Pythas

Release 0.1b1

Simon Plakolb

Mar 30, 2021

CONTENTS

With *Pythas* it is possible to import Haskell modules into Python just as if they were native modules. It automatically creates the FFI exports including all relevant type conversions, compiles and binds your Haskell code to Python at runtime. It was designed with ease of use in mind and sports a pythonic, intuitive interface.

CHAPTER

ONE

DEPENDENCIES

One development goal for *Pythas* was to keep users out of Haskell's package managing details ("cabal hell"). Thus, the package ships with all Haskell and C source files it needs. To compile its own source and other Haskell modules *Pythas* needs either *Stack* or a plain *GHC* installation. For a guide on how to install either refer to *Installation*. Both can be detected automatically if they are already installed. In case *Stack* is installed, it will always be used as the default.

CHAPTER

TWO

BASIC USAGE

Usage is as easy as importing any Python modules. If we have an `example/Example.hs` file containing a Hello World function `hello :: IO ()` which prints "Hello from Haskell" we would import and use it like that:

```
>>> import pythas
>>> import example.example as e
>>> e.hello()
Hello from Haskell!
```

In the *Usage* chapter you can find more details on how to use *Pythas*. More practical examples are collected in a separate repository Pythas-Examples.

CHAPTER

THREE

CONTENTS

3.1 Installation

While *Pythas* itself is a Python library, it contains a considerable amount of Haskell source code. This part of the library is compiled once at the first import. Therefore, it is paramount to provide a *GHC* installation. Note that in subsequent usages this compilation is not necessary, reducing the import time significantly. If you already have either *GHC* or *Stack* (or both) installed, skip ahead to *Install Pythas*.

3.1.1 Install Haskell

For compilation of Haskell source *Pythas* makes use of the *Glasgow Haskell Compiler (GHC)*. This is the most commonly used compiler for the Haskell programming language. It comprises a multitude of extensions to the language and most existing Haskell code should therefore be supported by *Pythas*. However, thorough support of all GHC language extensions cannot be guaranteed and is to be considered experimental at this stage (0.1b1).

Cabal/GHC vs Stack

The GHC compiler can be installed in various ways (without claim to completeness):

- As a bare bones compiler executable.
- As a minimal package also including the build tools *Cabal* and *Stack*.
- On top of the Stack build tool providing management of different GHC installs.
- With the full blown Haskell Platform installation providing the most common libraries and build tools altogether.

If a *Stack* install is available, *Pythas* will default to utilizing it. This means that available `stack.yaml` files are taken into consideration providing stable and consistent compilation results. This choice implies that any other *GHC* installations are ignored if the `stack` command is found to be available. If you prefer to use `ghc` instead you can tell *Pythas* to disable `stack`:

```
from pythas import compiler
compiler.stack_usage = False
print("Stack is used: {}".format(compiler.stack_usage))
```

This will use whichever `ghc` is in the path of the directory the source file is located in. Using *Cabal* you can thus add the `--write-ghc-environment-files=always` option when using `cabal build`. This will create a local `.ghc.environment` directory with its own *GHC* installation and link it to the local `ghc` command. Then this local instance of *GHC* and its libraries will also be utilized by *Pythas*.

3.1.2 Install Pythas

Once either `stack` and/or *GHC* are installed, you can install *Pythas*.

From pip

Pythas is available for download from the Python Package Index `pypi` via `pip`:

```
$ pip install pythas
$ python -c "import pythas" # Will compile the Haskell source
```

From source

If you want to be at the newest stage of development, you can install this package from source using the following commands:

```
$ git clone --recurse-submodules -j8 https://github.com/pinselimo/Pythas.git
$ cd Pythas && pip install .
$ cd ..
$ python -c "import pythas" # Will compile the Haskell source
```

3.2 Usage

With *Pythas* installed you have multiple options of compiling and importing Haskell source code into Python. In all cases `import pythas` is the precondition to any consequent use of Haskell.

Two majorly distinct use cases are supported: Importing static Haskell modules and inline Haskell source code from within Python. Both are outlined below.

Note: The first time you use `import pythas` *Pythas*' parts implemented in Haskell are compiled, which impacts import time. Therefore, it is suggested to run `python -c "import pythas"` directly after installation to prompt compilation. Following usages will not require this additional time. Only updates of *Pythas* which alter parts of its Haskell source will trigger recompilation.

3.2.1 Static Haskell Modules

For static Haskell source files *Pythas* aims to provide a pythonic import behavior. Consider you execute `python` in a directory which contains a Haskell module `Example.hs` in a `examples` subdirectory:

```
$ ls ./examples
Example.hs
$ cat ./examples/Example.hs
module Example where

increment :: Int -> Int
increment = (1+)
$ python
>>>
```

Then you can import this module simply by typing:

```
>>> import pythas
>>> import examples.example as example
```

The `example` module and the functions and constants it contains can now be accessed from python just like with any usual python package. Note how we use a capitalized module name in Haskell and a lower case one in Python. This way, module naming schemes stay consistent with regard to both languages. Another tweak is, that the `examples` directory does not need an `__init__.py` file to be considered in the module lookup. Instead, *Pythas* will trust your knowledge about the file path and search accordingly.

After the import, the module presents itself to the user just like a Python module would. Given the following code in `Example.hs`:

```
module Example where

increment :: Int -> Int
increment = (1+)
```

then this means you can call it from Python just as you'd expect:

```
>>> example.increment(1)
2
```

3.2.2 Inline Haskell Modules

Inspired by *pyCUDA* the `SourceModule` - Object was added as another option for compiling Haskell source directly from a Python context.

```
>>> from pythas import SourceModule
>>> m = SourceModule('''
        increment :: Int -> Int
        increment = (1+)
        ''')
>>> m.increment(1)
2
```

By default a new `SouceModule` will spawn its own `Compiler` instance. This is done to avoid irreproducible outcomes. However, all options configurable on a `Compiler` can also be set as key word arguments of the `SourceModule`. Furthermore, it also accepts a `compiler` key word argument where an existing `Compiler` instance can be passed. Options set on other key word arguments will override those of the `Compiler` instance passed to the `SourceModule`, but not alter the instance itself. Usage example:

```
>>> from pythas import SourceModule, compiler
>>> compiler.stack_usage = True
>>> m = SourceModule('''
        increment :: Int -> Int
        increment !i = 1 + i
        '''
        , compiler=compiler
        , flags=compiler.flags + ('-XBangPatterns',)
        )
>>> m.increment(1)
2
>>> compiler.stack_usage
True
```

(continues on next page)

(continued from previous page)

```
>>> compiler.flags
('-O2',)
```

The example shows how a `SourceModule` is compiled with individual compile time flags set using an existing instance of `Compiler`. However, the flags set on the `Compiler` instance are not altered permanently. (Note: By far not all language extensions can be used with Pythas, consider them experimental within this framework)

3.2.3 Limitations

In both cases, static and inline Haskell source, some limitations exist on which Haskell functions and constants can and will be imported. Most notably, type declarations are paramount for the imports. *Pythas* does not do its own type inference. All basic Haskell types are supported, including strings and nested lists and tuples.

Unsupported functions or constants will not be available from the Python context. However, they will not trigger any errors. Thus, they can be used within the Haskell context without risk. Checking what populates the namespace of a module imported through *Pythas* is as easy as for any Python module:

```
>>> dir(example)
[ ... , 'increment']
```

3.2.4 Call signatures

A note on peculiarities of call signatures of constants imported via *Pythas*. Consider two type annotations in Haskell:

```
a :: Int
b :: IO Int
```

Interfacing from Python through *Pythas* these constants/variables (let's just not go down that rabbit hole right now) will be available like:

```
>>> m.a
63
>>> m.b
<pythas.utils.PythasFunc object at 0x....>
>>> m.b()
63
```

Note how the second name `b` needs to be called in order to expose its value. This is actually somewhat convenient, as it exposes part of Haskells strict notion on purity in Python. However, it gets fuzzy when we try to use nested data types (i.e. anything that needs a pointer - Lists, Tuples & Strings). *Pythas* will need to wrap these using FFI memory operations. Thus, even pure code is lifted into the IO monad for data transfer. So, if we take `a` and `b` instead to be:

```
a :: [Int]
b :: IO [Int]
```

We will end up with the following on Python's side:

```
>>> m.a
<pythas.utils.PythasFunc object at 0x....>
>>> m.a()
[1, 2, 3]
>>> m.b()
[1, 2, 3]
```

The call signature of `b` doesn't change, but `a` requires unwrapping now and it shows. In effect, you lose the visible difference the IO monad would cause on Python's side in the first example.

Note that the purity of your code itself does not suffer under this restriction! It just makes the call syntax a little weird.

3.2.5 Custom types

Support for pointers to custom types defined with `newtype` or `data` within Haskell is currently **experimental**. To make the function or constant names accessible from a Python context, you will need to manually add `foreign export ccall` exports to your module. Within Python the values are then treated as NULL-pointers. Thus, you can hand them from one Haskell function to another. The `example/Example.hs` file contained in the repository of *Pythas* contains a trivial showcase for this feature:

```
>>> import pythas; import example.example as e
>>> e.fromCustom(e.toCustom(63))
63
```

Due to the immense simplicity of the "Custom" type just wrapping an `Int` this works. Note that otherwise it will be more effort to make `Custom` an instance of `Foreign.Storable (Storable)` and provide a pointer through the FFI. Future releases of *Pythas* may feature a more supportive implementation of custom types.

3.3 Advanced Topics

This chapter covers details for the interested reader. It is a great first read if you want to delve into *Pythas*' source and contribute. As is reading through the tests in the `test` directory. Be sure to first read *Installation* and *Usage* before taking in this deep dive.

3.3.1 How does this magic work?

As has been layed out, the Haskell source code is still compiled using the well established *glorious Glasgow Haskell Compiler* (GHC). So how is the compiled source accessed from Python?

Both Python and Haskell provide Foreign Function Interfaces (FFI) to facilitate communication across language boundaries. However, in both languages this is a tedious process. *Pythas* aims to automate it on both sides. The ease of use paradigm that *Pythas* development is subject to implied a minimal set of requirements; i.e. the users themselves should not be exposed to *Template Haskell*, "Cabal hell" or any foreign function interfacing.

The process of importing a Haskell module using *Pythas* comprises 4 major steps:

Parsing

Yes, *Pythas* ships its own parser. This is the main reason you'll need to wait a little longer for the first import. As parsing is famously convenient in Haskell, most of it is implemented using *Parsec*. This parser can be found in the Pythas-FFI repository.

A second minimal parser is contained in the `pythas.parser` package. This mini-parser will only parse `foreign export ccall` statements and derive the type conversion necessary on Python's side.

Type conversion

Haskell and Python types are not directly compatible. In both languages lists are a heavily used concept. However, these represent fundamentally different types in either idiom. Python's dynamic typing doesn't interface well with Haskell's strict and (as for the FFI) static type system.

Moreover, for some types there are no guarantees which exact representation on memory will be used. This can depend on the backend. Both language's FFIs provide rich sets of C data types which solve this problem. *Pythas* will wrap all compatible Haskell functions to accept and provide interfacing compatible types. This type conversion type checks with GHC at compile time! On the Python side it will then pack the data in the pythonic equivalent to your Haskell data type and provide it to you.

Code generation

To provide *GHC* with some static source code to compile, a temporary Haskell module is generated. All files additionally created will be removed directly after the compiled module is imported in Python. Haskell FFI exports contained in the original module will also be imported and are not affected by the generated code.

Additionally to the wrapped functions, *Pythas* will also add specific functions for freeing any memory allocated by the Haskell runtime.

Compilation and import

The module is ultimately compiled into a shared library stored in a temporary file. This is the actual binary imported into Python. Within Python it is again wrapped to provide a pythonic interface just as you would expect it from any other Python import. *Pythas* will also add automated calls to the functions freeing memory described above.

Voilá, you can use Haskell source from Python!

3.3.2 Interfacing with GHC

Regardless of the build tools utilized, a minimal interface to *GHC* is provided. The compiler is wrapped as `Compiler` object. Internally another abstraction step is taken with the `GHC` class which handles specifics of the actual compiler.

Compile time flags

Flags for compilation can be set using the `add_flag` method. Consequently, the `flags` attribute of the `Compiler` instance contains a tuple with the set flags. Note that flags used for the general functionality of *Pythas* are not exposed here. Thus, any flag contained within the `flags` attribute can be removed using `remove_flag` method without risk.

The default value for the optimisation flag is already set to `-O2`. However, if for some reason you want to change this value or remove it, you can do so by using `compiler.remove_flag('-O2')`.

3.3.3 Notes on faster execution times

Neither quick compilation nor execution are main objectives of *Pythas* development at this stage. In contrast, the aim is to emphasize Haskell's benefits and provide easier access to them for Python users. There are, however, some tricks that can speed up your usage of Haskell code through *Pythas* significantly:

Whenever the interface has to hand over a list, a new `struct` containing a C array and an integer with its length is created. This happens both in the direcetion Haskell - Python as well as in Python - Haskell direction. Even in cases where a list is handed in both directions, the pointer/array will not be re-used! Thus, to save execution time, consider moving `map`, `foldr` or similar calls into the Haskell source.

Similarly, pointers to the structs created for the transfer of tuples are not reused.

3.4 Contributing

The source code of *Pythas* is split among multiple repositories:

- The main repository contains the Python source handling all the interaction with the Python runtime system.
- Pythas-FFI contains the back end responsible for parsing Haskell modules and transpiling them into FFI exports.
- Pythas-Types defines the custom Haskell types required to exchange nested data types in between the two languages. Their Python equivalents are defined in `pythas.types`.
- C-structs is a Haskell package for variably typed, correctly aligned C structs.

Contributions are welcome in all of these repositories. Please be advised to checkout the respective *CONTRIBUTING.md* file first. The preferred contribution workflow is to first raise an issue on github, then issue a pull request only after the issue's discussion was successful.

3.5 License

Most code of the *Pythas* package is licensed under the LGPLv3 license. This license is valid for all source code if not explicitly stated otherwise. Some parts are licensed under the MIT license, notably the *C-structs* Haskell package. Please refer to the respective *COPYING* and *COPYING.LESSER* or *LICENSE* files for details.

3.6 User reference

User reference for the **Pythas** package.

The `Compiler` instance (imported as `compiler`) and `SourceModule` constructor are the major constituents of the API. Everything else, all the necessary setup, is executed automatically at import time.

3.6.1 Compiler

class pythas.compiler.**Compiler**(*flags=('-O2')*)
Interface for the compiler used to create shared libraries.

Parameters **flags** (*Tuple[str]*) – Compile time flags to append. Default value is using the "-O2" flag.

GHC_VERSION
Version number string of the used GHC instance.

Type str

flags
Flags for *compiler*.

Type tuple(str)

stack_usage
Enable the usage of stack for compilation. Will default to False if stack is not available.

Type bool

add_flag(*flag*)
Adds a flag to *flags*.

Parameters **flag** (*str*) – A valid compile time flag.

compile(*filename*)
Creates an FFI file, compiles and links it against the Python runtime.

Parameters **filename** (*str*) – Pathlike object to a Haskell source file.

Returns **ffi_libs** – List of tuples of linked libraries and their respective parsed infos.

Return type [(ctypes.CDLL, *pythas.parser.data.ParseInfo*)]

copy()
Creates a new instance of `Compiler` from an existing one. Flags and `stack_usage` will be consistent.

Returns **new** – The new Compiler instance.

Return type *Compiler*

remove_flag(*flag*)
Removes a flag from *flags*.

Parameters **flag** (*str*) – A flag contained within *flags*.

3.6.2 SourceModule

class pythas.compiler.**SourceModule**(*code*, *compiler=None*, *use_stack=True*, *flags=('-O2')*)
Module created from inline Haskell source code. Will instantiate its own instance of `Compiler` unless an alternative is provided. Other settings will not be permanently made in the compiler instance.

Parameters

- **code** (*str*) – The Haskell source code to construct the module from.
- **compiler** (*Compiler*) – Compiler instance to use settings from.
- **use_stack** (*bool*) – Use stack if available. Default value is True.

- **flags** (*Tuple[str]*) – Compile time flags to append. Default value is using the "-O2" flag.

3.7 Pythas package full reference

Full reference for the `Pythas` package. Targeting developers.

3.7.1 pythas module

Pythonically import Haskell modules into your Python context.

Pythas will automatically find a Haskell module and parse it. It will then create a module containing calls to the Haskell FFI wrapping everything compatible in the imported module. These calls are subsequently compiled into a shared library and parsed by the Python part of Pythas. Ultimately, Pythas imports this shared library as a module-like object into Python and wraps the function calls accordingly. In effect, for the user the calls to the Haskell runtime are almost indistinguishable to common Python calls.

Submodules

haskell : Containing all source written in Haskell and a wrapper around the GHC instance.

parser : A basic parser for exported Haskell type declarations.

3.7.2 pythas.compiler module

API for Haskell compilation and binding.

class pythas.compiler.**Compiler**(*flags=('-O2')*)

Interface for the compiler used to create shared libraries.

Parameters **flags** (*Tuple[str]*) – Compile time flags to append. Default value is using the "-O2" flag.

GHC_VERSION

Version number string of the used GHC instance.

Type str

flags

Flags for *compiler*.

Type tuple(str)

stack_usage

Enable the usage of stack for compilation. Will default to False if stack is not available.

Type bool

_compile(*name*)

Compiles an FFI file, links its library and parses for infos.

Parameters **name** (*str*) – Pathlike object to a Haskell file containing FFI exports.

Returns **(lib, parse_infos)**

Return type The linked library and its parsed infos

add_flag (*flag*)
Adds a flag to *flags*.

Parameters **flag** (*str*) – A valid compile time flag.

compile (*filename*)
Creates an FFI file, compiles and links it against the Python runtime.

Parameters **filename** (*str*) – Pathlike object to a Haskell source file.

Returns **ffi_libs** – List of tuples of linked libraries and their respective parsed infos.

Return type [(ctypes.CDLL, *pythas.parser.data.ParseInfo*)]

copy ()
Creates a new instance of `Compiler` from an existing one. Flags and `stack_usage` will be consistent.

Returns **new** – The new Compiler instance.

Return type *Compiler*

remove_flag (*flag*)
Removes a flag from *flags*.

Parameters **flag** (*str*) – A flag contained within *flags*.

class pythas.compiler.**SourceModule** (*code*, *compiler=None*, *use_stack=True*, *flags=('-O2')*)
Module created from inline Haskell source code. Will instantiate its own instance of `Compiler` unless an alternative is provided. Other settings will not be permanently made in the compiler instance.

Parameters

- **code** (*str*) – The Haskell source code to construct the module from.
- **compiler** (`Compiler`) – Compiler instance to use settings from.
- **use_stack** (*bool*) – Use stack if available. Default value is True.
- **flags** (`Tuple[str]`) – Compile time flags to append. Default value is using the "-O2" flag.

3.7.3 pythas.core module

Core module containing the main metaprogramming.

class pythas.core.**PythasLoader** (*compiler*, *filename*)
Creates the FFI, compiles and links Haskell modules.

Parameters

- **compiler** (`Compiler`) – The compiler used to create the linked library.
- **filename** (*str*) – Pathlike object locating the Haskell source file.

class pythas.core.**PythasMetaFinder** (*compiler*)
MetaPathFinder for Haskell source files.

Parameters **compiler** (`Compiler`) – The compiler used to create the linked library.

find_spec : Entry point for path finding

pythas.core.**install** (*compiler*)
Installer for the *PythasMetaFinder*.

Parameters **compiler** (`Compiler`) – The compiler used to create the linked library.

3.7.4 pythas.haskell module

The `pythas.haskell` module contains a wrapper around GHC and all the Haskell source Pythas is based on. After the installation of Pythas, the Haskell source files need to be compiled first. Subsequent imports of Pythas will **not** need any compilation and thus result in quicker import times.

Haskell source documentation

These parts of Python are split into three Haskell packages:

- C-structs

This package may be useful in many contexts and is therefore available on Hackage. It provides types representing structs as known from the C programming language in Haskell.

- Pythas-Types

The types in Pythas-Types are based on the struct types from C-structs. They provide a foundation for the interfacing over the FFIs of both Haskell and Python.

- Pythas-FFI

Parsing, wrapping and code generation are contained in here. First an AST of the type declarations is parsed. This AST is then extended to utilize Pythas-Types compatible input and output data. The extended AST is compiled into a Haskell source module and its path returned.

3.7.5 pythas.haskell.ghc module

Wrapping functionality for GHC and STACK.

exception pythas.haskell.ghc.**CompileError**

class pythas.haskell.ghc.**GHC**
Pythas interface class for GHC.

static check_version(*stack=False*)
Checks if the GHC version required is installed.

Parameters **stack** (*bool*) – If True check version of stack ghc.

:raises ImportError : Version-Number of GHC is too low:

static compile(*filepath*, *libpath*, *use_stack=False*, *add_flags=()*, *_redirect=False*)
Compiles a Haskell source file to a shared library.

Parameters

- **filepath** (*str*) – Pathlike object referencing the Haskell source file.
- **libpath** (*str*) – Pathlike object referencing the shared library file.
- **use_stack** (*bool*) – If True uses stack ghc as compile command if available. Defaults to availability.
- **add_flags** (Tuple[*str*]) – Additional flags handed to GHC. Default is empty.
- **_redirect** (*bool*) – If True internal binaries are redirect into Pythas' bin directory for clean pip uninstall. Default is False.

Returns **libpath** – Pathlike object referencing the shared library path.

Return type str

static compile_cmd(*use_stack*, *options*)
Generates the compile command to GHC.

Parameters

- **use_stack** (*bool*) – If True uses stack ghc as compile command.
- **options** (*tuple(str)*) – The flags handed to GHC.

Returns **command** – The entire command to initiate compilation.

Return type tuple(str)

static flags(*filename*, *libname*, *use_stack*, *_redirect=False*)
Creates the flags needed for successful compilation of Haskell FFI files using Pythas.

Parameters

- **filepath** (*str*) – Pathlike object referencing the Haskell source file.
- **libpath** (*str*) – Pathlike object referencing the shared library file.
- **use_stack** (*bool*) – If True uses stack ghc as compile command.
- **_redirect** (*bool = False*) – Internal binaries are redirect into Pythas' bin directory for clean pip uninstall.

Returns **flags** – Flags for compilation of *filepath* to shared library in *libpath*.

Return type tuple(str)

static get_version(*stack_ghc*)
Retrieves the GHC version number. Defaults to getting it from the command line, reverts to the *ghcversion.h* header.

Parameters **stack_ghc** (*bool*) – True if stack's GHC is used.

Returns **version** – Version number string.

Return type str

See also:

`get_ghc_version_from_cmdln`, `get_ghc_version_from_header`

:raises ImportError : Version-Number of GHC could not be found:

static get_version_from_cmdln(*stack_ghc*)
Retrieves the GHC version number from the command line.

Parameters **stack_ghc** (*bool*) – True if stack's GHC is used.

Returns **version** – Version number string.

Return type str

static get_version_from_header()
Retrieves the GHC version number from the *ghcversion.h* header.

Returns **version** – Version number string.

Return type str

:raises ImportError : Version-Number of GHC could not be found:

pythas.haskell.ghc.**has_stack**()
Looks for stack on the *$PATH*.

Returns **has_stack** – True if stack is in *$PATH*.

Return type bool

3.7.6 pythas.parser.data module

Data container classes for parsing.

class pythas.parser.data.**FuncInfo**(*name*, *argtypes*, *restype*, *constructors*, *reconstructor*, *htype*)
Container class for informations about a Haskell function.

name
The name of the function.

Type str

argtypes
List with types of the function arguments.

Type list(ctype)

restype
Return type of the function.

Type ctype

constructors
List of functions converting Python types to their respective *argtypes*.

Type list(callable)

reconstructor
Callable converting the *restype* back to a native Python type and releasing any memory allocated in the transferring process.

Type callable

htype
Type as given by the Haskell FFI export.

Type str

class pythas.parser.data.**ParseInfo**(*name*, *dir*, *exported_mod*, *exported_ffi*, *func_infos*)
Container class for informations about a Haskell source module.

name
Module name

Type str

dir
Pathlike object to the source file.

Type str

exported_mod
Exported functions according to the module header.

Type set(str)

exported_ffi
Functions with an FFI export.

Type set(str)

func_infos
Dictionary mapping function names to their respective FuncInfo instance.

Type dict(str, *FuncInfo*)

pythas.parser.data.**_FuncInfo**
alias of *pythas.parser.data.FuncInfo*

pythas.parser.data.**_ParseInfo**
alias of *pythas.parser.data.ParseInfo*

3.7.7 pythas.parser.parse_file module

Parse Haskell modules/files.

pythas.parser.parse_file.**_parse_haskell**(*hs_lines*, *parse_info*)
Parses lines of a Haskell source file.

Parameters

- **hs_lines** (*list (str)*) – Lines of a Haskell source file.
- **parse_info** (ParseInfo) – Container into which informations are to be stored.

Returns **parse_info** – Informations parsed from *hs_lines*.

Return type *ParseInfo*

pythas.parser.parse_file.**find_module_statement**(*hs_cont*, *name*)
Locates the *module* statement in a Haskell source file.

Parameters

- **hs_cont** (*str*) – Content of a Haskell source file.
- **name** (*str*) – Name of the Haskell module as given by file name.

Returns **posiiton** – Index of the *module* statement in the *hs_cont*.

Return type int

:raises SyntaxError : Haskell file module statement malformed:

pythas.parser.parse_file.**parse_haskell**(*hs_file*)
Parses a Haskell file for exported functions and ffi exports.

Parameters

- **hs_file** (*str*) – Pathlike object referring to the Haskell source file.
- **Reuturns** –
- **--------** –
- **parse_info** (ParseInfo) – Informations parsed from *hs_file*.

pythas.parser.parse_file.**parse_head**(*hs_lines*, *name*)
Finds all the names that are exported according to the module statement.

Parameters

- **hs_lines** (*list (str)*) – Lines of a Haskell source file
- **name** (*str*) – Name of the Haskell module as given by file name.

Returns **exports** – List of exported names or *None* if no exports statement is given.

Return type list(str)

pythas.parser.parse_file.**parse_line**(*line_nr*, *hs_line*, *parse_info*)
Parses a single line of Haskell source code.

Parameters

- **linr_nr** (*int*) – Line number of the current line.
- **hs_line** (*str*) – Line of Haskell code.
- **parse_info** (ParseInfo) – Container into which informations are to be stored.

3.7.8 pythas.parser.parse_type module

Parse Haskell type declarations.

pythas.parser.parse_type.**argtype**(*hs_type*)
Parser for the argument side types of a Haskell function.

Parameters **hs_type** (*str*) – Argument side Haskell type.

Returns **arg** – Tuple with the argument type and a callable that converts a conventional Python instance to an instance of the required type.

Return type (type, callable)

pythas.parser.parse_type.**hs2py**(*hs_type*)
Maps Haskell to Python types.

Parameters **hs_type** (*str*) – The Haskell type.

Returns **pytype** – The Python type.

Return type type

pythas.parser.parse_type.**parse_type**(*line_nr*, *name*, *hs_type*)
Parses the type of an FFI exported Haskell function or constant.

Parameters

- **line_nr** (*int*) – Line number of the parsed line.
- **name** (*str*) – Name of the function or constant.
- **hs_type** (*str*) – Type declaration of the Haskell entity.

Returns **func_info** – Parsed information about the function.

Return type *FuncInfo*

:raises TypeError : Functions as arguments are not supported:

pythas.parser.parse_type.**restype**(*hs_type*)
Parser for the result side type of a Haskell function.

Parameters **hs_type** (*str*) – Result side Haskell type.

Returns **res** – Tuple with the result type and a callable that converts the type to a conventional Python type.

Return type (type, callable)

pythas.parser.parse_type.**simple_hs_2_py**(*hs_type*)
Converts simple Haskell types to their Python equivalent.

Parameters **hs_type** (*str*) – The Haskell type.

Returns **pytype** – The Python type.

Return type type

> **Warning:** TypeWarning : Custom types are an experimental feature in Pythas.

3.7.9 pythas.parser.utils module

Utility functions for parsing Haskell.

exception pythas.parser.utils.**TypeWarning**
Warning reminding the user that the types used by them are currently only under experimental support.

See also:

`Warning`

pythas.parser.utils.**apply**(*fs*, *t*)
Weirdly similar to Haskell's ap for subscriptable types.

Parameters

- **fs** (*iterable*) – List of tuples which have a *callable* as second member.
- **t** (*iterable*) – Arguments for the *callable*.

Returns **applied** – Results of the application of the callables to the arguments in *t* with the same index.

Return type tuple

pythas.parser.utils.**lmap**(*f*, *xs*)
Like map but returns a list instead of a generator.

pythas.parser.utils.**match_parens**(*s*, *i*)
Given a string and the index of the opening parenthesis returns the index of the closing one.

Parameters

- **s** (*str*) – The string to match parentheses on.
- **i** (*int*) – The initial index to start from.

Returns **closing** – Index of the next matching closing paranthesis.

Return type int

pythas.parser.utils.**parse_generator**(*f_llist*, *f_carray*, *f_tuple*, *f_string*, *f_default*)
Parser generator for parsing Haskell type statements.

Parameters

- **f_llist** (*callable*) – Function to call in case of linked lists.
- **f_carray** (*callable*) – Function to call in case of arrays.
- **f_tuple** (*callable*) – Function to call in case of tuples.
- **f_string** (*callable*) – Function to call in case of string.
- **f_default** (*callable*) – Function to call in case of string.

Returns **parser** – Function taking a string object with a Haskell type statement and parsing it using the appropriate function.

Return type callable

pythas.parser.utils.**strip_io**(*tp*)
IO is somewhat disregarded in the FFI exports. IO CDouble looks the same as CDouble from Python's side. So we remove the monadic part from our type to process the rest.

Parameters **tp** (*str*) – Haskell type statement.

Returns **stripped** – Tuple of *'IO '* if there was an IO statement or the empty string if there was none and the rest of the type statement.

Return type (str, str)

pythas.parser.utils.**tuple_types**(*hs_type*)
Extracts the types declarations inside a Haskell tuple type statement.

Parameters **hs_type** (*str*) – Haskell type statement for a tuple.

Returns **types** – Haskell type statements inside the tuple.

Return type list(str)

3.7.10 pythas.types module

Custom types defined for Haskell FFI interfacing of complex, nested data.

class pythas.types.**Array**
Marker class for Pythas' array types

class pythas.types.**LinkedList**
Marker class for Pythas' linked lists

class pythas.types.**Tuple**
Marker class for Pythas' tuples

pythas.types.**from_c_array**(*cp_array*)
Reconstructor from Pythas c_arrays.

Parameters **cp_array** (*cl.POINTER(Array)*) – Pointer to an instance of a subclass of Array.

Returns **seq** – A list with the contents of *cp_array*'s array.

Return type list

pythas.types.**from_linked_list**(*ll*)
Reconstructor from Pythas linked lists.

Parameters **ll** (LinkedList) – Pointer to the first element of a LinkedList instance.

Returns **seq** – A list with the contents of *ll*.

Return type list

pythas.types.**from_tuple**(*cpt*)
Reconstructor from Pythas c_tuples.

Parameters **cpt** (*cl.POINTER(Tuple)*) – Pointer to an instance of a subclass of Tuple.

Returns **tup** – A tuple with the contents of *cpt*.

Return type tuple

pythas.types.**get_constructor**(*ctype*)
Finds the constructor for a standard or custom ctypes type. The custom types are checked against the marker classes:

- Array

- Tuple
- LinkedList

Parameters **ctype** (*ctypes type*) – (Sub-)Class of ctypes._SimpleCData or ctypes.Structure.

Returns **constructor** – Function creating an instance of *ctype*.

Return type callable

pythas.types.**new_c_array**(*ctype*)
Creates a Pythas array class of *ctype*.

Parameters **ctype** (*ctypes type*) – (Sub-)Class of *ctypes._SimpleCData* or *ctypes.Structure*.

Returns **c_array** – Subclass of *Array* and *ctypes.Structure*.

Return type ctypes type

pythas.types.**new_linked_list**(*ctype*)
Creates a Pythas linked list class of *ctype*.

Parameters **ctype** (*ctypes type*) – (Sub-)Class of *ctypes._SimpleCData* or *ctypes.Structure*.

Returns **c_linked_list** – Subclass of *LinkedList* and *ctypes.Structure*.

Return type ctypes type

pythas.types.**new_tuple**(*subtypes*)
Creates a constructor for a Pythas tuple from Python tuples of *subtypes*.

Parameters **subtypes** (*Sequence of ctypes types*) – (Sub-)Class of *ctypes._SimpleCData* or *ctypes.Structure*.

Returns **c_tuple** – Subclass of *Tuple* and *ctypes.Structure*.

Return type ctypes type

pythas.types.**to_c_array**(*cls*, *seq*)
Constructor function for Pythas array.

Any instance of Array must always be associated with its *cls*.

Parameters

- **cls** (*Array subclass*) – A array class created with *new_c_array*.
- **seq** (*Sequence*) – The sequence which needs to be converted to a c_array.

Returns **array** – An instance of *cls* constructed from *seq*.

Return type *Array*

See also:

new_c_array

pythas.types.**to_linked_list**(*cls*, *seq*)
Constructor function for Pythas linked lists.

Any instance of LinkedList must always be associated with its *cls*.

Parameters

- **cls** (*LinkedList subclass*) – A linked list class created with *new_linked_list*.
- **seq** (*Sequence*) – The sequence which needs to be converted to a linked list.

Returns **linked_list** – An instance of *cls* constructed from *seq*.

Return type *LinkedList*

See also:

new_linked_list

pythas.types.**to_tuple**(*cls*, *tup*)
Constructor function for Pythas tuples.

Any instance of LinkedList must always be associated with its *cls*.

Parameters

- **cls** (*Tuple subclass*) – A tuple class created with *new_tuple*.
- **tup** (*tuple*) – The tuple which needs to be converted to a Pythas Tuple.

Returns **tuple** – An instance of *cls* constructed from *tup*.

Return type *Tuple*

See also:

new_tuple

3.7.11 pythas.utils module

Utility module for the Pythas API.

class pythas.utils.**PythasFunc**(*name*, *func_info*, *funcPtr*, *destructorPtr=None*)
Wrapper class for functions imported from a compiled Haskell module.

Parameters

- **name** (*str*) – The name of the function.
- **func_info** (parser.data.FuncInfo) – Parsed information about the function.
- **funcPtr** (*ctypes._FuncPtr*) – The pointer to the function.
- **destructorPtr** (*ctypes._FuncPtr*) – Pointer to the function releasing any memory allocated by the function in *funcPtr*. None if no destruction is required.

pythas.utils.**check_has_ghc**()
Looks for a valid GHC or STACK installation is available and in the PATH variable.

:raises ImportError : No means of Haskell compilation is available.:

pythas.utils.**custom_attr_getter**(*obj*, *name*)
Pythas modules' __getattribute__ instance. Retrieves foreign functions imported from Haskell modules by their name. Creates a *PythasFunc* instance and calls the finalizer function if given.

Parameters

- **obj** (*module*) – The module of the overridden __getattribute__ method.
- **name** (*str*) – The name to be retrieved from the module.

Returns **res**

Return type Either a constant or an instance of PythasFunc

:raises AttributeError : No corresponding foreign import could be found for *name*.:

pythas.utils.**ffi_libs_exports**(*ffi_libs*)
Collects the exported function names of a sequence of FFI library tuples. (See below)

Parameters **ffi_libs** (*Sequence of tuples of the DLL import and its parser.data.ParseInfo*) – All imported modules and their parsed information.

Returns **exports** – Set with all the exported names of the imported modules.

Return type Set[str]

pythas.utils.**find_source**(*name*, *path*, *extension='.hs'*)
Discovery function for Haskell modules.

Parameters

- **name** (*str*) – The name of the module.
- **path** (*str or path-like object*) – The path of the source directory.
- **extension** (*str*) – The file extension to be used. Default value is '.hs'.

Returns **source** – List containing the source file path for module *name*. Empty list if it couldn't be discovered.

Return type List[str]

pythas.utils.**flatten**(*seq*)
Creates a list of all basal elements of a nested sequence.

Parameters **seq** (*Sequence*) – An arbitrarily deep nested sequence.

Returns **flatseq** – List of all basal elements of *seq*.

Return type List

pythas.utils.**is_constant**(*func_infos*)
Checks if an imported function is actually a pure constant. Constants that have to go be wrapped in the IO monad for head storage (like lists) are considered impure.

Parameters **func_infos** (parser.data.FuncInfo) – Parsed information about the function.

Returns **is_constant** – True if the function is actually a constant.

Return type bool

pythas.utils.**remove_created_files**(*filename*)
Removes all files created by *Pythas* during the compilation process.

Parameters **filename** (*str or path-like object*) – Path to the Haskell module source file.

pythas.utils.**shared_library_suffix**()
Defines the file suffix for shared library files depending on the host operating system.

Returns **suffix** – The correct suffix containing a dot like: '.so' .

Return type str

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

C

D

E

F

G

H

I

L

M

N

P

R

S

T