1 Introduction

In this document, I propose a new design for wasp's DSL. The description of the new DSL will be broke into three parts: a formal definition, my motivation for key decisions, and ideas on its potential implementation.

From here on, the new version of wasp's DSL will be referred to as **wasp** and the current version as **old wasp**.

2 Formal Definition

Wasp is composed of four parts: the syntax, a type system, evaluation method, and the standard library.

2.1 Syntax

```
<wasp>    ::= <include>* <stmt>*
<stmt>    ::= <decl> | <call> | <defn>

<decl>    ::= <ident> <ident> <expr> <eol>
<call>    ::= <ident> <expr> <eol>
<defn>    ::= <ident> '=' <expr>
<include> ::= 'include' <string> <eol>
            | 'include' <string> 'as' <ident> <eol>

<expr>    ::= <dict> | <list> | <dot>
            | <string> | <import> | <quoter>
            | <ident> | <num>
<dict>    ::= '{' <dictval> (',' <dictval>)* '}'
            | '{' '}'
<dictval> ::= <ident> : <literal>
<list>    ::= '[' <expr> (',' <expr>)* ']' | '[' ']'

<import>  ::= 'import' <name> 'from' <string>
<name>    ::= '{' <ident> '}' | <ident>
```

```
<dot>     ::= <ident> '.' <ident>
<quoter>  ::= '{=' <ident> <anything> <ident> '=}'
```

The nonterminals for strings, numbers, and identifers are not shown. The only significant whitespace in wasp is the `<eol>` at the end of a statement.

2.2 Type System

Wasp is a statically typed language. The following inference rules describe the type system. Some special conventions are followed:

- τ_i and σ_i are types
- e_i is an expression
- `ki`, `ji`, `xi` are identifiers
- `monospace` words are tokens from a wasp file
- The syntax $\Gamma = \Gamma, \texttt{x} : \tau$ updates Γ such that $\texttt{x} : \tau \in \Gamma$

2.2.1 Type Rules

$$\textsc{Var}\ \frac{\texttt{x} : \tau \in \Gamma}{\Gamma \vdash \texttt{x} : \tau} \qquad \textsc{Incl}\ \frac{\Gamma_{other} \vdash \texttt{x1} : \tau_1 \quad \ldots \quad \Gamma_{other} \vdash \texttt{xn} : \tau_n}{\texttt{include "other"} \quad \Gamma = \Gamma, \texttt{x1} : \tau_1, \ldots, \texttt{xn} : \tau_n}$$

$$\textsc{InclQ}\ \frac{\Gamma_{other} \vdash \texttt{x1} : \tau_1 \quad \ldots \quad \Gamma_{other} \vdash \texttt{xn} : \tau_n}{\texttt{include "other" as } Q \quad \Gamma = \Gamma, Q : \{\texttt{x1} : \tau_1, \ldots, \texttt{xn} : \tau_n\}}$$

$$\textsc{Call}\ \frac{\Gamma \vdash e : \tau \quad \texttt{func funcName =} \ \tau}{\texttt{funcName } e}$$

$$\textsc{Decl}\ \frac{\Gamma \vdash e : \tau \quad \texttt{data typeName =} \ \tau}{\texttt{typeName x =} \ e \quad \Gamma = \Gamma, \texttt{x} : typeName} \qquad \textsc{Defn}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{x =} \ e \quad \Gamma = \Gamma, \texttt{x} : \tau}$$

ENUM

$$\frac{\texttt{enum enumName = } \ldots \texttt{|} E_i \texttt{|} \ldots}{\Gamma \vdash E_i : enumName}$$

DICT

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \texttt{\{ k1:}e_1\texttt{, \ldots ,kn:}e_n\texttt{\}} : \{k1 : \tau_1, \ldots, kn : \tau_n\}}$$

DICTNONE

$$\frac{\Gamma \vdash e : \{\texttt{k1} : \tau_1, \ldots, \texttt{kn} : \tau_n\}}{\Gamma \vdash e : \{\texttt{k1} : \tau_1, \ldots, \texttt{kn} : \tau_n, \texttt{j} : \sigma?\}}$$

DICTSOME

$$\frac{\Gamma \vdash e : \{\texttt{k1} : \tau_1, \ldots, \texttt{kn} : \tau_n, \texttt{j} : \sigma\}}{\Gamma \vdash e : \{\texttt{k1} : \tau_1, \ldots, \texttt{kn} : \tau_n, \texttt{j} : \sigma?\}}$$

DICTINST

$$\frac{\Gamma \vdash e : \{\texttt{k1} : \tau_1, \ldots, \texttt{kn} : \tau_n, \texttt{j} : \sigma\}}{\Gamma \vdash e : \{\texttt{k1} : \tau_1, \ldots, \texttt{kn} : \tau_n\}}$$

LIST

$$\frac{\Gamma \vdash e_1, \ldots, e_n : \tau}{\Gamma \vdash \texttt{[} e_1\texttt{, \ldots ,}e_n\texttt{]} : [\tau]}$$

DOT

$$\frac{\Gamma \vdash e : \{\texttt{k} : \tau\}}{\Gamma \vdash e\texttt{.k} : \tau}$$

STRING

$$\frac{}{\Gamma \vdash \texttt{"string"} : string}$$

NUMBER

$$\frac{}{\Gamma \vdash \texttt{3.14} : number}$$

IMPORT

$$\frac{}{\begin{array}{l}\Gamma \vdash \texttt{import ImportName from "file"} : import \\ \Gamma \vdash \texttt{import importName from "file"} : import\end{array}}$$

QUOTE

$$\frac{}{\Gamma \vdash \texttt{\{=quot \ldots quot=\}} : quot}$$

DICTNONE allows omitting optional entries from a dictionary, and DICTSOME allows including optional entries. DICTINST allows including extra entries in dictionaries, and is generally required for proofs involving DOT.

An ambiguity that arises is whether `x = [{a: 1}, {}]` is well-typed, and if x's type is $[\{\}]$ or $[\{a : number?\}]$.

2.2.2 Type Checking

There are two phases to the type checker:

1. Included files are type checked, then INCL and INCLQ rules are applied.
2. Binding declaration types: The conclusion of DECL is assumed for all `<decl>`s, essentially forward declaring each of them.
3. Type checking of statements:
   (a) CALL, DEFN, and the premise of DECL are checked against statements in their lexical order.
   (b) Quoted text (ex. `{=json [1] json=}`) is parsed.
   (c) Recovery should be attempted on a type error, to allow reporting of multiple errors per compilation.

2.3 Evaluation

Since wasp has no control flow, evaluation is simple. Definitions must be evaluated in order. The evaluation order of declarations and calls is left unspecified. When declarations and calls are evaluated, so are their arguments.

Only the values of declarations and calls are visible to the caller of the wasp interpreter.

2.3.1 Includes

At the beginning of a wasp file, a list of include statements is allowed. When a wasp file is included in another, evaluated before the includer. The type rules for inclusion hints at how their variables are bound in the includer:

- `include "other.wasp"` brings other.wasp's bindings into the current file, over-writing any existing bindings to the same name. The declarations and calls from other.wasp are made available to the caller of the wasp interpreter.
- `include "other.wasp" as Q` binds the variables and declarations from other.wasp to a variable `Q` in the current file.

Note that in either case, expressions in other.wasp can not refer to any bindings in the includer.

2.4 Standard Library

Described below is the standard library for wasp. The syntax of the definitions here are the same as used in the type rules.

```
enum authMethod = EmailAndPassword
enum paramType = StringParam | NumberParam

data app = { title : string }
data page = { component : import
            , authRequired : bool?
            , params : [{ name : string, paramType : paramType }]?
            }
data entity = psl
data query = { fn : import, entities : [entity] }
data action = { fn : import, entities : [entity] }
data route = { path : string
             , page : page
             }

func auth = { userEntity : entity
            , methods : [authMethod]
            , onAuthFailedRedirectTo : route
            }
func dependencies = json
```

3 Motivation

The four main things considered in the design of wasp are:

1. Similarity to old wasp
2. Safety
3. Extensibility
4. Simplicity

3.1 Similarity to old wasp

This was the driving force for this design. Old wasp is very clear to read and works well for its purpose. Wasp is mostly a formalized version of old wasp.

Not counting new features, the two DSLs are very similar. When translating from old wasp to wasp, only the following changes must be kept in mind:

- `route` is now a declaration instead of a function
- `page` has an optional `params` parameter to specify path parameters
- The `onAuthFailedRedirectTo` field in `auth` now requires a `route` instead of a string

3.2 Safety

The type system introduces new mechanisms for safely writing wasp configurations: `route` was changed to a declaration to allow for type safe URLs; `page` specifies path parameters so that, in the future, a typescript codebase could generate types for path parameters.

3.3 Extensibility

Defining wasp's configuration objects in types instead of as syntax will make it easier for maintainers to add new objects to wasp. In old wasp's implementation, a custom parser had to be written for each type. Now, a developer just needs to declare a new type in the wasp library. A suggestion for how this might look is described in section 4.2.

Additionally, having all wasp objects described through the same mechanism will make it more worth the effort to develop a detailed error system for more helpful syntax and type errors when compiling a wasp project.

Finally, wasp's flexibility makes possible a plugin system to add more features to wasp without bloating its core library.

3.4 Simplicity

While wasp is much more complex than old wasp, it is still fairly simple. Notably, control flow is absent. I wanted wasp to be as simple as possible to make experimenting with its implementation less painful. Hopefully, when a good implementation is found, it will be flexible enough to bring in more complex features as needed.

4 Implementation Suggestions

The following sections describes my ideas for parts of the implementation of wasp.

4.1 Parsing

Issue #109 has good notes about parsing, but I think the best option would be a parser generator (Happy is mentioned in the issue). Writing a parser by hand, even with a parser combinator library, seems wasteful when there is a formal grammar available. On the other hand, parser combinators have the advantage of being able to directly produce a structure most convenient for the next phases.

4.2 Type checking

The actual strategy of type checking should be relatively simple. There are no polymorphic types or type inference of recursive definitions, thus no constraint solving is necessary to verify the program. This should make the type of each expression and statement easy to check.

I instead want to focus on defining types in wasp. It seems very possible that non-primitive types could be defined and used in the following way (omitting language extensions and imports):

```haskell
data ParamType = StringParam | NumParam
               deriving (Generic)
newtype PageParam = PageParam { name :: String
                              , paramType :: ParamType
                              } deriving (Generic)
newtype Page = Page { componenent :: WaspLang.Primitive.Import
                    , authRequired :: Maybe Bool
                    , params :: Maybe [PageParam]
                    } deriving (Generic)

newtype Dependencies = Dependencies WaspLang.Primitive.Json
                       deriving (Generic)

interpretWithLib :: String -> Either WaspLang.Error WaspLang.Result
interpretWithLib = WaspLang.interpret $ do
  WaspLang.registerEnumType @ParamType
  WaspLang.registerDataType @Page
  WaspLang.registerFuncType @Dependencies

-- To use pages declared by a wasp file
printPageComponents :: WaspLang.Result -> IO ()
printPageComponents result = do
  let pageDecls = WaspLang.Decl.take @Page $ WaspLang.Result.decls result
  mapM_
    (\(name, page) -> putStrLn $ name ++ ": " ++ show (component page))
    pageDecls
```

`registerEnumType`, `registerDataType`, and `registerFuncType` would all be implemented using GHC generics. Additionally, some type classes would be used to only allow types in specific forms:

```haskell
-- `IsWaspEnum` ensures `a` is in the form `data A = A1 | ... | An`
registerEnumType :: (Monad m, Generic a, IsWaspEnum a) => WaspMonad ()
-- `IsWaspType` ensures `a` has one constructor that either:
--   - Has one parameter
--   - Uses record syntax
--  And only contains:
--   - Wasp primitives
--   - Maybe
--   - IsWaspEnums
--   - IsWaspTypes
registerDeclType :: (Generic a, IsWaspType a) => WaspMonad ()
registerFuncType :: (Generic a, IsWaspType a) => WaspMonad ()
```

An important function is `WaspLang.Decl.take`, which extracts a specific type

from the existentially quantified type `Decl`:

```haskell
{-# LANGUAGE GADTs #-}
{-# LANGUAGE TypeApplications #-}
import Data.Typeable (Typeable, cast)
import Data.Maybe (mapMaybe)

data Decl where
  Decl :: (Typeable a) => String -> a -> Decl

take :: (Typeable a) => [Decl] -> [(String, a)]
take = mapMaybe $ \case
  Decl x -> cast x
```

The type for `Call` (complementary to `registerFuncType`) would look very similar.

4.3 Evaluation

Evaluation of definitions is simple, just compute them in order. Includes also are pretty intuitive. The only problem is circular inclusions. I think these should not be allowed.Evaluating declarations and calls is a bit more nuanced, since they can refer to declarations declared lexically after themselves.

One possible solution is sort the declarations and calls topographically before evaluation. This poses a problem for mutually recursive declarations, since sorting a cyclical directed graph isn't well defined. I have not spent much time thinking about this problem.

Alternatively, time travel could be used to make the final results of evaluation available to every single declaration and call, which solves the circular reference problem. If unfamiliar with time travel in Haskell, read this article for an introduction. This may have performance consequences.

Another interesting part is what structure to store the evaluated result in. Following from 4.2, it makes sense to use the registered ADTs as the result types. `Data.Data` seems to be the module to use for this.

The `Data.Data` allows constructing values of arbitrary constructors. If type checking succeeds, we can then create the proper data types to return in `Result.decls`

and `Result.calls`. I have investigated the `Data.Data` module and I am confident this will work, but `unsafeCoerce` might be needed.

5 Closing Thoughts

This is not a finished design. It definitely needs more work before implementing it, especially because its implementation will take a lot of work and require changes across the existing code base.

Because this implementation is quite large, I think it would make the most sense to start by defining its API and the boundaries between the three systems of the DSL. Then, the implementation could be divided into three sub-projects.