FOR TYPE JUNKIES

This section is meant to be read by programmers who can not live without a static type system for one reason or another. It will include a little bit more advanced theory and everything may not be understood by everyone. I will briefly describe tools used to do static type analysis in Erlang, defining custom types and getting more safety that way. These tools will be described for anyone to understand much later in the book, given that it is not necessary to use any of them to write reliable Erlang programs. Because we'll show them later, I'll give very little details about installing, running them, etc. Again, this section is for those who really can't live without advanced type systems.

Through the years, there were some attempts to build type systems on top of Erlang. One such attempt happened back in 1997, conducted by Simon Marlow, one of the lead developers of the Glasgow Haskell Compiler, and Philip Wadler, who worked on Haskell's design and has contributed to the theory behind monads (Read the paper on said type system). Joe Armstrong later commented on the paper:
One day Phil phoned me up and announced that a) Erlang needed a type system, b) he had written a small prototype of a type system and c) he had a one year's sabbatical and was going to write a type system for Erlang and "were we interested?" Answer —"Yes."

Phil Wadler and Simon Marlow worked on a type system for over a year and the results were published in [20]. The results of the project were somewhat disappointing. To start with, only a subset of the language was type-checkable, the major omission being the lack of process types and of type checking inter-process messages.

Processes and messages both being one of the core features of Erlang, it may explain why the system was never added to the language. Other attempts at typing Erlang failed. The efforts of the HiPE project (attempts to make Erlang's performances much better) produced Dialyzer, a static analysis tool still in use today, with its very own type inference mechanism.

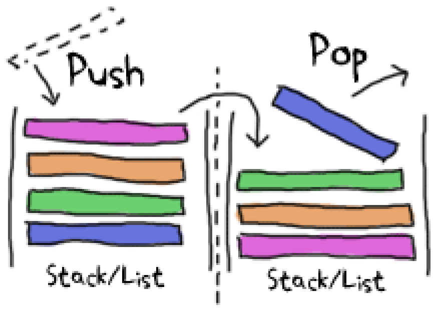
The type system that came out of it is based on success typings, a concept different from Hindley-Milner or soft-typing type systems. Success types are simple in concept: the type-inference will not try to find the exact type of every expression, but it will guarantee that the types it infers are right, and that the type errors it finds are really errors.

The best example would come from the implementation of the function `and`, which will usually take two Boolean values and return 'true' if they're both true, 'false' otherwise. In Haskell's type system, this would be written `and :: bool -> bool -> bool`. If the `and` function had to be implemented in Erlang, it could be done the following way:

```
and(false, _) -> false;
and(_, false) -> false;
and(true,true) -> true.
```

Under success typing, the inferred type of the function would be `and(_,_) -> bool()`, where _ means 'anything'. The reason for this is simple: when running an Erlang program and calling this function with the arguments `false`and `42`, the result would still be 'false'. The use of the `_` wildcard in pattern matching made it that in practice, any argument can be passed as long as one of them is 'false' for the function to work. ML types would have thrown a fit (and its users had a heart attack) if you had called the function this way. Not Erlang. It might make more sense to you if you decide to read the paper on the implementation of success types, which explains the rationale behind the behavior. I really encourage any type junkies out there to read it, it's an interesting and practical implementation definition.
The details about type definitions and function annotations are described in the Erlang Enhancement Proposal 8 (EEP 8). If you're interested in using success typings in Erlang, check out the TypEr application and Dialyzer, both part of the standard distribution. To use them, type in `$ typer --help` and `$ dialyzer --help` (`typer.exe --help`and `dialyzer.exe --help` for Windows, if they're accessible from the directory you are currently in).
TypEr will be used to generate type annotations for functions. Used on this small FIFO implementation, it spits the following type annotations:

```
%% File: fifo.erl
%% --------------
-spec new() -> {'fifo',[],[]}.
-spec push({'fifo',_,_},_) -
> {'fifo',nonempty_maybe_improper_list(),_}.
-spec pop({'fifo',_,maybe_improper_list()}) -> {_,{'fifo',_,_}}.
-spec empty({'fifo',_,_}) -> bool().
```



Which is pretty much right. Improper lists should be avoided because `lists:reverse/1` doesn't support them, but someone bypassing the module's interface would be able to get through it and submit one. In this case, the functions `push/2` and `pop/2` might still succeed for a few calls before they cause an exception. This either tells us to add guards or refine our type definitions manually. Suppose we add the signature `-spec push({fifo,list(),list()},_) -> {fifo,nonempty_list(),list()}.` and a function that passes an improper list to `push/2` to the module: when scanning it in Dialyzer (which checks and matches the types), the error message `"The call fifo:push({fifo,[1|2],[]},3) breaks the contract '<Type definition here>'` is output.

Dialyzer will complain only when code will break other code, and if it does, it'll usually be right (it will complain about more stuff too, like clauses that will never match or general discrepancies). Polymorphic data types are also possible to write and analyze with Dialyzer: the `hd()` function could be annotated with `-spec([A]) -> A.` and be analyzed correctly, although Erlang programmers seem to rarely use this type syntax.

Source : http://learnyousomeerlang.com/types-or-lack-thereof