# The Mercury Language Reference Manual

Version rotd-2026-06-28

**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

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

A Mercury program consists of a set of source files, each of which contains a module. A module consists of a sequence of tokens, each of which is a variable, name, literal, or punctuation symbol. Tokens may be separated by any amount of whitespace, comments, and line number directives. These separators are mostly ignored by the parser, but in some cases whitespace may be required to separate tokens that would otherwise be ambiguous. In other cases whitespace is not allowed, e.g. before the *open-ct* token, or after a '`.`' operator that would otherwise be interpreted as an *end* token.

## 2.2 Character set

Mercury program source files must be written using the UTF-8 encoding of the Unicode character set. In the rest of this chapter, "letters", "digits", "underscore" and other kinds of punctuation refer to characters in the Basic Latin code block.

## 2.3 Whitespace

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 Comments

The '`%`' character starts a comment that continues to the end of the line. The '`/*`' character sequence starts a comment that continues until the next occurrence of '`*/`'. For example:

```
% Calculate the answer.
Result = 42     % This is the answer!

/*
omit this declaration for now
:- mode append(out, in, in) is semidet.
*/
```

## 2.5 Line number directives

A line number directive consists of the character '`#`', a positive integer specifying the line number, and then a newline. Line number directives set the current line number; they are used in conjunction with the '`pragma source_file`' declaration (see Section 20.7 [Source file name], page 165) to indicate that errors in the Mercury code following the directive

should be reported relative to the line number set by the directive. This is useful if the code in question was generated by another tool, in which case the line number can be set to the corresponding location in the original source file from which the Mercury code was derived. The Mercury compiler can thereby issue more informative error messages using locations in the original source file. 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.

## 2.6 Variables

A variable is an uppercase letter or underscore followed by zero or more letters, underscores, and digits. For example, '`Sum`', '`_NotNeeded`', '`_a`', and '`_123`' are variables, whereas '`x`' and '`_#`' are not. A variable token consisting of a single underscore is treated specially: each instance of '`_`' denotes a distinct variable.

Variables starting with an uppercase letter are expected to occur more than once; the compiler will issue a warning if this is not the case, as it often indicates a simple error. Variables starting with an underscore are presumed to be "don't-care" variables; the compiler will issue a warning if a variable starting with an underscore, excluding single underscore variables, occurs more than once in the same scope.

## 2.7 Names

A name token is either an unquoted name, a quoted name, a graphic name, or a single semicolon character. 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. A graphic name is a sequence of one or more of the following characters

```
! & * + - : < = > ? @ ^ ~ \ # $ . /
```

where the first character is not '`#`'.

As a special case, the character sequences '`<<u`' and '`>>u`' are also graphic names. (They are intended to denote left and right shifts by unsigned amounts respectively.)

An unquoted name, graphic name, or semicolon is treated as equivalent to a quoted name containing the same sequence of characters.

## 2.8 Literals

The different literals in Mercury are as follows.

*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, '`\e`' for escape, '`\f`' for form-feed, '`\n`' for newline, '`\r`' for carriage-return, '`\t`' for tab, '`\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 sequences '`\u`' and '`\U`' begin a Unicode escape. '`\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.)

*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; 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, with one exception: underscores may occur between a digit and an ‘`e`’ or ‘`E`’ that introduces the exponent part of the number. The purpose of the underscores is to improve readability; 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.

## 2.9 Punctuation symbols

The following punctuation symbols are used in Mercury's syntax.

*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, ‘`}`’.

*backquote* A backquote character, ‘`` ` ``’.

*ht-sep* A “head-tail separator”, i.e. a vertical bar, ‘`|`’.

*comma* A comma, ‘`,`’.

*end* A full stop (period), ‘`.`’.

## 2.10 Operators

An operator is either a builtin operator or a user-defined operator. A user-defined operator is a name, module qualified name (see Section 10.1 [The module system], page 83), or variable, enclosed in backquotes (grave accents). User-defined operators are left-associative infix operators that bind more strongly than most other operators (see below).

The builtin operators, with the exception of comma, are all names, and as such they can be used without arguments supplied. For example, ‘`f(+)`’ is syntactically valid. In some cases parentheses may be required to limit the scope of an operator without arguments, e.g. if it appears as an argument to another operator. The comma operator is not a name and therefore requires single quotes in order to be used without arguments. An operator in single quotes is still an operator, so any requirement for parentheses will remain unchanged.

Operators are a syntactic concept. The ‘`+`’ infix operator, for example, is only a symbol; it does not mean addition, unless you write or import code that defines it as addition. Modules in the Mercury standard library, such as `int`, `uint` and `float`, provide such arithmetic

definitions. Other, non-arithmetic definitions can also be provided, for example, the '`-`' infix operator is defined as subtraction by those modules but is defined as a pair constructor by the `pair` module.

The following table lists all of Mercury's builtin operators, as well as user-defined operators of the form '*`op`*'. Operators with a higher priority bind more tightly than those with a lower priority. (This is a recent change; previously, Mercury followed the Prolog tradition in using higher priorities to denote operators that bind *less* tightly.) For example, given that `+` has priority 1000 and `*` has priority 1100, the term `2 * X + Y` parenthesises as `(2 * X) + Y`. Note that the module qualification operator, '`.`', binds more tightly than any other operator. Therefore, operator terms using builtin operators need to be parenthesized in order to be module qualified, for example, integer subtraction can be written as '`int.(A - B)`' whereas pair construction can be written as '`pair.(A - B)`'. (See Section 10.1 [The module system], page 83).

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 higher than that of the operator. "y" represents an argument whose priority is higher than 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 | 1490 |
| `!` | fx | 1460 |
| `!.` | fx | 1460 |
| `!:` | fx | 1460 |
| `@` | xfx | 1410 |
| `^` | xfy | 1401 |
| `^` | fx | 1400 |
| `event` | fx | 1400 |
| `:` | yfx | 1380 |
| '*`op`*' | yfx | 1380 |
| `**` | xfy | 1300 |
| `-` | fx | 1300 |
| `\` | fx | 1300 |
| `*` | yfx | 1100 |
| `/` | yfx | 1100 |
| `//` | yfx | 1100 |
| `<<` | yfx | 1100 |
| `<<u` | yfx | 1100 |
| `>>` | yfx | 1100 |
| `>>u` | yfx | 1100 |
| `div` | yfx | 1100 |
| `mod` | xfx | 1100 |
| `rem` | xfx | 1100 |
| `for` | xfx | 1000 |

| | | |
|---|---|---|
| + | fx | 1000 |
| + | yfx | 1000 |
| ++ | xfy | 1000 |
| - | yfx | 1000 |
| -- | yfx | 1000 |
| /\ | yfx | 1000 |
| \/ | yfx | 1000 |
| .. | xfx | 950 |
| := | xfx | 850 |
| =^ | xfx | 850 |
| < | xfx | 800 |
| = | xfx | 800 |
| =.. | xfx | 800 |
| =:= | xfx | 800 |
| =< | xfx | 800 |
| == | xfx | 800 |
| =\= | xfx | 800 |
| > | xfx | 800 |
| >= | xfx | 800 |
| @< | xfx | 800 |
| @=< | xfx | 800 |
| @> | xfx | 800 |
| @>= | xfx | 800 |
| \= | xfx | 800 |
| \== | xfx | 800 |
| ~= | xfx | 800 |
| is | xfx | 799 |
| and | xfy | 780 |
| or | xfy | 760 |
| func | fx | 700 |
| impure | fy | 700 |
| pred | fx | 700 |
| semipure | fy | 700 |
| \+ | fy | 600 |
| not | fy | 600 |
| when | xfx | 600 |
| ~ | fy | 600 |
| <= | xfy | 580 |
| <=> | xfy | 580 |
| => | xfy | 580 |
| all | fxy | 550 |
| arbitrary | fxy | 550 |
| atomic | fxy | 550 |
| disable_warning | fxy | 550 |
| disable_warnings | fxy | 550 |
| promise_equivalent_solutions | fxy | 550 |
| promise_equivalent_solution_sets | fxy | 550 |

| | | |
|---|---|---|
| promise_exclusive | fy | 550 |
| promise_exclusive_exhaustive | fy | 550 |
| promise_exhaustive | fy | 550 |
| promise_impure | fx | 550 |
| promise_pure | fx | 550 |
| promise_semipure | fx | 550 |
| require_complete_switch | fxy | 550 |
| require_switch_arms_det | fxy | 550 |
| require_switch_arms_semidet | fxy | 550 |
| require_switch_arms_multi | fxy | 550 |
| require_switch_arms_nondet | fxy | 550 |
| require_switch_arms_cc_multi | fxy | 550 |
| require_switch_arms_cc_nondet | fxy | 550 |
| require_switch_arms_erroneous | fxy | 550 |
| require_switch_arms_failure | fxy | 550 |
| require_det | fx | 550 |
| require_semidet | fx | 550 |
| require_multi | fx | 550 |
| require_nondet | fx | 550 |
| require_cc_multi | fx | 550 |
| require_cc_nondet | fx | 550 |
| require_erroneous | fx | 550 |
| require_failure | fx | 550 |
| trace | fxy | 550 |
| try | fxy | 550 |
| some | fxy | 550 |
| , | xfy | 500 |
| & | xfy | 475 |
| -> | xfy | 450 |
| ; | xfy | 400 |
| or_else | xfy | 400 |
| then | xfx | 350 |
| if | fx | 340 |
| else | xfy | 330 |
| :: | xfx | 325 |
| ==> | xfx | 325 |
| where | xfx | 325 |
| ---> | xfy | 321 |
| catch | xfy | 320 |
| type | fx | 320 |
| solver | fy | 319 |
| catch_any | xfy | 310 |
| end_module | fx | 301 |
| import_module | fx | 301 |
| include_module | fx | 301 |
| initialise | fx | 301 |
| initialize | fx | 301 |

| | | |
|---|---|---|
| `finalise` | `fx` | `301` |
| `finalize` | `fx` | `301` |
| `inst` | `fx` | `301` |
| `instance` | `fx` | `301` |
| `mode` | `fx` | `301` |
| `module` | `fx` | `301` |
| `pragma` | `fx` | `301` |
| `promise` | `fx` | `301` |
| `rule` | `fx` | `301` |
| `typeclass` | `fx` | `301` |
| `use_module` | `fx` | `301` |
| `-->` | `xfx` | `300` |
| `:-` | `fx` | `300` |
| `:-` | `xfx` | `300` |
| `?-` | `fx` | `300` |

## 2.11 Terms

Terms are the basic construct used in Mercury syntax. The term syntax is summarized by the following rules. (All of this information can be found in the descriptions below the rules.)

*term* = *core-term* | *special-term*

*core-term* = *variable* | *literal* | *functor-term*

*literal* = *string* | *integer* | *float* | *implementation-defined-literal*

*functor-term* = *name* | *name open-ct functor-args close*

*functor-args* = *functor-arg* | *functor-arg* ‘`,`’ *functor-args*

*functor-arg* = *arg* | *arg* ‘`::`’ *arg*

*args* = *arg* | *arg* ‘`,`’ *args*

*arg* = *term*, where the term is not an operator term with priority `>=` 1000

*special-term* = *operator-term* | *list-term* | *tuple-term* | *apply-term* | *paren-term*

*operator-term* = *term operator term* | *operator term* | *operator term term*,
where the term is constructed according to the requirements of the operator
(see Section 2.10 [Operators], page 5)

*list-term* = ‘`[`’ *list-body*? ‘`]`’

*list-body* = *arg* | *arg* ‘`,`’ *list-body* | *arg ht-sep term*

*tuple-term* = '`{`' *args*? '`}`'

*apply-term* = *term open-ct args close*,
where the term is not a name or operator term

*paren-term* = '`(`' *term* '`)`'

Terms can be described in the following way.

*term* A term is either a *core term* or a *special term*. A term normalization procedure, given below, translates terms that may contain special terms into terms that are only constructed from core terms; two terms are considered syntactically equivalent if they translate to the same term. Syntactically equivalent terms can be used interchangeably anywhere in a module (e.g. operator syntax can be used in declarations and clauses, in particular those that define an operator).

Note that there can be further equivalences in some contexts, e.g. an if-then-else can be written in either of two equivalent forms. Such equivalences will be covered in the relevant chapters.

*core-term* A core term is a *variable*, a *literal*, or a *functor-term*.

*literal* A literal is a *string*, an *integer*, a *float*, or an *implementation-defined-literal*.

*functor-term*
A functor term is either a name or a compound term. A compound term is a name followed without any intervening whitespace by an open parenthesis (i.e. an *open-ct* token), then followed by a functor argument list and a close parenthesis. E.g., '`foo(X,Y)`' is a compound term, whereas '`foo (X,Y)`' and '`foo()`' are not (the first because the space after '`foo`' is not allowed, the second because the parentheses must be omitted if there are no arguments).

The *principal functor* of a functor term is the name and arity of the term, separated by a slash, where the arity is the number of arguments (or zero if there are no arguments). For example, the principal functor of '`foo(bar,baz)`' is '`foo/2`', while the principal functor of '`foo`' is '`foo/0`'. The principal functor of a special term is determined *after* term normalization. For module qualified terms, the principal functor is defined slightly differently (see Section 10.1 [The module system], page 83).

Note that the word "functor" has a number of definitions, but in Mercury it just means a symbol to which arguments can be applied, and which has no intrinsic meaning of its own. It is a syntactic concept that applies to all functor terms. In specific contexts functors may also be referred to as type constructors, data constructors (or just constructors), predicates, functions, etc. The principal functor may also be referred to as the "top-level constructor".

*functor-args*
A functor argument list is a sequence of one or more functor arguments, separated by commas.

*functor-arg*
: A functor argument is either a single argument or two arguments separated by a '`::`' operator (the latter form is for mode qualifiers; see Section 5.4 [Different clauses for different modes], page 59).

*args*
: An argument list is a sequence of one or more arguments, separated by commas.

*arg*
: An argument is any term, except operator terms where the operator does not bind more tightly than comma (i.e. where the priority is greater than or equal to 1000). In such a situation parentheses can be used, e.g. '`f((A,B))`' is a compound term with one argument that is a parenthesized operator term, whereas '`f(A,B)`' is a compound term with two arguments (and no operators).

*special-term*
: A special term is an operator term, a list term, a tuple term, an apply term, or a parenthesized term. The term normalization procedure, below, defines how these terms are represented internally as core terms.

*operator-term*
: An operator term is a term constructed using an operator, which complies with the rules for constructing terms using that operator (see Section 2.10 [Operators], page 5). Operator terms can be infix, such as '`A + B`', unary-prefix, such as '`not P`', or binary-prefix, such as '`some Vars Goal`'.

*list-term*
: A list term is an open square bracket (an *open-list* token), followed by an optional list body, followed by a close square bracket (a *close-list* token). If the list body is omitted it is the empty list. If present, the list body is an argument list, optionally followed by a vertical bar (a *ht-sep* token) followed by a term. E.g., '`[]`', '`[X]`', and '`[1, 2 | Tail]`' are all list terms. The argument list gives the elements appearing at the front of the list. The term following the vertical bar, if present, gives the *tail* of the list (i.e. the remaining elements), otherwise the tail is the empty list. Note that technically the tail does not have to be a list for this to be syntactically valid, although generally it would need to be in order to be type correct.

*tuple-term*
: A tuple term is an open curly bracket (an *open-curly* token), followed by an optional argument list, followed by a close curly bracket (a *close-curly* token). If the argument list is omitted it is the empty tuple, otherwise the arguments give the components of the tuple. E.g., `{}` and `{1,'2',"three"}` are tuple terms.

*apply-term*
: An apply-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 (an *open-ct* token), an argument list, and a close parenthesis (a *close* token). E.g., '`A(B,C)`' is an apply-term. An apply-term represents the closure (i.e. a higher-order value) applied to the arguments.

  Note that although the closure term cannot be an operator term, it *can* be a parenthesized term. Thus '`(Var ^ foo)(Arg1, Arg2)`' is a valid apply-term, whereas '`Var ^ foo(Arg1, Arg2)`' is not (it is an operator term whose second argument is a compound term).

*paren-term*

A parenthesized term is just a term enclosed in parentheses. E.g., `(X-Y)` is a parenthesized term.

The term normalization procedure works by rewriting special terms that occur anywhere within a term (i.e. at the top level or as some descendant) according to a set of rewriting rules, and repeating until no rules can be further applied. The rules are as follows.

```
term1 `name` term2 ↦ name(term1, term2)
term1 `var` term2 ↦ var(term1, term2)
term1 operator term2 ↦ 'operator'(term1, term2)
operator term ↦ 'operator'(term)
operator term1 term2 ↦ 'operator'(term1, term2)

[ ] ↦ '[]'
[ arg ] ↦ '[|]'(arg, '[]')
[ arg , list-body ] ↦ '[|]'(arg, [list-body])
[ arg | term ] ↦ '[|]'(arg, term)

{ } ↦ '{}'
{ args } ↦ '{}'(args)

term(args) ↦ ''(term, args)

( term ) ↦ term
```

For example, the following terms are all syntactically equivalent (i.e. they are equal after term normalization). The last is constructed from core terms; the others all normalize to this term. The last one shows that the principal functor of all of them is `'[|]'/2`.

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

Similarly, the following terms are all syntactically equivalent. The principal functor in this case is `'+'/2`.

```
A * B + C
(A * B) + C
'+'('*'(A, B), C)
```

## 2.12 Items

Mercury modules are parsed as a sequence of *items*. Each item is a term followed by an *end* token (a period). If the principal functor of the term is '`:-/1`', it is a declaration item and the argument is the *declaration*. Otherwise it is a clause item and the term is the *clause*. Note that we often use "declaration" and "clause" informally to refer to the items themselves (i.e. including the *end* token).

Declarations are used in relation to a number of features. Details of their syntax are covered in the relevant chapters.

A clause provides part of the definition of a function or predicate, and takes one of the following forms. The first form is a DCG-rule and is not discussed further here (see Section 3.8 [Definite clause grammars], page 34).

```
DCG_Head --> DCG_Body.

Head :- Body.
Head.
```

*Head* is the *head* of the clause and *Body*, if present, is the *body* of the clause. If the principal functor is '`:-/2`', the clause is a *rule* and the body is a goal (see Section 3.2 [Goals], page 16). If the principal functor is not '`:-/1`', '`:-/2`', or '`-->/2`', the clause is a *fact*. A fact is equivalent to a rule that has the same head and a body of '`true`'.

A clause head takes one of the following forms.

```
FunctorTerm = Result
FunctorTerm
```

*`FunctorTerm`* is a functor term whose arguments are expressions (see Section 3.3 [Expressions], page 23), optionally annotated with mode qualifiers (see Section 5.4 [Different clauses for different modes], page 59). If the principal functor is '`=/2`', then the clause is a function rule or a function fact, and *`Result`* is an expression, optionally annotated with a mode qualifier. Otherwise, the clause is a predicate rule or a predicate fact. The principal functor of *`FunctorTerm`* determines which function or predicate is being defined.

For example, the following three items are clauses. The first is a function fact that defines a function named '`loop/1`', a not particularly useful function. The second is a predicate fact and the third is a predicate rule, that between them define a predicate named '`append/3`'.

```
loop(X) = 1 + loop(X).

append([], Bs, Bs).
append([X | As], Bs, [X | Cs]) :-
    append(As, Bs, Cs).
```

The following example contains a number of declaration and clause items, and forms a syntactically valid module. (The semantics of the clauses will be covered in the next chapter. Note that the `length/1` function in the standard library is implemented more efficiently.)

```
:- module slow_length.
:- interface.
:- import_module list.

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

:- implementation.
:- import_module int.       % for '+'

length([]) = 0.
length([_ | Xs]) = 1 + length(Xs).

:- end_module slow_length.
```

# 3 Clauses

This chapter covers the semantics of Mercury clauses, and the goals and expressions they contain. The first section gives an informal overview of the semantics; if you are already familiar with logic programming you may wish to skip it and start with Section 3.2 [Goals], page 16.

Full details of the language semantics can be found in Chapter 15 [Formal semantics], page 115.

## 3.1 Overview of Mercury semantics

There is no agreed upon definition of "declarative programming". One notable characteristic of Mercury as a declarative language, however, is that it has both a *declarative* and an *operational* semantics. The declarative semantics is conceptually the simpler of the two: it is only concerned with the relationship between inputs and outputs, and not the steps taken to execute a program. The operational semantics is additionally concerned with these steps. This is often expressed by saying that the declarative semantics is about "what" whereas the operational semantics is about "how".

In the remainder of this section we introduce each of these semantics.

### Declarative semantics

The declarative semantics is concerned with "truth". For example, it is true that 1 plus 1 is 2, and that the length of the list [1, 2, 3] is 3. Statements that are either true or false like this are known as *propositions*, e.g. `1 + 1 = 2` and `1 + 2 = 5` are both propositions; if `+` is interpreted as integer addition then the first proposition is true and the second is false.

Mercury clauses state things that are true about the function or predicate being defined. To illustrate we will use an example from the previous chapter. (Note that, here and below, some declarations would need to be added to make this compile.)

```
length([]) = 0.
length([_ | Xs]) = 1 + length(Xs).
```

Both of these clauses are facts about the function `length/1`. The first simply states that the length of an empty list is zero. The second states that no matter what expressions we substitute for the variables '`Xs`' and '`_`', the length of '`[_ | Xs]`' will be one greater than the length of '`Xs`'. In other words, the length of a non-empty list is one greater than the length of its tail.

These two statements are true according to our intuitive idea of length. Furthermore, we can see that the clauses cover every possible list, since every list is either empty or non-empty, and every non-empty list has a tail that is also a list. Perhaps surprisingly, this is enough to conclude that our implementation of list length is correct, at least as far as its arguments and return values are concerned.

As another example, the following clauses define a predicate named `append/3`, which is intended to be true if the third argument is the list that results from appending the first and second arguments. (Equivalently, we could say that it is intended to be true if it is possible to split the list in the third argument to produce the first and second arguments.)

```
append([], Bs, Bs).
```

```
append([X | As], Bs, [X | Cs]) :-
    append(As, Bs, Cs).
```

The first clause is a fact that states if you append the empty list and any other list, the result will be the same as that other list. The second clause is a rule; these are taken as logical implications in which the body implies the head (i.e. '`:-`' is interpreted as reverse implication). So this is stating that, for any substitution, if '`Cs`' is the result of appending '`As`' and '`Bs`', then '`[X | Cs]`' is the result of appending '`[X | As]`' and '`Bs`'.

Again, both clauses are true according to the *intended interpretation*, which is defined as all of the propositions about the functions and predicates in the program that the programmer intends to be true. And the definition is *complete*, meaning that for every proposition that is intended to be true there is either a fact that covers it, or a rule whose head covers it and (under the same substitution) whose body is intended to be true. Thus we can conclude in a similar way to above that our code is correct.

The declarative semantics of a Mercury program is defined as all of the propositions that can be inferred to be true from the clauses of the program (with some additional axioms). If the program is producing incorrect output, this implies that there is a difference between the declarative semantics and the intended interpretation. From the above discussion, there must be some clause that is false in the intended interpretation, or some definition that is incomplete.

This is the reason for having a declarative semantics. Despite it being relatively simple—you only need to know about which propositions are true and which are false, and not how the program actually executes—it is still effective in reasoning about your program, even so far as to be able to localize a bug observed in the output down to individual clauses or definitions.

## Operational semantics

The declarative semantics does not tell us whether our program will terminate, for example, or what its computational complexity is. For that we need the operational semantics, which tells us how the program will be executed.

Execution in Mercury starts with a *goal*, which is a proposition that may contain some variables. The aim of execution is to find a substitution for which the proposition is true. If it does, we refer to this as *success*, and we refer to the substitution that was found as a *solution*. If execution determines that there are no such substitutions, we refer to this as *failure*.

Say, for example, we start with a goal of '`N = length([1, 2])`'. Function evaluation is strict, depth-first, and left-to-right, so we want to call the '`length/1`' function first. To do this, we match the argument with the heads of the clauses that define the function to find the clause that is applicable. In this case the second clause matches, with the substitution of `Xs` ↦ `[2]` (the substitution for '`_`' is irrelevant, since any other occurrence of '`_`' is considered a distinct variable). Applying this substitution to the body then replacing it in the goal gives us a new goal, namely '`N = 1 + length([2])`'.

Repeating this process a second time gives us the goal '`N = 1 + 1 + length([])`'. When we call the function the third time it will match the *first* clause, and the new goal will be '`N = 1 + 1 + 0`'. At this point we can evaluate the '`+/2`' calls and get a result of '`N = 2`'. It is trivial to find a substitution that makes this proposition true: just map `N` to the literal `2`.

Now consider the goal '`append(As, Bs, [1])`'. In this case the first two arguments are *free*, meaning that they are variables that are not mapped to anything in the current substitution, and the third argument is *ground*, meaning that it does not contain any variables after applying the current substitution. In this case when we try to match (or *unify*) the goal with a clause, both clauses match. We arbitrarily pick the first one, but we also push a *choice point* onto a stack, which will allow us to return to this point later on and try the other clause if we need to. Matching with the first clause gives us the substitution `As` ↦ `[]`, `Bs` ↦ `[1]`; since this clause is a fact, we succeed with this substitution as our solution.

If a later goal fails, we pop the previous choice point off the stack in order to search for a different solution. This time we want to try unifying our goal with the head of the second clause, that is, we want to find a substitution such that

```
append(As, Bs, [1]) = append([X1 | As1], Bs1, [X1 | Cs1])
```

(the variables from the clause have been given a numerical suffix, which is to indicate that they came from a different scope and are not the same variables as those in the goal). The substitution we use is `As` ↦ `[1 | As1]`, `Bs` ↦ `Bs1`, `X1` ↦ `1`, `Cs1` ↦ `[]`; you can check that this does indeed unify the two terms. Note that information is effectively flowing in both directions: variables from both the goal and the clause (i.e. the caller and callee) are bound by this substitution. This is different from pattern matching in many other languages, in which only variables in the pattern are bound.

Applying this substitution to the body of the selected clause gives us our new goal, '`append(As1, Bs1, [])`'. Only the first clause matches, with the substitution of `As1` ↦ `[]`, `Bs1` ↦ `[]`, and the clause is a fact, so this is a solution to *this* call to append. To find the solution to the parent goal we compose this substitution with the one from before, giving `As` ↦ `[1]`, `Bs` ↦ `[]`. We have now found two solutions for our goal: one with `As` being the empty list and `Bs` being `[1]`, and the other with `As` being `[1]` and `Bs` being the empty list. These are all of the possible solutions; if we wanted to search for another we would find the choice point stack to be empty, hence we would fail.

## 3.2 Goals

A goal is a term that takes one of the following forms.

*`Call`* Any goal which is a functor term that does not match any of the other forms below is a first-order predicate call. The principal functor determines the predicate called, which must be visible (see Chapter 10 [Modules], page 83). The arguments, if present, are expressions.

```
call(Closure)
call(Closure1, Arg1)
call(Closure2, Arg1, Arg2)
call(Closure3, Arg1, Arg2, Arg3)
...
Closure
Closure1(Arg1)
Closure2(Arg1, Arg2)
Closure3(Arg1, Arg2, Arg3)
```

`...` A higher-order predicate call. The closure and arguments are expressions. '`call(Closure)`' and '`Closure`' just call the specified closure. The other forms append the specified arguments onto the argument list of the closure before calling it. A higher-order predicate call written using an apply term with `N` arguments is equivalent to the form using `call/N+1`. See Chapter 9 [Higher-order], page 75.

*`Expr1 = Expr2`*
: A unification. *Expr1* and *Expr2* are expressions.

`!`*`Var ^ field_list`* `:= Expr`
: A state variable field update. See Section 3.4 [State variables], page 26.

*`Goal1, Goal2`*
: A conjunction. *Goal1* and *Goal2* are goals.

*`Goal1 & Goal2`*
: A parallel conjunction. *Goal1* and *Goal2* are goals. This has the same declarative semantics as 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 7.1 [Determinism categories], page 63.

*`Goal1 ; Goal2`*
: A disjunction, where *Goal1* is not of the form '*`Goal1a -> Goal1b`*' (since that would make it an if-then-else, below). *Goal1* and *Goal2* are goals.

`true` The empty conjunction. Always succeeds exactly once.

`fail` The empty disjunction. Always fails.

`if` *`CondGoal`* `then` *`ThenGoal`* `else` *`ElseGoal`*
*`CondGoal -> ThenGoal ; ElseGoal`*
: An if-then-else. The two different syntaxes have identical semantics. *CondGoal*, *ThenGoal*, and *ElseGoal* are 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 is different, and it is treated differently for the purposes of determinism inference (see Chapter 7 [Determinism], page 63). Operationally, it executes the *CondGoal*, and if that succeeds, then execution continues with the *ThenGoal*; otherwise, i.e. if *CondGoal* fails without producing any solutions, it executes the *ElseGoal*. Note that *CondGoal* can be nondeterministic—if the *CondGoal* succeeds more than once then the *ThenGoal* is executed once for each of the solutions.

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* (see existential quantifications, below). 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.

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

*`Expr1`* `\=` *`Expr2`*
: An inequality. *Expr1* and *Expr2* are expressions. This is an abbreviation for '`not (`*`Expr1 = Expr2`*`)`'.

`try` *`Params Goal`* `...` `catch` *`Expr -> 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 Expr -> CatchGoal
...
catch_any CatchAnyVar -> CatchAnyGoal
```

See Chapter 14 [Exception handling], page 112 for the full details.

`some` *`Vars Goal`*
: An existential quantification. *Goal* is a goal and *Vars* is a list whose elements are either variables or state variables (a single list may contain both). The case where there are state variables is described in Section 3.4 [State variables], page 26; here we discuss the case where they are all plain variables.

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 3.6 [Implicit quantification], page 33) mean that variables are often implicitly existentially quantified. There is usually no need to write existential quantifiers explicitly.

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

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

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

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

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

`promise_impure` *`Goal`*
: A purity cast. *Goal* is a 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* is a list of variables and *Goal* is a 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 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.

`arbitrary` *`ArbVars ArbGoal`*
: Goals of this form are only allowed to occur inside `promise_equivalent_solution_sets` *`Vars Goal`* goals. See the preceding description for details.

`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* is a 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. *Var* is a variable and *Goal* is a goal. 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. *Var* is a variable and *Goal* is a 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 [`*`Warnings`*`] `*`Goal`*
`disable_warning [`*`Warnings`*`] `*`Goal`*

*Goal* is a goal and *Warnings* is a comma-separated list of names. The Mercury compiler can generate warnings about several kinds of constructs 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 [`*`Warnings`*`] `*`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 `[`*`Warnings`*`]`.

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

`singleton_vars`
: Disable the generation of warnings for variables that occur only once despite their names not starting with an underscore.

`repeated_singleton_vars`
: Disable the generation of warnings for variables that occur more than once despite their names starting with an underscore.

`suspected_occurs_check_failure`
: Disable the generation of warnings about code that looks like it unifies a variable with a term that contains that same variable.

`suspicious_recursion`
: Disable the generation of warnings about suspicious recursive calls.

`no_solution_disjunct`
: Disable the generation of warnings about disjuncts that can have no solution. This is usually done to shut up such a warning in a multi-mode predicate where the disjunct in question is a switch arm in another mode. (The difference is that a disjunct that cannot succeed has no meaningful use, but a switch arm that cannot succeed does have one: a switch may need that arm to make it complete.)

`unknown_format_calls`
: Disable the generation of warnings about calls to `string.format`, `io.format` or `stream.string_writer.format` for which the compiler cannot tell whether there are any mismatches between the format string and the supplied values.

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 warning category.

`trace` *`Params Goal`*
: A trace goal, typically used for debugging or logging. *Goal* is a goal and *Params* is a 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 effect 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 19 [Trace goals], page 156 for the details.

`event` *`Goal`*
: An event goal. *Goal* is 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.

## 3.3 Expressions

Syntactically, an expression is just a term. Semantically, an expression is a variable, a literal, a functor expression, or a special expression. A special expression is a conditional expression, a unification expression, a state variable, an explicit type qualification, a type conversion expression, a lambda expression, an apply expression, or a field access expression.

A literal is a string, an integer, a float, or an implementation-defined literal (note that character literals are just single character names; see below).

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 literals during compilation. Implementation-defined literals can only appear within clauses. The following 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.

A functor expression is a name or a compound expression. A compound expression is a compound term that does not match the form of a special expression, and whose arguments are expressions. If a functor expression is not a character literal, its principal functor must be the name of a visible function, predicate, or data constructor (except for field specifiers, for which the corresponding field access function must be visible; see below).

Character literals in Mercury are single character names, possibly quoted. Since they sometimes require quotes and sometimes require parentheses, for code consistency we recommend writing all character literals with quotes and (except where used as arguments) parentheses. For example, `Char = ('+') ; Char = ('''')`.

Special expressions (not including field access expressions, which are covered below) take one of the following forms.

`if` *Goal* `then` *ThenExpr* `else` *ElseExpr*
*Goal* `->` *ThenExpr* `;` *ElseExpr*

A conditional expression. *Goal* is a goal; *ThenExpr* and *ElseExpr* are both expressions. The two forms are equivalent. The meaning of a conditional expression is that if *Goal* is true it is equivalent to *ThenExpr*, otherwise it is equivalent to *ElseExpr*.

If *Goal* takes the form `some [X, Y, Z] ...` then the scope of *X*, *Y*, and *Z* includes *ThenExpr*. See the related discussion regarding if-then-else goals.

`X @ Y`
: A unification expression. *X* and *Y* are both expressions. 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 15 [Formal semantics], page 115) 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(Z, X) :-
    Z = X,
    Z = f(_, _).
```

  Unification expressions are particularly useful when writing switches (see Section 7.2 [Determinism checking and inference], page 65), as the arguments of a unification expression are examined when checking for switches. The arguments of an equivalent user-defined function would not be.

`!.S`
`!:S`
: A state variable. *S* is a variable. See Section 3.4 [State variables], page 26.

*Expr* `:` *Type*
: An explicit type qualification. *Expr* is an expression, and *Type* is a type (see Chapter 4 [Types], page 37). An explicit type qualification constrains the specified expression to have the specified type; these expressions are occasionally useful to resolve ambiguities that can arise from polymorphic types or overloading. Apart from that, the explicit type qualification is equivalent to *Expr*.

  Currently we also support *Expr* `` `with_type` `` *Type* as an alternative syntax for explicit type qualification.

`coerce(Expr)`
: A type conversion expression. *Expr* is an expression. See Chapter 13 [Type conversions], page 109.

```
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
```
: A lambda expression. *Arg1*, *Arg2*, ... are zero or more expressions, *Result* is an expression, *Goal* is a goal, *DCGGoal* is a DCG-goal, *Mode1*, *Mode2*, ..., *DCGMode0*, and *DCGMode1* are modes (see Chapter 5 [Modes], page 53), and *Det* is a determinism category (see Chapter 7 [Determinism], page 63). 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 9 [Higher-order], page 75.

A lambda expression introduces a new scope: any variables occurring in the arguments *Arg1*, *Arg2*, ... are locally quantified, i.e. they are distinct from other 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 3.6 [Implicit quantification], page 33).

The form of lambda expression using '`-->`' as its top level functor is a syntactic abbreviation. It 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 `transform(DCGVar0, DCGVar1, DCGGoal)` where `transform` is the function specified in Section 3.8 [Definite clause grammars], page 34.

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

An apply expression (i.e. a higher-order function call). *N* >= 0, *Func* is an expression of type '`func(T1, T2, ..., Tn) = T`', and *Arg1*, *Arg2*, ..., *ArgN* are expressions of types '`T1`', '`T2`', ..., '`Tn`'. The type of the apply expression is *T*. It denotes the result of applying the specified function to the specified arguments. See Chapter 9 [Higher-order], page 75.

## Field access expressions

Field access expressions provide a convenient way to select or update fields of data constructors, independent of the definition of the constructor. The compiler transforms field access expressions into sequences of calls to field selection or update functions (see Section 4.4 [Field access functions], page 48).

A field specifier is a functor expression. A field list is a sequence of field specifiers separated by `^` (circumflex). E.g., '`field`', '`field1 ^ field2`' and '`field1(A) ^ field2(B, C)`' are all field lists.

If the principal 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)`.

Field access expressions have one of the following forms. There are also DCG goals for field access (see Section 3.8 [Definite clause grammars], page 34), which provide similar functionality to field access expressions, except that they act on the DCG arguments of a DCG clause.

*`Expr ^ field_list`*

A field selection. *Expr* is an expression and *field_list* is a field list. For each field specifier in *field_list*, apply the corresponding selection function in turn.

A field selection is transformed using the following rules:

```
transform(Expr ^ field(Arg1, ...)) = field(Arg1, ..., Expr).
transform(Expr ^ field(Arg1, ...) ^ Rest) =
                    transform(field(Arg1, ..., Expr) ^ Rest).
```

*Expr* ^ *field_list* := *FieldExpr*

A field update. *Expr* and *FieldExpr* are expressions and *field_list* is a field list. Returns a copy of *Expr* with the value of the field specified by *field_list* replaced with *FieldExpr*.

A field update is transformed using the following rules:

```
transform(Expr ^ field(Arg1, ...) := FieldExpr) =
                'field :='(Arg1, ..., Expr, FieldExpr).

transform(Expr0 ^ field(Arg1, ...) ^ Rest := FieldExpr) = Expr :-
        OldFieldValue = field(Arg1, ..., Expr0),
        NewFieldValue = transform(OldFieldValue ^ Rest := FieldExpr),
        Expr = 'field :='(Arg1, ..., Expr0, NewFieldValue).
```

Examples:

```
transform(Expr ^ field) = field(Expr).

transform(Expr ^ field(Arg)) = field(Arg, Expr).

transform(Expr ^ field1(Arg1) ^ field2(Arg2, Arg3)) =
    field2(Arg2, Arg3, field1(Arg1, Expr)).

transform(Expr ^ field := FieldExpr) = 'field :='(Expr, FieldExpr).

transform(Expr ^ field(Arg) := FieldExpr) = 'field :='(Arg, Expr, FieldExpr).

transform(Expr0 ^ field1(Arg1) ^ field2(Arg2) := FieldExpr) = Expr :-
    OldField1 = field1(Arg1, Expr0),
    NewField1 = 'field2 :='(Arg2, OldField1, FieldExpr),
    Expr = 'field1 :='(Arg1, Expr0, NewField1).
```

## 3.4 State variables

Clauses may use *state variables* as a shorthand for naming intermediate values in a sequence. For example, the following clauses

```
main(IO0, IO) :-
    write_string("The answer is ", IO0, IO1),
    write_int(42, IO1, IO2),
    nl(IO2, IO).
```

could be written equivalently using state variable syntax as

```
main(!IO) :-
    write_string("The answer is ", !IO),
    write_int(42, !IO),
    nl(!IO).
```

One advantage of doing this is that if in future more operations need to be added in the middle of the sequence, the state variables will not need to be renumbered.

A state variable is written '`!.`*X*' or '`!:`*X*', denoting the "current" or "next" value of the sequence labelled *X*. A predicate argument '`!`*X*' is shorthand for two state variable

arguments '`!.X, !:X`'; that is, '`p(..., !X, ...)`' is equivalent to '`p(..., !.X, !:X, ...)`'. The variables '`!.X`' and '`!:X`' are referred to as the current and next components of '`!X`', respectively. Note that, since predicate arguments of the form '`!X`' stand for two arguments, the arity of a predicate may be greater than it appears. E.g. '`p(!X)`' is a call to the predicate `p/2`.

State variables obey special scope rules. A state variable $X$ must be explicitly introduced, and this can happen in one of four ways:

- As '`!X`' in the head of a predicate clause. In this case, references to state variable '`!X`' or to its components may appear in the clause body.
- As either '`!.X`' or '`!:X`' or both in the head of a predicate or function clause. Again, in this case, references to state variable '`!X`' or to its components may appear in the clause body.
- As either '`!.X`' or '`!:X`' or both in the head of a lambda expression. In this case, references to state variable '`!X`' or to its components may appear in the lambda expression body. (The reason that '`!X`' may not appear in the head of a lambda expression is that there is no syntax for specifying the modes of the two implied parameters.)
- In an explicit quantification such as '`some [!X] Goal`'. In this case, references to state variable '`!X`' or to its components may appear in '`Goal`'.

Only the current component of a state variable $X$ in the enclosing scope of a lambda or if-then-else expression may be referred to (unless the enclosing $X$ is shadowed by a more local state variable of the same name.)

For instance, the following clause employs a lambda expression and is illegal because it implicitly refers to the next component, '`!:S`', inside the lambda expression.

```
p(A, B, !S) :-
    P = ( pred(C::in, D::out) is det :-
            q(C, D, !S)
        ),
    ( if P(A, E) then
        B = E
    else
        B = A
    ).
```

However

```
p(A, B, !S) :-
    P = ( pred(C::in, D::out, !.S::in, !:S::out) is det :-
            q(C, D, !S)
        ),
    ( if P(A, E, !S) then
        B = E
    else
        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 component of a local state variable.

There are two restrictions concerning state variables in functions, whether they are defined by clauses or lambda expressions.

- '`!`*X*' is not a legal function result, because it stands for two arguments, rather than one.
- Neither '`!`*X*' nor '`!:`*X*' may appear as an argument in a function application, because this would not make sense given the usual interpretation of state variables and functions. (The default mode of functions is that all arguments are input, while in typical usage, '`!:`*X*' is output.)

Within each clause, the compiler replaces each occurrence of !*X* in an argument list with two arguments: !.*X*, !:*X*, where !.*X* represents the current version of the state of !*X*, and !:*X* represents its next state. It then replaces all occurrences of !.*X* and !:*X* with ordinary variables in a way that (in the general case) represents a sequence of updates to the state of *X* from an initial state to a final state.

This replacement is done by code that is equivalent to the '`transform_goal`' and '`transform_clause`' functions below. The basic operation used by these functions is substitution: '`substitute(`*Goal*`, [!.`*X*` ->` *CurX*`, !:`*X*` ->` *NextX*`])`' stands for a copy of *Goal* in which every free occurrence of '`!.`*X*' is replaced with *CurX*, and every free occurrence of '`!:`*X*' is replaced with *NextX*. (A free occurrence is one not bound by the head of a clause or lambda, or by an explicit quantification.)

The '`transform_goal(`*Goal*`, `*X*`, `*CurX*`, `*NextX*`)`' function's inputs are

- the goal to transform *Goal*,
- the name of the state variable *X*,
- and the ordinary variables *CurX* and *NextX* representing the current and next versions of that state variable.

It returns a transformed version of *Goal*.

'`transform_goal`' has a case for each kind of Mercury goal. These cases are as follows.

`Calls`
: Given a first order call such as '`predname(`*Arg1*`, ..., `*ArgN*`)`' or a higher-order call such as '`Expr(`*Arg1*`, ..., `*ArgN*`)`', if any of the arguments is !*X*, '`transform_goal`' replaces that argument with two arguments: !.*X* and !:*X*. It then checks whether '`!:`*X*' appears in the updated *Call*.

  - If it does, then it replaces *Call* with

    ```
    substitute(Call, [!.X -> CurX, !:X -> NextX])
    ```

  - If it does not, then it replaces *Call* with

    ```
    substitute(Call, [!.X -> CurX]),
    NextX = CurX
    ```

  Note that !.*X* can occur in *Call* on its own (i.e. without !:*X*). Likewise, !:*X* can occur in *Call* without !.*X*, but this does not need separate handling.

  The expression *Expr* in a higher-order call may not be of the form !*X*, !.*X* or !:*X*. It may be parenthesised as (!.*X*), however.

`Unifications`
: In a unification '`ExprA = ExprB`', each of *ExprA* and *ExprB* are expressions that may have one of the following four forms:

- The expression may be !.*S* for some state variable *S*. If *S* is *X*, then '`transform_goal`' replaces the expression with *CurX*.
- The expression may be !:*S* for some state variable *S*. If *S* is *X*, then '`transform_goal`' replaces the expression with *NextX*.
- The expression may be a name, a constant, or a variable that is not a state variable, '`transform_goal`' leaves such expressions unchanged.
- The expression may be a compound term, which means that it must have the form '`f(ArgTerm1, ..., ArgTermN)`'. '`transform_goal`' handles these the same way it handles function applications.

Note that *ExprA* and *ExprB* may not have the form !*S*.

`State variable field updates`

A state variable field update goal has the form

```
!S ^ field_list := Expr
```

where *field_list* is a valid field list See [Field access expressions], page 25. This means that

```
!S ^ field1 := Expr
!S ^ field1 ^ field2 := Expr
!S ^ field1 ^ field2 ^ field3 := Expr
```

are all valid field update goals. If *S* is *X*, '`transform_goal`' replaces such goals with

```
NextX = CurX ^ field_list := Expr
```

Otherwise, it leaves the goal unchanged.

`Conjunctions`

Given a nonempty conjunction, whether a sequential conjunction such as *Goal1*, *Goal2* or a parallel conjunction such as *Goal1* & *Goal2*, '`transform_goal`'

- creates a fresh variable *MidX*,
- replaces *Goal1* with

```
substitute(Goal1, [!.X -> CurX, !:X -> MidX])
```

- replaces *Goal2* with

```
substitute(Goal2, [!.X -> MidX, !:X -> NextX])
```

This implies that first *Goal1* updates the state of *X* from *CurX* to *MidX*, and then *Goal2* updates the state of *X* from *MidX* to *NextX*.

Given the empty conjunction, i.e. the goal '`true`', '`transform_goal`' will replace it with

```
NextX = CurX
```

`Disjunctions`

Given a disjunction such as *Goal1* ; *Goal2*, '`transform_goal`'

- replaces *Goal1* with

```
substitute(Goal1, [!.X -> CurX, !:X -> NextX])
```

- replaces *Goal2* with

```
substitute(Goal2, [!.X -> CurX, !:X -> NextX])
```

This shows that both disjuncts start with the *CurX*, and both end with *NextX*. If a disjunct has no update of !*X*, then the value of *NextX* in that disjunct will be set to *CurX*.

The empty disjunction, i.e. the goal '`fail`', cannot succeed, so what the value of !:*X* would be if it *did* succeed is moot. Therefore '`transform_goal`' returns empty disjunctions unchanged.

`Negations`

Given a negated goal of the form '`not NegatedGoal`', '`transform_goal`'

- creates a fresh variable *DummyX*, and then
- replaces '`not NegatedGoal`' with

```
'not' substitute('NegatedGoal', [!.X -> CurX, !:X -> Dum-
myX]),
NextX = CurX
```

It does this because negated goals may not generate any outputs visible from the rest of the code, which means that any output they *do* generate must be local to the negated goal.

Negations that use '`\+ NegatedGoal`' notation are handled exactly the same way.

`If-then-elses`

Given an if-then-else, whether it uses ( if *Cond* then *Then* else *Else* ) syntax or ( *Cond* -> *Then* ; *Else* ) syntax, '`transform_goal`'

- creates a fresh variable *MidX*,
- replaces *Cond* with

```
substitute(Cond, [!.X -> CurX, !:X -> MidX])
```

- replaces *Then* with

```
substitute(Then, [!.X -> MidX, !:X -> NextX])
```

- replaces *Else* with

```
substitute(Else, [!.X -> CurX, !:X -> NextX])
```

This effectively treats an if-then-else as being a disjunction, with the first disjunct being the conjunction of the *Cond* and *Then* goals, and the second disjunct being the *Else* goal. (The *Else* goal is implicitly conjoined inside the second disjunct with the negation of the existential closure of *Cond*, since the else case is executed only if the condition has no solution.)

`Bidirectional implications`

'`transform_goal`' treats a bidirectional implication goal, which has the form *GoalA* `<=>` *GoalB*, as if it were the conjunction of its two constituent unidirectional implications: *GoalA* `=>` *GoalB*, *GoalA* `<=` *GoalB*.

`Unidirectional implications`

'`transform_goal`' treats a unidirectional implication, which has one of the two forms '`GoalA => GoalB`' and '`GoalB <= GoalA`', as if they were written as '`not (GoalA, not GoalB)`'.

`Universal quantifications`
: '`transform_goal`' treats universal quantifications, which have the form '`all` *`Vars SubGoal`*' as if they were written as '`not (some` *`Vars`* `(not` *`SubGoal`*`))`'. Note that in universal quantifications, *Vars* must be a list of ordinary variables.

`Existential quantifications`
: In existential quantifications, which have the form '`some` *`Vars SubGoal`*', *Vars* must be a list, in which every element must be either an ordinary variable (such as *A*), or a state variable (such as !*B*). (Note that *Vars* may not contain any element whose form is !.*B* or !:*B*.)

  - If *Vars* does not contain !*X*, then '`transform_goal`' will replace *SubGoal* with

    ```
    substitute(SubGoal, [!.X -> CurX, !:X -> NextX])
    ```

  - If *Vars* does contain !*X*, then '`transform_goal`' will leave *SubGoal* unchanged, because any references to !.*X*, !:*X* and !*X* inside *SubGoal* refer to the state variable *X* introduced by this scope, not the one visible outside. Effectively, this state variable *X* *shadows* the one visible outside.

  Note that state variables in *Vars* are handled by '`transform_clause`' below.

The '`transform_clause`' function's input is a clause, which may be a non-DCG clause or a DCG clause, which have the forms

```
predname(ArgTerm1, ..., ArgTermN) :- BodyGoal.
```

and

```
predname(ArgTerm1, ..., ArgTermN) --> BodyGoal.
```

respectively. '`transform_clause`' handles both the same way.

- While any of the *ArgTerms* has one of the forms !.*X*, !:*X* and !*X*,
  - '`transform_clause`' will create two fresh variables, *InitX* and *FinalX*,
  - it will replace any one of the *ArgTerms* that is !.*X* with *InitX*, any one of the *ArgTerms* that is !:*X* with *FinalX*, and any one of the *ArgTerms* that is !*X* with the argument pair *InitX*, *FinalX*, and
  - it will replace *BodyGoal* with the result of '`transform_goal(`*`BodyGoal, X, InitX, FinalX`*`)`'.
- While *BodyGoal* contains a lambda expression whose argument list contains either !.*X* or !:*X* or both:
  - '`transform_clause`' will create two fresh variables, *InitX* and *FinalX*,
  - it will replace any one of the arguments that is !.*X* with *InitX*, and any one of the arguments that is !:*X* with *FinalX* (there may not be any argument that is !*X*), and
  - it will replace the lambda goal *BodyGoal* with the result of '`transform_goal(`*`BodyGoal, X, InitX, FinalX`*`)`'.
- While *BodyGoal* contains an existential quantification goal '`some` *`Vars SubGoal`*' where *Vars* contains a state variable such as !*B*,
  - '`transform_clause`' will create two fresh variables, *InitB* and *FinalB*,

- it will replace *SubGoal* with the result of '`transform_goal(SubGoal, B, InitB, FinalB)`', and then
- it will delete !*B* from *Vars*.

Actual application of this transformation would, in the general case, result in the generation of many different versions of each state variable, for which we need more names than just '*CurX*', '*MidX*' and '*NextX*'. The Mercury compiler therefore uses

- '*STATE_VARIABLE_X_0*' as the initial value of a state variable,
- '*STATE_VARIABLE_X_N*', where '*N*' is a nonzero positive integer, as its intermediate values, and
- '*STATE_VARIABLE_X*' as its final value.

This transformation can lead to the introduction of chains of unifications for variables that do not otherwise play a role in the definition, such as '*STATE_VARIABLE_X_5 = STATE_VARIABLE_X_6, STATE_VARIABLE_X_6 = STATE_VARIABLE_X_7, STATE_VARIABLE_X_7 = STATE_VARIABLE_X_8*'. Where possible, the compiler automatically shortcircuits such sequences by removing any unneeded intermediate variables. In the above case, this would yield '*STATE_VARIABLE_X_5 = STATE_VARIABLE_X_8*'.

The following code fragments illustrate some appropriate uses 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(Test, Update, !A, !B) :-
    ( if Test(!.A, !.B) then
        Update(!A, !B),
        iterate_while2(Test, Update, !A, !B)
    else
        true
    ).
```

**Introducing state**

```
compute_out(InA, InB, InC, Out) :-
    some [!State]
    (
        init_state(!:State),
        update_state_a(InA, !State),
```

```
        update_state_b(InB, !State),
        list.foldl(update_state_c, InC, !State),
        compute_output(!.State, Out)
    ).
```

## 3.5 Variable scoping

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

Variables occurring in types, type classes, and instances are called type variables. Variables occurring in insts or modes are called inst variables. Variables that occur in expressions, and that are not inst variables or type variables, are called ordinary variables.

Note that type variables can occur in expressions in the right-hand (*Type*) operand of an explicit type qualification. Inst variables can occur in expressions in the right-hand (*Mode*) operand of an explicit mode qualification. Apart from that, all other variables in expressions 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 3.2 [Goals], page 16) or implicitly (see Section 3.6 [Implicit quantification], page 33).

The scope of type variables in a predicate or function's type declaration extends over any explicit type qualifications (see Section 3.3 [Expressions], page 23) in the clauses for that predicate or function, and over '`pragma type_spec`' (see Section 20.2 [Type specialization], page 159) 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.

## 3.6 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 term 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 negated context.

## 3.7 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. See Chapter 5 [Modes], page 53.)

## 3.8 Definite clause grammars

A definite clause grammar (DCG) is a way of expressing parsing rules, and is intended for writing parsers and sequence generators. In the past it has also been used to thread an implicit state variable, typically the I/O state, through code. We now recommend that DCGs only be used for writing parsers and sequence generators, and that state variable syntax (see Section 3.4 [State variables], page 26), which performs a similar transformation but is more flexible, be used for other purposes such as threading state variables.

A DCG-rule is a clause that takes the form

```
DCG_Head --> DCG_Body.
```

where *DCG_Head* is a predicate head term and *DCG_Body* is a DCG-goal. It is an abbreviation for the rule

```
Head :- Body.
```

where *Head* is *DCG_Head* with two fresh variables, `V_in` and `V_out`, appended to the argument list, and *Body* is `transform(V_in, V_out, DCG_Body)`, where `transform` is the function defined below.

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

`some` *`Vars DCG-goal`*
: A DCG existential quantification. *Vars* is a list of variables and *DCG-goal* is a DCG-goal.

`all` *`Vars DCG-goal`*
: A DCG universal quantification. *Vars* is a list of variables and *DCG-goal* is a DCG-goal.

*`DCG-goal1, DCG-goal2`*
: A DCG sequence. *DCG-goal1* and *DCG-goal2* are DCG-goals. 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* `;` *DCG-goal2*
: A disjunction. *DCG-goal1* and *DCG-goal2* are DCG-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.)

`{` *Goal* `}`
: A brace-enclosed ordinary goal. *Goal* is a goal.

`[`*Expr*`, ...]`
: A DCG input match. *Expr* is an expression. Unifies the implicit DCG input variable V_in, which must have type '`list(_)`', with a list whose initial elements are the expressions specified and whose tail is the implicit DCG output variable V_out.

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

`not` *DCG-goal*
`\+` *DCG-goal*
: A DCG negation. *DCG-goal* is a DCG-goal. The two different syntaxes have identical semantics.

`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* are DCG-goals.

`=(`*Expr*`)`
: A DCG unification. *Expr* is an expression. Unifies *Expr* with the implicit DCG argument.

`:=(`*Expr*`)`
: A DCG output unification. *Expr* is an expression. Unifies *Expr* with the implicit DCG output argument, ignoring the input DCG argument.

*Expr* `=^` *field_list*
: A DCG field selection. *Expr* is an expression and *field_list* is a field list. Unifies *Expr* with the result of applying the field selection *field_list* to the implicit DCG argument. See [Field access expressions], page 25.

`^` *field_list* `:=` *Expr*
: A DCG field update. *Expr* is an expression and *field_list* is a field list. Replaces a field in the implicit DCG argument. See [Field access expressions], page 25.

*DCG-call*
: A term which does not match any of the above forms is a DCG predicate call. If *DCG-call* 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.

The semantics is given by the following function, where each occurrence of *V_new* is a fresh variable.

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

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

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

```
    (transform(V_in, V_new, DCG_goal1), transform(V_new, V_out, DCG_goal2))

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

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

transform(V_in, V_out, [Expr, ...]) = (V_in = [Expr, ... | V_out])

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

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

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

transform(V_in, V_out, =(Expr)) = (Expr = V_in, V_out = V_in)

transform(V_in, V_out, :=(Expr)) = (V_out = Expr)

transform(V_in, V_out, Expr =^ field_list) =
    (Expr = V_in ^ field_list, V_out = V_in)

transform(V_in, V_out, ^ field_list := Expr) =
    (V_out = V_in ^ field_list := Expr)

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

# 4 Types

The type system is based on many-sorted logic, and supports polymorphism, type classes (see Chapter 11 [Type classes], page 91), and existentially quantified types (see Chapter 12 [Existential types], page 101).

## 4.1 Builtin types

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

### 4.1.1 Primitive types

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

#### 4.1.1.1 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 $-(2^{N-1})$ to $2^{N-1}-1$, both inclusive; $N$ being the width of `int` in bits.

#### 4.1.1.2 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 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.

#### 4.1.1.3 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.

#### 4.1.1.4 Character type

There is one character type: `char`.

Values of this type represent Unicode code points.

#### 4.1.1.5 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, and using UTF-16 when generating code for C# or Java.

### 4.1.2 Other builtin types

#### 4.1.2.1 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 9 [Higher-order], page 75.

#### 4.1.2.2 Tuple types

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

A tuple type is equivalent to a discriminated union type (see Section 4.2.1 [Discriminated unions], page 39) with declaration

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

#### 4.1.2.3 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 conversion from `univ` to the original type will check that the value inside the `univ` has the expected type. The universal type is useful for situations where you need heterogeneous collections.

#### 4.1.2.4 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.

## 4.2 User-defined types

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

### 4.2.1 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 semicolons.

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*. (The reverse is not the case: a variable occurring in the *type* need not also occur in the *body*. Such variables are called '`phantom type parameters`', and their use is explained below.)

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 12 [Existential types], page 101. Existentially typed discriminated union definitions need not be transparent.

The arguments of constructor definitions may be named. These names 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 (see Section 4.4 [Field access functions], page 48). A named argument has the form *`fieldname`* `::` *`Type`*. It is an error for two fields in the same type definition to have the same name, even if the fields they name occur in different data constructors.

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 semicolon 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 semicolon. 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
''
```

Phantom type parameters are useful when you have two distinct concepts that you want to keep separate, but for which nevertheless you want to use the same representation. This is an example of their use, taken from the Mercury compiler:

```
:- type var(T)
        --->    ...

:- type prog_var == var(prog_var_type).
:- type type_var == var(type_var_type).

:- type prog_var_type
    --->    prog_var_type.
:- type type_var_type
    --->    type_var_type.
```

The `var` type represents the generic notion of a variable. It has a phantom type parameter, `T`, which does not occur in the body of its type definition. The `prog_var` and `type_var` types represent two different specific kinds of variables: program variables, which occur in code (clauses), and type variables, which occur in types, respectively. They each bind a different type to the type parameter `T`. These types, `prog_var_type` and `type_var_type`, each have a single function symbol of arity zero. This means that each type has only one value, which in turn means that values of these types contain no information at all. But containing information is not the purpose of these types. Their purpose is to ensure that

if a computation that expects program variables is ever accidentally given type variables, or vice versa, this mismatch is detected and reported by the compiler. Two variables can be unified only if they have the same type. While two `prog_var`s have the same type, and two `type_var`s have the same type, a `prog_var` and `type_var` have different types, due to having different types (`prog_var_type` and `type_var_type`) being bound to the phantom type parameter `T`.

### 4.2.2 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.

### 4.2.3 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.

### 4.2.4 Subtypes

(This is a new and experimental feature, subject to change.)

A subtype is a discriminated union type that is a subset of a supertype, in that every term of a subtype is a valid term in the supertype. It is possible to convert terms between subtype and supertype using type conversion expressions (see Chapter 13 [Type conversions], page 109).

As previously described, the syntax for non-subtype discriminated union types is

```
:- type type ---> body.
```

where *type* is the name of a type constructor applied to zero or more distinct type variables (the *parameters* of the type constructor), and *body* is a sequence of constructor definitions separated by semicolons. All universally quantified type variables that occur in *body* must be among *type*'s parameters.

The syntax for subtypes is similar but slightly different:

```
:- type subtype =< supertype ---> body.
```

Since a subtype is also a discriminated union type, the rules for discriminated union types apply to them as well: *subtype* must be the name of a type constructor applied to zero or more distinct type variables (its parameters), *body* must be a sequence of constructor definitions separated by semicolons, and all universally quantified type variables that occur in *body* must be among *subtype*'s parameters.

*supertype* must be a type constructor applied to zero or more argument types, which *may* be type variables, but they do not have to be, and if they are, do not need to be distinct. After expanding out equivalences, *supertype*'s principal type constructor must specify a discriminated union type whose definition is in scope where the subtype definition occurs, by normal module visibility rules.

The discriminated union type specified by *supertype* may itself be a subtype. Following the chain of subtype definitions, it must be possible to arrive at a *base type*, which is a discriminated union type but *not* a subtype.

The body of the subtype may differ from the body of its supertype in two ways.

- It may omit one or more constructor definitions. The ability to do this is the main motivation for the use of subtypes.

  Since the subtype cannot *add* definitions of constructors, the set of constructor definitions in the subtype must be a subset of the constructor definitions in the supertype. We recommend that they should appear in the same relative order as in the supertype definition.

- It may change the types of some of the arguments of some of the constructors, *provided* that each replacement replaces a type with one of its subtypes.

  Formally, this means that if the supertype '`t`' has a constructor '`f(T1, ..., Tn)`', and the subtype '`s =< t`' has a constructor '`f(S1, ..., Sn)`', then for each *Si*, the condition '`Si =< Ti`' must hold, where '`=<`' is the subtype relation below.

This is an example of the first kind of difference:

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

:- type citrus_fruit =< fruit
    --->    lemon
    ;       orange.
```

And this is an example of the second:

```
:- type fruit_basket
    --->    basket(fruit, int).
            % What kind of fruit, and how many.

:- type citrus_fruit_basket =< fruit_basket
    --->    basket(citrus_fruit, int).
```

(There are more examples below.)

If the subtype retains a constructor from the supertype that has one or more existentially quantified type variables, then the subtype constructor must repeat the list of existentially quantified type variables from the supertype constructor, and all existential class constraints, with no additions, removals, or reordering. (The type variables do not need to have the same names in the subtype as the supertype, but, stylistically, it makes more sense if they do.)

As mentioned above, any universally quantified type variable that occurs in *body* must occur also in *subtype*. However, this is the only restriction on the list of parameters in *subtype*. For example, it need not have any particular relationship with the list of parameters of the principal type constructor of *supertype*. For example, *subtype* may have a phantom type parameter (see Section 4.2.1 [Discriminated unions], page 39) that does not occur in *supertype*.

(In the following discussion, we assume that all equivalence types have been expanded out.)

The subtype relation '`S =< T`' has four cases to consider: when '`S`' and '`T`' are both discriminated union types, when they are both tuple types, when they are both higher-order types, and all other types.

1. For discriminated union types '`S`' and '`T`':

- If '`S`' and '`T`' have the same principal type constructor, say '`f/n`', which implies that '`S = f(S1, ..., Sn)`' and '`T = f(T1, ..., Tn)`', then '`S =< T`' holds if and only if for all *i* in '`1..n`', '`Si =< Ti`'.
- If '`S`' and '`T`' have different principal type constructors, and if '`S = f(S1, ..., Sn)`', '`S =< T`' holds if
  - there is a visible subtype definition starting with '`:- type f(R1, ..., Rn) =< U`',
  - for all *i* in '`1..n`', '`Si = Ri`' (unification), and
  - '`U =< T`'.

  In other words, if all occurrences of *Ri* in *U* are replaced by the corresponding *Si* to give *Usub*, then '`Usub =< T`' must hold.

2. For two tuple types '`S = {S1, ..., Sn}`' and '`T = {T1, ..., Tn}`', '`S =< T`' holds if and only if '`Si =< Ti`' for all '`i`' in '`1..n`'. This is analogous to the case for discriminated union types with the same principal type constructor.

3. A higher-order type '`S`' can be a subtype of another higher-order type '`T`' in only one way. Since subtype definitions do not apply to higher-order types, this way is analogous to the case for discriminated union types with the same principal type constructor.

- '`P =< Q`' holds for two higher-order types *P* and *Q* if and only if all of the following conditions hold:
  - *P* and *Q* are either both '`pred`' types, or both '`func`' types,
  - they have the same arity,
  - *P* and *Q* have the same argument types (the current implementation does not allow subtyping in higher-order arguments), and
  - if either of *P* and *Q* has higher-order inst information, then *P* and *Q* must have the *same* higher-order inst information, i.e. their higher-order inst information must specify the same argument modes, determinism, and purity.

4. For all other types, '`S =< T`' if and only if '`S = T`', i.e. they are syntactically identical.

A subtype may be exported as an abstract type by declaring only the name of the subtype in the interface section of a module (without the '`=< supertype`' part). Then the subtype definition must be given in the implementation section of the same module.

Example:

```
:- interface.

:- type non_empty_list(T).  % abstract type

:- implementation.

:- import list.

:- type non_empty_list(T) =< list(T)
    --->    [T | list(T)].
```

Subtypes must not have user-defined equality or comparison predicates. The base type of a subtype may have user-defined equality or comparison. In that case, values of the subtype will be tested for equality or compared using those predicates.

There is no special interaction between subtypes and the type class system.

Some more examples of subtypes:

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

:- type non_empty_list(T) =< list(T)
    --->    [T | list(T)].

:- type non_empty_list_of_foo =< list(foo)
    --->    [foo | list(foo)].

:- type maybe_foo
    --->    none
    ;       some [T] foo(T) => fooable(T).

:- type foo =< maybe_foo
    --->    some [T] foo(T) => fooable(T).

:- type task
   --->     create(pred(int::in, io::di, io::uo) is det)
   ;        delete(pred(int::in, io::di, io::uo) is det).

:- type create_task =< task
   --->     create(pred(int::in, io::di, io::uo) is det).
```

And one more complex example.

In the case of a set of mutually recursive types, omitting some constructor definitions from a type may not be enough; it may be necessary to replace some argument types with their subtypes as well. Consider this pair of mutually recursive types representing a bipartite graph, i.e. a graph in which there are two kinds of nodes, and edges always connect two nodes of different kinds. In this bipartite graph, the two kinds of nodes are *or* nodes and *and* nodes, and each kind of node can be connected to zero, two or more nodes of the other kind.

```
:- type or_node
    --->    or_source(source_id)
    ;       logical_or(and_node, and_node)
    ;       logical_or_list(and_node, and_node, and_node, list(and_node)).

:- type and_node
    --->    and_source(source_id)
    ;       logical_and(or_node, or_node)
    ;       logical_and_list(or_node, or_node, or_node, list(or_node)).
```

If we wanted a subtype to represent graphs in which no *or* node could be connected to more than two *and* nodes, one might think that it would be enough to delete the *logical_or_list* constructor from the *or_node* type, like this:

```
:- type binary_or_node =< or_node
    --->    or_source(source_id)
    ;       logical_or(and_node, and_node).
```

However, this would not work, because the *and_node*s have constructors whose arguments have type *or_node*, not *binary_or_node*. One would have to create a subtype of the *and_node* type that constructs *and* nodes from *binary_or_node*s, not plain *or_node*s, like this:

```
:- type binary_or_node =< or_node
    --->    or_source(source_id)
    ;       logical_or(binary_or_and_node, binary_or_and_node).

:- type binary_or_and_node =< and_node
    --->    and_source(source_id)
    ;       logical_and(binary_or_node, binary_or_node)
    ;       logical_and_list(binary_or_node, binary_or_node, binary_or_node,
                list(binary_or_node)).
```

## 4.3 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 9 [Higher-order], page 75) 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 11 [Type classes], page 91), 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).
:- func 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 12 [Existential types], page 101.

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 16 [Foreign language interface], page 117). 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 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 2.12 [Items], page 12) 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 expressions, 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 expression with an explicit type qualification (see Section 3.3 [Expressions], page 23) 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 the 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.

## 4.4 Field access functions

Fields of constructors of discriminated union types may be named (see Section 4.2.1 [Discriminated unions], page 39). These names 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 [Field access expressions], page 25) and to select and update fields of the DCG arguments of a clause (see Section 3.8 [Definite clause grammars], page 34).

[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 expressions to be identical, without knowing the exact types of the expressions. It also allows type qualifications to refer to the types of the results of existentially typed predicates or functions.

### 4.4.1 Field selection

*field*(*Term*)

Each field name '*field*' in a constructor tells the compiler to generate a field selection function '*field*/1', which takes an expression of the same type as the constructor and returns the value of the named 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 value from a field selection function, an explicit lambda expression must be used, unless a single mode declaration is supplied for the field selection function. The reason for this is that normally, field access functions are implemented directly as unifications, without the code of a function being generated for them. The declaration acts as the request for the generation of that code.

### 4.4.2 Field update

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

Each field name '*field*' in a constructor tells the compiler to generate a field update function ''*field* :='/2'. The first argument of this function is an expression of the same type as the constructor. The second argument is an expression of the same type as the named field. The return value is a copy of the first argument with the value of the named field replaced by the second argument. ''*field* :='/2' fails if the top-level constructor of the first argument is not the constructor containing the named 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 value from a field update function, users must write an explicit lambda expression, unless a single mode declaration is supplied for the field update function. The reason for this is that normally, as with field selection functions, field update functions are implemented directly as unifications, without the code of a function being generated for them. The declaration acts as the request for the compiler to generate that 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
                ).
```

### 4.4.3 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. The user-supplied declarations will be used instead of any automatically generated declarations. 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.

If a field occurs in the interface section of a module, then any declaration for a field access function for that field must also occur in the interface section.

If there are multiple fields with the same name in the same module, only one of those fields can have user-supplied declarations for its selection function. Similarly, only one of those fields can have user-supplied declarations for its update function.

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.

### 4.4.4 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 [Field access expressions], page 25, 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.
```

## 4.5 The standard ordering

For (almost) 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 one exception is higher-order types, which cannot be unified or compared; any attempt to do so will raise an exception.

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, and the C# `System.String.CompareOrdinal()` method, when compiling to C, Java and C# respectively.

For tuple types, corresponding arguments are compared, with the first argument being the most significant, then the second, and so on.

For discriminated union types (other than subtypes), if both values have the same principal constructor then corresponding arguments are compared in order, with the first argument being the most significant, then the second, and so on. If the values have different principal constructors, then the value whose principal constructor is listed first in the definition of the type will compare as less than the other value. There is one exception from this rule: in types that are subject to a `foreign_enum` pragma, the outcomes of comparisons are decided by the user's chosen foreign language representations, using the rules of the foreign language.

For subtypes, the two values compare as though converted to the base type. The ordering of constructors in a subtype definition does not affect the standard ordering.

# 5 Modes

## 5.1 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 18 [Solver types], page 152 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(free).
```

which is equivalent to writing

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

You can specify what type (actually what type constructor) an inst is intended to be used on by adding `for`, followed by the name and arity of that type constructor, after the name of the inst, like this:

```
:- inst maybeskel for maybe/1
    --->    no
    ;       yes(free).
```

This can be useful documentation, and the compiler will generate an error message when an inst that was declared to be for values of one type constructor is applied to values of another type constructor.

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.
```

Though we do not recommend this, 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. terms that contain both function symbols and free variables, such as '`f(a, b, X)`'. 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.md`' 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 for operation/0
    --->    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) for list/1
    --->    []
    ;       [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 been 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
```

## 5.2 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 9.4 [Higher-order insts and modes], page 78) 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 7 [Determinism], page 63.)

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, it 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.

## 5.3 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 9 [Higher-order], page 75).

## 5.4 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 does not 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 does not 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 17 [Impurity], page 146.

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).
```

# 6 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.

## 6.1 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 7 [Determinism], page 63).

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

## 6.2 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.
```

## 6.3 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 future.

# 7 Determinism

## 7.1 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 of 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 of 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*; aside from assisting in reasoning about the code, 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 throw an exception (see Chapter 14 [Exception handling], page 112), contain a trace goal (see Chapter 19 [Trace goals], page 156), or infinitely loop. In some cases these optimizations may not be desirable; the strict sequential semantics guarantees that they will not be performed (see Chapter 15 [Formal semantics], page 115).

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 5 [Modes], page 53 for more information on default modes of functions).

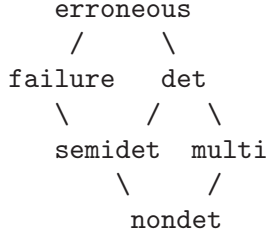
For example:

```
:- 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.

## 7.2 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 categories: 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 17 [Impurity], page 146), 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 the reachable arms of the switch can succeed at most zero times. (A switch arm is not reachable if it unifies the switched-on variable with a function symbol that is ruled out by that variable's initial instantiation state.)

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`.

## 7.3 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) :-
    ( if goal_that_should_never_fail(A, B0) then
        B = B0
    else
        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.)

## 7.4 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, then you need to use one of the predicates in the standard library module '`solutions`', such as '`solutions/2`' itself, which collects all of the solutions to a goal into a list — see Chapter 9 [Higher-order], page 75.

If you just want one solution from a predicate and don't care which, you should declare the relevant mode of the predicate to have determinism `cc_nondet` or `cc_multi` (depending on whether you are guaranteed at least one solution or not). This tells the compiler that this mode of this predicate may have more than one solution when viewed as a statement in logic, but the implementation should stop after generating the first solution. In other words, the implementation should *commit* to the first solution.

The commit to the first solution means that a piece of `cc_nondet` or `cc_multi` code can never be asked to generate a second solution. If e.g. a `cc_nondet` call is in a conjunction, then no later goal in that conjunction (after mode reordering) may fail, because that would ask the committed choice goal for a second solution. The compiler enforces this rule.

In the declarative semantics, which solution will be the first is unpredictable, but in the operational semantics, you *can* predict which solution will be the first, since Mercury does depth-first search with left-to-right execution of clause bodies, though that is not on the source code form of each clause body, but on its form *after* mode analysis reordering to put the producer of each variable before all its consumers.

The '`committed choice nondeterminism`' of a predicate has to be propagated up the call tree, making its callers, its callers' callers, and so on, also `cc_nondet` or `cc_multi`, until either you get to `main/2` at the top of the call tree, or you get to a location where you don't have to propagate the committed choice context upward anymore.

While `main/2` is usually declared to have determinism `det`, it may also be declared `cc_multi`. In the latter case, while the program's declarative semantics may admit more than one solution, the implementation will stop after the first, so alternative solutions to `main/2` (and hence also to `cc_nondet` or `cc_multi` predicates called directly or indirectly from '`main/2`') are implicitly pruned away. This is similar to the "don't care" nondeterminism of committed choice logic programming languages such as GHC.

One way to stop propagating committed choice nondeterminism is the one mentioned above: if a goal has no non-local output variables (i.e. none of the goal's outputs are visible from outside the goal), then the goal's solutions are indistinguishable from the outside, and the implementation will only attempt to satisfy the goal once, whether or not the goal is committed choice. Therefore if a `cc_multi` goal has all its outputs ignored, then the compiler considers it to be a `det` goal, while if a `cc_nondet` goal has all its outputs ignored, then the compiler considers it to be a `semidet` goal.

The other way to stop propagating committed choice nondeterminism is applicable when you know that all the solutions returned will be equivalent in all the ways that your program cares about. 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 iterate over them, the maximum value in the set should be the same.

If this condition is satisfied, i.e. if you know that there will only ever be at most one distinct solution under your equality theory of the output variables, then you can use a '`promise_equivalent_solutions`' determinism cast. If goal '`G`' is a `cc_multi` goal whose outputs are `X` and `Y`, then `promise_equivalent_solutions [X, Y] ( G )` promises the compiler that all solutions of `G` are equivalent, so that regardless of which solution of `G` the implementation happens to commit to, the rest of the program will compute either identical or (similarly) equivalent results. This allows the compiler to consider `promise_equivalent_solutions [X, Y] ( G )` to have determinism `det`. Likewise, the compiler will consider `promise_equivalent_solutions [X, Y] ( G )` where `G` is `cc_nondet` to have determinism `semidet`.

Note that specifying a user-defined equivalence relation as the equality predicate for user-defined types (see Chapter 8 [User-defined equality and comparison], page 72) 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.

## 7.5 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 8 [User-defined equality and comparison], page 72). And there are a variety of other applications.

# 8 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 representation. 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 (not including subtypes):

```
:- 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 16.4 [Using foreign types from Mercury], page 128) 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 17 [Impurity], page 146).
- 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 15 [Formal semantics], page 115), 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 16.4 [Using foreign types from Mercury], page 128) 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 17 [Impurity], page 146).
- 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.

# 9 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. This is because 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.)

## 9.1 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 `list(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 you 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.

## 9.2 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 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(in(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 if and only if
    all [X] (call(Pred, X) <=> list.member(X, List))
    and List is sorted without any duplicates.
```

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 is 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 if and only if 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.

## 9.3 Comparing higher-order terms

In Mercury, it is an error to attempt to unify or to compare two higher-order terms. This is because the question of whether two higher-order terms are equivalent is undecidable in the general case.

Note that the compiler will catch only *direct* attempts at unifications or comparisons of higher-order terms. Indirect attempts, using for example polymorphic predicates such as '`list.append([], [P], [Q])`', will result in an error at run-time rather than at compile-time.

## 9.4 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 that differ in their modes and in their determinisms, which is particularly useful for library predicates such as '`list.map`' and '`list.foldl`'.

Consider '`list.foldl`', one of the standard fold predicates on lists. The types of its arguments are given by this predicate declaration:

```
:- pred foldl(pred(L, A, A), list(L), A, A).
```

The first argument is a higher order value (a predicate in this case), whose types are the types bound by the caller to the type variables $L$, $A$ and $A$ respectively, where $L$ is the type of the elements in the list in the second argument, and $A$ is the type of the accumulator whose initial and final values are the third and fourth arguments. The job of the predicate passed in the first argument is to combine each element in the list with the current value of the accumulator to generate the next value of the accumulator, so in most calls to '`list.foldl`', the argument modes and the determinism of that predicate will be

```
pred(in, in, out) is det
```

This is a *higher order inst*: an inst describing the modes of the arguments and the determinism of a higher order value, which in this case is a predicate. A *higher order mode* is a mode in which the initial and/or final inst is a higher order inst. These *can* be written like this:

```
pred(in, in, out) is det >> pred(in, in, out) is det
```

for a predicate being passed to another predicate or function, and like this

```
free >> pred(in, in, out) is det
```

for a predicate being returned from another predicate or function. In practice, it is far more convenient to use the builtin mode constructors '`in/1`' and '`out/1`' to write

```
in(pred(in, in, out) is det)
out(pred(in, in, out) is det)
```

which are each equivalent to the corresponding example just above.

You can use higher order insts and modes in mode declarations like this:

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

The '`in()`' wrapper around the higher order inst here declares the first argument of '`list.foldl`' to be an input, but the higher order inst inside the wrapper goes further, by specifying the argument modes and the determinism of the predicate that '`list.foldl`' takes *in this mode*.

That last qualification is important, because '`list.foldl`' has several modes, each differing in the argument modes, in the determinism, or both. The set of mode declarations for '`list.foldl`' includes

```
:- mode foldl(in(pred(in, in, out) is det), in, in, out) is det.
:- mode foldl(in(pred(in, di, uo) is det), in, di, uo) is det.
:- mode foldl(in(pred(in, in, out) is semidet), in, in, out) is semidet.
:- mode foldl(in(pred(in, in, out) is cc_multi), in, in, out) is cc_multi.
:- mode foldl(in(pred(in, di, uo) is cc_multi), in, di, uo) is cc_multi.
```

That means you can pass predicates with several different argument mode/determinism combinations as the first argument. In the case of '`list.foldl`', the modes of the accumulator arguments and the determinism of the whole predicate will follow the argument modes and the determinism of the higher-order argument, but one can create predicates (and functions) where this would not be true.

You can give names to such insts and modes using declarations such as

```
:- inst std_fold_pred == (pred(in, in, out) is det).
:- mode std_fold_pred_in == in(pred(in, in, out) is det).
```

Note that the parentheses around the right hand side of the inst declaration are required, due to the relative precedences of the '`==`' and '`is`' operators.

Given these definitions, the declarations

```
:- mode foldl(in(std_fold_pred), in, in, out) is det.
:- mode foldl(std_fold_pred_in, in, in, out) is det.
```

are both equivalent to

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

but they may be more convenient to write, especially in the presence of more arguments.[1]

The general form of higher order insts follows one of two patterns, one for predicates, and one for functions.

The pattern for predicates is

[1] If all instances of a given type are expected to use a single inst or a single mode, there is a convention where programmers will give that inst or mode the same name as the type. This works because types, insts and modes belong to separate namespaces, so the names do not conflict. Nonetheless, there is usually no need to define a named mode. It is clearer to write a mode using '`in()`' or '`out()`' around a named inst.

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

while the pattern for functions is

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

In the case of zero-argument predicates and functions, the parentheses around '`pred`' and '`func`' are required to tell the compiler that these words are not being used as operators in this case. And, as explained above, one will usually need parentheses around any instances of these patterns in Mercury code.

As a convenience, the language allows you to write a higher order *mode* using the same syntax as a higher order *inst*. If *HOInst* has the form of a higher order inst, then writing *HOInst* where a mode is required is the same as writing '`in(HOInst)`', which is in turn equivalent to '`HOInst >> HOInst`'. Therefore, you can omit '`in()`' around the higher order inst of an input argument. For example,

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

can also be written as

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

though the former may be easier to understand.

As usual, if a predicate or function has only one mode, the '`pred`' or '`func`' declaration can be combined with the '`mode`' declaration. Consider the declaration of a function that computes the intersection of two maps from keys to values:

```
:- func intersect(func(V, V) = V, map(K, V), map(K, V)) = map(K, V).
```

One could declare the mode of this function either using a separate mode declaration, like this:

```
:- mode intersect(in(func(in, in) = out is det), in, in) = (out) is det.
```

or by combining the mode declaration with the function declaration, like this:

```
:- func intersect((func(V, V) = V)::in(func(in, in) = out is det),
    map(K, V)::in, map(K, V)::in) = (map(K, V)::out) is det.
```

In both cases, just as for the predicate examples above, the '`in()`' wrapper around the higher order inst is optional, though in the combined declaration the parentheses must stay.

Mercury also provides builtin '`inst`' values for use with solver types. These follow the patterns

```
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 18 [Solver types], page 152 for more details.

### 9.4.1 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 there is little value in doing so, because there is no default higher-order inst for predicates, and therefore it will not be possible to call those terms.

### 9.4.2 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 one is 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.

# 10 Modules

## 10.1 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 submodules, if any) and cannot be used by other modules. The implementation section must contain definitions for all abstract data types, abstract instance declarations, functions, predicates, and submodules exported by the module, as well as for all local types, type class instances, functions, predicates, and submodules. 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.

In order to constrain which entity a name refers to, functor terms representing predicates, functions, expressions, constructor fields, types, modes, insts, type classes, instances and sub-modules can be explicitly module qualified using the '`.`' operator, e.g. '`module.name`' or '`module.name(Args)`'. Operator terms (that is, functor terms written using operator syntax) may also be module qualified, though this requires putting parentheses around the operator and its arguments, e.g. '`module.(A + B)`'. The module name used in a module qualified term may itself be module qualified if it is a sub-module, e.g. '`module.submodule.name(Args)`'. A functor is *fully qualified* if every name that is not a builtin or top-level module is module qualified.

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

The principal functor of a module qualified term combines the name and qualifiers with the arity. For example, the principal functor of '`foo.bar.baz(Xs)`' is '`foo.bar.baz/1`', and the principal functor of '`quux.(A + B)`' is '`quux.'+'/2`'.

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.

You may need to module qualify a name if that name has several applicable definitions, and the context of its occurrence does not resolve this ambiguity. Module qualifiers are also useful for readability. Uses of entities imported using `use_module` declarations *must* be fully qualified.

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.

Exactly one module of the program 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.

## 10.2 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.
```

## 10.3 Submodules

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

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

### 10.3.1 Nested submodules

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

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

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

If a submodule is declared but not explicitly defined, then there is an implicit definition with an empty implementation section for that submodule. This empty implementation section will result in an error if the interface section of a submodule 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 submodule (which perforce has only an interface section, and no implementation section) and which contains any of the above.

### 10.3.2 Separate submodules

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

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

The module names need not be fully qualified.

Each of the named submodules 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. The Melbourne Mercury implementation requires that

- *either* every module must be in a file whose name is the fully qualified module name followed by '`.m`', (so a module named e.g. '`foo.bar.baz`', must be in a file named '`foo.bar.baz.m`')
- *or* that the programmer tell the implementation about which files contain which modules using a command such as '`mmc -f *.m`'. (Alternatively, you could replace the '`*.m`' in that command with a list of the file names of all the Mercury modules in the program.)

The source file of a separate submodule must contain the declaration (interface) and definition (implementation) of the submodule. It must start with a '`:- module`' declaration containing the fully qualified module name, followed by the interface and (if necessary) implementation sections, and it may optionally end with a '`:- end_module`' declaration. (The module name in the '`:- end_module`' declaration need not be fully qualified.)

The semantics of separate submodules are identical to those of nested submodules. The procedure to transform a separate submodule into a nested submodule is as follows:

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

### 10.3.3 Visibility rules

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

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 submodule 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 submodules of its parent module.

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

### 10.3.4 Implementation bugs and limitations

The current implementation of submodules has a couple of minor limitations.

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

## 10.4 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 submodules. 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`'.

## 10.5 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 signatures:

```
:- 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 submodules. 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`'.

## 10.6 Module-local mutable variables

Certain special cases require a module to have one or more mutable (i.e. destructively updateable) 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* are supported:

'`trailed`'/'`untrailed`'
: This attribute specifies whether the implementation should generate code to undo the effects of '`set_varname/1`' on backtracking ('`trailed`') or not ('`untrailed`'). 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 16.3.1 [C data passing conventions], page 122. 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 submodules 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`' will be invoked first,
- then '`bar`' will be set to its initial value of 561,
- and then '`baz/2`' will be 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.

# 11 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.

## 11.1 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 declaration of the method. The values of *most* parameter type variables are determined by having them occur in the declared type of an argument of the method. However, if either the typeclass named in the constraint, or its superclasses, include any functional dependencies, then the value of a variable may also be implied by the values of other variables (see Section 11.8 [Functional dependencies], page 98).

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 5 [Modes], page 53): 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.

## 11.2 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 [method_definition, method_definition, ...].
```

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).

There is no special interaction between subtypes and the typeclass system. A subtype is *not* automatically an instance of a typeclass if there is an '`instance`' declaration for its supertype.

Each *method_definition* 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 to define a method is by giving the name of the predicate or function which implements that method. In this case, the *method_definition* must have one of the following forms:

```
pred(method_name/arity) is predname
func(method_name/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 *method_name* 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 *method_definition* can be a clause. These clauses are just like the clauses used to define ordinary predicates or functions (see Section 2.12 [Items], page 12), 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 is 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 is 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.

Since methods may be defined using clauses, and the interface sections of modules may *not* include clauses, instance declarations that specify method definitions may appear only in the implementation section of a module. If you want to export the knowledge that a type, or a sequence of types, is a member of a given typeclass, then put a version of the instance declaration that omits all method definitions (see Section 11.4 [Abstract instance declarations], page 95) into the interface section of the module that contains the full instance declaration in its implementation section.

## 11.3 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.

## 11.4 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 in the implementation section of the same module by a corresponding non-abstract instance declaration that defines how the type class methods are implemented.

Here is 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.
```

## 11.5 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 11.8 [Functional dependencies], page 98). 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.

## 11.6 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 give 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.

## 11.7 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 to 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.

The abstract version of the above instance declaration would be

```
:- instance portrayable(list(T)) <= portrayable(T).
```

## 11.8 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 distinct 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 their concrete forms.

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`' for 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/3` 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`.

# 12 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.

## 12.1 Existentially typed predicates and functions

### 12.1.1 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 variable '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).
```

### 12.1.2 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 12.4 [Some idioms using existentially quantified types], page 106).

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.

### 12.1.3 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)
```

### 12.1.4 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)
```

## 12.2 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 11.5 [Type class constraints on predicates and functions], page 96 ; 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 12.1.1 [Syntax for explicit type quantifiers], page 101).

## 12.3 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 is an arbitrary example involving multiple type variables
% and multiple constraints.
:- typeclass foo(T1, T2) where [ /* ... */ ].
:- type bar(T)
    --->    f1
    ;       f2(T)
    ;       some [T1] f3(T1)
    ;       some [T1, T2] f4(T1, T2, T) => (showable(T1), showable(T2))
    ;       some [T1, T2] f5(list(T1), T2) => foo(T1, 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 '`showable_list`' 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(showable_list) = list(string).
process_list(nil) = [].
process_list(cons(Head, Tail)) = [show(Head) | process_list(Tail)].
```

There are some restrictions on the forms that existentially typed data constructors can take.

The first restriction is that no type variable may be quantified both universally, by being listed as an argument of the type constructor, and existentially, by being listed in the existential type quantifier before the data constructor. The type '`t12`' violates this restriction:

```
:- type t12(T)
    --->    f1(T)
    ;       some [T] f2(T).
```

The reason for the restriction is simple: the reference of '`T`' in the '`f2`' data constructor being simultaneously inside the scope of more than one quantification can mislead readers who see one of the quantifications, and stop looking for the other. The simplest way to avoid such confusion is to require the programmer to avoid having one quantification shadow another.

The second restriction is that type variables listed in the existential type quantifier before the data constructor cannot be repeated. Type variables in the argument list of the type constructor also cannot be repeated, whether or not the data constructors of that type have existential types. The type '`t34`' violates both these restrictions:

```
:- type t34(A, B, A)
    --->    f3(A, B)
    ;       some [C, D, D] f4(C, D).
```

The third and final restriction is that every existentially quantified type variable must occur

- either in one of the argument types of the data constructor,
- or in one of the type class constraints on the data constructor, in the range of a functional dependency.

This means that the type '`t5`' in

```
:- type t5
    --->    some [T1, T2] f5(T1) => xable(T1, T2).
```

violates this restriction *unless* the type class '`xable`' has a functional dependency that determines the type bound to its second argument from the type bound to its first.

The reason for this restriction is that the identity of the type bound to the existential type variable must somehow be decided at runtime. It can either be given by the type of an argument, or determined through a functional dependency from the types bound to one or more other existential type variables.

## 12.4 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’ does not 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 = "foo", Univ = 'new mkshowable'(42)
    ; Name = "bar", Univ = 'new mkshowable'("blah")
    ),
    Univ = mkshowable(Showable).
```

The issue can also arise for mode-specific clauses (see Section 5.4 [Different clauses for different modes], page 59). 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).
```

# 13 Type conversions

(This is a new and experimental feature, subject to change.)

A term may be converted from one type *FromType* to another type *ToType* using a type conversion expression of the form:

```
coerce(Term)
```

The expression is type-correct if and only if *FromType* and *ToType* are both discriminated union types, and after replacing the principal type constructors with base types (see Section 4.2.4 [Subtypes], page 42) the two types have the same type constructor, and the arguments of the common type constructor satisfy the type parameter variance restrictions below.

Let *FromType* expand out to '`base(S1, ..., Sn)`' and *ToType* expand out to '`base(T1, ..., Tn)`', where '`base(B1, ..., Bn)`' is the common base type, and *Bi* is the i'th type parameter, which is bound to *Si* in *FromType* and *Ti* in *ToType*.

For each pair of corresponding type arguments, one of the following must be true:

- '`Si = Ti`' if the two types are the same
- '`Si < Ti`' if *Si* is a subtype of *Ti* by the relation below
- '`Ti < Si`' if *Ti* is a subtype of *Si* by the relation below

Otherwise, the `coerce` expression is not type-correct.

Furthermore, '`Si = Ti`' must be true if *Bi* occurs in one or more of these locations in the '`base/n`' type definition:

- in a higher-order type
- in a foreign type
- in an abstract type
- in a solver type
- in a discriminated union type, other than a recursive type of the exact form '`base(B1, ..., Bn)`'

The relation '`S < T`' is true when '`S != T`' and either:

- '`S`' and '`T`' are both discriminated union types, and '`S`' is a subtype of '`T`' by visible subtype definitions; or
- '`S`' and '`T`' are both tuple types of the same arity, and for each pair of corresponding argument types '`Si`' and '`Ti`', '`Si < Ti`' is true.

## Mode checking

Type conversion expressions must also be mode-correct. Intuitively, conversion from a subtype to its supertype is safe, but a conversion from a supertype to one of its subtypes is safe only if the inst approximating the term to be converted indicates that the result would also be valid in the subtype.

Mode checking proceeds by simultaneously traversing the inst tree of the `coerce` argument, the type tree of the `coerce` argument, and the type tree of the result term, and producing the inst tree of the result term if the conversion is valid. Let

- *InstX* be the current node in the `coerce` argument's inst tree,
- *InstY* be the current node in the result inst tree,
- *TypeX* be the current node in the `coerce` argument's type tree,
- *TypeY* be the current node in the result type tree,
- *TypeCtorX* be the principal type constructor of *TypeX*,
- *TypeCtorY* be the principal type constructor of *TypeY*.

In the following, $X$ `<` $Y$ means $X$ is a subtype of $Y$ by visible subtype definitions.

For each node *InstX*:

- If *InstX* is a recursive node in the inst tree (i.e. it is its own ancestor), then we require *TypeX* `=<` *TypeY*. Let *InstY* = *InstX*.
- Otherwise, if *InstX* is a `bound` node:
  - If *TypeX* is an existentially quantified type variable, then *InstY* = *InstX*.
  - If *TypeX* is not an existentially quantified type variable, then each of the function symbols listed in *InstX* must name a constructor in *TypeCtorY*. Let *InstY* be a `bound` inst containing those same function symbols; the insts for the arguments of each function symbol are then checked and constructed recursively.
- Otherwise, if *InstX* is a `ground` node:
  - If *TypeX* = *TypeY*, or if *TypeX* is an existentially quantified type variable, then let *InstY* = *InstX*.
  - If *TypeX* `<` *TypeY*, then let *InstY* be the `bound` node constructed using the process below.
- Otherwise, the `coerce` expression is not mode-correct.

To construct a '`bound`' node *InstY* from a '`ground`' node *InstX*:

- If *TypeX* = *TypeY* or if *TypeX* is a recursive node in the type tree (i.e. it is its own ancestor), then let *InstY* be `ground`.
- Otherwise, let *InstY* be a `bound` inst containing all of the constructors in *TypeCtorX*; the insts for the arguments of each function symbol are constructed recursively.

## Examples

Assume we have:

```
:- type fruit
   --->    apple
   ;       lemon
   ;       orange.

:- type citrus =< fruit
   --->    lemon
   ;       orange.
```

This function is type and mode-correct:

```
:- func f1(citrus) = fruit.
```

```
f1(X) = coerce(X).
```

This function is type-correct but not mode-correct because some `fruit`s are not `citrus`:

```
:- func f2(fruit) = citrus.

f2(X) = coerce(X).  % incorrect
```

This function is mode-correct because the initial inst of the input argument limits the range of `fruit` values to those that would also be valid in `citrus`:

```
:- inst citrus for fruit/0
    --->    lemon
    ;       orange.

:- func f3(fruit) = citrus.
:- mode f3(in(citrus)) = out is det.

f3(X) = coerce(X).
```

Finally, this function is type-incorrect because in the coerce expression, the type parameter $T$ of `wrap/1` is bound to `fruit` in the input type, but `citrus` in the result type.

```
:- type wrap(T)
    --->    wrap(T).

:- func f4(func(fruit) = int) = (func(citrus) = int).

f4(X) = Y :-
    wrap(Y) = coerce(wrap(X)).  % incorrect
```

# 14 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)
        ).
```

One common use for exceptions is to check the input and throw an exception if it is invalid. It might be tempting to implement this with a predicate such as the following:

```
:- pred check_target(target::in) is det.

check_target(Target) :-
    ( if ... then
        true
    else
        throw("invalid target")
    ).
```

This code warrants caution, however. Consider the following usage:

```
shoot(Target, !IO) :-
    check_target(Target),
    unsafe_shoot(Target, !IO).
```

Mercury may reorder conjunctions, which is (probably) not what the user intended in this case. Furthermore, Mercury may optimize away the call to '`check_target/1`' entirely, since the mode-determinism assertion for a '`det`' predicate with no outputs essentially states that it is equivalent to '`true`'.

The strict sequential semantics can be used to guarantee that these changes will not occur (see Chapter 15 [Formal semantics], page 115). However, we recommend implementing checks like these in the following way, to avoid depending on the choice of operational semantics:

```
shoot(Target0, !IO) :-
    check_target(Target0, Target),
    unsafe_shoot(Target, !IO).

:- pred check_target(target::in, target::out) is det.

check_target(Target0, Target) :-
    ( if ... then
        Target = Target0
    else
        throw("invalid target")
    ).
```

# 15 Formal 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 7 [Determinism], page 63), 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* semantics. In this semantics, the program is executed top-down using SLDNF resolution (or something equivalent), starting from '`main/2`' preceded by any module initialisation goals (as per Section 10.4 [Module initialisation], page 87), and followed by any module finalisation goals (as per Section 10.5 [Module finalisation], page 88). Function calls, 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: the sub-goals are reordered, not split and interleaved. Function application is strict, not lazy. Predicate calls are strict in the sense that goals will be executed irrespective of any mode-determinism assertions, even if they loop, are '`erroneous`', or are '`det`' and contain no outputs.

Mercury implementations are required to provide a method of processing Mercury programs which is equivalent to the strict sequential semantics.

There is another operational semantics of Mercury programs called the *strict commutative* semantics. This semantics is equivalent to the strict sequential 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 differ each time a particular goal is entered.

As well as providing the strict sequential semantics, Mercury implementations may provide one or more implementation-defined operational semantics, as long as any such implementation-defined operational semantics is at least as complete as the strict commutative semantics. An implementation-defined operational semantics is "at least as complete" as the strict commutative semantics if and only if the implementation-defined operational 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 function calls, conjunctions and disjunctions).

Thus, to summarize, there are in fact a variety of different operational semantics for Mercury. One of them, the strict sequential semantics, is deterministic—the behaviour is always specified exactly. Programs are executed top-down, mode analysis does "minimal" reordering (in a precisely defined sense), function calls, conjunctions and disjunctions are executed depth-first left-to-right, and function and predicate 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 be non-deterministic) as long as they are sound with respect to the declarative semantics, and meet a minimum level of completeness.

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. In some ways this is not 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. On the other hand, if you do write code which works for all possible executions under the strict commutative semantics, then you can be guaranteed that it will work correctly on every implementation, under every possible implementation-defined operational semantics.

The Melbourne Mercury implementation offers eight different operational semantics, which can be selected with different combinations of the following options:

`--no-reorder-conj`
: Only do minimal reordering of conjunctions.

`--no-reorder-disj`
: Do not reorder disjunctions.

`--no-fully-strict`
: Predicate calls are not strict (function application is always strict in the current implementation). This option allows the compiler to improve completeness by optimizing away infinite loops, goals with determinism `erroneous`, and goals with determinism `det` and no outputs.

The default semantics is the strict commutative semantics. The strict sequential semantics can be selected with the '`--no-reorder-conj`' and '`--no-reorder-disj`' options.

Future implementations of Mercury may wish to offer other implementation-defined 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.

# 16 Foreign language interface

This chapter documents the foreign language interface.

## 16.1 Calling foreign code from Mercury

Mercury procedures can be implemented using fragments of foreign language code using '`pragma foreign_proc`'.

### 16.1.1 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 17 [Impurity], page 146).

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 is 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) :-
    ( if NextS1Len = S3Len then
        append_3(NextS1Len, S3Len, S1, S2, S3)
    else
        (
            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);
").
```

### 16.1.2 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 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 17 [Impurity], page 146). 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 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 for-

eign 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 21.3 [Termination analysis], page 171 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 21.6 [Trailing], page 177). 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 21.2 [Tabled evaluation], page 167). 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 to make its own judgement in the matter, based on factors such as the size of the code fragment.

'`may_export_body/may_not_export_body`'
: This attribute tells the compiler whether it is allowed to duplicate the foreign code fragment outside of the target file for the module that defines the foreign procedure. The '`may_export_body`' attribute says that it may; the '`may_not_export_body`' attribute says that it may not. The default is '`may_export_body`'.

## 16.2 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`.

## 16.3 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.

### 16.3.1 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 were 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 types 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 16.4 [Using foreign types from Mercury], page 128) 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 16.10.1.8 [Memory management for C], page 138).

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)`'.

### 16.3.2 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.SByte   sbyte
int16         System.Int16   short
int32         System.Int32   int
int64         System.Int64   long
uint          System.UInt32  uint
uint8         System.Byte    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 16.4 [Using foreign types from Mercury], page 128) 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.

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
```

### 16.3.3 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 16.4 [Using foreign types from Mercury], page 128) 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
```

## 16.4 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*. *MercuryTypeName* must be the name of either an abstract type or a discriminated union type. In both cases, *MercuryTypeName* must be declared with '`:- type`' 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).

If *MercuryTypeName* names a discriminated union type, that type cannot be the base type of any subtypes, nor can it be a subtype itself (see Section 4.2.4 [Subtypes], page 42).

*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*.

The same type may have a foreign language definition for more than one foreign language. The definition used in the generated code will be the one for the foreign language that is most appropriate for the target language of the compilation (see the language specific information below for details). All the foreign language definitions must have the same visibility.

A type which has one or more foreign language definitions may also have a Mercury definition, which must define a discriminated union type. The constructors for this Mercury 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 21.2 [Tabled evaluation], page 167) 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 8 [User-defined equality and comparison], page 72):

```
:- 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.

## 16.5 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*. Because of this, the names of the constants need not and must not be module qualified.

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).

## 16.6 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.

## 16.7 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 backend supports 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.

## 16.8 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 submodules. By default, they are not visible to the foreign code in '`pragma foreign_code`' or '`pragma foreign_proc`' declarations in any other module, but this default can be overridden (giving access to all other modules) 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.

Note that the Melbourne Mercury implementation often behaves as if '`pragma foreign_import_module`' declarations were implicitly added to modules. However, programmers should *not* depend on this behaviour; they should always write explicit '`pragma foreign_import_module`' declarations wherever they are needed.

## 16.9 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.

## 16.10 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 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.

### 16.10.1 Interfacing with C

#### 16.10.1.1 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, 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),
    [promise_pure, will_not_call_mercury],
"
    Structure = init_struct();
").

:- pragma foreign_proc("C",
    do_calculation(Value::in, Structure0::di, Structure::uo),
    [promise_pure, will_not_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.

#### 16.10.1.2 Using pragma foreign_enum for C

Foreign enumeration values in C must be constants of type `MR_Integer`. A foreign enumeration value may be specified by one of the following:

- An integer literal.
- An enumeration constant.

- A preprocessor macro that expands to either an integer literal or an enumeration constant.

#### 16.10.1.3 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 16.3.1 [C data passing conventions], page 122, 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`.

#### 16.10.1.4 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 16.3.1 [C data passing conventions], page 122.

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.

#### 16.10.1.5 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 16.3.1 [C data passing conventions], page 122. 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 16.10.1.6 [Using pragma foreign_decl for C], page 138.

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.

#### 16.10.1.6 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, then 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).
```

#### 16.10.1.7 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.

#### 16.10.1.8 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. The latter is suitable only for programs with very short running times (less than a second), which makes the former the standard method for almost all Mercury programs.

Conservative garbage collection makes inter-language calls simplest. Mercury uses the Boehm-Demers-Weiser conservative garbage collector, which we also call simply Boehm gc. This has its own set of functions for allocating memory blocks, such as '`MR_GC_NEW`', which are documented in '`runtime/mercury_memory.h`'. Memory blocks allocated by these functions, either in C code generated by the Mercury compiler or in C code hand written by programmers, are automatically reclaimed when they are no longer referred to either from the stack, from global variables, or from other memory blocks allocated by Boehm gc functions. Note that these are the *only* places where Boehm gc looks for pointers to the blocks it has allocated. If the only pointers to such a block occur in other parts of memory, such as in memory blocks allocated by '`malloc`', the Boehm collector won't see them, and may collect the block prematurely. Programmers can avoid this either by not storing pointers to Boehm-allocated memory in malloc-allocated blocks, or by storing them e.g. on the stack as well.

Boehm gc recognizes pointers to the blocks it has allocated only if they point either to the start of the block, or to a byte in the first word of the block; pointers into the middle of a block beyond the first word won't keep the block alive.

Pointers to Boehm-allocated memory blocks can be passed freely between Mercury and C code provided these restrictions are observed.

Note that the Boehm collector cannot and does not recover memory allocated by other methods, such as '`malloc`'.

When using no garbage collection, heap storage is reclaimed only on backtracking. This requires programmers to 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. They 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.

#### 16.10.1.9 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 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_function.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.

### 16.10.2 Interfacing with C#

#### 16.10.2.1 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*.

#### 16.10.2.2 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`.

#### 16.10.2.3 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`.

#### 16.10.2.4 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 16.3.2 [C# data passing conventions], page 123.

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.

#### 16.10.2.5 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 16.3.2 [C# data passing conventions], page 123.

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`'.

#### 16.10.2.6 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"");
").
```

#### 16.10.2.7 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;
").
```

### 16.10.3 Interfacing with Java

#### 16.10.3.1 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*.

### 16.10.3.2 Using pragma foreign_enum for Java

'`pragma foreign_enum`' is currently not supported for Java.

### 16.10.3.3 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`.

### 16.10.3.4 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 16.3.3 [Java data passing conventions], page 125.

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`).

### 16.10.3.5 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 16.3.3 [Java data passing conventions], page 125.

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`'.

#### 16.10.3.6 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();
    // ...
").
```

#### 16.10.3.7 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;
").
```

# 17 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**.

## 17.1 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 4 [Types], page 37) 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 not 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.

## 17.2 Purity ordering

The three levels of purity (which we will simply call the purity) have a total ordering defined upon them: `pure` > `semipure` > `impure`.

## 17.3 Impurity 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 Mercury draws this distinction 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 15 [Formal semantics], page 115). *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.

## 17.4 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, the declarations

```
:- impure pred Pred(Arguments...).
:- semipure pred Pred(Arguments...).
:- impure func Func(Arguments...) = Result.
:- semipure func Func(Arguments...) = Result.
```

declare the predicate *Pred* and the function *Func* to be impure and 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.

## 17.5 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 in 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.

## 17.6 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 3.2 [Goals], page 16 may be used to promise purity at the finer level of goals within clauses.

## 17.7 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
    ).
```

## 17.8 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.)

### 17.8.1 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 explicitly "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.

### 17.8.2 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)`'.

### 17.8.3 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, Xs).
```

# 18 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).

## 18.1 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.

## 18.2 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.

## 18.3 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 10.6 [Module-local mutable variables], page 88), 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.

Solver types may be exported from their defining module, but only in an abstract form. This requires the full definition to appear in the implementation section of the module, and an abstract declaration like the following in its interface:

```
:- solver type solver_type.
```

If a solver type is exported, then its representation type, and, if specified, its equality and/or comparison predicates must also be exported from the same module.

If a solver type has no equality predicate specified, then the compiler will generate an equality predicate that throws an exception of type '`exception.software_error/0`' when called.

Likewise, if a solver type has no comparison predicate 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, []).
```

## 18.4 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 21.6 [Trailing], page 177) 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 17 [Impurity], page 146),
- 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).

## 18.5 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 if and only if 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 does not 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 17.8 [Higher-order impurity], page 150).

# 19 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),
        ( if !.LoggingLevel > 1 then
            io.write_string("Input is ", !IO),
            io.write(In, !IO),
            io.nl(!IO)
        else
            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 '`GradeTest`'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.

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 to be set when the program starts execution. (It does not 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 does not 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, even though they are '`det`' and have no outputs. 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. Note that trace goals inside a procedure do not prevent calls to that procedure from being optimized away. For example, if a predicate definition contains a single trace goal in order to factor out the details of that goal, calls to it may be optimized away. This will render them ineffective; the strict sequential semantics can be used to inhibit such optimizations (see Chapter 15 [Formal semantics], page 115).

# 20 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 16 [Foreign language interface], page 117) and impurity (see Chapter 17 [Impurity], page 146). As an extension, implementations may also choose to support additional pragmas with implementation-dependent semantics (see Chapter 21 [Implementation-dependent extensions], page 167).

## 20.1 Inlining

Declarations of these forms

```
:- pragma inline(pred(Name/Arity)).
:- pragma inline(func(Name/Arity)).
```

are a hint to the compiler that all calls to the predicate or function with name *Name* and arity *Arity* should be inlined.

The current Mercury implementation is smart enough to inline simple predicates even without this hint.

Declarations of these forms

```
:- pragma no_inline(pred(Name/Arity)).
:- pragma no_inline(func(Name/Arity)).
```

tell the compiler not to inline the named predicate or function. 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.

## 20.2 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.

### 20.2.1 Syntax and semantics of type specialization pragmas

A declaration of the form

```
:- pragma type_spec(pred(Name/Arity), Subst).
:- pragma type_spec(func(Name/Arity), Subst).
```

suggests to the compiler that it should create a specialized version of the predicate or function with name *Name* and arity *Arity* with the type substitution given by *Subst* applied to the argument types. 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 conjunction must have parentheses around it if it contains two or more bindings.)

For example, the declarations

```
:- pred map.lookup(map(K, V), K, V).
:- pragma type_spec(pred(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.

The pragma also has a form that suggests specialization of only one mode of the predicate or function, instead of all of them:

```
:- pragma type_spec(Name(m1, ... mn), Subst).
:- pragma type_spec(Name(m1, ... mn) = mr, Subst).
```

where *m1* etc are the modes of the arguments. If the '`= mr`' part is present, it gives the mode of the function result; if it is absent, this indicates that *Name* is a predicate.

### 20.2.2 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.

### 20.2.3 Implementation specific details

The 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.

## 20.3 Obsolescence

Declarations of the forms

```
:- pragma obsolete(pred(Name/Arity)).
:- pragma obsolete(func(Name/Arity)).
:- pragma obsolete(pred(Name/Arity),
    [ReplName1/ReplArity1, ..., ReplNameN/ReplArityN]).
:- pragma obsolete(func(Name/Arity),
    [ReplName1/ReplArity1, ..., ReplNameN/ReplArityN]).
```

declare that the predicate or function with name *Name* and arity *Arity* is "obsolete": they instruct the compiler to issue a warning whenever the named predicate or function is used. The forms with a second argument tell the compiler to suggest the use of one of the listed possible replacements.

Declarations of the forms

```
:- pragma obsolete_proc(PredName(ArgMode1, ..., ArgModeN)).
:- pragma obsolete_proc(PredName(ArgMode1, ..., ArgModeN),
```

```
    [ReplName1/ReplArity1, ..., ReplNameN/ReplArityN]).
:- pragma obsolete_proc(FuncName(ArgMode1, ..., ArgModeN) = RetMode).
:- pragma obsolete_proc(FuncName(ArgMode1, ..., ArgModeN) = RetMode,
    [ReplName1/ReplArity1, ..., ReplNameN/ReplArityN]).
```

similarly declare that the predicate named *PredName* with arity *N*, or the function named *FuncName* with arity *N*, is obsolete when called in the specified mode. These forms also allow the specification of an optional list of possible replacements.

These 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 or procedure, they should leave its old version in the library, but mark it as obsolete using one of these declarations, and, if possible, use the suggested replacements to steer users to their replacements (either partial or total) in 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 believes that users have had sufficient warning, they can remove the old version entirely.

## 20.4 No determinism warnings

Declarations of the forms

```
:- pragma no_determinism_warning(pred(Name/Arity)).
:- pragma no_determinism_warning(func(Name/Arity)).
```

tell the compiler not to generate any warnings about the determinism declarations of procedures of the predicate or function with name *Name* and arity *Arity* not being 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 throw 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.

## 20.5 No dead predicate warnings

Declarations of the forms

```
:- pragma consider_used(pred(Name/Arity)).
:- pragma consider_used(func(Name/Arity)).
```

tell 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 function, *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.

## 20.6 Format calls

The '`format_call`' pragma asks the compiler to perform the same checks on calls to the named predicate or function as it performs for calls to the following formatting predicates and function in the Mercury standard library.

The Mercury standard library has a function and a predicate to format a given sequence of values according to a format string, like this:

```
string.format("Count = %d, Total = %5.2f, Average = %5.2f\n",
    [i(Count), f(Total), f(Total/float(Count))], FormatStr)

FormatStr = string.format("Count = %d, Total = %5.2f, Average = %5.2f\n",
    [i(Count), f(Total), f(Total/float(Count))])
```

and predicates that immediately output the resulting formatted string, like this:

```
io.format(
    "Count = %d, Total = %5.2f, Average = %5.2f\n",
    [i(Count), f(Total), f(Total/float(Count))], !IO)

io.format(OutputStream,
    "Count = %d, Total = %5.2f, Average = %5.2f\n",
    [i(Count), f(Total), f(Total/float(Count))], !IO)

stream.string_writer.format(StringWriterStream,
    "Count = %d, Total = %5.2f, Average = %5.2f\n",
    [i(Count), f(Total), f(Total/float(Count))], !StringWriterState)
```

These four predicates and one function all take a format string argument, and a values_to_format argument. The format string argument in all the examples above is

```
"Count = %d, Total = %5.2f, Average = %5.2f\n"
```

while the values_to_format argument is

```
[i(Count), f(Total), f(Total/float(Count))]
```

In this example, `%d` means that the corresponding value should be output as a signed integer, using as many characters as needed, and `%5.2f` means that the corresponding value should be formatted as a floating point number, using five characters, of which two are after the decimal point.

Mercury does not allow values of different types to be stored in the same list, so the elements of that list are actually values of the `poly_type` type, whose definition in the `string` module of the standard library is

```
:- type poly_type
    --->    f(float)
    ;       i(int)
    ;       i8(int8)
    ;       i16(int16)
    ;       i32(int32)
    ;       i64(int64)
    ;       u(uint)
    ;       u8(uint8)
```

```
;       u16(uint16)
;       u32(uint32)
;       u64(uint64)
;       s(string)
;       c(char).
```

As you can see, each function symbol in this type wraps up a value of a builtin type, and thus converts it to the same `poly_type` type.

With one exception, every occurrence of a percent sign in the format string is intended to start a *conversion specifier*, which specifies how the corresponding entry in the value list should be converted to a string. (The exception is that the double percent sign `%%` simply says that the output should be a single percent sign; the doubling up escapes the special conversion-introduction role of the character.) Each conversion specifier consists of a percent sign, zero or more non-alphabetic characters such as '`5`', and an alphabetic *conversion character* such as `d` or `f`.

The format string and the values list must match. There should be exactly one element in the value list for each conversion specifier, and the Nth value in the value list must satisfy the requirements of the Nth conversion specifier in the format string.

For example, the call

```
string.format("Count = %d, Total = %5.2f, Average = %5.2f\n",
    [i(Count), f(Total/float(Count))], FormatStr)
```

would not work, because the format string contains three conversion specifiers, but the value list contains only two values.

The call

```
string.format("Count = %f, Total = %d, Average = %5.2f\n",
    [i(Count), f(Total), f(Total/float(Count))], FormatStr)
```

would not work either, because the first specifier's conversion character, `f`, requires the corresponding value to have the form `f(Float)`, but the corresponding value is `i(Count)`. The conversion character in the second specifier, `d`, accepts any signed integer as the value, so the second value could be any of `i(Int)`, `i8(Int8)`, `i16(Int16)`, `i32(Int32)`, or `i64(Int64)`, but not `f(Float)`.

For the full set of the rules, please see the documentation of `string.format` in the Mercury Library Reference Manual; the same rules apply to `io.format` and `stream.string_writer.format` as well.

The Mercury compiler checks calls to the formatting predicates and function in the Mercury standard library, specifically

predicate `io.format/4`
predicate `io.format/5`
predicate `stream.string_writer.format/5`
function `string.format/2`
predicate `string.format/3`

for inconsistencies between the format string and the value list, generating a report for each such inconsistency. This improves both program reliability (because programmers get the problem brought to their attention even if the bad call is never executed) and programmer productivity (because programmers don't have to write test cases just to force

the execution of all calls to these predicates). However, the compiler can perform this check only if the predicate or function containing the call to the formatting predicate or function also constructs the format string and the value list. Most calls to formatting predicates satisfy this condition, but not all.

The main exceptions are predicates intended for logging, whose logic follows this general pattern:

```
maybe_log_message(LogInfo, FormatString, Values, !IO) :-
    log_info_should_log(LogInfo, ShouldLog),
    (
        ShouldLog = no
    ;
        ShouldLog = yes,
        io.format(FormatString, Values, !IO),
        io.flush_output(!IO)
    ).
```

In this case, the compiler cannot check the call to `io.format` in this predicate, because the format string and the values list both come from the caller. If some caller supplies format string and values list arguments that are inconsistent with one another, the call to `io.format` will throw an exception.

The purpose of the '`format_call`' pragma is to remedy this. The pragma

```
:- pragma format_call(pred(maybe_log_message/5),
    format_string_values(2, 3)).
```

tells the compiler that calls to this predicate should be checked the same way as calls to the four formatting predicates and one formatting function listed above, with the format string in the second argument, and the values in the list in the third. This way, while `maybe_log_message` cannot ensure that `FormatString` and `Values` will match, its *callers* can do so.

In general, this pragma may take these four forms.

```
:- pragma format_call(pred(Name/Arity),
    format_string_values(FormatStringArgNum, ValuesArgNum)).
:- pragma format_call(func(Name/Arity),
    format_string_values(FormatStringArgNum, ValuesArgNum)).
:- pragma format_call(pred(Name/Arity),
    [format_string_values(FS1, V1), ...]).
:- pragma format_call(func(Name/Arity),
    [format_string_values(FS1, V1), ...]).
```

The first form is the one in the example above, while the second is the function version. The third and fourth forms differ from the first and second respectively by having the second argument being not a single `format_string_values(`*`FormatStringArgNum`*`, `*`ValuesArgNum`*`)` term, but a list of two or more such terms. (One such term is also allowed, but in that case, there is no need for the list.) This latter capability is intended for use in situations where the predicate or function concerned contains several (possibly conditionally executed) calls to formatting predicates or functions whose format strings and values come from the caller, like this:

```
maybe_quad_log_message(LogInfo,
        FmtA, ValuesA1, ValuesA2,
        FmtB, ValuesB1, ValuesB2, !IO) :-
    ...
    io.format(FmtA, ValuesA1, !IO),
    ...
    io.format(FmtA, ValuesA2, !IO),
    ...
    io.format(FmtB, ValuesB1, !IO),
    ...
    io.format(FmtB, ValuesB2, !IO),
    ...
```

In such cases, programmers can include a `format_string_values` entry describing the argument numbers that act as the sources for each such call, like this:

```
:- pragma format_call(pred(maybe_quad_log_message/9),
    [format_string_values(2, 3), format_string_values(2, 4),
    format_string_values(5, 6), format_string_values(5, 7)]).
```

This will ask the compiler to check the compatibility of all four listed argument pairs at all call sites.

## 20.7 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 an 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.

## 20.8 Old pragma syntax

Many of the pragmas above specify the identity of a predicate or a function as the entity to which the pragma applies. Their documentation shows these pragmas to have this syntax:

```
:- pragma pragma_name(pred(Name/Arity)).
:- pragma pragma_name(func(Name/Arity)).
```

New code should use this syntax. However, old versions of the Mercury compiler supported only a simpler version of this syntax, like this:

```
:- pragma pragma_name(Name/Arity).
```

Since this syntax does not specify whether the pragma is supposed to apply to a predicate or to a function, it is ambiguous in the event that the program contains both a predicate and a function with the given name and arity.

For backwards compatibility, the Mercury compiler still supports this syntax, but it will now generate a warning when the program contains both a predicate and a function with the given name and arity. It can also be asked to generate a warning for all pragmas that should specify whether they apply to a predicate or to a function but do not do so.

A later version of Mercury will deprecate this syntax, and a still later version will stop supporting it.

# 21 Implementation-dependent extensions

The Melbourne Mercury implementation supports the following extensions to the Mercury language:

## 21.1 Fact tables

Large tables of facts can be compiled using a different algorithm that is more efficient and can produce more efficient code.

Declarations of the forms

```
:- pragma fact_table(pred(Name/Arity), FileName).
:- pragma fact_table(func(Name/Arity), FileName).
```

tell 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`.

## 21.2 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 an 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.)

When targeting the generation of C code, the current Mercury implementation supports three different pragmas for tabling, to cover these three cases: '`loop_check`', '`memo`', and '`minimal_model`'. (None of these are supported when targeting the generation of C# or Java code.)

- 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 no 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(pred(Name/Arity)).
:- pragma memo(pred(Name/Arity), [list of tabling attributes]).
:- pragma loop_check(pred(Name/Arity)).
:- pragma loop_check(pred(Name/Arity), [list of tabling attributes]).
:- pragma minimal_model(pred(Name/Arity)).
:- pragma minimal_model(pred(Name/Arity), [list of tabling attributes]).
```

and the corresponding versions in which '`pred`' is replaced with '`func`'. The '`pred(`*`Name/Arity`*`)`' or '`func(`*`Name/Arity`*`)`' part specifies the predicate or function to

which the declaration applies. At most one of these declarations may be specified for any given predicate or function.

Pragmas using the above syntax specify a declaration that applies to all the modes of a predicate or function. Programmers can request the application of tabling to only 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]).
```

All the above example pragmas are for predicates. For functions, the first argument of the pragma would include the mode of the function result as well, like this:

```
:- pragma memo(Name(in, in) = 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 16.4 [Using foreign types from Mercury], page 128).

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.

As mentioned above, the Mercury compiler implements tabling only when targeting the generation of C code. In other grades, the compiler normally generates a warning for each

tabling pragma that it is forced to ignore. The '`disable_warning_if_ignored`' attribute tells the compiler not to generate such a warning for the pragma it is attached to. Since the '`loop_check`' and '`minimal_model`' pragmas affect the semantics of the program, and such changes should not be made silently, this attribute may not be specified for them. But this attribute may be specified for '`memo`' pragmas, since these affect only the program's performance, not its semantics.

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 a declaration like this can also be specified without giving the argument modes:

```
:- pragma memo(pred(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 184. 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 Eighth 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.

## 21.3 Termination analysis

The compiler includes a termination analyser which can be used to prove termination of predicates and functions. Details of the analysis are available in "Termination Analysis for Mercury" by Chris Speirs, Zoltan Somogyi and Harald Sondergaard. See [[1]], page 184.

The analysis is based on 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 184.

For an introduction to termination analysis for logic programs, please refer to "Termination Analysis for Logic Programs" by Chris Speirs. See [[3]], page 184.

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(pred(Name/Arity)).
:- pragma terminates(func(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(pred(Name/Arity)).
:- pragma does_not_terminate(func(Name/Arity)).
```

This declaration may be used to inform the compiler that this predicate or function may not terminate. This declaration affects not only the predicate or function specified but also any other predicates and/or functions that are mutually recursive with it.

```
:- pragma check_termination(pred(Name/Arity)).
:- pragma check_termination(func(Name/Arity)).
```

This pragma tells the compiler that it should try to prove the termination of this predicate or function, and if it fails, then it should quit with an error message.

## 21.4 Ensuring tail recursion

In declarative languages such as Mercury, iteration must be implemented using recursive code, in which a predicate or function calls itself, with (hopefully) changed input arguments,

either directly (*self* recursion), or indirectly through other predicates or functions (*mutual* recursion). In general, each call consumes some stack space, which means that without special treatment, the amount of stack space consumed by an iteration is proportional to the number of iterations. If the number of iterations is large enough, the amount of stack space required can exceed the amount available, and this *stack exhaustion* will lead to a crash.

The standard technique for avoiding this is *tail call optimization*. This technique is applicable whenever a call is the very last thing that a predicate or function does. This means not just that it is not followed by any computation, but also that it is not followed by any argument shuffling, because the vector of the output arguments of the call matches exactly the vector of the output arguments of the procedure containing the call. Tail call optimization exploits this by freeing the stack space of the caller just before the call, allowing the callee to reuse its stack space.

*Tail recursion optimization* is the specific case of *tail call optimization* where the tail call is (directly or mutually) recursive. However, it is a very important specific case, because it can significantly improve both robustness and performance. The reason for this is that when tail recursion optimization is applied to *all* the recursive calls in an iteration, then the optimized code can handle an unbounded number of iterations using constant (and therefore bounded) stack space. This is often critical in code that has to handle large amounts of data. Even in situations where stack exhaustion is not an issue, tail recursion can still be critical for good performance, because it leads to much better memory locality for references to the stack, and therefore makes better use of caches.

The '`require_tail_recursion`' pragma asks the compiler to generate a diagnostic when any recursive call in a named predicate or function (with the exception mentioned below) is not *tail* recursive. This can be useful both

- when initially writing the predicate or function, to verify that its recursive calls are tail recursive, and
- when maintaining the predicate or function, to verify that changes to its code have not destroyed this property.

The exception applies to conjunctions that contain two or more recursive calls, as in the recursive clause of quicksort. Since obviously only the last of these calls can be a tail call, generating a diagnostic for each non-last call would be just a nuisance. Therefore this pragma applies only to the last recursive call in each conjunction.

Note that when compiling a module, the compiler cannot see

- the code of predicates and functions defined in other modules, because of separate compilation (though intermodule optimization allows some exceptions to this), or
- the code or even the identities of the callees of higher-order calls, including type class method calls, since the callee is selected at runtime, not compile time.

Thus, for the purposes of warning about non-tail recursion, the compiler considers both intermodule calls and higher-order calls to be non-recursive.

The simplest form of this pragma has a single argument that specifies the name and arity of the predicate or a function it is intended to apply to:

```
:- pragma require_tail_recursion(pred(PredName/Arity)).
:- pragma require_tail_recursion(func(FuncName/Arity)).
```

Such a pragma asks the compiler to generate a warning if any recursive call in the body of the named predicate or function, whether it be self-recursive or mutually recursive, is not a *tail* call. It also asks for the warning to be generated only in grades that permit tail recursion at all. (The implementation of some grades, including grades that permit debugging or perform deep profiling, need to get control for their own purposes at the end of every clause, which means that *no* call can be a tail call.)

(The '`pred()`' and '`func()`' wrappers around the name/arity pair are optional in the usual case where there is no risk of confusion, but it is good practice to include them anyway.)

By default, the pragma applies to all the modes of the specified predicate or function. If you want it to apply to only a specific subset of those modes, you can use the first form below for a predicate and the second form for a function:

```
:- pragma require_tail_recursion(Name(Modes)).
:- pragma require_tail_recursion(Name(Modes) = ReturnMode).
```

These forms are almost never needed, because in practice, the vast majority of predicates and functions have just one mode.

The default is for a '`require_tail_recursion`' pragma to ask the compiler to generate a warning if any recursive call in the body of the named predicate or function (with the exception of non-last recursive calls in a conjunction) whether it be self-recursive or mutually recursive, is not a *tail* call. It also asks for the warning to be generated only in grades that permit tail recursion at all. However, programmers can specify options that change one or more of these choices. They can do so by adding to the pragma a second argument that consists of a list of one or more of the options listed in this table, which represent three possible binary choices.

`warn` Every recursive call (other than a non-last recursive call in a conjunction) should generate a warning if it is not *tail* recursive. This option, which is the default, is incompatible with '`error`'.

`error` Every recursive call (other than a non-last recursive call in a conjunction) should generate an error if it is not *tail* recursive. This option is incompatible with '`warn`'.

`self_or_mutual_recursion`
Allow the compiler to generate a diagnostic (warning or error) for both self-recursive and mutually recursive calls. This option, which is the default, is incompatible with '`self_recursion_only`'.

`self_recursion_only`
Allow the compiler to generate a diagnostic (warning or error) only for self-recursive calls; do not generate diagnostics for mutually recursive calls that are not tail recursive. This option is incompatible with '`self_or_mutual_recursion`'.

`in_tailrec_grades_only`
Tell the compiler to generate a diagnostic for non-tail recursion only if the grade allows for the possibility of tail recursion. This disables all such diagnostics in grades in which *no* call can be a tail call. This option, which is the default, is incompatible with '`in_all_grades`'.

`in_all_grades`
: Tell the compiler to generate a diagnostic for non-tail recursion even in grades that do not allow the possibility of tail recursion. In such grades, which includes debugging, deep profiling, and minimal model grades, this will generate a diagnostic for *every* recursive call. The resulting diagnostics can be a useful reminder if you were intending to debug or to profile the program with data sets that would exhaust the stack without tail recursion, but they are likely to be nuisances otherwise. This option is incompatible with '`in_tailrec_grades_only`'.

If you want to be told about *all* recursive calls in a module that are not tail calls, you *could* add a '`require_tail_recursion`' pragma for every predicate and function in the module. However, there is an easier way to achieve the same effect: using the '`--warn-non-tail-recursion`' option. This option takes an argument which can be one of two words: either '`self-and-mutual`', or just '`self`'. The first specifies the '`self_or_mutual_recursion`' for all the implicitly generated pragmas, while the second likewise specifies '`self_recursion_only`'. In both cases, the other two choices will be set to '`warn`' and '`in_tailrec_grades_only`'.

If you think that the implicit pragmas specified by either form of this option are right for *almost, but not quite all* of the predicates and functions in a module, you can still use the option, and override its effect where necessary. You can do this either

- by explicitly specifying a '`require_tail_recursion`' pragma for a predicate or function, which will override the one that would be implicitly specified by the option,
- or by specifying a '`disable_non_tail_recursion_reports`' pragma for a predicate or function, which will turn off all non-tail recursion diagnostics for that predicate or function. These have the same forms as '`require_tail_recursion`' pragmas:

  ```
  :- pragma disable_non_tail_recursion_reports(pred(PredName/Arity)).
  :- pragma disable_non_tail_recursion_reports(func(FuncName/Arity)).
  :- pragma disable_non_tail_recursion_reports(Name(Modes)).
  :- pragma disable_non_tail_recursion_reports(Name(Modes) = ReturnMode).
  ```

  just with a different pragma name, and with no second argument allowed.

It is worth mentioning that the compiler may generate different sets of warnings (or errors, if you asked for them) in two different grades even if both grades support tail recursion.

The kind of code which is most likely to cause users to encounter this fact is a set of two or more mutually recursive predicates in which some recursive calls are not tail calls. As an example, consider predicates `p1` and `p2`, where `p1` calls `p2` via a tail call, but `p2` calls `p1` via a non-tail call.

- With '`--no-high-level-code`', the Mercury compiler generates code that does its own stack management, which means that the compiler can apply tail call optimization to the call from `p1` to `p2`, and will generate a warning only for the call from `p2` to `p1`.
- With '`--high-level-code`', the Mercury compiler generates code in another high level language (C, C#, or Java) and stack management is the responsibility of the target language. Since the Mercury compiler cannot ensure the application of tail call optimization to the call from `p1` to `p2`, it will generate a warning for both calls.

While the set of diagnostics you get differs in the two cases, their overall message is much the same. This is because the iteration represented by these two predicates will consume an amount of stack space that is linear in the number of iterations *even with* '`--no-high-level-code`'. This is because in that case, the compiler can optimize away half of the iteration's stack frames (`p1`'s stack frames), but it can do nothing about the other half (`p2`'s stack frames). The presence or absence of '`--high-level-code`' thus alters the constant factor, but not the overall memory complexity function, which is still linear. Either way, programmers who wish to exclude the possibility of stack exhaustion, or at least make it much less likely, will need to modify the code.

One technique that can reduce the need for stack space, though without reducing it to a constant amount, is two level iteration. Consider a non-tail-recursive loop that processes a list of M items, where the stack is not big enough to store M stack frames at the same time. You can replace this with

- an inner loop that handles the first (up to) N items of the list (where N is much less than M), and then returns to its caller both the results so far, and the list of remaining items;
- and an outer loop, which calls the inner loop repeatedly to process the next chunk of items, until all items have been processed.

The key insight here is that having the inner loop return after processing N items replaces N stack frames of the inner loop predicate with *one* stack frame of the outer loop predicate. This technique can handle up to N*N items while never using more than 2*N stack frames (N frames of the inner and N frames of the outer predicate).

## 21.5 Feature sets

The Melbourne Mercury implementation supports a number of optional compilation model features, such as Section 21.6 [Trailing], page 177 or Section 21.2 [Tabled evaluation], page 167. 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 using 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 21.2 [Tabled evaluation], page 167).

‘`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 21.6 [Trailing], page 177. 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 submodules.

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.

## 21.6 Trailing

In certain compilation grades (see the “Compilation model options” section of the Mercury User’s Guide), the 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.

### 21.6.1 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.

### 21.6.2 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 addresses of Mercury data structures that have been passed to C via the foreign language interface are guaranteed to be appropriately aligned.)

### 21.6.3 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 the current choice point, or backtracks past the current choice point to some earlier choice point, 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.

### 21.6.4 Delayed goals and floundering

Another use for the function trail is to 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 there 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.

### 21.6.5 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 choice point is pruned away by a commit, but is not meaningful after backtracking past the point where the choice point was created (since choice point 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)` returns true iff the choice point indicated by *x* is newer than (i.e. was created more recently than) the choice point indicated by *y*. The null ChoicepointId is considered older than any non-null ChoicepointId. If either of the choice points has 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 choice point 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_choicepoint_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 21.2 [Tabled evaluation], page 167), 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;
    ").
```

# 22 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 https://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 https://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 https://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 from https://user.it.uu.se/~kostis/Papers/cat.ps.gz.