REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Syntax vs Semantic types

In any program you have lots of data floating around your system, you may have a GUID like *<<"286e56f0-2d64-4b4d-9b7f-876952eea532">>* in your system. Now a GUID may well be an ID of something, but what what?

As it is the type system can't tell if that is a User id, a Store id, shopping cart id or something else. What would be really nice is if we could leverage the type system to ensure that we don't cross contaminate our data.

If you use an ID that should be a user id as a store id, or an integer that should be a height as a counter then you will have a very hard to track down bug.

Thankfully between pattern matching and dialyzer there is an easy solution, to this problem.

If instead of having a value like *<<"286e56f0-2d64-4b4d-9b7f-876952eea532">>* which could be anything, use a tuple with an atom like *{user_id, <<"286e56f0-2d64-4b4d-9b7f-876952eea532">>}* then, the data has moved from being a binary, to being a userid, which carries semantic value.

You can then use dialyzer or a pattern match (or both) to ensure that there is no cross contamination.

If you use a pattern match in the header of a function like in this example, then if a value that is not correctly formatted is passed the code will fail with a pattern match error.

Pattern Match

```
-module(user_pattern).

-type(user_id() :: {user_id, binary()}).

find_user(User = {user_id, _}) ->
    not_found.
```

This will ensure that if an error is introduced into the program it will not cause data to propagate across the system. This fits well with the Erlang "Fail Fast" mantra.

If you provide type specs in your code, dialyzer [?] will also flag any possible errors that it can find.

Using a Spec

```
-module(user_type).

-type(user_id() :: {user_id, binary()}).

-spec(find_user(user_id()) -> maybe(user)).
find_user(User) ->
    not_found.
```

Of course doing both would be even better to do both.

Both Type and Guard

```
-module(user_both).

-type(user_id() :: {user_id, binary()}).


-spec(find_user(user_id()) -> not_found).
find_user(User = {user_id, _}) ->
    not_found.
```

Of course this definition of a tuple is exactly the same as an Erlang record. So instead of defining *user_id()* as the tuple *{user_id, binary()}*. you could define a record like this.

And then use a pattern match or a guard to ensure the correct types, as in these two examples:

record

```
-record(user_id, {user_id :: binary()}).

-type(user_id() :: #user_id{}).

find_user(User)  when is_record(User, user_id)->
    not_found.
```

record

```
-record(user_id, {user_id :: binary()}).

-type(user_id() :: #user_id{}).

find_user(User = #user_id{}) ->
    not_found.
```

As to which you choose, it is really up to which makes more sense in the context of your program.

1.1 Putting this to use

In general when data comes into your system you want to convert it as quickly as possible to a semantic type.

In this example, which is working with WebMachine, the user id is extracted from the data passed into the web request,and then converted to a user id.

```
-spec resource_exists(rd(), state()) -> {boolean() | halt(), rd(), state()}.
resource_exists(ReqData, State) ->
    Uid     = wrq:path_info('user_id', ReqData),
    IsValid = subscription:is_valid({user_id,list_to_binary(Uid)}),
    {IsValid, ReqData,State}.
```

Of course, far better would be to wrap all of that into an independent function which can be tested and checked.

```
-spec(resource_exists(rd()) -> {ok, user_id()}).
get_user_id(ReqData) ->
    UserId     = wrq:path_info('user_id', ReqData),
    {ok,{user_id,list_to_binary(UserId)}}.

-spec resource_exists(rd(), state()) -> {boolean() | halt(), rd(), state()}.
resource_exists(ReqData, State) ->
    {ok, UserId}   = get_user_id(ReqData)
    IsValid        = subscription:is_valid(UserId),
    {IsValid, ReqData, State}.
```