

Language versioning

This document is extremely a draft. It needs some editing and discussion before it can be made into a useful thing. It's been simply copy pasted out of the pad in its current form.

See also

- FIXME: piegames langver ideas

musings

puck: honestly, having language version as part of a scopedImport-style primop would be funny
horrors: we're shitposting about setting language version from the source accessor

- horrors: `use features...;` file head clause
 - jade: this can be combined into feature sets like editions or such. we might become ghc haskell but whatever.
- horrors: [some kind of file head clause and/or propagation is the] only real way out of this mess that doesn't require a package manager in the package manager
jade: yeah. doing it from flakes seems initially sane until you realise you can import below the flake in the same git repo and then blegh
horrors: an ambient minimum-language-version binding in builtins that can be scopedImport'ed for flake support on top of this

horrific writeup

basic mechanism

add a new syntactic element that is only valid at the head of a file and used only to declare language requirements. nix versions that cannot satisfy all requirements must reject this element to situations in which two nix versions parse the same file differently, or even evaluating the same file to different derivation hashes. any kind of comment as used by eg GHC is not viable for nix for this reason.

proposed syntax for the first implementation: `use $($feature: ident)+;`

anything ahead of this directive could be either unversioned nix code or versioned nix code (see below for details), but since the directive is only valid at the *head* of a file or expression this "code" can only be comments. this kind of locks us into supporting the current comment syntax forever,

but the comment syntax is rather fine so this won't be a problem.

each feature may declare a syntactical requirement for the file, a semantic requirement, or possible both (cf rust editions, or perl `use v<something>`).

features may be global, namespaced to their implementations, or live in a reserved `experimental` namespace an implementation can add to and remove from as it wishes with ***absolutely no guarantee of future evaluatility***.

syntactic features

syntax is entirely local to the file itself and has few to no intercompatibility constraints with other code. a very useful syntax requirement would something like `no-url-literals`, which might strip the syntactic ability to parse url-like sequences of characters into strings and, rather than nix currently does the experimental feature of the same name simply throwing a parse error, parse them as eg a lambda with a sequence of divisions in its body.

(realistically `no-url-literals` would not appear in practice, instead it should be implied by `use` itself since url literals are such an obvious misfeature)

semantic features

semantic features produce evaluation changes that could be achieved any other way. examples of this are:

- the recent change that evalutes `x` in `inherit (x) names...` at most once overall rather than once per inherited name accessed
- potential extensions to the string context mechanism
- new types of values

semantic changes may escape the expression that requires them and usually some of amount of cross-compatibility with other semantic versions must be given. using the same examples as above, considerations can include:

- observable side-effects changing (if `x` includes a call to `trace`)
- `getContext` returning sets an outside use may not expect
- value types being unknown to outside users and causing failures

this is in fact a full classification of cross-compatibility issues: side-effects changing, evaluation outputs changing, and evaluation inputs changing. side-effects need not be considered very much since nixlang is supposed to be *pure* and all side-effects that are not part of the store interface must already be considered incidental. evaluation outputs changing can be handled by optional lint or runtime warnings when a versioned evaluation structure passes a semantic version boundary without being annotated as an intentional behavioral leak. evaluation inputs changing is a non-issue because nix plugins and the `ExternalValue` infrastructure already make it impossible to rely on the type system being fully specified at the time an expression is written

inter-file inter-actions

by default language features **must not** be propagated across an unadorned `import` boundary to retain compatibility with existing nix code (eg nixpkgs, which will not be able to switch for quite some time). in some circumstances it is however *required* to propagate language features across imports to provide a consistent and meaningful interface, eg in the case of a hypothetical `requiredLanguageFeatures` attribute for a flake. to allow for both of these requirements to peacefully coexist we add a new primop:

```
scopedImportUsing
:: { features ? <current language features> :: AttrSetOf bool
  ## ^ language features as would be specified by `use ...;`.
  ## selecting a default-off feature is achieved by setting its key to `true`,
  ## deselecting a default-on feature is achieved by setting its key to `false`.
  ## nesting is not needed because features are identifiers. future changes to
  ## the use interface may extend the type of this set.
, newGlobals ? env: env :: AttrSetOf Any -> AttrSetOf Any
  ## ^ function to produce the new global environment. it receives the default globals
  ## set for the target expression language features (as calculated from `features` and
  ## the target `use` clause) and produces a new set.
  ## `scopedImport` behavior is recovered by setting this to `const newEnv`.
}
-> PathLike
## ^ imported path as in `scopedImport`
-> Any
## ^ import result. may be cached, most immediately using the intransparent internal
## object id of the provided features and the globals set. this mimics the behavior
## or `import` in cppnix
```

if the imported expression selects a different set of language features the features specified by `scopedImportUsing` are ignored.

`scopedImportUsing` is available in the `builtins` set and crucially, *can be replaced*. this allows a hypothetical flake implementation to replace both `scopedImportUsing` and `import` with its own versions that provide propagation behaviors that might be expected from such a library:

- importing within the same flake simply propagates the language features as-is
- importing across flake boundaries first resolves the language versions used by the imported-from flake, then applies and propagates using these features. if the imported-from flake then imports code from elsewhere this cycle repeats and can eventually restore the language features set to its original value when importing code next to the code importing the importing code

- importing out of a flake boundary (as might be possible in an impure mode) resets the propagated language feature set as if it had never been set in the first place

additionally the current language features might be made available through a builtin value `languageFeatures` by such a replacement of `scopedImportUsing`.

builtins versioning, global versions

a language feature may add or remove elements of `builtins` or the global environment. as mentioned earlier this does not pose a large hazard since evaluation is sufficiently unspecified that this must *already* be expected to happen.

interactions with eg nixpkgs lib

nixpkgs lib (and other libraries) will have to cater to the smallest common denominator when exposing library functions/constants as they do now. if we change a function to have a different prototype and a library reexports it from builtins to its own namespace the language features used by the code importing the library do not matter. to make this problem less unbearable we may want to introduce a concept of library objects and a "use library" directive like eg python `from ... import ...` that can pass language features down to the library being imported in some way.

as a first approximation it would be sufficient to encourage libraries to version their namespaces in such a way that accessing a namespace that relies on language features not present in the current evaluator will fail to evaluate (eg by providing the library itself as a plain set and each version as an attribute that (lazily) imports the specific version of the library needed to fulfill the requested version).

bad ideas for features to remove/change in the first langver

- remove url literals
- remove `with`
- remove `rec` (including `__overrides`)
- remove `let { body = ...; ... }`
- remove `or` contextual keyword, either rework or make a real keyword
- extend `listToAttrs` prototype to also accept 2-tuples instead of name-value-attrpairs
- remove `__sub` and similar overloading

Feature detection

jade: I think we might want to be able to feature detect certain features, e.g. new builtin args, which can be done without, but we would like to know if they are there.

`builtins.nixVersion` has been defanged, which means that an alternate cross impl compatible mechanism needs to be created.

Minimally thought-through proposal

`builtins.features` is an attribute set, where individual attribute names are exposed with the value true if they are implemented by a given implementation.

Attribute names are of the format:

"domainname.feature", for example, "systems.lix.somefeature".

Revision #2

Created 7 April 2024 23:09:49 by jade

Updated 7 April 2024 23:23:02 by jade