# Pragmas

GHC supports several pragmas, or instructions to the compiler placed in the source code. Pragmas don’t normally affect the meaning of the program, but they might affect the efficiency of the generated code.

Pragmas all take the form {-# word ... #-} where ⟨word⟩ indicates the type of pragma, and is followed optionally by information specific to that type of pragma. Case is ignored in ⟨word⟩. The various values for ⟨word⟩ that GHC understands are described in the following sections; any pragma encountered with an unrecognised ⟨word⟩ is ignored.

Certain pragmas are file-header pragmas:

* A file-header pragma must precede the module keyword in the file.
* There can be as many file-header pragmas as you please, and they can be preceded or followed by comments.
* File-header pragmas are read once only, before pre-processing the file (e.g. with cpp).
* The file-header pragmas are: {-# LANGUAGE #-}, {-# OPTIONS\_GHC #-}, and {-# INCLUDE #-}.

## 1. LANGUAGE pragma

**{-# LANGUAGE ⟨ext⟩, ⟨ext⟩, ... #-}**

|  |  |
| --- | --- |
| **Where:** | file header |

Enable or disable a set of language extensions.

The LANGUAGE pragma allows language extensions to be enabled in a portable way. The LANGUAGE pragma should be used instead of OPTIONS\_GHC, if possible.

For example, to enable the FFI and preprocessing with CPP:

*{-# LANGUAGE ForeignFunctionInterface, CPP #-}*

LANGUAGE is a file-header pragma.

A list of all supported language extensions can be obtained by invoking ghc --supported-extensions (see **--supported-extensions**).

Any extension from the Extension type defined in Haskell Language may be used. GHC will report an error if any of the requested extensions is not supported.

## 2. OPTIONS\_GHC pragma

**{-# OPTIONS\_GHC ⟨flags⟩ #-}**

|  |  |
| --- | --- |
| **Where:** | file header |

The OPTIONS\_GHC pragma is used to specify additional options that are given to the compiler when compiling this source file. Previous versions of GHC accepted OPTIONS rather than OPTIONS\_GHC, but that is now deprecated.

***OPTIONS\_GHC is a file-header pragma.***

## 3. INCLUDE pragma

The INCLUDE used to be necessary for specifying header files to be included when using the FFI and compiling via C. It is no longer required for GHC, but is accepted (and ignored) for compatibility with other compilers.

## 4. WARNING and DEPRECATED pragmas

**{-# WARNING #-}**

|  |  |
| --- | --- |
| **Where:** | declaration |

The **WARNING** pragma allows you to attach an arbitrary warning to a particular function, class, or type.

**{-# DEPRECATED #-}**

|  |  |
| --- | --- |
| **Where:** | declaration |

A **DEPRECATED** pragma lets you specify that a particular function, class, or type is deprecated.

There are two ways of using these pragmas.

* You can work on an entire module thus:
* **module** Wibble *{-# DEPRECATED "Use Wobble instead" #-}* **where**
* **...**

Or:

**module** Wibble *{-# WARNING "This is an unstable interface." #-}* **where**

**...**

When you compile any module that import Wibble, GHC will print the specified message.

* You can attach a warning to a function, class, type, or data constructor, with the following top-level declarations:
* *{-# DEPRECATED f, C, T "Don't use these" #-}*
* *{-# WARNING unsafePerformIO "This is unsafe; I hope you know what you're doing" #-}*

When you compile any module that imports and uses any of the specified entities, GHC will print the specified message.

You can only attach to entities declared at top level in the module being compiled, and you can only use unqualified names in the list of entities. A capitalised name, such as T refers to either the type constructor T or the data constructor T, or both if both are in scope. If both are in scope, there is currently no way to specify one without the other (c.f. fixities Infix type constructors, classes, and type variables).

Also note that the argument to DEPRECATED and WARNING can also be a list of strings, in which case the strings will be presented on separate lines in the resulting warning message,

*{-# DEPRECATED foo, bar ["Don't use these", "Use gar instead"] #-}*

*Warnings and deprecations are not reported for (a) uses within the defining module, (b) defining a method in a class instance, and (c) uses in an export list. You can suppress the warnings with the flag*[***-Wno-warnings-deprecations***](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/using-warnings.html#ghc-flag--Wwarnings-deprecations)*.*

## 5. MINIMAL pragma

**{-# MINIMAL ⟨name⟩ | ⟨name⟩ , ... #-}**

|  |  |
| --- | --- |
| **Where:** | in class body |

Define the methods needed for a minimal complete instance of a class.

The MINIMAL pragma is used to specify the minimal complete definition of a class, i.e. specify which methods must be implemented by all instances. If an instance does not satisfy the minimal complete definition, then a warning is generated. This can be useful when a class has methods with circular defaults. For example

**class** **Eq** a **where**

**(==)** **::** a **->** a **->** **Bool**

**(/=)** **::** a **->** a **->** **Bool**

x **==** y **=** not **(**x **/=** y**)**

x **/=** y **=** not **(**x **==** y**)**

*{-# MINIMAL (==) | (/=) #-}*

Without the MINIMAL pragma no warning would be generated for an instance that implements neither method.

The syntax for minimal complete definition is:

mindef **::=** name

**|** '(' mindef ')'

**|** mindef '|' mindef

**|** mindef ',' mindef

A vertical bar denotes disjunction, i.e. one of the two sides is required. A comma denotes conjunction, i.e. both sides are required. Conjunction binds stronger than disjunction.

## 6. INLINE and NOINLINE pragmas

These pragmas control the inlining of function definitions.

### INLINE pragma

**{-# INLINE ⟨name⟩ #-}**

|  |  |
| --- | --- |
| **Where:** | any function definition |

Force GHC to inline a value.

GHC (with [**-O**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/using-optimisation.html#ghc-flag--O), as always) tries to inline (or “unfold”) functions/values that are “small enough,” thus avoiding the call overhead and possibly exposing other more-wonderful optimizations. The sledgehammer you can bring to bear is the INLINE pragma, used thusly:

key\_function **::** **Int** **->** **String** **->** **(Bool,** **Double)**

*{-# INLINE key\_function #-}*

The major effect of an INLINE pragma is to declare a function’s “cost” to be very low. The normal unfolding machinery will then be very keen to inline it. However, an INLINE pragma for a function “f” has a number of other effects:

* While GHC is keen to inline the function, it does not do so blindly. For example, if you write
* map key\_function xs

there really isn’t any point in inlining key\_function to get

map **(**\x **->** body**)** xs

In general, GHC only inlines the function if there is some reason (no matter how slight) to suppose that it is useful to do so.

* Moreover, GHC will only inline the function if it is fully applied, where “fully applied” means applied to as many arguments as appear (syntactically) on the LHS of the function definition. For example:
* comp1 **::** **(**b **->** c**)** **->** **(**a **->** b**)** **->** a **->** c
* *{-# INLINE comp1 #-}*
* comp1 f g **=** \x **->** f **(**g x**)**
* comp2 **::** **(**b **->** c**)** **->** **(**a **->** b**)** **->** a **->** c
* *{-# INLINE comp2 #-}*
* comp2 f g x **=** f **(**g x**)**

The two functions comp1 and comp2 have the same semantics, but comp1 will be inlined when applied to two arguments, while comp2 requires three. This might make a big difference if you say

map **(**not **`**comp1**`** not**)** xs

which will optimise better than the corresponding use of comp2.

* It is useful for GHC to optimise the definition of an INLINE function f just like any other non-INLINE function, in case the non-inlined version of f is ultimately called. But we don’t want to inline the optimised version of f; a major reason for INLINE pragmas is to expose functions in f’s RHS that have rewrite rules, and it’s no good if those functions have been optimised away.

So GHC guarantees to inline precisely the code that you wrote, no more and no less. It does this by capturing a copy of the definition of the function to use for inlining (we call this the “inline-RHS”), which it leaves untouched, while optimising the ordinarily RHS as usual. For externally-visible functions the inline-RHS (not the optimised RHS) is recorded in the interface file.

* An INLINE function is not worker/wrappered by strictness analysis. It’s going to be inlined wholesale instead.

#### 7. INLINE pragma effects on various locations

Syntactically, an INLINE pragma for a function can be put anywhere its type signature could be put. This means a INLINE pragma can really be put on any definition site for a binding. This includes top-level, let and where bindings as well as default class methods and instance declarations. The pragma itself will only have an effect when the RHS of the binding.

If we consider a class definition with two instances like this:

**class** **C** a **where**

op1 **::** a **->** a

op2 **::** **[**a**]** **->** **[**a**]**

op2 xs **=** reverse **(**xs **++** xs**)**

*{-# INLINE op2 #-}*

**instance** **C** **T1** **where**

op1 x **=** **...**blah**...**

**instance** **C** **T2** **where**

*{-# INLINE op1 #-}*

op1 x **=** **...**blah**...**

op2 xs **=** **...**blah**...**

Then op2 for the T1 instance will get an implicit INLINE pragma. This is because the RHS of the default method is used for op2 which retains it’s INLINE pragma.

In the T2 instance op1 gets an INLINE pragma and behaves accordingly. However op2 for T2 is **not** implemented by the default method. This means the pragma in the class definition doesn’t apply to this instance. With no pragma being explicitly applied GHC will then decide on a proper inlining behaviour for T2s op2 method on it’s own.

### 8. INLINABLE pragma

**{-# INLINABLE ⟨name⟩ #-}**

|  |  |
| --- | --- |
| **Where:** | any function definition |

Suggest that the compiler always consider inlining **name**.

An {-# INLINABLE f #-} pragma on a function f has the following behaviour:

* While INLINE says “please inline me”, the INLINABLE says “feel free to inline me; use your discretion”. In other words the choice is left to GHC, which uses the same rules as for pragma-free functions. Unlike INLINE, that decision is made at the call site, and will therefore be affected by the inlining threshold, optimisation level etc.
* Like INLINE, the INLINABLE pragma retains a copy of the original RHS for inlining purposes, and persists it in the interface file, regardless of the size of the RHS.

### 9. NOINLINE pragma

**{-# NOINLINE ⟨name⟩ #-}**

|  |  |
| --- | --- |
| **Where:** | any function definition |

Instructs the compiler not to inline a value.

The [**NOINLINE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html#pragma-NOINLINE) pragma does exactly what you’d expect: it stops the named function from being inlined by the compiler. You shouldn’t ever need to do this, unless you’re very cautious about code size.

NOTINLINE is a synonym for NOINLINE (NOINLINE is specified by Haskell 98 as the standard way to disable inlining, so it should be used if you want your code to be portable).

### 10. CONLIKE modifier

**{-# CONLIKE #-}**

|  |  |
| --- | --- |
| **Where:** | modifies [**INLINE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html#pragma-INLINE) or [**NOINLINE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html#pragma-NOINLINE) pragma |

Instructs GHC to consider a value to be especially cheap to inline.

An [**INLINE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html#pragma-INLINE) or [**NOINLINE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html#pragma-NOINLINE) pragma may have a [**CONLIKE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html#pragma-CONLIKE) modifier, which affects matching in [**RULE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/rewrite_rules.html#pragma-RULES)s.

### 11. Phase control

Sometimes you want to control exactly when in GHC’s pipeline the [**INLINE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html#pragma-INLINE) pragma is switched on. Inlining happens only during runs of the simplifier. Each run of the simplifier has a different phase number; the phase number decreases towards zero.

The same information is summarised here:

-- Before phase 2 Phase 2 and later

{-# INLINE [2] f #-} -- No Yes

{-# INLINE [~2] f #-} -- Yes No

{-# NOINLINE [2] f #-} -- No Maybe

{-# NOINLINE [~2] f #-} -- Maybe No

{-# INLINE f #-} -- Yes Yes

{-# NOINLINE f #-} -- No No

By “Maybe” we mean that the usual heuristic inlining rules apply (if the function body is small, or it is applied to interesting-looking arguments etc). Another way to understand the semantics is this:

* For both [**INLINE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html#pragma-INLINE) and [**NOINLINE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html#pragma-NOINLINE), the phase number says when inlining is allowed at all.
* The [**INLINE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html#pragma-INLINE) pragma has the additional effect of making the function body look small, so that when inlining is allowed it is very likely to happen.

The same phase-numbering control is available for [**RULE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/rewrite_rules.html#pragma-RULES)s ([Rewrite rules](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/rewrite_rules.html#rewrite-rules)).

## 12. OPAQUE pragma

**{-# OPAQUE ⟨name⟩ #-}**

|  |  |
| --- | --- |
| **Where:** | top-level |

Instructs the compiler to ensure that every call of **name** remains a call of **name**, and not some name-mangled variant.

The [**OPAQUE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html#pragma-OPAQUE) pragma is an even stronger variant of the [**NOINLINE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html#pragma-NOINLINE) pragma. Like the [**NOINLINE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html#pragma-NOINLINE), named functions annotated with a [**OPAQUE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html#pragma-OPAQUE) pragma are not inlined, nor will they be be specialized. Unlike the [**NOINLINE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html#pragma-NOINLINE), named functions annotated with a [**OPAQUE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html#pragma-OPAQUE) pragma are left untouched by the Worker/Wrapper transformation. Unlike [**NOINLINE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html#pragma-NOINLINE), [**OPAQUE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html#pragma-OPAQUE) has no phase control.

## 13. LINE pragma

**{-# LINE ⟨lineno⟩ "⟨file⟩" #-}**

|  |  |
| --- | --- |
| **Where:** | anywhere |

Generated by preprocessors to convey source line numbers of the original source.

This pragma is similar to C’s #line pragma, and is mainly for use in automatically generated Haskell code

## 14. COLUMN pragma

This is the analogue of the LINE pragma and is likewise intended for use in automatically generated Haskell code. It lets you specify the column number of the original code; for example

foo **=** **do**

*{-# COLUMN 42 #-}*pure ()

pure ()

This adjusts all column numbers immediately after the pragma to start at 42. The presence of this pragma only affects the quality of the diagnostics and does not change the syntax of the code itself.

## 15. RULES pragma

The [**RULES**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/rewrite_rules.html#pragma-RULES) pragma lets you specify rewrite rules.

## 16. SPECIALIZE pragma

**{-# SPECIALIZE ⟨name⟩ :: ⟨type⟩ #-}**

Ask that GHC specialize a polymorphic value to a particular type. For key overloaded functions, you can create extra versions (NB: at the cost of larger code) specialised to particular types. Thus, if you have an overloaded function:

hammeredLookup **::** **Ord** key **=>** **[(**key**,** value**)]** **->** key **->** value

If it is heavily used on lists with Widget keys, you could specialise it as follows:

*{-# SPECIALIZE [0] hammeredLookup :: [(Widget, value)] -> Widget -> value #-}*

generates a specialisation rule that only fires in Phase 0 (the final phase). If you do not specify any phase control in the SPECIALIZE pragma, the phase control is inherited from the inline pragma (if any) of the function. For example:

foo **::** **Num** a **=>** a **->** a

foo **=** **...**blah**...**

*{-# NOINLINE [0] foo #-}*

*{-# SPECIALIZE foo :: Int -> Int #-}*

The NOINLINE pragma tells GHC not to inline foo until Phase 0; and this property is inherited by the specialisation RULE, which will therefore only fire in Phase 0.

Here are some examples (where we only give the type signature for the original function, not its code):

f **::** **Eq** a **=>** a **->** b **->** b

*{-# SPECIALISE f :: Int -> b -> b #-}*

g **::** **(Eq** a**,** **Ix** b**)** **=>** a **->** b **->** b

*{-# SPECIALISE g :: (Eq a) => a -> Int -> Int #-}*

h **::** **Eq** a **=>** a **->** a **->** a

*{-# SPECIALISE h :: (Eq a) => [a] -> [a] -> [a] #-}*

The last of these examples will generate a RULE with a somewhat-complex left-hand side (try it yourself), so it might not fire very well. If you use this kind of specialisation, let us know how well it works.

### 17. SPECIALIZE INLINE

**{-# SPECIALIZE INLINE ⟨name⟩ :: ⟨type⟩ #-}**

|  |  |
| --- | --- |
| **Where:** | top-level |

A SPECIALIZE pragma can optionally be followed with a INLINE or NOINLINE pragma, optionally followed by a phase, as described in [INLINE and NOINLINE pragmas](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html#inline-noinline-pragma). The INLINE pragma affects the specialised version of the function (only), and applies even if the function is recursive. The motivating example is this:

*-- A GADT for arrays with type-indexed representation*

**data** **Arr** e **where**

**ArrInt** **::** **!Int** **->** **ByteArray#** **->** **Arr** **Int**

**ArrPair** **::** **!Int** **->** **Arr** e1 **->** **Arr** e2 **->** **Arr** **(**e1**,** e2**)**

**(!:)** **::** **Arr** e **->** **Int** **->** e

*{-# SPECIALISE INLINE (!:) :: Arr Int -> Int -> Int #-}*

*{-# SPECIALISE INLINE (!:) :: Arr (a, b) -> Int -> (a, b) #-}*

**(ArrInt** **\_** ba**)** **!:** **(I#** i**)** **=** **I#** **(**indexIntArray**#** ba i**)**

**(ArrPair** **\_** a1 a2**)** **!:** i **=** **(**a1 **!:** i**,** a2 **!:** i**)**

Here, (!:) is a recursive function that indexes arrays of type Arr e. Consider a call to (!:) at type (Int,Int). The second specialisation will fire, and the specialised function will be inlined. It has two calls to (!:), both at type Int. Both these calls fire the first specialisation, whose body is also inlined. The result is a type-based unrolling of the indexing function.

### 18. SPECIALIZE for imported functions

Generally, you can only give a [**SPECIALIZE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html#pragma-SPECIALIZE) pragma for a function defined in the same module. However if a function f is given an [**INLINABLE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html#pragma-INLINABLE) pragma at its definition site, then it can subsequently be specialised by importing. For example

**module** Map**(** lookup**,** blah blah **)** **where**

lookup **::** **Ord** key **=>** **[(**key**,**a**)]** **->** key **->** **Maybe** a

lookup **=** **...**

*{-# INLINABLE lookup #-}*

**module** Client **where**

**import** Map**(** lookup **)**

**data** **T** **=** **T1** **|** **T2** **deriving(** **Eq,** **Ord** **)**

*{-# SPECIALISE lookup :: [(T,a)] -> T -> Maybe a*

Here, lookup is declared [**INLINABLE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html#pragma-INLINABLE), but it cannot be specialised for type T at its definition site, because that type does not exist yet. Instead a client module can define T and then specialise lookup at that type.

Moreover, every module that imports Client (or imports a module that imports Client, transitively) will “see”, and make use of, the specialised version of lookup. You don’t need to put a [**SPECIALIZE**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html#pragma-SPECIALIZE) pragma in every module.

## 19. SPECIALIZE instance pragma

**{-# SPECIALIZE instance ⟨instance head⟩ #-}**

|  |  |
| --- | --- |
| **Where:** | instance body |

Same idea, except for instance declarations. For example:

**instance** **(Eq** a**)** **=>** **Eq** **(Foo** a**)** **where** **{**

*{-# SPECIALIZE instance Eq (Foo [(Int, Bar)]) #-}*

**...** usual stuff **...**

**}**

The pragma must occur inside the where part of the instance declaration.

## 20. UNPACK pragma

**{-# UNPACK #-}**

|  |  |
| --- | --- |
| **Where:** | data constructor field |

Instructs the compiler to unpack the contents of a constructor field into the constructor itself. The UNPACK indicates to the compiler that it should unpack the contents of a constructor field into the constructor itself, removing a level of indirection. For example:

**data** **T** **=** **T** *{-# UNPACK #-}* **!Float**

*{-# UNPACK #-}* **!Float**

will create a constructor T containing two unboxed floats. This may not always be an optimisation: if the T constructor is scrutinised and the floats passed to a non-strict function for example, they will have to be reboxed (this is done automatically by the compiler).

## 21. NOUNPACK pragma

**{-# NOUNPACK #-}**

|  |  |
| --- | --- |
| **Where:** | top-level |

Instructs the compiler not to unpack a constructor field.

The NOUNPACK pragma indicates to the compiler that it should not unpack the contents of a constructor field. Example:

**data** **T** **=** **T** *{-# NOUNPACK #-}* **!(Int,Int)**

Even with the flags [**-funbox-strict-fields**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/using-optimisation.html#ghc-flag--funbox-strict-fields) and [**-O**](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/using-optimisation.html#ghc-flag--O), the field of the constructor T is not unpacked.

## 22. SOURCE pragma

**{-# SOURCE #-}**

|  |  |
| --- | --- |
| **Where:** | after **import** statement |

Import a module by **hs-boot** file to break a module loop.

The {-# SOURCE #-} pragma is used only in import declarations, to break a module loop. It is described in detail in [How to compile mutually recursive modules](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/separate_compilation.html#mutual-recursion).

## 23. COMPLETE pragmas

**{-# COMPLETE #-}**

|  |  |
| --- | --- |
| **Where:** | at top level |

Specify the set of constructors or pattern synonyms which constitute a total match.

The COMPLETE pragma is used to inform the pattern match checker that a certain set of patterns is complete and that any function which matches on all the specified patterns is total.

The most common usage of COMPLETE pragmas is with [Pattern synonyms](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pattern_synonyms.html#pattern-synonyms). On its own, the checker is very naive and assumes that any match involving a pattern synonym will fail. As a result, any pattern match on a pattern synonym is regarded as incomplete unless the user adds a catch-all case.

For example, the data types 2 \* A and A + A are isomorphic but some computations are more naturally expressed in terms of one or the other. To get the best of both worlds, we can choose one as our implementation and then provide a set of pattern synonyms so that users can use the other representation if they desire. We can then specify a COMPLETE pragma in order to inform the pattern match checker that a function which matches on both LeftChoice and RightChoice is total.

**data** **Choice** a **=** **Choice** **Bool** a

pattern **LeftChoice** **::** a **->** **Choice** a

pattern **LeftChoice** a **=** **Choice** **False** a

pattern **RightChoice** **::** a **->** **Choice** a

pattern **RightChoice** a **=** **Choice** **True** a

*{-# COMPLETE LeftChoice, RightChoice #-}*

foo **::** **Choice** **Int** **->** **Int**

foo **(LeftChoice** n**)** **=** n **\*** **2**

foo **(RightChoice** n**)** **=** n **-** **2**

COMPLETE pragmas are only used by the pattern match checker. If a function definition matches on all the constructors specified in the pragma then the compiler will produce no warning.

## 24. OVERLAPPING, OVERLAPPABLE, OVERLAPS, and INCOHERENT pragmas

**{-# OVERLAPPING #-}**

**{-# OVERLAPPABLE #-}**

**{-# OVERLAPS #-}**

**{-# INCOHERENT #-}**

|  |  |
| --- | --- |
| **Where:** | on instance head |

The pragmas OVERLAPPING, OVERLAPPABLE, OVERLAPS, INCOHERENT are used to specify the overlap behavior for individual instances, as described in Section [Overlapping instances](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/instances.html#instance-overlap). The pragmas are written immediately after the instance keyword, like this: