

The Morrow Interpreter

Daan Leijen

(Version 1.0)

1 Introduction

This document is written as a short introduction to using the Morrow interpreter and does not explain the type system or language behind it. We refer the interested reader to some relevant publications [2, 3, 5, 4, 6].

Morrow is small programming language designed for experimentation with advanced type system concepts. In particular, extensible polymorphic records and variants, combined with impredicative higher-ranked type inference. This combination leads to an expressive core language that is able to express many programming language concepts directly without specific extensions. For example, we hope to use these features to elegantly express first-class pattern matching and first-class modules.

Currently, we have only implemented a type checker that can handle some of the features of Morrow: impredicative higher-ranked types, polymorphic kinds, and extensible records and variants. None of these are implemented in a Haskell-like system and the Morrow interpreter can be a nice vehicle to experiment with these features. The interpreter evaluates expressions if it can find the OCaml compiler in the path (`ocamlc`).

2 Examples

If you have build a version of the Morrow interpreter yourself, you can start it using `make`:

```
~/dev/morrow> make interactive
  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
| \ / | / _ \ | ' _ | ' _ | _ \ \ / \ / / | welcome to the morrow interpreter
| \ / | ( ) | | | | | ( ) \ v v / | http://www.cs.uu.nl/~daan/morrow
| | \ / | \ _ _ / | _ | | _ \ _ _ / \ _ \ _ / | type "?" for help
| | | |
| _ | | _ | version 1.0

loading: prelude
loaded :
  c:\daan\dev\morrowF\lib\prelude.mw
>
```

In other cases, one needs to pass the library path to the Morrow interpreter explicitly. For example:

```
~/dev/morrow> out/debug/morrow-1.0 -i lib
```

At the prompt, we can type a Morrow expression which the interpreter will evaluate:

```
> 1
1

> \x -> 1
(0,0): Warning: Only values of a basic type can be shown automatically
Type: forall a. a -> Int
```

Using the `:t` command, we can show the type of expressions:

```
> :t 1
Int

> :t \x -> 1
forall a. a -> Int
```

The syntax of Morrow is evolving, but currently it is a heavily restricted subset of Haskell: no operators, no `case` and `data`, but it does support `newtype` and the corresponding patterns. The lack of `data` types is especially inconvenient, but in the future Morrow will have variants and records to offer a more general approach to data types.

It is possible to create definitions at the interpreter prompt:

```
> let identity x = 1
identity :: forall a. a -> Int
```

The above definition is not the correct identity function, and we can override it by giving a new definition:

```
> let identity x = x
identity :: forall a. a -> a
```

We can view all interactively defined values as source code with the `:d` command:

```
> :d
identity :: forall a. a -> a
identity x = x
```

A full list of interpreter commands is given in section 4.

2.1 Higher-ranked types

Of course, we want to experiment with higher-ranked types. A nice example of higher-ranked types is the `auto` function that applies a function to itself:

```
> let auto f = f f
              ^
((2),16): Unable to unify types (due to an infinite type)
```

```

context      : f f
term        : f
inferred type : a -> b
does not match: a
probable cause: A value is used (recursively) with different types
hint        : Add a polymorphic type signature to the definition?

> let auto = (\f -> f f) :: (forall a. a->a) -> (forall a. a->a)
auto :: (forall a. a -> a) -> (forall a. a -> a)

> let auto (f :: forall a. a -> a) = f f
auto :: (forall a. a -> a) -> (forall a. a -> a)

```

The first definition fails as we try to use argument `f` polymorphically. In that case we need to give an explicit type signature [2]. The second definition gives a type signature for the entire definition. The last definition is an example where we just annotate the argument itself.

We can now pass a polymorphic function like `id` to the `auto` function:

```

> :t auto identity
forall a. a -> a

```

Morrow has an *impredicative* type system. This means that type variables can be instantiated by type schemes. Therefore, we can maintain proper abstraction and do not have to instantiate equal implementations for different types. Here is an example:

```

> let apply f x = f x
apply :: forall a b. (a -> b) -> a -> b

> :t apply auto identity
forall a. a -> a

```

Here we see that we can abstract from application via the `apply` function; note that the type variable `b` of `apply` is instantiated with the type scheme of `id`.

2.2 More polymorphism

Certain functions will get a more polymorphic type than a standard Hindley-Milner type inferencer will infer. Take for example the `const` function (defined in the prelude):

```

> :t const
const :: forall a. a -> (forall b. b -> a)

```

A Haskell compiler would infer the type `forall a b. a -> b -> a`. However, the inferred Morrow type is strictly more general, as a partial application of `const` can be passed as a polymorphic argument. In a higher-ranked system à la GHC [6], the partial application must be bound to a `let` expression to generalize properly.

Just like ML^F [2], Morrow uses *bounded polymorphism* to implement the impredicative type system. The interpreter takes care to hide the bounded types as they are less readable than standard types. However, sometimes the bounded polymorphism can not be hidden:

```
> let choose x y = if True then x else y
choose :: forall a. a -> a -> a

> choose id
forall (a > forall b. b -> b). a -> a
```

In the type of `choose id`, we see that the type variable `a` is *bounded*. The bound `(a > forall b. b -> b)` means that `a` can be any instance of the type scheme `forall b. b -> b`. This means that we can for example pass the argument `id` again (to get a function of type `forall a. a -> a`), but we could also pass an argument of type `Int -> Int` for example.

The bound `>` is called *flexible* and can be read as “must be an instance of”. There is one other bound, `=`, that is called *rigid*, and is read as “must be the same as”. For example, in contrast to `choose id`, the `auto` function expects a truly polymorphic argument, and we can write the type of `auto` more explicitly as:

```
auto :: forall (a = forall b. b -> b) (c > forall d. d -> d). a -> c
```

When we give the interpreter the option `--show-bounded-types`, the interpreter shows the expanded bounded types automatically:

```
> :set --show-bounded-types

> :t auto
forall (a = forall b. b -> b) (b > forall c. c -> c). a -> b
```

Normally, the interpreter inlines the bounded types: rigid bounds are inlined when the type variable occurs only once in a (syntactical) contra-variant position, and flexible bounds are inlined when the type variable occurs only once in a (syntactical) co-variant position. The variance is only determined by function arrows. For example:

```
> List (forall a. a -> a) -> List (forall a. a -> a)
```

The above function type expects a list of real polymorphic identity functions (rigid), but returns a list of polymorphic functions that can still be instantiated (flexible). The explicit bounds after translation are:

```
> forall (b = forall a. a -> a) (c > forall a. a -> a). List b -> List c
```

In general, the automatic translation improves readability enormously for complicated higher-ranked types. A good example is the encoding of church numerals (see `samples/type/churchenc`).

You can find more examples of higher-ranked types in the `samples/type` directory of the Morrow distribution.

3 Polymorphic kinds

Morrow automatically infers the kinds of types. For example:

```
> newtype Id x = Id x
Id : * -> *
```

The kind `*` (star) is the kind of types that denote values. Our new `Id` type takes types of kind `*` to a new value type (of kind `star`).

Morrow also does polymorphic kind inference. Here is an example of a type with a polymorphic kind:

```
> newtype Const x y = Const x
                          ^
((5),17): Warning: Type variable y is unused

Const : kforall k. * -> k -> *
```

As we can see, the constructor `Const` is polymorphic in the kind of the second argument. In Haskell, such argument would be assigned the kind `star` by default. A particularly nice example of the use of polymorphic kinds is the equality data type described by Baars and Swierstra [1]:

```
> newtype Eq a b = Eq (forall f. f a -> f b)
((5),17): Warning: Type variable y is unused

Eq : kforall k. k -> k -> *
```

In contrast to Haskell, the equality data type can now encode equality between arbitrary types, including higher-order functions. Using the new equality data type, we can encode laws like reflection and transitivity:

```
> let reflex = Eq id
reflex :: kforall k. forall (a :: k). Eq a a

> let compose f g = \x -> f (g x)
compose :: forall a b. (a -> b) -> (forall c. (c -> a) -> c -> b)

> let trans (Eq f) (Eq g) = Eq (compose f g)
trans :: kforall k. forall (a :: k) (b :: k).
      Eq b a -> (forall (c :: k). Eq c b -> Eq c a)
```

4 Commands

The following table contains a summary of all interpreter commands. Optional arguments are given between [brackets].

<i>expression</i>		Show the type of the <i>expression</i>
<code>let</code>	<i>definition</i>	Add a value definition
<code>type</code>	<i>definition</i>	Add a type synonym
<code>newtype</code>	<i>definition</i>	Add a new type definition
<code>:q[uit]</code>		Quit the interpreter
<code>:l[oad]</code>	<i>filenames</i>	Load source files
<code>:a[ls]</code>	<i>filenames</i>	Add source files
<code>:r[eload]</code>		Reload the current file
<code>:f[ind]</code>	<i>identifier</i>	Edit the source file containing the <i>identifier</i>
<code>:e[dit]</code>	<i>filename</i>	Edit the source file <i>filename</i>
<code>:e[dit]</code>		Edit the current file (at the last error location)
<code>:cd</code>		Show the current directory
<code>:cd</code>	<i>directory</i>	Change the current directory
<code>:! </code>	<i>command</i>	Execute a shell command
<code>:set</code>	<i>options</i>	Set (command line) options
<code>:?</code>		Show command help
<code>:t[ype]</code>		Show the type signatures of all value definitions
<code>:t[ype]</code>	<i>expression</i>	Show the type of the <i>expression</i>
<code>:k[ind]</code>		Show the kind signatures of all type definitions
<code>:s[ource]</code>		Show the current source file
<code>:s[ource]</code>	<i>identifier</i>	Show the source of the definition of <i>identifier</i>
<code>:d[efines]</code>		Show the source code of interactive definitions
<code>:syms</code>		Show type synonym signatures
<code>:env</code>		Show the Morrow environment variables
<code>:version</code>		Show the interpreter version and warranty

Note that `set` can be used to change the behaviour of the interpreter dynamically.

```
> :set --no-color --show-bounded-types
```

4.1 Command line options

<code>-?,-h</code>	<code>--help</code>	Show command line help
<code>-c</code>	<code>--compiler</code>	Non-interactive mode
	<code>--version</code>	Show the version
<code>-i dirs</code>	<code>--include=dirs</code>	Add <i>dirs</i> to include path (empty resets)
	<code>--outdir=dir</code>	Put intermediate files in <i>dir</i>
	<code>--editor=cmd</code>	Use <i>cmd</i> as editor
	<code>--ocamlc=cmd</code>	Use <i>cmd</i> as the ocaml backend compiler
	<code>--set-color=colors</code>	Set colors
	<code>--warn-shadow</code>	Warn when a variable is shadowed
	<code>--show-kinds</code>	Show full kind annotations
	<code>--show-kind-sigs</code>	Show kind signatures of type definitions
	<code>--show-type-sigs</code>	Show type signatures of definitions
	<code>--show-synonyms</code>	Show expanded type synonyms in types
	<code>--show-bounded-types</code>	Show expanded type bounds
	<code>--show-rows</code>	Show expanded rows
	<code>--show-core</code>	Show core
	<code>--evaluate</code>	Evaluate expressions
	<code>--color</code>	Use color

Boolean flags can be negated using the `no-` prefix. For example, `--no-color` to disable colors, and `--no-evaluate` to disable evaluation and only show types.

4.2 Colors

Colors can be set using a comma separated list of `<category>=<color>` pairs.

```
> morrow-1.0 --set-color=range=red,marker=white
```

Color categories are: interpreter, marker, warning, error, range, source, type, kind, and keyword. For syntax highlighting, the following categories are used: typecon, sep, comment, reserved, reservedop, special, string, and number.

Valid color names are: black darkred, darkgreen, darkyellow, darkblue, darkmagenta, darkcyan, lightgray, gray, red, green, yellow, blue, magenta, cyan, white, default, lightgrey, grey, navy, teal, maroon, purple, olive, silver, lime, aqua, fuchsia, darkgray, and darkgrey.

4.3 Environment variables

Morrow looks for the following environment variables:

<code>morrowoptions</code>	Standard command line options.
<code>morroweditor</code>	The standard editor. <code>%l</code> , <code>%c</code> , and <code>%s</code> are substituted for the line number, column number, and source file.
<code>morrowdir</code>	The Morrow directory. Sets the library path and output directory to standard values (<code>\$morrowdir/lib</code> and <code>\$morrowdir/out/lib</code>).

References

- [1] A. Baars and D. Swierstra. Typing dynamic typing. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 157–166. ACM Press, 2002.

- [2] D. Le Botlan and D. Rémy. MLF: Raising ML to the power of System-F. In *Proceedings of the International Conference on Functional Programming (ICFP 2003)*, Uppsala, Sweden, pages 27–38. ACM Press, aug 2003.
- [3] D. Leijen. Extensible records with scoped labels. Unpublished, Apr. 2005.
- [4] D. Leijen. First-class labels for extensible rows. Draft paper, Jan. 2005.
- [5] D. Leijen and A. Löh. Qualified types for mlf. Unpublished, Apr. 2005.
- [6] S. Peyton-Jones and M. Shields. Practical type inference for arbitrary-rank types. Submitted to the *Journal of Functional Programming (JFP)*, 2004.

A Syntax of Morrow

This appendix specifies the formal syntax of Morrow. Note that this is still work in progress and the syntax is subject to change.

A.1 Notational conventions

These notational conventions are used for presenting syntax:

<i>production</i>	→	[<i>p</i>]	optional
		(<i>p</i>)	grouping
		{ <i>p</i> }*	zero or more <i>p</i>
		{ <i>p</i> } ⁺	one or more <i>p</i>
		(<i>p</i> <i>q</i>) [*]	zero or more <i>p</i> , separated by <i>q</i>
		(<i>p</i> <i>q</i>) ⁺	one or more <i>p</i> , separated by <i>q</i>
		⟨ <i>p</i> <i>q</i> ⟩ [*]	zero or more <i>p</i> , separated or terminated by <i>q</i>
		⟨ <i>p</i> <i>q</i> ⟩ ⁺	one or more <i>p</i> , separated or terminated by <i>q</i>
		<i>p</i> <i>q</i>	choice: <i>p</i> or <i>q</i>
		<i>p</i> (<i>q</i>)	difference: <i>p</i> except those in <i>q</i>
		terminal	terminals are in typewriter font
		x0D	hexadecimal character code

*production*_[*lex*] → ... lexemes are drawn recursively from *lex*

A.2 Program

<i>program</i> _[<i>lex</i>]	→	{ ⟨ <i>declaration</i> ; ⟩ [*] }
<i>declaration</i>	→	<i>fixitydecl</i> <i>newtype</i> <i>synonym</i> <i>termdecl</i>
		<i>external</i>

A.3 Terms

<i>decls</i>	→	{ ⟨ <i>decl</i> ; ⟩ ⁺ }
<i>decl</i>	→	<i>termdecl</i>
<i>termdecl</i>	→	<i>variable</i> { <i>binder</i> } [*] = <i>term</i>

A.3.1 Expressions

<i>term</i>	→	<i>compound</i> <i>annot</i>
<i>compound</i>	→	let <i>decls</i> in <i>term</i> let expression
		if <i>term</i> then <i>term</i> else <i>term</i> conditional
		λ { <i>binder</i> } ⁺ → <i>term</i> lambda expression
<i>annot</i>	→	<i>expr</i> [:: <i>type</i>] type annotation

<i>expr</i>	→ { <i>atom</i> } ⁺ [<i>compound</i>]	operator expression
<i>atom</i>	→ <i>atom</i> . <i>label</i> <i>variable</i> <i>constructor</i> <i>operator</i> <i>literal</i> { } { (<i>field</i> ,) ⁺ [<i>term</i>] } < <i>label</i> { <i>binder</i> } [*] = <i>term</i> > < (<i>label</i> ,) ⁺ <i>term</i> > (<i>term</i> { , <i>term</i> } ⁺) (<i>term</i>)	record selection empty record record construction variant injection variant embedding tuple parenthesized term
<i>field</i>	→ <i>label</i> { <i>binder</i> } [*] (= :=) <i>term</i> - <i>label</i>	extension and update restriction
<i>literal</i>	→ <i>natural</i> <i>float</i> <i>string</i> <i>char</i>	

A.3.2 Patterns

<i>binder</i>	→ <i>patatom</i>
<i>pattern</i>	→ <i>patatom</i> [:: <i>type</i>]
<i>patatom</i>	→ <i>variable</i> <i>constructor patatom</i> (<i>pattern</i>)

A.3.3 Identifiers and operators

<i>label</i>	→ <i>variable</i>	labels
<i>variable</i>	→ <i>varid</i> (<i>varsym</i>)	
<i>operator</i>	→ <i>varsym</i> ‘ <i>varid</i> ‘	
<i>constructor</i>	→ <i>conid</i>	

A.4 Types

<i>newtype</i>	→ newtype <i>tdef</i> { <i>tpar</i> } [*] = <i>constructor tatom</i>
<i>synonym</i>	→ type <i>tdef</i> { <i>tpar</i> } [*] = <i>type</i>

A.4.1 Type expressions

<i>type</i>	→ kforall { <i>kvar</i> } ⁺ . <i>type</i> forall { <i>tbinder</i> } ⁺ . <i>type</i> <i>tfun</i>	kind quantification type quantification
<i>tfun</i>	→ (<i>tapp</i> ->) ⁺	function type
<i>tapp</i>	→ { <i>tatom</i> } ⁺	type application
<i>tatom</i>	→ to <i>tatom</i> <i>tatom</i> <i>tvar</i> <i>tcon</i> { [<i>trow</i>] } < [<i>trow</i>] > ([<i>tfields</i>]) (<i>type</i>)	type operator record type variant type row type parenthesized type
<i>trow</i>	→ <i>tfields</i> <i>tatom</i>	row
<i>tfields</i>	→ (<i>tfield</i> ,) ⁺ [<i>type</i>]	sequence of labeled types
<i>tfield</i>	→ <i>label</i> :: <i>type</i>	labeled type

A.4.2 Type identifiers

<i>tdef</i>	→ <i>tcon</i> (<i>tcon</i> [:: <i>kscheme</i>])	type constructor definition
<i>tpar</i>	→ <i>tvar</i> (<i>tvar</i> [:: <i>kind</i>])	type parameter
<i>tbinder</i>	→ <i>tvar</i> (<i>tvar</i> [:: <i>kind</i>] [<i>bound</i>])	quantified type variable
<i>bound</i>	→ > <i>type</i> = <i>type</i>	flexible bound rigid bound
<i>tvar</i>	→ <i>varid</i>	
<i>tcon</i>	→ <i>conid</i>	

A.4.3 Kind expressions

<i>kscheme</i>	→ kforall { <i>kvar</i> } ⁺ . <i>kscheme</i> <i>kind</i>	kind scheme
<i>kind</i>	→ (<i>katom</i> ->) ⁺	arrow kinds
<i>katom</i>	→ <i>kvar</i> <i>kcon</i> (<i>kind</i>)	kind variable kind constructor parenthesized kind
<i>kcon</i>	→ * Row	types and rows
<i>kvar</i>	→ <i>varid</i>	

A.5 Fixity

fixitydecl → *fixity natural* (*operator* ,)⁺ fixity declaration
fixity → **infixl** | **infixn** | **infixr** association: left, none, right

A.6 External

external → **external** *variable* :: *type* = *string*

B Lexical structure

B.1 Lexemes

lex → { *whitespace* | *lexeme* }^{*}
lexeme → *varid* | *varsym* | *conid*
| *reserved* | *reservedsym* | *special*
| *natural* | *float* | *string* | *char*

B.2 Identifiers

conid → *upperid* constructors
varid → *lowerid*_(*reserved*) variables

lowerid → (*lower* | *_*) { *idchar* }^{*}
upperid → *upper* { *idchar* }^{*}
idchar → *alphanum* | *_* | *'*

reserved → *type* | *newtype*
| *forall* | *kforall*
| *to* | *from*
| *let* | *in*
| *if* | *then* | *else*
| *external*

B.3 Symbols

varsym → *symbolid*_(*reservedsym*) variable symbol
symbolid → *symbol*_(*:*) { *symbol* }^{*}
symbol → *unisymbol*
| \$ | % | & | * | +
| ? | @ | ! | # | / | ^ | ~
| < | > | = | \ | | | . | - | :
reservedsym → < | > | = | \ | | | . | -> | :: | :=
special → (|) | { | } | ; | , | ' |

B.4 Literals

string → " { *graphic*_(*"* | **) | *space* | *escape* | *gap* }^{*} "
char → ' (*graphic*_(*'* | **) | *space* | *escape*) '

escape → \ (*charesc* | *decimal* | *x hexadecimal*)
charesc → a | b | f | n | r | t | v | \ | " | '
gap → \ *whitespace* \

float → *decimal . decimal [exponent]*
exponent → *(e | E) [- | +] decimal*

natural → *decimal | 0 (x | X) hexadecimal*
decimal → *{digit}⁺*
hexadecimal → *{hexdigit}⁺*

B.5 White space

whitespace → *{white | linecomment | blockcomment}⁺*

linecomment → *-- space {linechar}^{*}*
linechar → *graphic | space | tab*

blockcomment → *{- blockchars {blockcomment blockchars}^{*} -}*
blockchars → *many_{(many ({- | -}) many)}*

many → *{any}^{*}*

B.6 Character classes

<i>alphanum</i>	→	<i>alpha digit</i>	
<i>alpha</i>	→	<i>upper lower</i>	
<i>lower</i>	→	<i>a...z unilower</i>	
<i>upper</i>	→	<i>A...Z uniupper</i>	
<i>digit</i>	→	<i>0...9</i>	
<i>hexdigit</i>	→	<i>a...f A...F digit</i>	
<i>any</i>	→	<i>graphic white</i>	
<i>white</i>	→	<i>return linefeed space</i>	
<i>space</i>	→	<i>x20</i>	a space
<i>tab</i>	→	<i>x09</i>	a horizontal tab (<i>\t</i>)
<i>linefeed</i>	→	<i>x0A</i>	a line feed (<i>\n</i>)
<i>return</i>	→	<i>x0D</i>	a carriage return (<i>\r</i>)
<i>graphic</i>	→	<i>x21...x7F unigraphic</i>	visible character
<i>unilower</i>	→	<i>...</i>	unicode lowercase (<i>Ll</i>)
<i>uniupper</i>	→	<i>...</i>	unicode uppercase or titlecase (<i>Lu</i> or <i>Lt</i>)
<i>unisymbol</i>	→	<i>...</i>	unicode symbol (<i>Sm</i>)
<i>unigraphic</i>	→	<i>x80...x10FFFF</i>	any unicode character