heap after Mercury has backtracked to before the point where that memory was allocated. You must also avoid the use of the macros `MR_list_empty()` and `MR_list_cons()`. (The reason for this is that they may access Mercury's '`MR_hp`' register, which might not be valid in C code. Using them in the bodies of procedures defined using '`pragma foreign_proc`' with '`will_not_call_mercury`' would probably work, but we don't advise it.) Instead, you can write Mercury functions to perform these actions and use '`pragma foreign_export`' to access them from C. This alternative method also works with conservative garbage collection.

Future Mercury implementations may use non-conservative methods of garbage collection. For such implementations, it will be necessary to explicitly register pointers passed to C with the garbage collector. The mechanism for doing this has not yet been decided on. It would be desirable to provide a single memory management interface for use when interfacing with other languages that can work for all methods of memory management, but more implementation experience is needed before we can formulate such an interface.

14.10.1.9 Linking with C object files

A Mercury implementation should allow you to link with object files or libraries that were produced by compiling C code. The exact mechanism for linking with C object files is implementation-dependent. The following text describes how it is done for the University of Melbourne Mercury implementation.

To link an existing object file or archive of object files into your Mercury code, use the command line option '`--link-object`'. For example, the following will link the object file '`my_function.o`' from the current directory when compiling the program '`prog`':

```
mmc --link-object my_functions.o prog
```

The command line option '`--library`' (or '`-l`' for short) can be used to link an existing library into your Mercury code. For example, the following will link the library file '`libfancy_library.a`', or perhaps the shared version '`libfancy_library.so`', from the directory '`/usr/local/contrib/lib`', when compiling the program '`prog`':

```
mmc -R/usr/local/contrib/lib -L/usr/local/contrib/lib -lfancy_library prog
```

As illustrated by the example, the command line options '`-R`', '`-L`' and '`-l`', have the same meaning as they do with the Unix linker.

For more information, see the "Libraries" chapter of the Mercury User's Guide.

14.10.2 Interfacing with C#

14.10.2.1 Using pragma foreign_type for C#

A C# '`pragma foreign_type`' declaration has the form:

```
:- pragma foreign_type("C#", MercuryTypeName, "C#-Type").
```

The *C#-Type* can be any accessible C# type.

The effect of this declaration is that Mercury values of type *MercuryTypeName* will be passed to and from C# foreign_procs as having type *C#-Type*.

Furthermore, any Mercury procedure exported with '`pragma foreign_export`' will use *C#-Type* as the type for any parameters whose Mercury type is *MercuryTypeName*.

14.10.2.2 Using pragma foreign_enum for C#

Foreign enumeration values in C# must be a constant value expression which is a valid initializer within an enumeration of underlying type `int`.

14.10.2.3 Using pragma foreign_export_enum for C#

For C# the symbolic names generated by a '`pragma foreign_export_enum`' must form valid C# identifiers. These identifiers are used as the names of static class members.

The default mapping used by '`pragma foreign_export_enum`' declarations for C# is to use the Mercury constructor name as the base of the symbolic name. For example, the symbolic name for the Mercury constructor '`foo`' would be `foo`.

14.10.2.4 Using pragma foreign_proc for C#

The C# code from C# '`pragma foreign_proc`' declarations will be placed in the bodies of static member functions of an automatically generated C# class. Since such C# code will become part of a static member function, it must not refer to the `this` keyword. It may however refer to static member variables or static member functions declared with '`pragma foreign_code`'.

The input and output variables for a C# '`pragma foreign_proc`' will have C# types corresponding to their Mercury types. The exact rules for mapping Mercury types to C# types are described in Section 14.3.2 [C# data passing conventions], page 104.

C# code in a `pragma foreign_proc` declaration for any procedure whose determinism indicates that it can fail must assign a value of type `bool` to the variable `SUCCESS_INDICATOR`. For example:

```
:- pred string.contains_char(string, character).
:- mode string.contains_char(in, in) is semidet.

:- pragma foreign_proc("C#",
    string.contains_char(Str::in, Ch::in),
    [will_not_call_mercury, promise_pure],
"
    SUCCESS_INDICATOR = (Str.IndexOf(Ch) != -1);
").
```

C# code for procedures whose determinism indicates that they cannot fail should not access `SUCCESS_INDICATOR`.

Arguments whose mode is input will have their values set by the Mercury implementation on entry to the C# code. If the procedure succeeds, the C# code must set the values of all output arguments. If the procedure fails, the C# code need only set `SUCCESS_INDICATOR` to false.

14.10.2.5 Using pragma foreign_export for C#

A '`pragma foreign_export`' declaration for C# has the form:

```
:- pragma foreign_export("C#", MercuryMode, "C#_Name").
```

For example,

```
:- pragma foreign_export("C#", foo(in, in, out), "FOO").
```

The type signature of the C# interface to a Mercury procedure is as described in Section 14.3.2 [C# data passing conventions], page 104.

Calling polymorphically typed Mercury procedures from C# is a little bit more difficult than calling ordinary (monomorphically typed) Mercury procedures. The simplest method is to just create monomorphic forwarding procedures that call the polymorphic procedures, and export them, rather than exporting the polymorphic procedures.

If you do export a polymorphically typed Mercury procedure, the compiler will prepend one '`type_info`' argument to the parameter list of the C# interface function for each distinct type variable in the Mercury procedure's type signature. The caller must arrange to pass in appropriate '`type_info`' values corresponding to the types of the other arguments passed. These '`type_info`' arguments can be obtained using the Mercury '`type_of`' function in the Mercury standard library module '`type_desc`'.

14.10.2.6 Using pragma foreign_decl for C#

'`pragma foreign_decl`' declarations for C# can be used to provide any top-level C# declarations (e.g. '`using`' declarations or auxiliary class definitions) which are needed by C# code in '`pragma foreign_proc`' declarations in that module.

For example:

```
:- pragma foreign_decl("C#", "
using System;
```

```
").
:- pred hello(io.state::di, io.state::uo) is det.
:- pragma foreign_proc("C#",
     hello(_IO0::di, _IO::uo),
     [will_not_call_mercury, promise_pure],
"
    // here we can refer directly to Console rather than System.Console
    Console.WriteLine(""hello world"");
").
```

14.10.2.7 Using pragma foreign_code for C#

The C# code from '`pragma foreign_proc`' declarations for C# will be placed in the bodies of static member functions of an automatically generated C# class. '`pragma foreign_code`' can be used to define additional members of this automatically generated class, which can then be referenced by '`pragma foreign_proc`' declarations for C# from that module.

For example:

```
:- pragma foreign_code("C#", "
    static int counter = 0;
").

:- impure pred incr_counter is det.
:- pragma foreign_proc("C#",
    incr_counter,
    [will_not_call_mercury], "
    counter++;
").

:- semipure func get_counter = int.
:- pragma foreign_proc("C#",
    get_counter = (Result::out),
    [will_not_call_mercury, promise_semipure],
"
    Result = counter;
").
```

14.10.3 Interfacing with Java

14.10.3.1 Using pragma foreign_type for Java

A Java '`pragma foreign_type`' declaration has the form:

```
:- pragma foreign_type("Java", MercuryTypeName, "JavaType").
```

The *JavaType* can be any accessible Java type.

The effect of this declaration is that Mercury values of type *MercuryTypeName* will be passed to and from Java foreign_procs as having type *JavaType*.

Furthermore, any Mercury procedure exported with '`pragma foreign_export`' will use *JavaType* as the type for any parameters whose Mercury type is *MercuryTypeName*.

14.10.3.2 Using pragma foreign_enum for Java

'`pragma foreign_enum`' is currently not supported for Java.

14.10.3.3 Using pragma foreign_export_enum for Java

For Java the symbolic names generated by a '`pragma foreign_export_enum`' must form valid Java identifiers. These identifiers are used as the names of static class members which are assigned instances of the enumeration class.

The `equals` method should be used for equality testing of enumeration values in Java code.

The default mapping used by '`pragma foreign_export_enum`' declarations for Java is to use the Mercury constructor name as the base of the symbolic name. For example, the symbolic name for the Mercury constructor '`foo`' would be `foo`.

14.10.3.4 Using pragma foreign_proc for Java

The Java code from Java '`pragma foreign_proc`' declarations will be placed in the bodies of static member functions of an automatically generated Java class. Since such Java code will become part of a static member function, it must not refer to the `this` keyword. It may however refer to static member variables or static member functions declared with '`pragma foreign_code`'.

The input and output variables for a Java '`pragma foreign_proc`' will have Java types corresponding to their Mercury types. The exact rules for mapping Mercury types to Java types are described in Section 14.3.3 [Java data passing conventions], page 106.

The Java code in a `pragma foreign_proc` declaration for a procedure whose determinism indicates that it can fail must assign a value of type `boolean` to the variable `SUCCESS_INDICATOR`. For example:

```
:- pred string.contains_char(string, character).
:- mode string.contains_char(in, in) is semidet.

:- pragma foreign_proc("Java",
    string.contains_char(Str::in, Ch::in),
    [will_not_call_mercury, promise_pure],
"
    SUCCESS_INDICATOR = (Str.IndexOf(Ch) != -1);
").
```

Java code for procedures whose determinism indicates that they cannot fail should not refer to the `SUCCESS_INDICATOR` variable.

Arguments whose mode is input will have their values set by the Mercury implementation on entry to the Java code. With our current implementation, the Java code must set the values of all output variables, even if the procedure fails (i.e. sets the `SUCCESS_INDICATOR` variable to `false`).

14.10.3.5 Using pragma foreign_export for Java

A '`pragma foreign_export`' declaration for Java has the form:

```
:- pragma foreign_export("Java", MercuryMode, "Java_Name").
```

For example,

```
:- pragma foreign_export("Java", foo(in, in, out), "FOO").
```

The type signature of the Java interface to a Mercury procedure is as described in Section 14.3.3 [Java data passing conventions], page 106.

Calling polymorphically typed Mercury procedures from Java is a little bit more difficult than calling ordinary (monomorphically typed) Mercury procedures. The simplest method is to just create monomorphic forwarding procedures that call the polymorphic procedures, and export them, rather than exporting the polymorphic procedures.

If you do export a polymorphically typed Mercury procedure, the compiler will prepend one '`type_info`' argument to the parameter list of the Java interface function for each distinct type variable in the Mercury procedure's type signature. The caller must arrange to pass in appropriate '`type_info`' values corresponding to the types of the other arguments passed. These '`type_info`' arguments can be obtained using the Mercury '`type_of`' function in the Mercury standard library module '`type_desc`'.

14.10.3.6 Using pragma foreign_decl for Java

'`pragma foreign_decl`' declarations for Java can be used to provide any top-level Java declarations (e.g. '`import`' declarations or auxiliary class definitions) which are needed by Java code in '`pragma foreign_proc`' declarations in that module.

For example:

```
:- pragma foreign_decl("Java", "
import javax.swing.*;
import java.awt.*;

class MyApplet extends JApplet {
    public void init() {
        JLabel label = new JLabel(""Hello, world"");
        label.setHorizontalAlignment(JLabel.CENTER);
        getContentPane().add(label);
    }
}
").
:- pred hello(io.state::di, io.state::uo) is det.
:- pragma foreign_proc("Java",
    hello(_IO0::di, _IO::uo),
    [will_not_call_mercury],
"
    MyApplet app = new MyApplet();
    // ...
").
```

14.10.3.7 Using pragma foreign_code for Java

The Java code from '`pragma foreign_proc`' declarations for Java will be placed in the bodies of static member functions of an automatically generated Java class. '`pragma foreign_code`' can be used to define additional members of this automatically generated class, which can then be referenced by '`pragma foreign_proc`' declarations for Java from that module.

For example:

```
:- pragma foreign_code("Java", "
    static int counter = 0;
").

:- impure pred incr_counter is det.
:- pragma foreign_proc("Java",
    incr_counter,
    [will_not_call_mercury],
"
    counter++;
").

:- semipure func get_counter = int.
:- pragma foreign_proc("Java",
    get_counter = (Result::out),
    [will_not_call_mercury, promise_semipure],
"
    Result = counter;
").
```

14.10.4 Interfacing with Erlang

14.10.4.1 Using pragma foreign_type for Erlang

An Erlang '`pragma foreign_type`' declaration has the form:

```
:- pragma foreign_type("Erlang", MercuryTypeName, "").
```

The effect of this declaration is that Mercury values of type *MercuryTypeName* will be passed to and from Erlang foreign_procs as having some representation unknown to Mercury.

14.10.4.2 Using pragma foreign_export_enum for Erlang

Values of Mercury enumeration types may be referred to from Erlang code using the convention for discriminated union types described in Section 14.3.4 [Erlang data passing conventions], page 108. As such, '`pragma foreign_export_enum`' declarations are neither required or supported for Erlang.

14.10.4.3 Using pragma foreign_proc for Erlang

The input and output variables for a Erlang '`pragma foreign_proc`' will be the Erlang representations as described in Section 14.3.4 [Erlang data passing conventions], page 108.

The Erlang code in a `pragma foreign_proc` declaration for a procedure whose determinism indicates that it can fail must assign either '`true`' or '`false`' to the variable `SUCCESS_INDICATOR`. For example:

```
:- pred contains_char(list(char)::in, char::in) is semidet.

:- pragma foreign_proc("Erlang",
```

```
        contains_char(Str::in, Ch::in),
        [will_not_call_mercury, promise_pure, thread_safe],
    "
        SUCCESS_INDICATOR = (string:chr(Str, Ch) =/= 0)
    ").
```

Arguments whose mode is input will have their values set by the Mercury implementation on entry to the Erlang code. The Erlang code must set the values of all output variables, even if the procedure fails (i.e. sets the `SUCCESS_INDICATOR` variable to `false`).

14.10.4.4 Using pragma foreign_export for Erlang

A '`pragma foreign_export`' declaration for Erlang has the form:

```
:- pragma foreign_export("Erlang", MercuryMode, "Erlang_Name").
```

For example,

```
:- pragma foreign_export("Erlang", foo(in, in, out), "foo").
```

The type signature of the Erlang interface to a Mercury procedure is described in Section 14.3.4 [Erlang data passing conventions], page 108.

Calling polymorphically typed Mercury procedures from Erlang is a little bit more difficult than calling ordinary (monomorphically typed) Mercury procedures. The simplest method is to just create monomorphic forwarding procedures that call the polymorphic procedures, and export them, rather than exporting the polymorphic procedures.

If you do export a polymorphically typed Mercury procedure, the compiler will prepend one '`type_info`' argument to the parameter list of the Erlang interface function for each distinct type variable in the Mercury procedure's type signature. The caller must arrange to pass in appropriate '`type_info`' values corresponding to the types of the other arguments passed. These '`type_info`' arguments can be obtained using the Mercury '`type_of`' function in the Mercury standard library module '`type_desc`'.

14.10.4.5 Using pragma foreign_decl for Erlang

'`pragma foreign_decl`' declarations for Erlang can be used to provide any top-level Erlang declarations (e.g. '`-define`' macro declarations) which are needed by Erlang code.

'`pragma foreign_decl`' blocks which do not have the '`local`' attribute will be copied into the '`.hrl`' header file for that module, and automatically included by other modules that import the module. Therefore '`-export`' directives and Erlang module attributes should only appear in '`local`' blocks.

For example:

```
:- pragma foreign_decl("Erlang", "
    -define(FOO, 42).
").
:- pred hello(io.state::di, io.state::uo) is det.
:- pragma foreign_proc("Erlang",
    hello(_IO0::di, _IO::uo),
    [will_not_call_mercury, promise_pure],
"
    io:format(""FOO = ~w~n"", [?FOO])
").
```

14.10.4.6 Using pragma foreign_code for Erlang

'`pragma foreign_code`' can be used to define additional Erlang functions which can then be referenced by '`pragma foreign_proc`' declarations for Erlang from that module. By adding '`-export`' directives inside '`pragma foreign_decl`' declarations, those functions can additionally be called from outside the defining module.

For example:

```
:- pragma foreign_code("Erlang", "
    foo() -> io:put_chars(""Foo."").
").

:- impure pred say_foo is det.
:- pragma foreign_proc("Erlang",
    say_foo,
    [will_not_call_mercury],
"
    foo()
").
```

15 Impurity declarations

In order to efficiently implement certain predicates, it is occasionally necessary to venture outside pure logic programming. Other predicates cannot be implemented at all within the paradigm of logic programming, for example, all solutions predicates. Such predicates are often written using the foreign language interface. Sometimes, however, it would be more convenient, or more efficient, to write such predicates using the facilities of Mercury. For example, it is much more convenient to access arguments of compound Mercury terms in Mercury than in C, and the ability of the Mercury compiler to specialize code can make higher-order predicates written in Mercury significantly more efficient than similar C code.

One important aim of Mercury's impurity system is to make the distinction between the pure and impure code very clear. This is done by requiring every impure predicate or function to be so declared, and by requiring every call to an impure predicate or function to be flagged as such. Predicates or functions that are implemented in terms of impure predicates or functions are assumed to be impure themselves unless they are explicitly promised to be pure.

Please note that the facilities described here are needed only very rarely. The main intent is for implementing language primitives such as the all solutions predicates, or for implementing interfaces to foreign language libraries using the foreign language interface. Any other use of '`impure`' or '`semipure`' probably indicates either a weakness in the Mercury standard library, or the programmer's lack of familiarity with the standard library. Newcomers to Mercury are hence encouraged to **skip this section**.

15.1 Choosing the right level of purity

Mercury distinguishes three "levels" of purity:

pure For pure procedures, the set of solutions depends only on the values of the input arguments. They do not interact with the "real" world (i.e., do any input/output) without taking an io.state (see Chapter 3 [Types], page 28) as input and returning one as output, and do not change the value of any data structure that will not be undone on backtracking (unless the data structure would be unreachable on backtracking). Note that equality axioms are important when considering the value of data structures. The declarative semantics of pure predicates is never affected by the invocation of other predicates. It is possible for the invocation of pure predicates to affect the operational behaviour of non-pure predicates and vice versa.

By default, Mercury predicates and functions are pure. Without using the foreign language interface, writing mode-specific clauses or calling other impure predicates and functions it is impossible to write impure code in Mercury.

semipure Semipure predicates are just like pure predicates, except that their declarative semantics may be affected by the invocation of impure predicates. That is, they are sensitive to the state of the computation other than as reflected by their input arguments, though they do not affect the state themselves.

impure Impure predicates may perform I/O or modify hidden state, even if these side effects alter the operational semantics of other code. However, impure predi-

cates may not change the declarative semantics of pure code. They must be type-, mode-, determinism- and uniqueness correct.

15.2 Purity ordering

The three levels of purity have a total ordering defined upon them (which we will simply call the purity), where `pure > semipure > impure`.

15.3 Semantics

It is important to the proper operation of impure and semipure code, to the flexibility of the compiler to optimize pure code, and to the semantics of the Mercury language, that a clear distinction be drawn between ordinary Mercury code and imperative code written with Mercury syntax. How this distinction is drawn will be explained below; the purpose of this section is to explain the semantics of programs with impure predicates.

A *declarative* semantics of impure Mercury code would be largely useless, because the declarative semantics cannot capture the intent of the programmer. Impure predicates are executed for their side-effects, which by definition are not part of their declarative semantics. Thus it is the *operational* semantics of impure predicates that Mercury must specify, and Mercury implementations must respect.

The operational semantics of a Mercury predicate which invokes *impure* code is a modified form of the *strict sequential* semantics (see Chapter 13 [Semantics], page 96). *Impure* goals may not be reordered relative to any other goals; not even "minimal" reordering as implied by the modes is permitted. If any such reordering is needed, this is a mode error. However, *pure* and *semipure* goals may be reordered as the compiler desires (within the bounds of the semantics the user has specified for the program) as long as they are not moved across an impure goal. Execution of impure goals is strict: they must be executed if they are reached, even if it can be determined that the computation cannot lead to successful termination.

Semipure goals can be given a "contextual" declarative semantics. They cannot have any side-effects, so it is expected that, given the context in which they are called (relative to any impure goals in the program), their declarative semantics fully captures the intent of the programmer. Thus a semipure goal has a perfectly consistent declarative semantics, until an impure goal is reached. After that, it has another (possibly different) declarative semantics, until the next impure goal is executed, and so on. Mercury implementations must respect this contextual nature of the semantics of semipure goals; within a single context, an implementation may treat a semipure goal as if it were pure.

15.4 Declaring impure functions and predicates

Every Mercury predicate or function has exactly two purity values associated with it. One is the *declared* purity of the predicate or function, which is given by the programmer. The other value is the *inferred* purity, which is calculated from the purity of goals in the body of the predicate or function.

A predicate is declared to be impure or semipure by preceding the word `pred` in its `pred` declaration with `impure` or `semipure`, respectively. Similarly, a function is declared impure or semipure by preceding the word `func` in its `func` declaration with `impure` or `semipure`. That is, a declaration of the form:

```
:- impure pred Pred(Arguments...).
:- semipure pred Pred(Arguments...).
```

or

```
:- impure func Func(Arguments...) = Result.
:- semipure func Func(Arguments...) = Result.
```

declares the predicate *Pred* to be impure and the function *Func* to be semipure, respectively.

Type class methods may also be declared as `impure` or `semipure` by preceding the word `pred` or `func` with the appropriate purity level. An instance of the type class must provide method implementations that are at least as pure as the method declaration.

15.5 Marking a goal as impure

If predicate `p/N` is declared to be `impure` (`semipure`) then all calls to `p/N` must be annotated with `impure` (`semipure`):

```
impure p(X1, X2, ..., XN)
```

If function `f/N` is declared to be `impure` (`semipure`) then all applications of `f/N` must be obtained by unification with a variable and the unification goal as a whole be annotated with `impure`

```
impure X = f(X1, X2, ..., XN)
```

Any call or unification goal containing a non-local variable with inst any that appears in a negated context (i.e., in a negation or the condition of an if-then-else goal) must be given an `impure` annotation because it may violate referential transparency.

Compound goals should not have purity annotations.

The compiler will report an error if a required purity annotation is omitted from a call or unification goal or if a `semipure` annotation is used where an `impure` annotation is required. The compiler will report a warning if a semipure goal is annotated with `impure` or a pure goal is annotated with `semipure`

The requirement that impure or semipure calls be marked with `impure` or `semipure` allows someone reading the code to tell which goals are not pure, making code which relies on side effects somewhat less mysterious. Furthermore, it means that if a call is *not* preceded by `impure` or `semipure`, then the reader can rely on the call having a proper declarative semantics, without hidden side-effects.

15.6 Promising that a predicate is pure

Predicates that are implemented in terms of impure or semipure predicates are assumed to have the least of the purity of the goals in their body. The declared purity of a predicate must not be more pure than the inferred purity; if it is, the compiler must generate an error. If the declared purity is less pure than the inferred purity, the compiler should issue a warning (this is similar to the above case for goals). Because the inferred purity of the predicate is calculated from the declared purity of the calls it executes, the lowest purity bound is propagated up from callee to caller through the program.

In some cases the impurity of a predicate's body is an implementation detail which should not be exposed to callers. These predicates are pure or semipure even though they call impure or semipure predicates. The only way for the programmer to stop the

propagation of impurity is to explicitly promise that the predicate or function is pure or semipure.

Of course, the Mercury compiler cannot verify that the predicate's purity matches the promise, so it is the programmer's responsibility to ensure this. If a predicate is promised pure or semipure and is not, the behaviour of the program is undefined.

The programmer may promise that a predicate or function is pure or semipure using the `promise_pure` and `promise_semipure` pragmas:

```
:- pragma promise_pure(Name/Arity).
:- pragma promise_semipure(Name/Arity).
```

Programmers should be very careful about mixing code that is promised pure with impure predicates or functions that may manipulate the same hidden state (for example, the impure predicates used to implement a predicate that is promised pure); the '`promise_pure`' declaration is supposed to promise that impure code cannot change the declarative semantics of pure code. The module system can be used to minimize the possibility of making errors with such code, by keeping impure predicates or functions behind the interface where code is promised pure.

Note that the '`promise_pure`', '`promise_semipure`', and '`promise_impure`' scopes described in Section 2.11 [Goals], page 10 may be used to promise purity at the finer level of goals within clauses.

15.7 An example using impurity

The following example illustrates how a pure predicate may be implemented using impure code. Note that this code is not reentrant, and so is not useful as is. It is meant only as an example.

```
:- pragma foreign_decl("C", "#include <limits.h>").
:- pragma foreign_decl("C", "extern MR_Integer max;").

:- pragma foreign_code("C", "MR_Integer max;").

:- impure pred init_max is det.
:- pragma foreign_proc("C",
    init_max,
    [will_not_call_mercury],
"
    max = INT_MIN;
").

:- impure pred set_max(int::in) is det.
:- pragma foreign_proc("C",
   set_max(X::in),
   [will_not_call_mercury],
"
    if (X > max) max = X;
").
```

```
:- semipure func get_max = (int::out) is det.
:- pragma foreign_proc("C",
    get_max = (X::out),
    [promise_semipure, will_not_call_mercury],
"
    X = max;
").

:- pragma promise_pure(max_solution/2).
:- pred max_solution(pred(int), int).
:- mode max_solution(pred(out) is multi, out) is det.

max_solution(Generator, Max) :-
    impure init_max,
    (
        Generator(X),
        impure set_max(X),
        fail
    ;
        semipure Max = get_max
    ).
```

15.8 Using impurity with higher-order code

Higher-order code can manipulate impure or semipure predicates and functions, provided that explicit purity annotations are used in three places: on the higher-order types, on lambda expressions, and on higher-order calls. (There are no purity annotations on higher-order insts and modes, however.)

15.8.1 Purity annotations on higher-order types

Ordinary higher-order types, such as '`pred(T1, T2)`' and '`func(T1, T2) = T`', represent only pure predicates or pure functions. But for each ordinary higher-order type *Foo*, there are two corresponding types '`semipure` *`Foo`*' and '`impure` *`Foo`*'. These types can be used for higher-order code that needs to manipulate impure or semipure procedures. For example the type '`impure func(int) = int`' represents impure functions from `int` to `int`.

There are no implicit conversions and no subtyping relationship between ordinary higher-order types and the corresponding impure or semipure higher-order types. However, a value of an ordinary higher-order type can be explicit "converted" to a value of an impure (or semipure) higher-order type by wrapping it in an impure (or semipure) lambda expression that just calls the pure higher-order term.

15.8.2 Purity annotations on lambda expressions

Purity annotations are required on lambda expressions that call semipure or impure code. Lambda expressions can be declared as '`semipure`' or '`impure`' by including such an annotation before the '`pred`' or '`func`' identifier in the lambda expression. Such lambda expressions have the corresponding '`semipure`' or '`impure`' higher-order type. For example, the expression

```
(impure func(X) = Y :- semipure get_max(Y), impure set_max(X))
```

is an example of an impure function lambda expression with type '`(impure func(int) = int)`', and the expression

```
(impure pred(X::in, Y::out) is det :-
    semipure get_max(Y),
    impure set_max(X))
```

is an example of an impure predicate lambda expression with type '`impure pred(int, int)`'.

15.8.3 Purity annotations on higher-order calls

Any calls to impure or semipure higher-order terms must be explicitly annotated as such. For impure or semipure higher-order predicates, the annotation is indicated by putting '`impure`' or '`semipure`' before the call. For example:

```
:- func foo(impure pred(int)) = int.
:- mode foo(in(pred(out) is det)) = out is det.

foo(ImpurePred) = X1 + X2 :-
    % Using higher-order syntax.
    impure ImpurePred(X1),
    % Using the call/N syntax.
    impure call(ImpurePred, X2).
```

For calling impure or semipure higher-order functions, the notation is different than what you might expect. In addition to using an '`impure`' or '`semipure`' operator on the unification which invokes the higher-order function application, you must also use '`impure_apply`' or '`semipure_apply`' rather than using '`apply`' or higher-order syntax. For example:

```
:- func map(impure func(T1) = T2, list(T1)) = list(T2).

map(_ImpureFunc, []) = [].
map(ImpureFunc, [X|Xs]) = [Y|Ys] :-
    impure Y = impure_apply(ImpureFunc, X),
    impure Ys = map(ImpureFunc, Ys).
```

16 Solver types

Solver types are an experimental addition to the language supporting the implementation of constraint solvers. A program may place constraints on and between variables of a solver type, limiting the values those variables may take on before they are actually bound. For example, if `X` and `Y` are variables belonging to a constrained integer solver type, we might place constraints upon them such that `X > 3 + Y` and `Y =< 7`. A later attempt to unify `Y` with `10` will fail (it would violate the second constraint); similarly an attempt to unify `X` with `5` and `Y` with `4` would fail (it would violate the first constraint).

16.1 The 'any' inst

Variables with solver types can have one of three possible insts: `free`, `ground` or `any`. A variable with a solver type with inst `any` may not (yet) be semantically ground, in the following sense: if a variable is semantically ground then the set of values it unifies with form an equivalence class; if a variable is non-ground then the set of values it unifies with do not form an equivalence class.

More formally, `X` is ground if for values `Y` and `Z` that unify with `X`, it is the case that `Y` and `Z` also unify with each other. `X` is non-ground if there are values `Y` and `Z` that unify with `X`, but which do not unify with each other.

A non-solver type value will have inst `any` if it is constructed using one or more inst `any` values.

The builtin modes `ia` and `oa` are equivalent to `in(any)` and `out(any)` respectively.

16.2 Abstract solver type declarations

The type declarations

```
:- solver type t1.
:- solver type t2(T1, T2).
```

declare types `t1/0` and `t2/2` to be abstract solver types. Abstract solver type declarations are identical to ordinary abstract type declarations except for the `solver` keyword.

16.3 Solver type definitions

A solver type definition takes the following form:

```
:- solver type solver_type
    where   representation   is representation_type,
            ground           is ground_inst,
            any              is any_inst,
            constraint_store is mutable_decls,
            equality         is equality_pred,
            comparison       is comparison_pred.
```

The `representation` attribute is mandatory. The *ground_inst* and *any_inst* attributes are optional and default to `ground`. The `constraint_store` attribute is mandatory: *mutable_decls* must be either a single mutable declaration (see Section 9.6 [Module-local mutable variables], page 72), or a comma separated list of mutable declarations in brackets. The

`equality` and `comparison` attributes are optional, although a solver type without equality would not be very useful. The attributes that are not omitted must appear in the order shown above.

The *representation_type* is the type used to implement the *solver_type*. A two-tier scheme of this kind is necessary for a number of reasons, including

- a semantic gap is necessary to accommodate the fact that non-ground *solver_type* values may be represented by ground *representation_type* values (in the context of the corresponding constraint solver state);
- this approach greatly simplifies the definition of equality and comparison predicates for the *solver_type*.

The *ground_inst* is the inst associated with *representation_type* values denoting `ground` *solver_type* values.

The *any_inst* is the inst associated with *representation_type* values denoting `any solver_type` values.

The compiler constructs four impure functions for converting between *solver_type* values and *representation_type* values (*name* is the function symbol used to name *solver_type* and *arity* is the number of type parameters it takes):

```
:- impure func 'representation of ground name/arity'(solver_type) =
                        representation_type.
:-        mode 'representation of ground name/arity'(in) =
                        out(ground_inst) is det.

:- impure func 'representation of any name/arity'(solver_type) =
                        representation_type.
:-        mode 'representation of any name/arity'(in(any)) =
                        out(any_inst) is det.

:- impure func 'representation to ground name/arity'(representation_type) =
                        solver_type.
:-        mode 'representation to ground name/arity'(in(ground_inst)) =
                        out is det.

:- impure func 'representation to any name/arity'(representation_type) =
                        solver_type.
:-        mode 'representation to any name/arity'(in(any_inst)) =
                        out(any) is det.
```

These functions are impure because of the semantic gap issue mentioned above.

These functions are constructed in-line as part of a source-to-source transformation, hence it is an error to define a solver type in the interface section of a module.

If *solver_type* is exported then it is a requirement that *representation_type*, and, if specified, *equality_pred* and *comparison_pred* are also exported from the same module.

If *equality_pred* is not specified then the compiler will generate an equality predicate that throws an exception of type '`exception.software_error/0`' when called.

Likewise, if *comparison_pred* is not specified then the compiler will generate a comparison predicate that throws an exception of type '`exception.software_error/0`' when called.

If provided, any mutable declarations given for the `constraint_store` attribute are equivalent to separate mutable declarations; their association with the solver type is for the purposes of documentation. That is,

```
:- solver type t
    where ...,
          constraint_store is [ mutable(a, int, 42, ground, []),
                                mutable(b, string, "Hi", ground, [])
                              ],
          ...
```

is equivalent to

```
:- solver type t
    where ...
:- mutable(a, int, 42, ground, []).
:- mutable(b, string, "Hi", ground, []).
```

16.4 Implementing solver types

A solver type is an abstraction, implemented using a combination of a private representation type and a constraint store.

The constraint store is an (impure) piece of state used to keep track of the extant constraints on variables of the solver type. This will typically be implemented using foreign code.

It is important that changes to the constraint store are properly trailed (see Section 19.5 [Trailing], page 152) so that changes can be undone on backtracking.

The solver type implementation should provide functions and predicates to

- construct and deconstruct solver type values,
- to place constraints on solver type variables,
- to convert `any` solver type variables to `ground` if possible (this is obviously an impure operation — see Chapter 15 [Impurity], page 129),
- to convert solver type values to non-solver type values (again, this is impure and requires the argument solver type values be sufficiently ground),
- to ask questions about the extant constraints on solver type variables without constraining them further (this too is impure because the set of constraints on a variable may change during execution of the program).

16.5 Solver types and negated contexts

Mercury's negation and if-then-else goals (and hence also inequalities and universal quantifications) are implemented using *negation as failure*, meaning that the failure to find a proof of one statement is regarded as a proof of its negation. Negation as failure is sound provided that no non-local variable becomes further bound during the execution of a goal which may be negated. This includes negated goals themselves, as well as the conditions of if-then-elses, which are negated iff they fail without producing any solution, and the bodies

of pred or func expressions, which may be called or applied in one of the other contexts, or indeed in another pred or func expression.

Mercury checks that any solver variables that are used in the above contexts are used in such a way that negation as failure remains sound. In the case of negation and if-then-else goals, if any non-local solver type variable or higher-order variable with inst `any` is used in a negated context, the goal must be placed inside a `promise_pure`, `promise_semipure`, or `promise_impure` scope. The first two promises assert that (among other things) no solver variable becomes further bound in the negated context. The third promise makes the weaker assertion that the goal satisfies the requirements of all impure goals (namely, that it doesn't interfere with the semantics of other pure goals).

In the case of pred and func expressions, Mercury allows three possibilities. The higher-order value may be considered `ground`, which means that all non-local variables used in the body of the expression (including those with other higher-order values) must themselves be ground. Higher-order values that are ground can be safely called or applied in any context, including negated contexts, since none of their (ground) non-local variables can become further bound by doing so. Alternatively, the higher-order value may be considered to have inst `any`, which allows non-local variables used in the body of the expression to have inst `any`. Calling or applying these values *may* further bind non-local variables, so if this occurs in a negated context then, as in the case of solver variables, a promise will be required around the negation or if-then-else.

Pred and func expressions with inst `any` are written using `any_pred` and `any_func` in place of `pred` and `func`, respectively.

The third possibility is that the higher-order value can be given an impure type (see Section 15.8 [Higher-order impurity], page 133).

17 Trace goals

Sometimes, programmers find themselves needing to perform some side-effects in the middle of declarative code. One example is an operation that takes so long that users may think the program has gone into an infinite loop: periodically printing a progress message can give them reassurance. Another example is a program that is too long-running for its behaviour to be analyzed via debuggers and too complex for analysis via profilers; a programmable logging facility generating data for analysis by a specially-written program may be the best option. However, inserting arbitrary side effects into declarative code is against the spirit of Mercury. Trace goals exist to provide a mechanism to code these side effects in a disciplined fashion.

The format of trace goals is `trace` *`Params Goal`*. *Goal* must be a valid goal; *Params* must be a valid list of one or more trace parameters. The following example shows all four of the available kinds of parameters: '`compile_time`', '`run_time`', '`io`' and '`state`'. (In practice, it is far more typical to have just one parameter, '`io`'.)

```
:- mutable(logging_level, int, 0, ground, []).

:- pred time_consuming_task(data::in, result::out) is det.

time_consuming_task(In, Out) :-
    trace [
        compile_time(flag("do_logging") or grade(debug)),
        run_time(env("VERBOSE")),
        io(!IO),
        state(logging_level, !LoggingLevel)
    ] (
        io.write_string("Time_consuming_task start\n", !IO),
        ( !.LoggingLevel > 1 ->
            io.write_string("Input is ", !IO),
            io.write(In, !IO),
            io.nl(!IO)
        ;
            true
        )
    ),
    ...
    % perform the actual task
```

The '`compile_time`' parameter says under what circumstances the trace goal should be included in the executable program. In the example, at least one of two conditions has to be true: either this module has to be compiled with the option '`--trace-flag=do_logging`', or it has to be compiled in a debugging grade.

In general, the single argument of the '`compile_time`' function symbol is a boolean expression of primitive compile-time conditions. Valid boolean operators in these expressions are '`and`', '`or`' and '`not`'. There are three kinds of primitive compile-time conditions. The first has the form '`flag(`*`FlagName`*`)`', where *FlagName* is an arbitrary name picked by the programmer; this condition is true if the module is compiled with the

option '`--trace-flag=`*`FlagName`*'. The second has the form '`tracelevel(shallow)`', or '`tracelevel(deep)`'; this condition is true (irrespective of grade) if the module is compiled with at least the specified trace level. The third has the form '`grade(GradeTest)`'. The supported '`GradeTests`'s and their meanings are as follows.

'`debug`' True if the module is compiled with execution tracing enabled.

'`ssdebug`' True if the module is compiled with source-to-source debugging enabled.

'`prof`' True if the module is compiled with non-deep profiling enabled.

'`profdeep`'
True if the module is compiled with deep profiling enabled.

'`par`' True if the module is compiled with parallel execution enabled.

'`trail`' True if the module is compiled with trailing enabled.

'`llds`' True if the module is compiled with '`--highlevel-code`' disabled.

'`mlds`' True if the module is compiled with '`--highlevel-code`' enabled.

'`c`' True if the target language of the compilation is C.

'`csharp`' True if the target language of the compilation is C#.

'`java`' True if the target language of the compilation is Java.

'`erlang`' True if the target language of the compilation is Erlang.

The '`run_time`' parameter says under what circumstances the trace goal, if included in the executable program, should actually be executed. In this case, the environment variable '`VERBOSE`' has be to set when the program starts execution. (It doesn't matter what value it is set to.)

In general, the single argument of the '`run_time`' function symbol is a boolean expression of primitive run-time conditions. Valid boolean operators in these expressions are '`and`', '`or`' and '`not`'. There is just one primitive run-time condition. It has the form '`env(`*`EnvVarName`*`)`', this condition is true if the environment variable *EnvVarName* exists when the program starts execution.

The '`compile_time`' and '`run_time`' parameters may not appear in the parameter list more than once; programmers who want more than one condition have to specify how (with what boolean operators) their values should be combined. However, it is ok for them not to appear in the parameter list at all. If there is no '`compile_time`' parameter, the trace goal is always compiled into the executable; if there is no '`run_time`' parameter, the trace goal is always executed (if it is compiled into the executable).

Since the trace goal may end up either not compiled into the executable or just not executed, it cannot bind any variables that occur in the surrounding code. (If it were allowed to bind such variables, then those variables would stay unbound if either the compile time or the run time condition were false.) This greatly restricts what trace goals can do.

The usual reason for including a trace goal in a procedure body is to perform some I/O, which requires access to the I/O state. The '`io`' parameter supplies this access. Its argument must be the name of a state variable prefixed by '`!`'; by convention, it is usually '`!IO`'. The language implementation supplies the initial unique value of the I/O state as

the value of '`!.IO`' at the start of the trace goal; it requires the trace goal to give back the final unique value of the I/O state as the value of '`!.IO`' current at the end of the trace goal.

Note that trace goals that wish to do I/O must include this parameter in their parameter list *even if* the surrounding code already has access to an I/O state. This is because otherwise, doing any I/O inside the trace goal would destroy the value of the current I/O state, changing the instantiation state of the variable holding it, and trace goals are not allowed to do that.

The '`io`' parameter may appear in the parameter list at most once, since it doesn't make sense to have two copies of the I/O state available to the trace goal.

Besides doing I/O, trace goals may read and possibly write the values of mutable variables. Each mutable the trace goal wants access to should be listed in its own '`state`' parameter (which may therefore appear in the parameter list more than once). Each '`state`' parameter has two arguments: the first gives the name of the mutable, while the second must be the name of a state variable prefixed by '`!`', e.g. '`!LoggingLevel`'. The language implementation supplies the initial value of the mutable as the value of (in this case) '`!.LoggingLevel`' at the start of the trace goal; at the end of the trace goal, it puts the value of '`!.LoggingLevel`' current then back into the mutable.

The intention here is that trace goals should be able to access mutables that give them information about the parameters within which they should operate. The ability of trace goals to actually *update* the values of mutables is intended to allow the implementation of trace goals whose actions depend on the actions executed by previous trace goals. For example, a trace goal could test whether the current input value is the same as the previous input value, and if it is, then it can say so instead of printing the value out again. Another possibility is a progress message which is printed not for every item processed, but for every 1000th item, reassuring users without overwhelming them with a deluge of output.

This kind of code is the *only* intended use of this ability. Any program in which the value of a mutable set by a trace goal is inspected by code that is not itself within a trace goal is explicitly violating the intended uses of trace goals. Only the difficulty of implementing the required global program analysis prevents the language design from outlawing such programs in the first place.

The compiler will not delete trace goals from the bodies of the procedures containing them. However, trace goals inside a procedure don't prevent calls to that procedure from being optimized away, if such optimization is otherwise possible. (There is no point in debugging or logging operations that don't actually occur.) In their effect on program optimizations, trace goals function as a kind of impure code, but one with an implicit promise_pure around the clause in which they occur.

18 Pragmas

The pragma declarations described below are a standard part of the Mercury language, as are the pragmas for controlling the foreign language interface (see Chapter 14 [Foreign language interface], page 98) and impurity (see Chapter 15 [Impurity], page 129). As an extension, implementations may also choose to support additional pragmas with implementation-dependent semantics (see Chapter 19 [Implementation-dependent extensions], page 146).

18.1 Inlining

A declaration of the form

```
:- pragma inline(Name/Arity).
```

is a hint to the compiler that all calls to the predicate(s) or function(s) with name *Name* and arity *Arity* should be inlined.

The current Mercury implementation is smart enough to inline simple predicates even without this hint.

A declaration of the form

```
:- pragma no_inline(Name/Arity).
```

ensures the compiler will not inline this predicate. This may be used simply for performance concerns (inlining can cause unwanted code bloat in some cases) or to prevent possibly dangerous inlining when using low-level C code.

18.2 Type specialization

The overhead of polymorphism can in some cases be significant, especially where polymorphic predicates make heavy use of class method calls or the builtin unification and comparison routines. To avoid this, the programmer can suggest to the compiler that a specialized version of a procedure should be created for a specific set of argument types.

18.2.1 Syntax and semantics of type specialization pragmas

A declaration of the form

```
:- pragma type_spec(Name/Arity, Subst).
:- pragma type_spec(Name(Modes), Subst).
```

suggests to the compiler that a specialized version of predicate(s) or function(s) with name *Name* and arity *Arity* should be created with the type substitution given by *Subst* applied to the argument types. The second form of the declaration only suggests specialization of the specified mode of the predicate or function.

The substitution is written as a conjunction of bindings of the form '`TypeVar = Type`', for example '`K = int`' or '`(K = int, V = list(int))`'.

The declarations

```
:- pred map.lookup(map(K, V), K, V).
:- pragma type_spec(map.lookup/3, K = int).
```

give a hint to the compiler that a version of '`map.lookup/3`' should be created for integer keys.

Implementations are free to ignore '`pragma type_spec`' declarations. Implementations are also free to perform type specialization even in the absence of any '`pragma type_spec`' declarations.

18.2.2 When to use type specialization

The set of types for which a predicate or function should be specialized is best determined by profiling your application. Overuse of type specialization will result in code bloat.

Type specialization of predicates or functions which unify or compare polymorphic variables is most effective when the specialized types are builtin types such as `int`, `float` and `string`, or enumeration types, since their unification and comparison procedures are simple and can be inlined.

Predicates or functions which make use of type class method calls may also be candidates for specialization. Again, this is most effective when the called type class methods are simple enough to be inlined.

18.2.3 Implementation specific details

The University of Melbourne Mercury compiler performs user-requested type specializations when invoked with '`--user-guided-type-specialization`', which is enabled at optimization level '`-O2`' or higher. However, for the Java back-end, user-requested type specializations are ignored.

18.3 Obsolescence

A declaration of the form

```
:- pragma obsolete(Name/Arity).
```

declares that the predicate(s) or function(s) with name *Name* and arity *Arity* are "obsolete": it instructs the compiler to issue a warning whenever the named predicate(s) or function(s) are used.

'`pragma obsolete`' declarations are intended for use by library developers, to allow gradual (rather than abrupt) evolution of library interfaces. If a library developer changes the interface of a library predicate, they should leave the old version of that predicate in the library, but mark it as obsolete using a '`pragma obsolete`' declaration, and document how library users should modify their code to suit the new interface. The users of the library will then get a warning if they use obsolete features, and can consult the library documentation to determine how to fix their code. Eventually, when the library developer deems that users have had sufficient warning, they can remove the old version entirely.

18.4 No determinism warnings

A declaration of the form

```
:- pragma no_determinism_warning(Name/Arity).
```

tells the compiler not to generate any warnings that the determinism declarations of procedures of the predicate or function with name *Name* and arity *Arity* are not as tight as they could be.

'`pragma no_determinism_warning`' declarations are intended for use in situations in which the code of a predicate has one determinism, but the declared determinism of the

procedure must be looser due to some outside requirement. One such situation is when a set of procedures are all possible values of the same higher-order variable, which requires them to have the same argument types, modes, and determinisms. If (say) most of the procedures are `det` but some are `erroneous` (that is, they always throws an exception), the procedures that are declared `det` but whose bodies have determinism `erroneous` will get a warning saying their determinism declaration could be tighter, unless the programmer specifies this pragma for them.

18.5 No dead predicate warnings

A declaration of the form

```
:- pragma consider_used(Name/Arity).
```

tells the compiler to consider the predicate or function with name *Name* and arity *Arity* to be used, and not generate any dead procedure/predicate/function warnings either for the named predicate, *or* for the other predicates and functions that it calls, either directly or indirectly.

'`pragma consider_used`' declarations are intended for use in situations in which the code that was intended to call such a predicate or function is not yet written.

18.6 Source file name

The '`source_file`' pragma and '`#line`' directives provide support for preprocessors and other tools that generate Mercury code. The tool can insert these directives into the generated Mercury code to allow the Mercury compiler to report diagnostics (error and warning messages) at the original source code location, rather than at the location in the automatically generated Mercury code.

A '`source_file`' pragma is a declaration of the form

```
:- pragma source_file(Name).
```

where *Name* is a string that specifies the name of the source file.

For example, if a preprocessor generated a file '`foo.m`' based on a input file '`foo.m.in`', and it copied lines 20, 30, and 31 from '`foo.m.in`', the following directives would ensure that any error or warnings for those lines copied from '`foo.m`' were reported at their original source locations in '`foo.m.in`'.

```
:- module foo.
:- pragma source_file("foo.m.in").
#20
% this line comes from line 20 of foo.m
#30
% this line comes from line 30 of foo.m
% this line comes from line 31 of foo.m
:- pragma source_file("foo.m").
#10
% this automatically generated line is line 10 of foo.m
```

Note that if a generated file contains some text which is copied from a source file, and some which is automatically generated, it is a good idea to use '`pragma source_file`'

and '`#line`' directives to reset the source file name and line number to point back to the generated file for the automatically generated text, as in the above example.

19 Implementation-dependent extensions

The University of Melbourne Mercury implementation supports the following extensions to the Mercury language:

19.1 Fact tables

Large tables of facts can be compiled using a different algorithm that is more efficient and produces more efficient code.

A declaration of the form

```
:- pragma fact_table(Name/Arity, FileName).
```

tells the compiler that the predicate or function with name *Name* and arity *Arity* is defined by a set of facts in an external file *FileName*. Defining large tables of facts in this way allows the compiler to use a more efficient algorithm for compiling them. This algorithm uses less memory than would normally be required to compile the facts so much larger tables are possible.

Each mode is indexed on all its input arguments so the compiler can produce very efficient code using this technique.

In the current implementation, the table of facts is compiled into a separate C file named '*FileName*`.c`'. The compiler will automatically generate the correct dependencies for this file when the command '`mmake` *main_module*`.depend`' is invoked. This ensures that the C file will be compiled to '*FileName*`.o`' and then linked with the other object files when '`mmake` *main_module*' is invoked.

The main limitation of the '`fact_table`' pragma is that in the current implementation, predicates or functions defined as fact tables can only have arguments of types `string`, `int` or `float`.

Another limitation is that the '`--high-level-code`' back-end does not support '`pragma fact_table`' for procedures with determinism `nondet` or `multi`.

19.2 Tabled evaluation

(Note: "Tabled evaluation" has no relation to the "fact tables" described above.)

Ordinarily, the results of each procedure call are not recorded; if the same procedure is called with the same arguments, then the answer(s) must be recomputed again. For some procedures, this recomputation can be very wasteful.

With tabled evaluation, the implementation keeps a table containing the previously computed results of the specified procedure; this table is sometimes called the memo table (since it "remembers" previous answers). At each procedure call, the implementation will search the memo table to check whether the answer(s) have already been computed, and if so, the answers will be returned directly from the memo table rather than being recomputed. This can result in much faster execution, at the cost of additional space to record answers in the table.

The implementation can also check at runtime for the situation where a procedure calls itself recursively with the same arguments, which would normally result in a infinite loop;

if this situation is encountered, it can (at the programmer's option) either throw an exception, or avoid the infinite loop by computing solutions using a "minimal model" semantics. (Specifically, the minimal model computed by our implementation is the perfect model.)

The current Mercury implementation supports three different pragmas for tabling, to cover these three cases: '`loop_check`', '`memo`', and '`minimal_model`'.

- The '`loop_check`' pragma asks only for loop checking. With this pragma, the memo table will map each distinct set of input arguments only to a single boolean saying whether a call with those arguments is currently active or not; the pragma's only effect is to cause the predicate to throw an exception if this boolean says that the current call has the same arguments as one of its ancestors, which indicates an infinite recursive loop.

  Note that loop checking for nondet and multi predicates assumes that calls to these predicates generate all their solutions and then fail. If a caller asks them only for some solutions and then cuts away all later solutions (e.g. via a quantification that only asks whether a solution satisfying a particular test exists), then the cut-away call never gets a chance to record the fact that it is not longer active. The next call to that predicate with the same arguments will therefore think that the previous call is still active, and will consider this call to be an infinite loop.
- The '`memo`' pragma asks for both loop checking and memoization. With this pragma, the memo table will map each distinct set of input arguments either to the set of results computed previously for those arguments, or to an indication that the call is still active and thus those results are still being computed. This predicate will thus look for infinite recursive loops (and throw an exception if and when it finds one) but it will also record all its solutions in the memo table, and will avoid recomputing solutions that are already available in the memo table.
- The '`minimal_model`' pragma asks for the computation of a "minimal model" semantics. These differ from the '`memo`' pragma in that the detection of what appears to be an infinite recursive loop is not fatal. The implementation will consider the apparently infinitely recursive calls to fail if the call concerned has no way of computing any solutions it has not already computed and recorded, and if it does have such a way, then it switches the execution to explore those ways before coming back to the apparently infinitely recursive call.

  Minimal model evaluation is applicable only to procedures that can succeed more than once, and only in grades that explicitly support it.

The syntax for each of these declarations is

```
:- pragma memo(Name/Arity).
:- pragma memo(Name/Arity, [list of tabling attributes]).
:- pragma loop_check(Name/Arity).
:- pragma loop_check(Name/Arity, [list of tabling attributes]).
:- pragma minimal_model(Name/Arity).
:- pragma minimal_model(Name/Arity, [list of tabling attributes]).
```

where *Name*/*Arity* specifies the predicate or function to which the declaration applies. The declaration applies to all modes of the predicate and/or function named. At most one of these declarations may be specified for any given predicate or function.

Programmers can also request the application of tabling only to one particular mode of a predicate or function, via declarations such as these:

```
:- pragma memo(Name(in, in, out)).
:- pragma memo(Name(in, in, out), [list of tabling attributes]).
:- pragma loop_check(Name(in, out)).
:- pragma loop_check(Name(in, out), [list of tabling attributes]).
:- pragma minimal_model(Name(in, in, out, out)).
:- pragma minimal_model(Name(in, in, out, out), [list of tabling attributes]).
```

Because all variants of tabling record the values of input arguments, and all except '`loop_check`' also record the values of output arguments, you cannot apply any of these pragmas to procedures whose arguments' modes include any unique component.

Tabled evaluation of a predicate or function that has an argument whose type is a foreign type will result in a run-time error unless the foreign type is one for which the '`can_pass_as_mercury_type`' and '`stable`' assertions have been made (see Section 14.4 [Using foreign types from Mercury], page 109).

The optional list of attributes allows programmers to control some aspects of the management of the memo table(s) of the procedure(s) affected by the pragma.

The '`allow_reset`' attribute asks the compiler to define a predicate that, when called, resets the memo table. The name of this predicate will be "table_reset_for", followed by the name of the tabled predicate, followed by its arity, and (if the predicate has more than one mode) by the mode number (the first declared mode is mode 0, the second is mode 1, and so on). These three or four components are separated by underscores. The reset predicate takes a di/uo pair of I/O states as arguments. The presence of these I/O state arguments in the reset predicate, and the fact that tabled predicates cannot have unique arguments together imply that a memo table cannot be reset while a call using that memo table is active.

The '`statistics`' attribute asks the compiler to define a predicate that, when called, returns statistics about the memo table. The name of this predicate will be "table_statistics_for", followed by the name of the tabled predicate, followed by its arity, and (if the predicate has more than one mode) by the mode number (the first declared mode is mode 0, the second is mode 1, and so on). These three or four components are separated by underscores. The statistics predicate takes three arguments. The second and third are a di/uo pair of I/O states, while the first is an output argument that contains information about accesses to and modifications of the procedure's memo table, both since the creation of the table, and since the last call to this predicate. The type of this argument is defined in the file table_builtin.m in the Mercury standard library. That module also contains a predicate for printing out this information in a programmer-friendly format.

The remaining two attributes, '`fast_loose`' and '`specified`', control how arguments are looked up in the memo table. The default implementation looks up the *value* of each input argument, and thus requires time proportional to the number of function symbols in the input arguments. This is the only implementation allowed for minimal model tabling, but for predicates tabled with the '`loop_check`' and '`memo`' pragmas, programmers can also choose some other tabling methods.

The '`fast_loose`' attribute asks the compiler to generate code that looks up only the *address* of each input argument in the memo table, which means that the time required is linear

only in the *number* of input arguments, not their *size*. The tradeoff is that '`fast_loose`' does not recognize calls as duplicates if they involve input arguments that are logically equal but are stored at different locations in memory. The following declaration calls for this variant of tabling.

```
:- pragma memo(Name(in, in, in, out),
        [allow_reset, statistics, fast_loose]).
```

The '`specified`' attribute allows programmers to choose individually, for each input argument, whether that argument should be looked up in the memo table by value or by address, or whether it should be looked up at all:

```
:- pragma memo(Name(in, in, in, out), [allow_reset, statistics,
        specified([value, addr, promise_implied, output])]).
```

The '`specified`' attribute should have an argument which is a list, and this list should contain one element for each argument of the predicate or function concerned (if a function, the last element is for the return value). For output arguments, the list element should be '`output`'. For input arguments, the list element may be '`value`', '`addr`' or '`promise_implied`'. The first calls for tabling the argument by value, the second calls for tabling the argument by address, and the third calls for not tabling the argument at all. This last course of action promises that any two calls that agree on the values of the value-tabled input arguments and on the addresses of the address-tabled input arguments will behave the same regardless of the values of the untabled input arguments. In most cases, this will mean that the values of the untabled arguments are implied by the values of the value-tabled arguments and the addresses of the address-tabled arguments, though the promise can also be fulfilled if the table predicate or function does not actually use the untabled argument for computing any of its output. (It is ok for it to use the untabled argument to decide what exception to throw.)

If the tabled predicate or function has only one mode, then this declaration can also be specified without giving the argument modes:

```
:- pragma memo(Name/Arity, [allow_reset, statistics,
        specified([value, addr, promise_implied, output])]).
```

Note that a '`pragma minimal_model`' declaration changes the declarative semantics of the specified predicate or function: instead of using the completion of the clauses as the basis for the semantics, as is normally the case in Mercury, the declarative semantics that is used is a "minimal model" semantics, specifically, the perfect model semantics. Anything which is true or false in the completion semantics is also true or false (respectively) in the perfect model semantics, but there are goals for which the perfect model specifies that the result is true or false, whereas the completion semantics leaves the result unspecified. For these goals, the usual Mercury semantics requires the implementation to either loop or report an error message, but the perfect model semantics requires a particular answer to be returned. In particular, the perfect model semantics says that any call that is not true in *all* models is false.

Programmers should therefore use a '`pragma minimal_model`' declaration only in cases where their intended interpretation for a procedure coincides with the perfect model for that procedure. Fortunately, however, this is usually what programmers intend.

For more information on tabling, see K. Sagonas's PhD thesis *The SLG-WAM: A Search-Efficient Engine for Well-Founded Evaluation of Normal Logic Programs.* See [[4]], page 159.

The operational semantics of procedures with a '`pragma minimal_model`' declaration corresponds to what Sagonas calls "SLGd resolution".

In the general case, the execution mechanism required by minimal model tabling is quite complicated, requiring the ability to delay goals and then wake them up again. The Mercury implementation uses a technique based on copying relevant parts of the stack to the heap when delaying goals. It is described in *Tabling in Mercury: design and implementation* by Z. Somogyi and K. Sagonas, Proceedings of the Eight International Symposium on Practical Aspects of Declarative Languages.

Please note: the current implementation of tabling does not support all the possible compilation grades (see the "Compilation model options" section of the Mercury User's Guide) allowed by the Mercury implementation. In particular, minimal model tabling is incompatible with high level code and the use of trailing.

19.3 Termination analysis

The compiler includes a termination analyser which can be used to prove termination of predicates and functions. Details of the analysis is available in "Termination Analysis for Mercury" by Chris Speirs, Zoltan Somogyi and Harald Sondergaard. See [[1]], page 159.

The analysis is based around an algorithm proposed by Gerhard Groger and Lutz Plumer in their paper "Handling of mutual recursion in automatic termination proofs for logic programs." See [[2]], page 159.

For an introduction to termination analysis for logic programs, please refer to "Termination Analysis for Logic Programs" by Chris Speirs. See [[3]], page 159.

Information about the termination properties of a predicate or function can be given to the compiler. Pragmas are also available to require the compiler to prove termination of a given predicate or function, or to give an error message if it cannot do so.

The analyser is enabled by the option '`--enable-termination`', which can be abbreviated to '`--enable-term`'. When termination analysis is enabled, any predicates or functions with a '`check_termination`' pragma defined on them will have their termination checked, and if termination cannot be proved, the compiler will emit an error message detailing the reason that termination could not be proved.

The option '`--check-termination`', which may be abbreviated to '`--check-term`' or '`--chk-term`', forces the compiler to check the termination of all predicates in the module. It is common for the compiler to be unable to prove termination of some predicates and functions because they call other predicates which could not be proved to terminate or because they use language features (such as higher-order calls) which cannot be usefully analysed. In this case, the compiler only emits a warning for these predicates and functions if the '`--verbose-check-termination`' option is enabled. For every predicate or function that the compiler cannot prove the termination of, a warning message is emitted, but compilation continues. The '`--check-termination`' option implies the '`--enable-termination`' option.

The accuracy of the termination analysis is substantially degraded if intermodule optimization is not enabled. Unless intermodule optimization is enabled, the compiler must assume that any imported predicate may not terminate.

By default, the compiler assumes that a procedure defined using the foreign language interface will terminate for all input if it does not call Mercury. If it does call Mercury then by default the compiler will assume that it may not terminate.

The foreign code attributes '`terminates`'/'`does_not_terminate`' may be used to force the compiler to treat a foreign_proc as terminating/non-terminating irrespective of whether it calls Mercury. As a matter of style, it is preferable to use foreign code attributes for foreign_procs rather than the termination pragmas described below.

The following declarations can be used to inform the compiler of the termination properties of a predicate or function.

```
:- pragma terminates(Name/Arity).
```

This declaration may be used to inform the compiler that this predicate or function is guaranteed to terminate for any input. This is useful when the compiler cannot prove termination of some predicates or functions which are in turn preventing the compiler from proving termination of other predicates or functions. This declaration affects not only the predicate specified but also any other predicates that are mutually recursive with it.

```
:- pragma does_not_terminate(Name/Arity).
```

This declaration may be used to inform the compiler that this predicate may not terminate. This declaration affects not only the predicate specified but also any other predicates that are mutually recursive with it.

```
:- pragma check_termination(Name/Arity).
```

This pragma forces the compiler to prove termination of this predicate. If it cannot prove the termination of the specified predicate or function then the compiler will quit with an error message.

19.4 Feature sets

The University of Melbourne Mercury implementation supports a number of optional compilation model features, such as Section 19.5 [Trailing], page 152 or Section 19.2 [Tabled evaluation], page 146. Feature sets allow the programmer to assert that a module requires the presence of one or more optional features in the compilation model. These assertions can be made use a '`pragma require_feature_set`' declaration.

The '`require_feature_set`' pragma declaration has the following form:

```
:- pragma require_feature_set(Features).
```

where *Features* is a (possibly empty) list of features.

The supported features are:

'`concurrency`'
: This specifies that the compilation model must support concurrent execution of multiple threads.

'`single_prec_float`'
: This specifies that the compilation model must use single precision floats. This feature cannot be specified together with the '`double_prec_float`' feature.

‘`double_prec_float`’,
: This feature specifies that the compilation model must use double precision floats. This feature cannot be specified together with the ‘`single_prec_float`’ feature.

‘`memo`’
: This feature specifies that the compilation model must support memoisation (see Section 19.2 [Tabled evaluation], page 146).

‘`parallel_conj`’
: This feature specifies that the compilation model must support parallel execution of conjunctions. This feature cannot be specified together with the ‘`trailing`’ feature.

‘`trailing`’
: This feature specifies that the compilation model must support trailing, see Section 19.5 [Trailing], page 152. This feature cannot be specified together with the ‘`parallel_conj`’ feature.

‘`strict_sequential`’
: This feature specifies that a semantics that is equivalent to the strict sequential operational semantics must be used.

‘`conservative_gc`’
: This feature specifies that a module requires conservative garbage collection. This feature is only checked when using the C backends It is ignored by the non-C backends.

When a module containing a ‘`pragma require_feature_set`’ declaration is compiled, the implementation checks to see that the specified features are supported by the compilation model. It emits an error if they are not.

A ‘`pragma require_feature_set`’ may only occur in the implementation section of a module.

A ‘`pragma require_feature_set`’ affects only the module in which it occurs; in particular it does not affect any sub-modules

If a module contains multiple ‘`pragma require_feature_set`’ declarations then the implementation should emit an error if any of them specifies a feature that is not supported by the compilation model.

19.5 Trailing

In certain compilation grades (see the “Compilation model options” section of the Mercury User’s Guide), the University of Melbourne Mercury implementation supports trailing. Trailing is a means of having side-effects, such as destructive updates to data structures, undone on backtracking. The basic idea is that during forward execution, whenever you perform a destructive modification to a data structure that may still be live on backtracking, you should record whatever information is necessary to restore it on a stack-like data structure called the “trail”. Then, if a computation fails, and execution backtracks to before those updates were performed, the Mercury runtime engine will traverse the trail back to the most recent choice point, undoing all those updates.

The interface used is a set of C functions (which are actually implemented as macros) and types. Typically these will be called from C code within '`pragma foreign_proc`' or '`pragma foreign_code`' declarations in Mercury code.

For an example of the use of this interface, see the module '`extras/trailed_update/tr_array.m`' in the Mercury extras distribution.

19.5.1 Choice points

A "choice point" is a point in the computation to which execution might backtrack when a goal fails or throws an exception. The "current" choice point is the one that was most recently encountered; that is also the one to which execution will branch if the current computation fails.

When you trail an update, the Mercury engine will ensure that if execution ever backtracks to the choice point that was current at the time of trailing, then the update will be undone.

If the Mercury compiler determines that it will never need to backtrack to a particular choice point, then it will "prune" away that choice point. If a choice point is pruned, the trail entries for those updates will not necessarily be discarded, because in general they may still be necessary in case we backtrack to a prior choice point.

19.5.2 Value trailing

The simplest form of trailing is value trailing. This allows you to trail updates to memory and have the Mercury runtime engine automatically undo them on backtracking.

- `MR_trail_value()`

  Prototype:

  ```
  void MR_trail_value(MR_Word *address, MR_Word value);
  ```

  Ensures that if future execution backtracks to the current choice point, then *value* will be placed in *address*.

- `MR_trail_current_value()`

  Prototype:

  ```
  void MR_trail_current_value(MR_Word *address);
  ```

  Ensures that if future execution backtracks to the current choice point, the value currently in *address* will be restored.

  '`MR_trail_current_value(address)`' is equivalent to '`MR_trail_value(address, *address)`'.

Note that *address* must be word aligned for both `MR_trail_current_value()` and `MR_trail_value()`. (The address of Mercury data structures that have been passed to C via the foreign language interface are guaranteed to be appropriately aligned.)

19.5.3 Function trailing

For more complicated uses of trailing, you can store the address of a C function on the trail and have the Mercury runtime call your function back whenever future execution backtracks to the current choice point or earlier, or whenever that choice point is pruned,

because execution commits to never backtracking over that point, or whenever that choice point is garbage collected.

Note the garbage collector in the current Mercury implementation does not garbage-collect the trail; this case is mentioned only so that we can cater for possible future extensions.

- `MR_trail_function()`

  Prototype:

```
typedef enum {
        MR_undo,
        MR_exception,
        MR_retry,
        MR_commit,
        MR_solve,
        MR_gc
} MR_untrail_reason;

void MR_trail_function(
        void (*untrail_func)(void *, MR_untrail_reason),
        void *value
);
```

  A call to '`MR_trail_function(untrail_func, value)`' adds an entry to the function trail. The Mercury implementation ensures that if future execution ever backtracks to current choicepoint, or backtracks past the current choicepoint to some earlier choicepoint, then `(*untrail_func)(value, reason)` will be called, where *reason* will be '`MR_undo`' if the backtracking was due to a goal failing, '`MR_exception`' if the backtracking was due to a goal throwing an exception, or '`MR_retry`' if the backtracking was due to the use of the "retry" command in '`mdb`', the Mercury debugger, or any similar user request in a debugger. The Mercury implementation also ensures that if the current choice point is pruned because execution commits to never backtracking to it, then `(*untrail_func)(value, MR_commit)` will be called. It also ensures that if execution requires that the current goal be solvable, then `(*untrail_func)(value, MR_solve)` will be called. This happens in calls to `solutions/2`, for example. (`MR_commit` is used for "hard" commits, i.e. when we commit to a solution and prune away the alternative solutions; `MR_solve` is used for "soft" commits, i.e. when we must commit to a solution but do not prune away all the alternatives.)

  MR_gc is currently not used — it is reserved for future use.

Typically if the *untrail_func* is called with *reason* being '`MR_undo`', '`MR_exception`', or '`MR_retry`', then it should undo the effects of the update(s) specified by *value*, and then free any resources associated with that trail entry. If it is called with *reason* being '`MR_commit`' or '`MR_solve`', then it should not undo the update(s); instead, it may check for floundering (see the next section). In the '`MR_commit`' case it may, in some cases, be possible to also free resources associated with the trail entry. If it is called with anything else (such as '`MR_gc`'), then it should probably abort execution with an error message.

Note that the address of the C function passed as the first argument of `MR_trail_function()` must be word aligned.

19.5.4 Delayed goals and floundering

Another use for the function trail is check for floundering in the presence of delayed goals.

Often, when implementing certain kinds of constraint solvers, it may not be possible to actually solve all of the constraints at the time they are added. Instead, it may be necessary to simply delay their execution until a later time, in the hope the constraints may become solvable when more information is available. If you do implement a constraint solver with these properties, then at certain points in the computation — for example, after executing a negated goal — it is important for the system to check that their are no outstanding delayed goals which might cause failure, before execution commits to this execution path. If there are any such delayed goals, the computation is said to "flounder". If the check for floundering was omitted, then it could lead to unsound behaviour, such as a negation failing even though logically speaking it ought to have succeeded.

The check for floundering can be implemented using the function trail, by simply calling '`MR_trail_function()`' to add a function trail entry whenever you create a delayed goal, and putting the appropriate check for floundering in the '`MR_commit`' and '`MR_solve`' cases of your function. The Mercury extras distribution includes an example of this: see the '`ML_var_untrail_func()`' function in the file '`extras/trailed_update/var.m`'.) If your function does detect floundering, then it should print an error message and then abort execution.

19.5.5 Avoiding redundant trailing

If a mutable data structure is updated multiple times, and each update is recorded on the trail using the functions described above, then some of this trailing may be redundant. It is generally not necessary to record enough information to recover the original state of the data structure for *every* update on the trail; instead, it is enough to record the original state of each updated data structure just once for each choice point occurring after the data structure is allocated, rather than once for each update.

The functions described below provide a means to avoid redundant trailing.

- `MR_ChoicepointId`

  Declaration:

  ```
  typedef ... MR_ChoicepointId;
  ```

  The type `MR_ChoicepointId` is an abstract type used to hold the identity of a choice point. Values of this type can be compared using C's '`==`' operator or using '`MR_choicepoint_newer()`'.

- `MR_current_choicepoint_id()`

  Prototype:

  ```
  MR_ChoicepointId MR_current_choicepoint_id(void);
  ```

  `MR_current_choicepoint_id()` returns a value indicating the identity of the most recent choice point; that is, the point to which execution would backtrack if the current computation failed. The value remains meaningful if the choicepoint

is pruned away by a commit, but is not meaningful after backtracking past the point where the choicepoint was created (since choicepoint ids may be reused after backtracking).

- `MR_null_choicepoint_id()`

  Prototype:

  ```
  MR_ChoicepointId MR_null_choicepoint_id(void);
  ```

  `MR_null_choicepoint_id()` returns a "null" value that is distinct from any value ever returned by `MR_current_choicepoint_id`. (Note that `MR_null_choicepoint_id()` is a macro that is guaranteed to be suitable for use as a static initializer, so that it can for example be used to provide the initial value of a C global variable.)

- `MR_choicepoint_newer()`

  Prototype:

  ```
  bool MR_choicepoint_newer(MR_ChoicepointId, MR_ChoicepointId);
  ```

  `MR_choicepoint_newer(x, y)` true iff the choicepoint indicated by x is newer than (i.e. was created more recently than) the choicepoint indicated by y. The null ChoicepointId is considered older than any non-null ChoicepointId. If either of the choice points have been backtracked over, the behaviour is undefined.

The way these functions are generally used is as follows. When you create a mutable data structure, you should call `MR_current_choicepoint_id()` and save the value it returns as a '`prev_choicepoint`' field in your data structure. When you are about to modify your mutable data structure, you can then call `MR_current_choicepoint_id()` again and compare the result from that call with the value saved in the '`prev_choicepoint`' field in the data structure using `MR_choicepoint_newer()`. If the current choicepoint is newer, then you must trail the update, and update the '`prev_choicepoint`' field with the new value; furthermore, you must also take care that on backtracking the previous value of the '`prev_choicepoint`' field in your data structure is restored to its previous value, by trailing that update too. But if `MR_current_choice_id()` is not newer than the `prev_choicepoint` field, then you can safely perform the update to your data structure without trailing it.

If your mutable data structure is a C global variable, then you can use `MR_null_choicepoint_id()` for the initial value of the '`prev_choicepoint`' field. If on the other hand your mutable data structure is created by a predicate or function that uses tabled evaluation (see Section 19.2 [Tabled evaluation], page 146), then you *should* use `MR_null_choicepoint_id()` for the initial value of the field. Doing so will ensure that the data will be reset to its initial value if execution backtracks to a point before the mutable data structure was created, which is important because this copy of the mutable data structure will be tabled and will therefore be produced again if later execution attempts to create another instance of it.

For an example of avoiding redundant trailing, see the sample module below.

Note that there is a cost to this — you have to include an extra field in your data structure for each part of the data structure which you might update, you need to perform a test for each update to decide whether or not to trail it, and if you do need to trail the

update, then you have an extra field that you need to trail. Whether or not the benefits from avoiding redundant trailing outweigh these costs will depend on your application.

```
:- module trailing_example.
:- interface.

:- type int_ref.

    % Create a new int_ref with the specified value.
    %
:- pred new_int_ref(int_ref::uo, int::in) is det.

    % update_int_ref(Ref0, Ref, OldVal, NewVal).
    % Ref0 has value OldVal and Ref has value NewVal.
    %
:- pred update_int_ref(int_ref::mdi, int_ref::muo, int::out, int::in)
    is det.

:- implementation.

:- pragma foreign_decl("C", "

typedef struct {
    MR_ChoicepointId prev_choicepoint;
    MR_Integer data;
} C_IntRef;

").

:- pragma foreign_type("C", int_ref, "C_IntRef *").

:- pragma foreign_proc("C",
    new_int_ref(Ref::uo, Value::in),
    [will_not_call_mercury, promise_pure],
"
    C_Intref *x = malloc(sizeof(C_IntRef));
    x->prev_choicepoint = MR_current_choicepoint_id();
    x->data = Value;
    Ref = x;
").

:- pragma foreign_proc("C",
    update_int_ref(Ref0::mdi, Ref::muo, OldValue::out, NewValue::in),
    [will_not_call_mercury, promise_pure],
"
    C_IntRef *x = Ref0;
    OldValue = x->data;
```

```
    /* Check whether we need to trail this update. */
    if (MR_choicepoint_newer(MR_current_choicepoint_id(),
        x->prev_choicepoint))
    {
        /*
        ** Trail both x->data and x->prev_choicepoint,
        ** since we're about to update them both.
        */
        assert(sizeof(x->data) == sizeof(MR_Word));
        assert(sizeof(x->prev_choicepoint) == sizeof(MR_Word));
        MR_trail_current_value((MR_Word *)&x->data);
        MR_trail_current_value((MR_Word *)&x->prev_choicepoint);

        /*
        ** Update x->prev_choicepoint to indicate that
        ** x->data's previous value has been trailed
        ** at this choice point.
        */
        x->prev_choicepoint = MR_current_choicepoint_id();
    }
    x->data = NewValue;
    Ref = Ref0;
").
```

20 Bibliography

[1]

Chris Speirs, Zoltan Somogyi and Harald Sondergaard, *Termination Analysis for Mercury*. In P. Van Hentenryck, editor, *Static Analysis: Proceedings of the 4th International Symposium*, Lecture Notes in Computer Science. Springer, 1997. A longer version is available for download from http://www.mercurylang.org/documentation/papers/mu_97_09.ps.gz.

[2]

Gerhard Groger and Lutz Plumer, *Handling of mutual recursion in automatic termination proofs for logic programs.* In K. Apt, editor, *The Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 336–350. MIT Press, 1992.

[3]

Chris Speirs, *Termination Analysis for Logic Programs*, Technical Report 97/23, Department of Computer Science, The University of Melbourne, Melbourne, Australia, 1997. Available from http://www.mercurylang.org/documentation/papers/mu_97_23.ps.gz.

[4]

K. Sagonas, *The SLG-WAM: A Search-Efficient Engine for Well-Founded Evaluation of Normal Logic Programs*, PhD thesis, SUNY at Stony Brook, 1996. Available from http://user.it.uu.se/~kostis/Thesis/thesis.ps.gz.

[5]

B. Demoen and K. Sagonas, *CAT: the Copying Approach to Tabling*, In C. Palamidessi, H. Glaser and K. Meinke, editors, *Principles of Declarative Programming, 10th International Symposium, PLILP'98*, Lecture Notes in Computer Science, Springer, 1998. Available form http://user.it.uu.se/~kostis/Papers/cat.ps.gz.