29

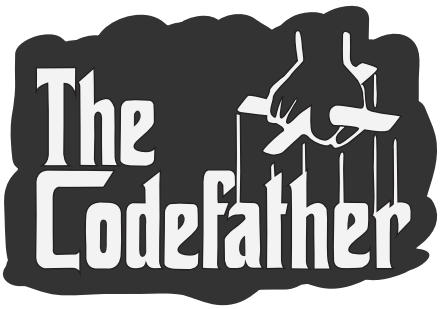
MNESIA AND THE ART OF REMEMBERING

You're the closest friend of a man with many friends. He has known some of those friends for a very long time, as have you. These friends come from all around the world, ranging from Sicily to New York. They pay their respects, and care about you and your friend, and you both care about them back.

In exceptional circumstances, they ask for favors because you're people of power and trust. They're your good friends, so you oblige. However, friendship has a cost. Each favor realized is duly noted, and at some point in the future, you may ask for a service in return.

You always keep your promises—you're a pillar of reliability. That's why they call your friend *boss*, and they call you *consigliere*. You're helping to lead one of the most respected Mafia families.

However, it becomes a pain to remember all your friendships, and as your areas of influence grow across the world, it is increasingly harder to keep track of what friends owe to you and what you owe to friends.



Because you're a helpful counselor, you decide to upgrade the traditional system from notes secretly kept in various places to something using Erlang.

At first, you figure using ETS and DETS tables will be perfect. However, when you're out on an overseas trip away from the boss, it becomes somewhat difficult to keep things synchronized.

You could write a complex layer on top of your ETS and DETS tables to keep everything in check. You could do that, but being human, you know that you might make mistakes and write buggy software. Such mistakes are to be avoided when friendship is so important, so you look online to find how to make sure your system works correctly.

This is when you start reading this chapter, which explains Mnesia, an Erlang distributed database built to solve such problems.

What's Mnesia?

Mnesia is a layer built on top of ETS and DETS to add a lot of functionality to those two databases, which were introduced in Chapter 25. It mostly contains features that many developers might end up creating on their own if they wanted to use ETS and DETS intensively. Mnesia allows you to write to both ETS and DETS automatically, to have both DETS's persistence and ETS's performance, and to be able to replicate the database to many different Erlang nodes automatically.

Another useful feature provided by Mnesia is *transactions*. Basically, transactions let you perform multiple operations on one or more tables as if the process doing them were the only one to have access to the tables—something ETS doesn't allow. This proves vital when you need to have concurrent operations that mix read and writes as part of a single unit. One example would be reading in the database to see if a username is taken, and then creating the user if that name is available. Without transactions, looking inside the table for the value and then registering it counts as two distinct operations that can be messing with each other. Given the right timing, more than one process might believe it has the right to create the unique user, which will lead to a lot of confusion. Transactions solve this problem by allowing many operations to act as a single unit.

Mnesia is pretty much the only full-featured database available that will natively store and return any Erlang term out of the box (at the time of this writing). The downside is that it will inherit all the limitations of DETS tables in some modes, such as not being able to store more than 2GB of data for a single table on disk (but this can be bypassed with a feature called *fragmentation*, described at *http://www.erlang.org/doc/apps/mnesia/Mnesia_chap5.html#id75194*).

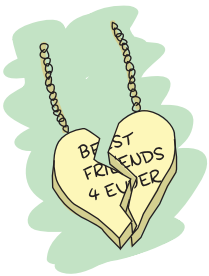
If we consider the CAP theorem (discussed in Chapter 26), Mnesia sits on the CP side, rather than the AP side. This means that it won't do eventual consistency and will react rather badly to netsplits in some cases, but it will give you strong consistency guarantees if you expect the network to be reliable (and you sometimes shouldn't).

Mnesia is not meant to replace your standard SQL database, nor is it meant to handle terabytes of data across a large number of data centers, as often claimed by the giants of the NoSQL world. Mnesia is made for smaller amounts of data, on a limited number of nodes. While it is possible to use it on a ton of nodes, most people find that their practical limits center around 10 or so. You will want to use Mnesia when you know it will run on a fixed number of nodes, have an idea of how much data it will require, and know that you will primarily need to access your data from Erlang in ways ETS and DETS would allow in usual circumstances.

Just how close to Erlang is it? Mnesia is centered on the idea of using a record to define a table's structure. Each table can store a bunch of similar records, and anything that goes in a record can be stored in an Mnesia table, including atoms, pids, references, and so on.

What Should the Store Store?

The first step in using Mnesia is to figure out the information you'll need and the table structure you'll use for storing that information.



The Data to Store

For our Mafia friend-tracking application (which I decided to name `mafiapp`), the information we might want to store includes the following:

- The friend's name, to know who we're talking to when we ask for a service or provide one.
- The friend's contact information, to know how to reach that friend. This might take various forms, such as an email address, a cell phone number, or even notes regarding where that person likes to hang out.
- Additional information, such as when the person was born, occupation, hobbies, special traits, and so on.
- A unique expertise—our friend's forte. This field stands on its own because it's something we want to know explicitly. If someone's expertise is in cooking, and we're in dire need of a caterer, we know who to call. If we are in trouble and need to disappear for a while, we may look for friends who are pilots, camouflage experts, or excellent magicians.

Then we need to think about the services being exchanged. What will we want to know about them? Here are a few items:

- Who gave the service. Maybe it's you, the *consigliere*. Maybe it's the *padrino*. Maybe it's a friend of a friend, on your behalf. Maybe it's someone who then becomes your friend. We need to know.
- Who received the service.
- When the service was given. It's generally useful to be able to refresh someone's memory, especially when asking for a favor payback.
- Details regarding the services. It's much nicer (and more intimidating) to remember every tiny detail of the services we gave, as well as the date.

Table Structure

As I mentioned in the previous section, Mnesia is based on records and tables (ETS and DETS). To be exact, you can define an Erlang record and tell Mnesia to turn its definition into a table.

For example, suppose we're working on a recipe app, and we decided to have our record take this form:

```
-record(recipe, {name, ingredients=[], instructions=[]}).
```

We can then tell Mnesia to create a recipe table, which would store any number of `#recipe{}` records as table rows. We could have a recipe for pizza noted as follows:

```
#recipe{name=pizza,
        ingredients=[sauce,tomatoes,meat,dough],
        instructions=["order by phone"]}
```

And a recipe for soup might look like this:

```
#recipe{name=soup,
        ingredients=["who knows"],
        instructions=["open unlabeled can, hope for the best"]}
```

We could insert both of these in the recipe table, as is. We could then fetch the same records from the table and use them in the same way as any other records.

The primary key—the unique field by which it is the fastest to look up things in a table—would be the recipe name. That's because `name` is the first item in the record definition for `#recipe{}`. You'll also notice that in the pizza recipe, we use atoms as ingredients, and in the soup recipe, we use a string. As opposed to SQL tables, Mnesia tables *have no built-in type constraints*, as long as you respect the structure of the table itself (all entries in an Mnesia table must be the same kind of record, or tuples of the same size with the same first element).

So, how should we represent our friends and services information for our mafia application? Maybe as one table doing everything?

```
-record(friends, {name,
                  contact=[],
                  info=[],
                  expertise,
                  service=[]}). % {To, From, Date, Description} for services?
```

But this isn't the best choice possible. Nesting the data for services within friend-related data means that adding or modifying service-related information will require us to change friends at the same time. This might be annoying, especially since services imply at least two people. For each service, we would need to fetch the records for two friends and update them, even if there is no friend-specific information that needs to be modified.

A more flexible model is to use one table for each kind of data we need to store:

```
-record(mafiapp_friends, {name,
                          contact=[],
                          info=[],
                          expertise}).
-record(mafiapp_services, {from,
                           to,
                           date,
                           description}).
```

Having two tables should give us all the flexibility we need to search for information and modify it, with little overhead.

> **DON'T DRINK TOO MUCH KOOL-AID**
>
> You'll notice that I prefixed both the `friends` and `services` records with `mafiapp_`. While records are defined locally within our module, Mnesia tables are global to all the nodes that will be part of its cluster. This implies a high potential for name clashes if you're not careful. Therefore, it is a good idea to manually namespace your tables.

From Record to Table

Now that we know what we want to store, the next logical step is to decide how we're going to store it. Remember that Mnesia is built using ETS and DETS tables. This gives us two means of storage: on disk or in memory. We need to pick a strategy!

Here are the options:

`ram_copies`
: This option makes it so all data is stored exclusively in ETS, so in memory only. Memory should be limited to a theoretical 4GB (and practically around 3GB) for VMs compiled on 32 bits, but this limit is pushed further away on 64-bit (and half-word) VMs, assuming there is more than 4GB of memory available.

`disc_only_copies`
: This option means that the data is stored only in DETS, on disk only, and the storage is limited to DETS's 2GB limit.

`disc_copies`
: This option means that the data is stored both in ETS and on disk, so both in memory and on the hard disk. `disc_copies` tables are not restricted by DETS limits, as Mnesia uses a complex system of transaction logs and checkpoints that allow creating a disk-based backup of the table in memory.

For our current application, we will go with with `disc_copies`. The relationships we built with our friends need to be long-lasting, so it makes sense to be able to store things persistently. It would be quite annoying to wake up after a power failure, only to find out you've lost all the friendships you worked so hard to make. "Why just not use `disc_only_copies`?" you might ask. Well, having copies in memory is usually nice when we want to do more somewhat complex queries and searches, given they can be done without needing to access the disk, which is often the slowest part of any computer memory access, especially if it's a hard disk.

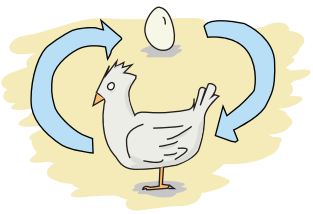
There's another hurdle on our path to filling the database with our precious data. Because of how ETS and DETS work, we need to define a table type. The types available bear the same definition as their ETS and DETS counterparts. The options are `set`, `bag`, and `ordered_set`, although `ordered_set` is not supported for `disc_only_copies` tables. (Tables of type `duplicate_bag` are not available for any of the storage types.) If you don't remember what these types do, look them up in Chapter 25.

The good news is that we're pretty much finished deciding how we're going to store things. The bad news is that there are still more things to understand about Mnesia before truly getting started.

Of Mnesia Schemas and Tables

Although Mnesia can work fine on isolated nodes, it does support distribution and replication to many nodes. To know how to store tables on disk, how to load them, and which other nodes they should be synchronized with, Mnesia needs to have a *schema* that holds all that information.

By default, Mnesia creates a schema directly in memory when it's started. It works fine for tables that need to live in RAM only, but when your schema needs to survive across many VM restarts, on all the nodes that are part of the Mnesia cluster, things get a bit more complex.



Mnesia depends on the schema, but Mnesia should also create the schema. This sets up a weird situation where the schema needs to be created by Mnesia without running Mnesia first! Fortunately, it's rather simple to deal with this in practice. We just need to call the function `mnesia:create_schema(ListOfNodes)` *before* starting Mnesia. It will create a bunch of files on each node, storing all the table information required. You don't need to be connected to the other nodes when calling the function, but the nodes need to be running; the function will set up the connections and get everything working for you.

By default, the schema will be created in the current working directory, wherever the Erlang node is running. To change this, the Mnesia application has a `dir` variable that can be set to pick where the schema will be stored. You can start your node as `erl -name SomeName -mnesia dir where/to/store/the/db`, or set it dynamically with `application:set_env(mnesia, dir, "where/to/store/the/db")`.

NOTE *A schema may fail to be created because one already exists, Mnesia is running on one of the nodes the schema should be on, you can't write to the directory Mnesia wants to write to, or another common file-handling problem occurred.*

Once the schema has been created, we can start Mnesia and begin creating tables. The function `mnesia:create_table/2` is what we need to use for this task. It takes two arguments: the table name and a list of options. The following are some of the options available:

`{attributes, List}`
: This is a list of all the items in a table. By default, it takes the form `[key, value]`, meaning you would need a record of the form `-record(TableName, {key,val})`. to work. Pretty much everyone cheats a little and uses a special construct (a compiler-supported macro) that extracts the element names from a record. The construct looks like a function call. To do it with our friends record, we would pass it as `{attributes, record_info(fields, mafiapp_friends)}`.

`{disc_copies, NodeList}`, `{disc_only_copies, NodeList}`, and `{ram_copies, NodeList}`
: This is where you specify how to store the tables, as explained in the previous section. Note that you can have many of these options present at once. As an example, you could define a table to be stored on disk and RAM on your master node, only in RAM on all of the slaves, and only on disk on a dedicated backup node by using all three of the options.

`{index, ListOfIntegers}`
: Mnesia tables let you have *indexes* on top of the basic ETS and DETS functionality. This is useful in cases where you are planning to build searches on record fields other than the primary key. As an example, our friends table will need an index for the `expertise` field. We can declare such an index as `{index, [#mafiapp_friends.expertise]}`. In general—and

this is true for many, many databases—you want to build indexes only on fields where the data is not too similar across all records. On a table with hundreds of thousands of entries, if your index at best splits your table in two groups to sort through, indexing will take a lot of resources (RAM and processor time) for very little benefit. An index that would split the same table in *N* groups of 10 or fewer elements, as an example, would be more useful for the resources it uses. Note that you do not need to put an index on the first field of the record (the second element of the tuple), as this is done for you by default.

`{record_name, Atom}`
: This is useful if you want to have a table that has a different name than the one your record uses. However, using this option forces you to use different functions to operate on the table than those commonly used. I don't recommend using this option, unless you really know you need it.

`{type, Type}`
: `Type` can be `set`, `ordered_set`, or `bag`, which are the same as the types used by ETS and described in Chapter 25.

`{local_content, true | false}`
: By default, all Mnesia tables have this option set to `false`. You should leave it that way if you want the tables and their data replicated on all nodes that are part of the schema (and those specified in the `disc_copies`, `disc_only_copies`, and `ram_copies` options). Setting this option to `true` will create all the tables on all the nodes, but the content will be the local content only; nothing will be shared. In this case, Mnesia becomes an engine to initialize similar empty tables on many nodes.

Here's the sequence of events that can happen when setting up your Mnesia schema and tables:

- Starting Mnesia for the first time creates a schema in memory, which is good for `ram_copies`. Other kinds of tables won't work with it.
- If you create a schema manually before starting Mnesia (or after stopping it), you will be able to create tables that sit on disk.
- Start Mnesia, and you can then start creating tables. Tables can't be created while Mnesia is not running.

NOTE *There is another way to do things. Whenever you have an Mnesia node running and tables created that you would want to port to disk, the function* `mnesia:change_table_copy_type(Table, Node, NewType)` *can be called to move a table to disk. More particularly, if you forgot to create the schema on disk, by calling* `mnesia:change_table_copy_type(schema, node(), disc_copies)`*, you'll be taking your RAM schema and turning it into a disk schema.*

Now it's time to get started with our application and see Mnesia in action.

Creating Tables

We'll handle creating the application and its tables with some weak TDD-style programming, using Common Test. Now you might dislike the idea of TDD, but stay with me, we'll do it in a relaxed manner, just as a way to guide our design more than anything else. None of that "run tests to make sure they fail" business (although you can feel free to do it that way). That we have tests in the end will just be a nice side effect, not an objective in itself. We'll mostly care about defining the interface of how `mafiapp` should behave and look, without doing it all from the Erlang shell. The tests won't even be distributed, but this example will still be a decent opportunity to get some practical use out of Common Test while learning Mnesia at the same time.

For this example, create a directory named *mafiapp-1.0.0* following the standard OTP structure:

```
ebin/
logs/
src/
test/
```

Installing the Database

We'll start by figuring out how we want to install the database. Because there is a need for a schema and initializing tables the first time around, we'll set up all the tests with an installation function that will ideally install things in Common Test's *priv_dir* directory. Let's begin with a basic test suite, `mafiapp_SUITE`, stored under the *test/* directory:

```
-module(mafiapp_SUITE).
-include_lib("common_test/include/ct.hrl").
-export([init_per_suite/1, end_per_suite/1,
         all/0]).
all() -> [].

init_per_suite(Config) ->
    Priv = ?config(priv_dir, Config),
    application:set_env(mnesia, dir, Priv),
    mafiapp:install([node()]),
    application:start(mnesia),
    application:start(mafiapp),
    Config.

end_per_suite(_Config) ->
    application:stop(mnesia),
    ok.
```

This test suite doesn't have any tests yet, but it gives us our first specification of how things should be done. We first pick where to put the Mnesia schema and database files by setting the `dir` variable to the value of `priv_dir`. This will put each instance of the schema and database in a private directory

generated with Common Test, guaranteeing that we will not have problems and clashes from earlier test runs.

We named the installation function `install` and gave it a list of nodes on which to install the records. Using this type of list is generally a better way to do things than hardcoding the information within the `install` function, as it is more flexible. Once this is done, Mnesia and `mafiapp` should be started.

We can now get into *src/mafiapp.erl* and start figuring out how the `install` function should work. First, we'll need to take the record definitions we had earlier and bring them back in:

```
-module(mafiapp).
-export([install/1]).

-record(mafiapp_friends, {name,
                          contact=[],
                          info=[],
                          expertise}).
-record(mafiapp_services, {from,
                           to,
                           date,
                           description}).
```

This looks good enough. Here's the `install/1` function:

```
install(Nodes) ->
    ok = mnesia:create_schema(Nodes),
    application:start(mnesia),
    mnesia:create_table(mafiapp_friends,
                        [{attributes, record_info(fields, mafiapp_friends)},
                         {index, [#mafiapp_friends.expertise]},
                         {disc_copies, Nodes}]),
    mnesia:create_table(mafiapp_services,
                        [{attributes, record_info(fields, mafiapp_services)},
                         {index, [#mafiapp_services.to]},
                         {disc_copies, Nodes},
                         {type, bag}]),
    application:stop(mnesia).
```

First, we create the schema on the nodes specified in the `Nodes` list. Then we start Mnesia, which is a necessary step in order to create tables. We create the two tables, named after the records `#mafiapp_friends{}` and `#mafiapp_services{}`. There's an index on the `expertise` field because we do expect to search friends by expertise.

You'll also see that the services table is of type `bag`. This is because it's possible to have multiple services with the same senders and receivers. Using a `set` table, we could deal with only unique senders, but `bag` tables handle this fine. Then there's an index on the `to` field of the table. That's because we expect to look up services either by who received them or who gave them, and indexes allow us to make any field faster to search.

The last thing to note is that the code stops Mnesia after creating the tables. This is just to match the behavior I decided I wanted in the test. What was in the test is how I expect to use the code, so I better make the code fit that idea. There is nothing wrong with just leaving Mnesia running after the installation, though.

Now, if we had successful test cases in our Common Test suite, the initialization phase would succeed with this `install` function. However, trying it with many nodes would bring failure messages to our Erlang shells. Do you have any idea why? Here's what it would look like:

```
Node A                      Node B
------                      ------
create_schema -----------> create_schema
start Mnesia
creating table ----------> ???
creating table ----------> ???
stop Mnesia
```

For the tables to be created on all nodes, Mnesia needs to run on all nodes. For the schema to be created, Mnesia must not be running on any nodes. Ideally, we could start Mnesia and stop it remotely. The good news is we can. Remember the `rpc` module introduced in Chapter 26? We have the function `rpc:multicall(Nodes, Module, Function, Args)` to do it for us. Let's change the `install/1` function definition to this one:

```
install(Nodes) ->
    ok = mnesia:create_schema(Nodes),
    rpc:multicall(Nodes, application, start, [mnesia]),
    mnesia:create_table(mafiapp_friends,
                        [{attributes, record_info(fields, mafiapp_friends)},
                         {index, [#mafiapp_friends.expertise]},
                         {disc_copies, Nodes}]),
    mnesia:create_table(mafiapp_services,
                        [{attributes, record_info(fields, mafiapp_services)},
                         {index, [#mafiapp_services.to]},
                         {disc_copies, Nodes},
                         {type, bag}]),
    rpc:multicall(Nodes, application, stop, [mnesia]).
```

Using `rpc` allows us to do the Mnesia action on all nodes. The scheme now looks like this:

```
Node A                      Node B
------                      ------
create_schema -----------> create_schema
start Mnesia ------------> start Mnesia
creating table ----------> replicating table
creating table ----------> replicating table
stop Mnesia -------------> stop Mnesia
```

Good, very good.

Starting the Application

The next part of the `init_per_suite/1` function we need to take care of is starting `mafiapp`. Actually, we don't need to do this, because our entire application depends on Mnesia. Starting Mnesia starts our application. However, there can be a noticeable delay between the time Mnesia starts and the time it finishes loading all tables from disk, especially if they're large. In such circumstances, a function such as `mafiapp`'s `start/2` might be the perfect place to do that kind of waiting, even if we need no process at all for normal operations.

We'll make *mafiapp.erl* implement the application behavior (`-behavior(application).`) and add the two following callbacks in the file (remember to export them):

```
start(normal, []) ->
    mnesia:wait_for_tables([mafiapp_friends,
                            mafiapp_services], 5000),
    mafiapp_sup:start_link().

stop(_) -> ok.
```

The secret is the `mnesia:wait_for_tables(TableList, TimeOut)` function. This function will wait for at most 5 seconds (an arbitrary number; replace it with what you think fits your data) or until the tables are available.

This doesn't tell us much regarding how the supervisor should behave, because `mafiapp_sup` doesn't have much to do at all:

```
-module(mafiapp_sup).
-behaviour(supervisor).
-export([start_link/1]).
-export([init/1]).

start_link(Tables) ->
    supervisor:start_link(?MODULE, Tables).

%% This does absolutely nothing, only there to
%% allow waiting for tables.
init(Tables) ->
    {ok, {{one_for_one, 1, 1}, []}}.
```

The supervisor does nothing, but because the starting of OTP applications is synchronous, it's actually one of the best places to put such synchronization points.

Last, add the following *mafiapp.app* file in the *ebin/* directory to make sure the application can be started:

```
{application, mafiapp,
 [{description, "Help the boss keep track of his friends"},
  {vsn, "1.0.0"},
  {modules, [mafiapp, mafiapp_sup]},
  {applications, [stdlib, kernel, mnesia]}]}.
```

We're now ready to write actual tests and implement our application. Or are we?

Access and Context

Before getting to the implementation of our app, let's take a look at how to use Mnesia to work with tables.

All modifications, or even reads, to a database table must be done in an *activity access context*. These contexts represent different types of transactions, or ways to run queries. Here are the options:

`transaction`
: A Mnesia transaction allows you to run a series of database operations as a single functional block. The whole block will run on all nodes or none of them; it succeeds entirely or fails entirely. When the transaction returns, we're guaranteed that the tables were left in a consistent state, and that different transactions didn't interfere with each other, even if they tried to manipulate the same data.

  This type of activity context is partially asynchronous. It will be synchronous for operations on the `local` node, but it will wait only for the confirmation from other nodes that they *will* commit the transaction, not that they *have* done it. With Mnesia, if the transaction worked locally and everyone else agreed to do it, it should work everywhere else. If it doesn't, possibly due to failures in the network or hardware, the transaction will be reverted at a later point in time. The protocol tolerates this for some efficiency reasons, but might give you confirmation that a transaction succeeded when it will be rolled back later.

`sync_transaction`
: This activity context is pretty much the same as `transaction`, but it is synchronous. If the guarantees of `transaction` aren't enough for you because you don't like the idea of a transaction telling you it succeeded when it may have failed due to weird errors, especially if you want to do things that have side effects (like notifying external services, spawning processes, and so on) related to the transaction's success, `sync_transaction` is what you want. Synchronous transactions will wait for the final confirmation for all other nodes before returning, making sure everything went fine 100 percent of the way.

  An interesting use case is that if you're doing a lot of transactions—enough to overload other nodes—switching to a synchronous mode should force things to go at a slower pace with less backlog accumulation, pushing the problem of overload up a level in your application.

`async_dirty`
: The `async_dirty` activity context basically bypasses all the transaction protocols and locking activities (although it will wait for active transactions to finish before proceeding). It will, however, keep on doing

everything that includes logging, replication, and so on. An `async_dirty` activity context will try to perform all actions locally, and then return, leaving other nodes' replication to take place asynchronously.

`sync_dirty`
: This activity context is to `async_dirty` what `sync_transaction` is to `transaction`. It will wait for the confirmation that things went fine on remote nodes, but will still stay out of all locking or transaction contexts. Dirty contexts are generally faster than transactions, but absolutely riskier by design. Handle these with care.

`ets`
: The `ets` activity context is basically a way to bypass everything Mnesia does and perform a series of raw operations on the underlying ETS tables, if there are any. No replication will be done. The `ets` activity context isn't something you usually need to use. It's yet another case of "you'll know when you need it; if in doubt, don't use it."

These are all the contexts within which common Mnesia operations can be run. These operations themselves are wrapped in a `fun` and executed by calling `mnesia:activity(Context, Fun)`. The `fun` can contain any Erlang function call, but be aware that it is possible for a transaction to be executed many times in case of failures or interruption by other transactions.

This means that if a transaction that reads a value from a table also sends a message before writing something back, it is entirely possible for the message to be sent dozens of times. As such, *no side effects of the kind should be included in the transaction.*

Reads, Writes, and More

I've referred to the table-modifying functions a lot, and it is now time to define them. Most of them are unsurprisingly similar to what ETS and DETS offer.

`write`
: By calling `mnesia:write(Record)`, where the name of the record is the name of the table, we're able to insert `Record` in the table. If the table is of type `set` or `ordered_set`, and the primary key (the second field of the record, not its name, under a tuple form), the element will be replaced. For `bag` tables, the whole record will need to be similar.

If the write operation is successful, `write/1` will return `ok`. Otherwise, it throws an exception that will abort the transaction. Throwing such an exception shouldn't be something that occurs frequently. It should mostly happen when Mnesia is not running, the table cannot be found, or the record is invalid.

`delete`
: This function is called as `mnesia:delete(TableName, Key)`. The record(s) that share this key will be removed from the table. It either returns `ok` or throws an exception, with semantics similar to `mnesia:write/1`.

`read`
: Called as `mnesia:read({TableName, Key})`, this function will return a list of records with their primary key matching `Key`. Much like `ets:lookup/2`, it will always return a list, even with tables of type `set` that can never have more than one result that matches the key. If no record matches, an empty list is returned. Similar to `delete` and `write` operations, in case of a failure, an exception is thrown.

`match_object`
: This function is similar to ETS's `match_object` function. It uses patterns such as those described in Chapter 25 to return entire records from the database table. For example, a quick way to look for friends with a given expertise is to use `mnesia:match_object(#mafiapp_friends{_ = '_', expertise = given})`. It will return a list of all matching entries in the table. Once again, failures end up in exceptions being thrown.

`select`
: This is similar to the ETS `select` function. It works using match specifications or `ets:fun2ms` as a way to do queries. (If you don't remember how this works, see "You Have Been Selected" on page 428 to brush up on your matching skills.) The function can be called as `mnesia:select(TableName, MatchSpec)`, and it will return a list of all items that fit the match specification. And again, in case of failure, an exception will be thrown.

Many other operations are available for Mnesia tables. However, those explained so far constitute a solid base for us to move forward. If you're interested in other operations, you can head to the Mnesia reference manual to find functions such as `first`, `last`, `next`, and `prev` for individual iterations, and `foldl` and `foldr` for folds over entire tables. You might also be interested in functions to manipulate tables themselves, such as `transform_table` (which is especially useful to add or remove fields in a record or a table) and `add_table_index`.

That makes for a lot of functions. To see how to use them realistically, we'll drive the tests forward a bit.

Implementing the First Requests

Now we'll add some test cases for adding data to our mafia application and for using it to look up friends and services.

A Test for Adding Services

To implement the requests, we'll first write a somewhat simple test demonstrating the behavior we want from our application. The test will be about adding services, but will contain implicit tests for more functionality.

We begin with the standard initialization stuff we need to add in most Common Test suites.

```
...
-export([init_per_suite/1, end_per_suite/1,
         init_per_testcase/2, end_per_testcase/2,
         all/0]).
-export([add_service/1]).

all() -> [add_service].
...

init_per_testcase(add_service, Config) ->
    Config.

end_per_testcase(_, _Config) ->
    ok.
```

Now for the test itself:

```
%% Services can go both ways: from a friend to the boss, or
%% from the boss to a friend! A boss friend is required!
add_service(_Config) ->
    {error, unknown_friend} = mafiapp:add_service("from name",
                                                  "to name",
                                                  {1946,5,23},
                                                  "a fake service"),
    ok = mafiapp:add_friend("Don Corleone", [], [boss], boss),
    ok = mafiapp:add_friend("Alan Parsons",
                            [{twitter,"@ArtScienceSound"}],
                            [{born, {1948,12,20}},
                             musician, 'audio engineer',
                             producer, "has projects"],
                            mixing),
    ok = mafiapp:add_service("Alan Parsons", "Don Corleone",
                             {1973,3,1}, "Helped release a Pink Floyd album").
```

Because we're adding a service, we should include both of the friends who will be part of the exchange. We'll use the function `mafiapp:add_friend(Name, Contact, Info, Expertise)` for that. Once the friends are added, we can actually add the service.

NOTE *If you've ever read other Mnesia tutorials, you'll find that some people are very eager to use records directly in the functions (say* `mafiapp:add_friend(#mafiapp_friend{name=...})`*). This is something that I try to avoid, as records are often better kept private. Changes in implementation might break the underlying record representation. This is not a problem in itself, but whenever you'll be changing the record definition, you'll need to recompile and, if possible, atomically update all modules that use that record so that they can keep working in a running application. Simply wrapping things in functions gives a somewhat cleaner interface that won't require any module using your database or application to include records through* .hrl *files, which is frankly annoying.*

You'll notice that the test we just defined doesn't actually look for services. This is because we'll search for them when looking up users. For now, we can try to implement the functionality required for the test using Mnesia transactions. The first function we'll add to *mafiapp.erl* will be used to add a user to the database:

```
add_friend(Name, Contact, Info, Expertise) ->
    F = fun() ->
        mnesia:write(#mafiapp_friends{name=Name,
                                      contact=Contact,
                                      info=Info,
                                      expertise=Expertise})
    end,
    mnesia:activity(transaction, F).
```

We're defining a single function that writes the record `#mafiapp_friends{}`. This is a somewhat simple transaction. `add_services/4` should be a little more complex:

```
add_service(From, To, Date, Description) ->
    F = fun() ->
        case mnesia:read({mafiapp_friends, From}) =:= [] orelse
             mnesia:read({mafiapp_friends, To}) =:= [] of
            true ->
                {error, unknown_friend};
            false ->
                mnesia:write(#mafiapp_services{from=From,
                                               to=To,
                                               date=Date,
                                               description=Description})
        end
    end,
    mnesia:activity(transaction,F).
```

In the transaction, we first do one or two reads to see if the friends we're trying to add are already in the database. If either friend is not there, the tuple `{error, unknown_friend}` is returned, as per the test specification.

If both members of the transaction are found, we'll write the service to the database instead.

NOTE *Validating the input is left to your discretion. Doing so requires only writing custom Erlang code like anything else you would program with the language. If it is possible, doing as much validation as possible outside the transaction context is a good idea. Code in the transaction might run many times and compete for database resources.*

Based on this, we should be able to run the first test batch. To do so, we'll use the following test specification, *mafiapp.spec* (placed at the root of the project):

```
{alias, root, "./test/"}.
{logdir, "./logs/"}.
{suites, root, all}.
```

And we need the following Emakefile (also at the root):

```
{["src/*", "test/*"],
 [{i,"include"}, {outdir, "ebin"}]}.
```

Then we can run the tests:

```
$ erl -make
Recompile: src/mafiapp_sup
Recompile: src/mafiapp
$ ct_run -pa ebin/ -spec mafiapp.spec
... <snip> ...
Common Test: Running make in test directories...
Recompile: mafiapp_SUITE
... <snip> ...
Testing learn-you-some-erlang.mafiapp: Starting test, 1 test cases
... <snip> ...
Testing learn-you-some-erlang.mafiapp: TEST COMPLETE, 1 ok, 0 failed of 1 test
cases
... <snip> ...
```

All right, it passes. That's good. Let's move on to the next tests.

NOTE *When running the Common Test suite, you might get errors saying that some directories are not found. The solution is to use* `ct_run -pa ebin/` *or* `erl -name ct -pa `pwd`/ebin` *(or full paths). While starting the Erlang shell makes the current working directory the node's current working directory, calling* `ct:run_test/1` *changes the current working directory to a new one. This breaks relative paths such as* `./ebin/`*. Using absolute paths solves the problem.*

Tests for Lookups

The add_service/1 test lets us add both friends and services. The next tests should focus on making it possible to look up information. For the sake of simplicity, we'll add the boss to all possible future test cases:

```
init_per_testcase(add_service, Config) ->
    Config;
init_per_testcase(_, Config) ->
    ok = mafiapp:add_friend("Don Corleone", [], [boss], boss),
    Config.
```

The use case we'll want to emphasize is looking up friends by their name. While we could very well search through services only, in practice, we might want to look up people by name more than actions. Very rarely will the boss ask, "Who delivered that guitar to whom, again?" No, he will more likely ask, "Who is it that delivered the guitar to our friend Pete Cityshend?" and try to look up this friend's history through his name to find details about the service. For this case, the next test is friend_by_name/1:

```
-export([add_service/1, friend_by_name/1]).

all() -> [add_service, friend_by_name].
...
friend_by_name(_Config) ->
    ok = mafiapp:add_friend("Pete Cityshend",
                            [{phone, "418-542-3000"},
                             {email, "quadrophonia@example.org"},
                             {other, "yell real loud"}],
                            [{born, {1945,5,19}},
                             musician, popular],
                            music),
    {"Pete Cityshend",
     _Contact, _Info, music,
     _Services} = mafiapp:friend_by_name("Pete Cityshend"),
    undefined = mafiapp:friend_by_name(make_ref()).
```

This test verifies that we can insert a friend and look him up, as well as what should be returned when we don't know any friend by that name. We'll have a tuple structure returning all kinds of details, including services, which we do not care about for now. We mostly want to find people, although duplicating the information would make the test stricter.

We can implement mafiapp:friend_by_name/1 by using a single Mnesia read. Our record definition for #mafiapp_friends{} put the friend name as the primary key of the table (the first one defined in the record). By using mnesia:read({Table, Key}), we can get things going easily, with minimal wrapping to make it fit the test:

```
friend_by_name(Name) ->
    F = fun() ->
```

```
        case mnesia:read({mafiapp_friends, Name}) of
            [#mafiapp_friends{contact=C, info=I, expertise=E}] ->
                {Name,C,I,E,find_services(Name)};
            [] ->
                undefined
        end
    end,
    mnesia:activity(transaction, F).
```

This function alone should be enough to get the tests to pass, as long as you remember to export it. We do not care about `find_services(Name)` for now, so we'll just stub it out:

```
%%% PRIVATE FUNCTIONS
find_services(_Name) -> undefined.
```

That being done, the new test should also pass:

```
$ erl -make
... <snip> ...
$ ct_run -pa ebin/ -spec mafiapp.spec
... <snip> ...
Testing learn-you-some-erlang.wiptests: TEST COMPLETE, 2 ok, 0 failed of 2
test cases
... <snip> ...
```

It would be nice to put a bit more details into the services area of the request. Here's the test to do that:

```
-export([add_service/1, friend_by_name/1, friend_with_services/1]).

all() -> [add_service, friend_by_name, friend_with_services].
...
friend_with_services(_Config) ->
    ok = mafiapp:add_friend("Someone", [{other, "at the fruit stand"}],
                            [weird, mysterious], shadiness),
    ok = mafiapp:add_service("Don Corleone", "Someone",
                             {1949,2,14}, "Increased business"),
    ok = mafiapp:add_service("Someone", "Don Corleone",
                             {1949,12,25}, "Gave a Christmas gift"),
    %% We don't care about the order. The test was made to fit
    %% whatever the functions returned.
    {"Someone",
     _Contact, _Info, shadiness,
     [{to, "Don Corleone", {1949,12,25}, "Gave a Christmas gift"},
      {from, "Don Corleone", {1949,2,14}, "Increased business"}]} =
    mafiapp:friend_by_name("Someone").
```

In this test, Don Corleone helped a shady person with a fruit stand to grow his business. Said shady person at the fruit stand later gave a Christmas gift to the boss, who never forgot about it.

You can see that we still use `friend_by_name/1` to search entries. Although the test is overly generic and not too complete, we can probably figure out what we want to do. Fortunately, the total absence of maintainability requirements kind of makes it okay to do something this incomplete.

The `find_service/1` implementation will need to be a bit more complex than the previous one. While `friend_by_name/1` could work just by querying the primary key, the friend's name in services is only the primary key when searching in the `from` field. We still need to deal with the `to` field. There are many ways to handle this one, such as using `match_object` many times or reading the entire table and filtering data manually. I chose to use match specifications and the `ets:fun2ms/1` parse transform:

```
-include_lib("stdlib/include/ms_transform.hrl").
...
find_services(Name) ->
    Match = ets:fun2ms(
            fun(#mafiapp_services{from=From, to=To, date=D, description=Desc})
                when From =:= Name ->
                    {to, To, D, Desc};
               (#mafiapp_services{from=From, to=To, date=D, description=Desc})
                when To =:= Name ->
                    {from, From, D, Desc}
            end
    ),
    mnesia:select(mafiapp_services, Match).
```

This match specification has two clauses: whenever `From` matches `Name`, we return a `{to, ToName, Date, Description}` tuple. Whenever `Name` matches `To` instead, the function returns a tuple of the form `{from, FromName, Date, Description}`, allowing us to have a single operation that includes both services given and received.

Note that `find_services/1` does not run in any transaction. That's because the function is called only within `friend_by_name/1`, which runs in a transaction already. Mnesia can run nested transactions, but it's useless to do so in this case.

NOTE *When writing larger applications that use Mnesia, it can be interesting to separate the operations on the data stored in Mnesia from the part of the code that actually runs the operations (using `mnesia:activity/2`). That way, you can specify multiple operations independently, and have calling code decide whether to run them as synchronous or asynchronous transactions, or use any other context.*

Running the tests again should reveal that all three of them work.

The last use case is in regard to searching for friends through their expertise. The following test case illustrates how we might find our friend the red panda when we need a climbing expert for some task:

```
-export([add_service/1, friend_by_name/1, friend_with_services/1,
         friend_by_expertise/1]).
```

```
all() -> [add_service, friend_by_name, friend_with_services,
          friend_by_expertise].
...
friend_by_expertise(_Config) ->
    ok = mafiapp:add_friend("A Red Panda",
                            [{location, "in a zoo"}],
                            [animal,cute],
                            climbing),
    [{"A Red Panda",
      _Contact, _Info, climbing,
      _Services}] = mafiapp:friend_by_expertise(climbing),
    [] = mafiapp:friend_by_expertise(make_ref()).
```

To implement this, we'll need to read something other than the primary key. We could use match specifications, but we've already done that. Plus, we need to match on only one field. The `mnesia:match_object/1` function is well adapted for this:

```
friend_by_expertise(Expertise) ->
    Pattern = #mafiapp_friends{_ = '_',
                               expertise = Expertise},
    F = fun() ->
        Res = mnesia:match_object(Pattern),
        [{Name,C,I,Expertise,find_services(Name)} ||
            #mafiapp_friends{name=Name,
                             contact=C,
                             info=I} <- Res]
    end,
    mnesia:activity(transaction, F).
```

In this case, we first declare the pattern. We need to use `_ = '_'` to declare all undefined values as a match-all specification (`'_'`). Otherwise, the `match_object/1` function will look only for entries where everything but the expertise is the atom `undefined`.

Once the result is obtained, we format the record into a tuple, in order to respect the test. Again, compiling and running the tests will reveal that this implementation works. Hooray, we implemented the whole specification!

Accounts and New Needs

No software project is ever really finished. Users using the system bring new needs to light or break it in unexpected ways. The boss, even before using our brand-new software, decided that he wants a feature that lets him quickly go through all of our friends and see who we owe things to, and who owes us things.

Here's the test for that one:

```
...
init_per_testcase(accounts, Config) ->
    ok = mafiapp:add_friend("Consigliere", [], [you], consigliere),
    Config;
```

```
...
accounts(_Config) ->
    ok = mafiapp:add_friend("Gill Bates", [{email, "ceo@macrohard.com"}],
                            [clever,rich], computers),
    ok = mafiapp:add_service("Consigliere", "Gill Bates",
                             {1985,11,20}, "Bought 15 copies of software"),
    ok = mafiapp:add_service("Gill Bates", "Consigliere",
                             {1986,8,17}, "Made computer faster"),
    ok = mafiapp:add_friend("Pierre Gauthier", [{other, "city arena"}],
                            [{job, "sports team GM"}], sports),
    ok = mafiapp:add_service("Pierre Gauthier", "Consigliere", {2009,6,30},
                             "Took on a huge, bad contract"),
    ok = mafiapp:add_friend("Wayne Gretzky", [{other, "Canada"}],
                            [{born, {1961,1,26}}, "hockey legend"],
                            hockey),
    ok = mafiapp:add_service("Consigliere", "Wayne Gretzky", {1964,1,26},
                             "Gave first pair of ice skates"),
    %% Wayne Gretzky owes us something so the debt is negative.
    %% Gill Bates' services and debts are equal.
    %% Gauthier is owed a service.
    [{-1,"Wayne Gretzky"},
     {0,"Gill Bates"},
     {1,"Pierre Gauthier"}] = mafiapp:debts("Consigliere"),
    [{1, "Consigliere"}] = mafiapp:debts("Wayne Gretzky").
```

We're adding three test friends: Gill Bates, Pierre Gauthier, and hockey Hall of Famer Wayne Gretzky. There is an exchange of services going on with each of them and you, the consigliere. (We didn't pick the boss for this test because he is being used by other tests, and it would mess with the results!)

The `mafiapp:debts(Name)` function looks for a name, and counts all the services where the name is involved. When someone owes us something, the value is negative. When we're even, it's 0. When we owe something to someone, the value is 1. We can thus say that the `debt/1` function returns the number of services owed to different people.

The implementation of this function is going to be a bit more complex:

```
-export([install/1, add_friend/4, add_service/4, friend_by_name/1,
         friend_by_expertise/1, debts/1]).
...
debts(Name) ->
    Match = ets:fun2ms(
        fun(#mafiapp_services{from=From, to=To}) when From =:= Name ->
            {To,-1};
           (#mafiapp_services{from=From, to=To}) when To =:= Name ->
            {From,1}
        end),
    F = fun() -> mnesia:select(mafiapp_services, Match) end,
    Dict = lists:foldl(fun({Person,N}, Dict) ->
                         dict:update(Person, fun(X) -> X + N end, N, Dict)
                       end,
```

```
                    dict:new(),
                    mnesia:activity(transaction, F)),
    lists:sort([{V,K} || {K,V} <- dict:to_list(Dict)]).
```

Whenever Mnesia queries become more complex, match specifications are usually going to be part of your solution. They let you run basic Erlang functions, which make them invaluable when it comes to specific result generation. In this function, the match specification is used to find that whenever the service given comes from `Name`, its value is -1 (we gave a service; they owe us one). When `Name` matches `To`, the value returned will be 1 (we received a service; we owe one). In both cases, the value is coupled to a tuple containing the name.

Including the name is necessary for the second step of the computation, where we'll try to count all the services given for each person and give a unique cumulative value. Again, there are many ways to obtain this value. I picked one that required me to stay as little time as possible within a transaction, to allow as much as possible of my code to be separated from the database, which allows more transactions to be done in general. This is useless for `mafiapp`, but in high-performance cases, it can reduce the contention for resources in major ways.

The solution I picked is to take all the values, put them in a dictionary, and use the function `dict:update(Key, Operation, Dict)` to increment or decrement the value based on whether a move is for us or from us. By putting this into a fold over the results given by Mnesia, we get a list of all the values required.

The final step is to flip the values around (from `{Key, Debt}` to `{Debt, Key}`) and sort based on this, which will give the results desired.

Meet the Boss

Our software product should at least be tried once in production. We'll do this by setting up the node the boss will use, and then the consigliere's node.

```
$ erl -name corleone -pa ebin/
```

```
$ erl -name genco -pa ebin/
```

Once both nodes are started, you can connect them and install the app:

```
(corleone@ferdmbp.local)1> net_kernel:connect('genco@ferdmbp.local').
true
(corleone@ferdmbp.local)2> mafiapp:install([node()|nodes()]).
{[ok,ok],[]}
(corleone@ferdmbp.local)3>
```

```
=INFO REPORT==== 8-Apr-2013::20:02:26 ===
    application: mnesia
    exited: stopped
    type: temporary
```

You can then start running Mnesia and `mafiapp` on both nodes by calling `application:start(mnesia)`, `application:start(mafiapp)`. Once it's running, you can see if everything is working as it should by calling `mnesia:system_info()`, which will display status information about your whole setup:

```
(genco@ferdmbp.local)2> mnesia:system_info().
===> System info in version "4.7", debug level = none <===
opt_disc. Directory "/Users/ferd/.../Mnesia.genco@ferdmbp.local" is used.
use fallback at restart = false
running db nodes   = ['corleone@ferdmbp.local','genco@ferdmbp.local']
stopped db nodes   = []
master node tables = []
remote             = []
ram_copies         = []
disc_copies        = [mafiapp_friends,mafiapp_services,schema]
disc_only_copies   = []
[{'corleone@...',disc_copies},{'genco@...',disc_copies}] = [schema,
mafiapp_friends,
mafiapp_services]
5 transactions committed, 0 aborted, 0 restarted, 2 logged to disc
0 held locks, 0 in queue; 0 local transactions, 0 remote
0 transactions waits for other nodes: []
yes
```

You can see that both nodes are in the running database nodes, and that both tables and the schema are written to disk and in RAM (`disc_copies`). We can start writing and reading data from the database. Of course, adding Don Corleone to the database is a good starting step:

```
(corleone@ferdmbp.local)4> ok = mafiapp:add_friend("Don Corleone", [], [boss], boss).
ok
(corleone@ferdmbp.local)5> mafiapp:add_friend(
(corleone@ferdmbp.local)5>     "Albert Einstein",
(corleone@ferdmbp.local)5>     [{city, "Princeton, New Jersey, USA"}],
(corleone@ferdmbp.local)5>     [physicist, savant,
(corleone@ferdmbp.local)5>         [{awards, [{1921, "Nobel Prize"}]}]],
(corleone@ferdmbp.local)5>     physicist).
ok
```

All right, we added friends from the `corleone` node. Let's try adding a service from the `genco` node:

```
(genco@ferdmbp.local)3> mafiapp:add_service("Don Corleone",
(genco@ferdmbp.local)3>                     "Albert Einstein",
(genco@ferdmbp.local)3>                     {1905, '?', '?'},
(genco@ferdmbp.local)3>                     "Added the square to E = MC").
ok
```

```
(genco@ferdmbp.local)4> mafiapp:debts("Albert Einstein").
[{1,"Don Corleone"}]
```

And all these changes can also be reflected back to the `corleone` node:

```
(corleone@ferdmbp.local)6> mafiapp:friend_by_expertise(physicist).
[{"Albert Einstein",
  [{city,"Princeton, New Jersey, USA"}],
  [physicist,savant,[{awards,[{1921,"Nobel Prize"}]}]],
  physicist,
  [{from,"Don Corleone",
         {1905,'?','?'},
         "Added the square to E = MC"}]}]
```

Now, if you shut down one of the nodes and start it again, things should still be fine:

```
(corleone@ferdmbp.local)7> init:stop().
ok

$ erl -name corleone -pa ebin
... <snip> ...
(corleone@ferdmbp.local)1> net_kernel:connect('genco@ferdmbp.local').
true
(corleone@ferdmbp.local)2>
application:start(mnesia), application:start(mafiapp).
ok
(corleone@ferdmbp.local)3> mafiapp:friend_by_expertise(physicist).
[{"Albert Einstein",
... <snip> ...
         "Added the square to E = MC"}]}]
```

Isn't it nice? We've now used Mnesia successfully!

NOTE *If you end up working on a system where tables start to get messy, or if you're just curious about looking at entire tables, call the function* `tv:start()`*. It will start a graphical table viewer that lets you interact with tables visually, rather than through code.*

Deleting Stuff, Demonstrated

Wait—did we just entirely skip over deleting records from a database? Oh no! Let's add a table and use it to see how to get rid of stuff.

We'll create a little feature for you and the boss that lets you store personal enemies, for personal reasons:

```
-record(mafiapp_enemies, {name,
                          info=[]}).
```

Because this is personal information, we'll need to use slightly different table settings, with `local_content` set as an option when installing the table. This will let the table be private to each node, so that no one can read anyone else's personal enemies accidentally (although `rpc` would make it trivial to circumvent).

Here's the new `install` function, preceded by `mafiapp`'s `start/2` function, changed for the new table:

```
start(normal, []) ->
    mafiapp_sup:start_link([mafiapp_friends,
                            mafiapp_services,
                            mafiapp_enemies]).
...
install(Nodes) ->
    ok = mnesia:create_schema(Nodes),
    application:start(mnesia),
    mnesia:create_table(mafiapp_friends,
                        [{attributes, record_info(fields, mafiapp_friends)},
                         {index, [#mafiapp_friends.expertise]},
                         {disc_copies, Nodes}]),
    mnesia:create_table(mafiapp_services,
                        [{attributes, record_info(fields, mafiapp_services)},
                         {index, [#mafiapp_services.to]},
                         {disc_copies, Nodes},
                         {type, bag}]),
    mnesia:create_table(mafiapp_enemies,
                        [{attributes, record_info(fields, mafiapp_enemies)},
                         {disc_copies, Nodes},
                         {local_content, true}]),
    application:stop(mnesia).
```

The `start/2` function now sends `mafiapp_enemies` through the supervisor to keep things alive there. The `install/1` function will be useful for tests and fresh installations, but if you're doing things in production, you can call `mnesia:create_table/2` in production to add tables. Depending on the load on your system and how many nodes you have, you might want to have a few practice runs in staging first, though.

Now, we can write a simple test to work with our database and see how it goes, still in `mafiapp_SUITE`:

```
...
-export([add_service/1, friend_by_name/1, friend_by_expertise/1,
         friend_with_services/1, accounts/1, enemies/1]).

all() -> [add_service, friend_by_name, friend_by_expertise,
          friend_with_services, accounts, enemies].
...
enemies(_Config) ->
    undefined = mafiapp:find_enemy("Edward"),
    ok = mafiapp:add_enemy("Edward", [{bio, "Vampire"},
                                      {comment, "He sucks (blood)"}]),
```

```
    {"Edward", [{bio, "Vampire"},
                {comment, "He sucks (blood)"}]} =
       mafiapp:find_enemy("Edward"),
    ok = mafiapp:enemy_killed("Edward"),
    undefined = mafiapp:find_enemy("Edward").
```

This is going to be similar to previous runs for `add_enemy/2` and `find_enemy/1`. All we'll need to do is a basic insertion for the former, and an `mnesia:read/1` based on the primary key for the latter:

```
add_enemy(Name, Info) ->
    F = fun() -> mnesia:write(#mafiapp_enemies{name=Name, info=Info}) end,
    mnesia:activity(transaction, F).

find_enemy(Name) ->
    F = fun() -> mnesia:read({mafiapp_enemies, Name}) end,
    case mnesia:activity(transaction, F) of
        [] -> undefined;
        [#mafiapp_enemies{name=N, info=I}] -> {N,I}
    end.
```

The `enemy_killed/1` function is the one that's a bit different:

```
enemy_killed(Name) ->
    F = fun() -> mnesia:delete({mafiapp_enemies, Name}) end,
    mnesia:activity(transaction, F).
```

And that's pretty much it for basic deletions. You can export the functions and run the test suite, and all the tests should still pass.

When trying the tests on two nodes (after deleting the previous schemas, or possibly just calling the `create_table` function), we should be able to see that data between tables isn't shared:

```
$ erl -name corleone -pa ebin
```

```
$ erl -name genco -pa ebin
```

With the nodes started, reinstall the database:

```
(corleone@ferdmbp.local)1> net_kernel:connect('genco@ferdmbp.local').
true
(corleone@ferdmbp.local)2> mafiapp:install([node()|nodes()]).
=INFO REPORT==== 8-Apr-2013::21:21:47 ===
... <snip> ...
{[ok,ok],[]}
```

Start the apps and get going:

```
(genco@ferdmbp.local)1> application:start(mnesia), application:start(mafiapp).
ok
```

```
(corleone@ferdmbp.local)3> application:start(mnesia), application:start(mafiapp).
ok
(corleone@ferdmbp.local)4> mafiapp:add_enemy("Some Guy", "Disrespected his family").
ok
(corleone@ferdmbp.local)5> mafiapp:find_enemy("Some Guy").
{"Some Guy","Disrespected his family"}
```

```
(genco@ferdmbp.local)2> mafiapp:find_enemy("Some Guy").
undefined
```

And you can see that no data is shared. Deleting the entry is also as simple:

```
(corleone@ferdmbp.local)6> mafiapp:enemy_killed("Some Guy").
ok
(corleone@ferdmbp.local)7> mafiapp:find_enemy("Some Guy").
undefined
```

Finally!

Query List Comprehensions

If you've followed this chapter (or worse, skipped right to this part!), thinking to yourself, "Damn, I don't like the way Mnesia looks," you might like this section. If you liked how Mnesia looked, you might also like this section. And if you like list comprehensions, you'll definitely like this section.

Query list comprehensions (QLCs) are basically a compiler trick using parse transforms that let you use list comprehensions for any data structure that can be searched and iterated through. They're implemented for Mnesia, DETS, and ETS, but can also be implemented for things like `gb_trees`.

Once you add `-include_lib("stdlib/include/qlc.hrl").` to your module, you can start using list comprehensions with something called a *query handle* as a generator. The query handle allows any iterable data structure to work with QLCs. In the case of Mnesia, you can use `mnesia:table(TableName)` as a list comprehension generator, and from that point on, you can use list comprehensions to query any database table by wrapping them in a call to `qlc:q(...)`.

This will return a modified query handle, with more details than the one returned by the table. This newest one can subsequently be modified some more by using functions like `qlc:sort/1-2`, and can be evaluated by using `qlc:eval/1` or `qlc:fold/1`.

Let's experiment with this approach. We'll rewrite a few of the `mafiapp` functions. You can make a copy of *mafiapp-1.0.0* and call it *mafiapp-1.0.1* (don't forget to bump the version in the *.app* file).

First, we'll rework `friend_by_expertise`. This function is currently implemented using `mnesia:match_object/1`. Here's a version using a QLC:

```
friend_by_expertise(Expertise) ->
    F = fun() ->
        qlc:eval(qlc:q(
            [{Name,C,I,E,find_services(Name)} ||
             #mafiapp_friends{name=Name,
                              contact=C,
                              info=I,
                              expertise=E} <- mnesia:table(mafiapp_friends),
             E =:= Expertise]))
    end,
    mnesia:activity(transaction, F).
```

You can see that except for the part where we call `qlc:eval/1` and `qlc:q/1`, this is a normal list comprehension. You have the final expression in `{Name,C,I,E,find_services(Name)}`, the generator in `#mafiapp{...} <- mnesia:table(...)`, and finally, a condition with `E =:= Expertise`. Searching through database tables is now a bit more natural, Erlang-wise.

Now let's try a slightly more complex example. We'll take a look at the `debts/1` function. It was implemented using a match specification and then a fold over to a dictionary. Here's how that might look using a QLC:

```
debts(Name) ->
    F = fun() ->
        QH = qlc:q(
            [if Name =:= To -> {From,1};
                Name =:= From -> {To,-1}
             end || #mafiapp_services{from=From, to=To} <-
                      mnesia:table(mafiapp_services),
                    Name =:= To orelse Name =:= From]),
        qlc:fold(fun({Person,N}, Dict) ->
                   dict:update(Person, fun(X) -> X + N end, N, Dict)
                 end,
                 dict:new(),
                 QH)
    end,
    lists:sort([{V,K} || {K,V} <- dict:to_list(mnesia:activity(transaction, F))]).
```

The match specification is no longer necessary. The list comprehension (saved to the `QH` query handle) does that part. The fold has been moved into the transaction, and it is used as a way to evaluate the query handle. The resulting dictionary is the same as the one that was formerly returned by `lists:foldl/3`. The last part, sorting, is handled outside the transaction by taking whatever dictionary `mnesia:activity/1` returned and converting it to a list.

And there you go. If you write these functions in your *mafiapp-1.0.1* application and run the test suite, all six tests should still pass.

Remember Mnesia

That's it for Mnesia. It's quite a complex database, and we've covered only a moderate portion of everything it can do. Pushing further ahead will require you to read the Erlang manuals and dive into the code. Programmers who have true production experience with Mnesia in large, scalable systems that have been running for years are rather rare. You can find a few of them on mailing lists, sometimes answering a few questions, but they're generally busy people.

Otherwise, Mnesia is always a very nice tool for smaller applications where you find picking a storage layer to be very annoying, or even larger ones where you will have a known number of nodes. Being able to store and replicate Erlang terms directly is a very neat thing—something other languages tried to write for years using object-relational mapping.

Interestingly enough, if you put your mind to it, you could likely write QLC selectors for SQL databases or any other kind of storage that allows iteration.

Mnesia and its toolchain have all the potential to be very useful in some of your future applications. For now though, we'll move on to additional tools to help you develop Erlang systems with Dialyzer.

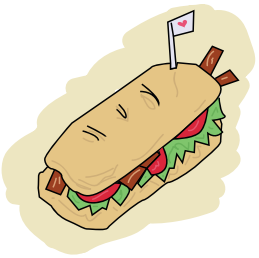
30

TYPE SPECIFICATIONS AND DIALYZER

This chapter focuses on Dialyzer, which is a very effective tool when it comes to analyzing Erlang code. It's used to find all kinds of discrepancies, such as code that will never be executed, but its main use is to detect type errors in your Erlang code base. We'll look at why Dialyzer was created, the guiding principles behind it, and its capabilities to find type-related errors. Of course, we'll also work through a few examples of Dialyzer in use.

PLTs Are the Best Sandwiches

Our first step is to create Dialyzer's *persistent lookup table (PLT)*, which is a compilation of all the details Dialyzer can identify about the applications and modules that are part of your standard Erlang distribution, as well as code outside OTP.



It takes quite a while to compile everything, especially if you're working on a platform that does not provide native compilation through HiPE (namely Windows) or on older versions of Erlang. Fortunately, things tend to get faster with time, and the newest releases of Erlang (R15B02 onward) have parallel PLT building to make it even faster.

Enter the following command into a terminal, and let it run as long as it needs (in my case, that tends to be under 10 minutes):

```
$ dialyzer --build_plt --apps erts kernel stdlib crypto mnesia sasl common_test eunit
Compiling some key modules to native code... done in 1m19.99s
Creating PLT /Users/ferd/.dialyzer_plt ...
eunit_test.erl:302: Call to missing or unexported function eunit_test:nonexisting_function/0
Unknown functions:
compile:file/2
compile:forms/2
... <snip> ...
xref:stop/1
Unknown types:
compile:option/0
done in 6m39.15s
done (warnings were emitted)
```

This command builds the PLT by specifying which OTP applications we want to include in it. You can ignore the warnings if you want, as Dialyzer can deal with unknown functions when looking for type errors (this has to do with how its type inference algorithm works, as discussed in the next section). Some Windows users will see an error message saying "The HOME environment variable needs to be set so that Dialyzer knows where to find the default PLT." This is because Windows doesn't always come with the `HOME` environment variable set, and Dialyzer doesn't know where to dump the PLT. Set the variable to wherever you want Dialyzer to place its files.

If you want, you can include applications like `ssl` or `reltool` by adding them to the sequence that follows `--apps`, or if your PLT is already built, by calling the following:

```
$ dialyzer --add_to_plt --apps ssl reltool
```

If you want to add your own applications or modules to the PLT, you can do so by using `-r Directories`, which will look for all *.erl* or *.beam* files (as long as they are compiled with `debug_info`).

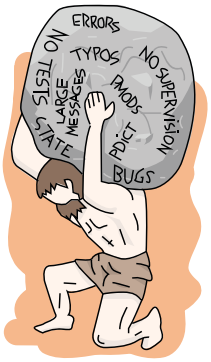
Moreover, Dialyzer lets you have many PLTs by specifying them with the `--plt Name` option in any of the commands you use, and pick a specific PLT. Alternatively, if you built many *disjoint* PLTs, where none of the included modules are shared between PLTs, you can "merge" them by using `--plts Name1 Name2 ... NameN`. This is especially useful when you want to have different PLTs in your system for different projects or Erlang versions.

While the PLT is still building, let's get acquainted with Dialyzer's mechanism for finding type errors.

Success Typing

As with most other dynamic programming languages, Erlang programs are always at risk of suffering from type errors. A programmer passes in some arguments to a function he shouldn't have, and maybe he forgot to test things properly. The program gets deployed, and everything seems to be going okay. Then at four in the morning, your company's operations guy's cell phone starts ringing because your piece of software is repeatedly crashing—enough that the supervisors can't cope with the sheer weight of your mistakes.

The next morning, you get to the office, and you find your computer has been reformatted, your car is keyed, and your commit rights have been revoked, all by the operations guy who has had enough of you accidentally controlling his work schedule.



That entire debacle could have been prevented by a compiler that has a static type analyzer to verify programs before they run.

While Erlang doesn't crave a type system as much as other dynamic languages, thanks to its reactive approach to runtime errors, it is definitely nice to benefit from the additional safety provided by early type-related error discovery.

Usually, languages with static type systems are designed that way. The semantics of the languages is heavily influenced by what their type systems allow and don't allow. For example, consider this function:

```
foo(X) when is_integer(X) -> X + 1.
foo(X) -> list_to_atom(X).
```

Most type systems are unable to properly represent the types of this function. They can see that it can take an integer or a list and return an integer or an atom, but they won't track the dependency between the input type of the function and its output type (conditional types and intersection types are able to, but they can be verbose). This means that writing such functions, which is entirely normal in Erlang, can result in some uncertainty for the type analyzer when these functions are used later in the code.

Generally speaking, analyzers will want to actually prove that there will be no type errors at runtime, as in *mathematically* prove. This means that in a few circumstances, the type checker will disallow certain practically valid operations for the sake of removing uncertainty that could lead to crashes.

Implementing such a type system would likely mean forcing Erlang to change its semantics.

The problem is that by the time Dialyzer came around, Erlang was already well in use for very large projects. For any tool like Dialyzer to be accepted, it needed to respect Erlang's philosophies. If Erlang allows pure nonsense in its types that can be solved only at runtime, so be it. The type

checker doesn't have a right to complain. No programmer likes a tool that tells him his program cannot run when it has been doing so in production for a few months already!

The other option is to have a type system that will not prove the absence of errors, but will do a best effort at detecting whatever it can. You can make such detection really good, but it will never be perfect—it's a trade-off.

Dialyzer's type system designers made the decision not to prove that a program is error-free when it comes to types, but only to find as many errors as possible without ever contradicting what happens in the real world. As the "Practical Type Inference Based on Success Typings" paper (*http://www.it.uu.se/research/group/hipe/papers/succ_types.pdf*) behind Dialyzer explains, a type checker for a language like Erlang should work without type declarations being there (although it accepts hints), should be simple and readable, should adapt to the language (and not the other way around), and complain only about type errors that would guarantee a crash.

> Our main goal is to make uncover [sic] the implicit type information in Erlang code and make it explicitly available in programs. Because of the sizes of typical Erlang applications, the type inference should be completely automatic and faithfully respect the operational semantics of the language. Moreover, it should impose no code rewrites of any kind. The reason for this is simple. Rewriting, often safety critical, applications consisting of hundreds of thousand lines of code just to satisfy a type inferencer is not an option which will enjoy much success. However, large software applications have to be maintained, and often not by their original authors. By automatically revealing the type information that is already present, we provide automatic documentation that can evolve together with the program and will not rot. We also think that it is important to achieve a balance between precision and readability. Last but not least, the inferred typings should never be wrong.

Dialyzer begins each analysis optimistically, assuming that all functions are good. It will see them as always succeeding, accepting anything, and possibly returning anything. No matter how an unknown function is used, it's a good way to use it. This is why warnings about unknown functions are not a big deal when generating PLTs. It's all good anyway; Dialyzer is a natural optimist when it comes to type inference.

As the analysis goes, Dialyzer gets to know your functions better and better. As it does so, it can analyze the code and see some interesting things.

Suppose that one of your functions has a + operator between both of its arguments and that it returns the value of the addition. Dialyzer no longer assumes that the function takes anything and returns anything, but will now expect the arguments to be numbers (either integers or floating-point values), and the returned values will similarly be numbers. This function will have a basic type associated with it saying that it accepts two numbers and returns a number.

Now let's say one of your functions calls the one described previously with an atom and a number. Dialyzer will think about the code and say, "Wait a minute, you can't use an atom and a number with the + operator!" It will then freak out because where the function could return a number before, it cannot return anything given how you use it.

In more general circumstances, though, you might find that Dialyzer keeps silent about many things that you know will *sometimes* cause an error. For example, consider the following snippet of code:

```
main() ->
    X = case fetch() of
        1 -> some_atom;
        2 -> 3.14
    end,
    convert(X).

convert(X) when is_atom(X) -> {atom, X}.
```

This bit of code assumes the existence of a `fetch/0` function that returns either 1 or 2. Based on this, we either return an atom or a floating-point number.

From our point of view, it appears that at some point in time, the call to `convert/1` will fail. We would likely expect a type error there whenever `fetch()` returns 2, which sends a floating-point value to `convert/1`. Dialyzer doesn't think so. Remember that Dialyzer is optimistic. It has figurative faith in your code, and because there is the possibility that the function call to `convert/1` succeeds at some point, Dialyzer will keep silent. No type error is reported in this case.

Type Inference and Discrepancies

For a practical example of the principles described in the previous section, let's try Dialyzer on a few modules: *discrep1.erl*, *discrep2.erl*, and *discrep3.erl*. Here's *discrep1.erl*:

```
-module(discrep1).
-export([run/0]).

run() -> some_op(5, you).

some_op(A, B) -> A + B.
```

The error in this example is kind of obvious. You can't add 5 to the `you` atom. We can try Dialyzer on that piece of code, assuming the PLT has been created:

```
$ dialyzer discrep1.erl
  Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis...
```

```
discrep1.erl:4: Function run/0 has no local return
discrep1.erl:4: The call discrep1:some_op(5,'you') will never return
since it differs in the 2nd argument from the success typing arguments:
(number(),number())
discrep1.erl:6: Function some_op/2 has no local return
discrep1.erl:6: The call erlang:'+'(A::5,B::'you') will never return
since it differs in the 2nd argument from the success typing arguments:
(number(),number())
  done in 0m0.62s
done (warnings were emitted)
```

Oh bloody fun—Dialyzer found stuff. What the hell does it mean?

The first one is an error you will see a lot when using Dialyzer. "Function Name/Arity has no local return" is the standard Dialyzer warning emitted whenever a function provably doesn't return anything (other than perhaps raising an exception) because one of the functions it calls happens to trip Dialyzer's type-error detector or raises an exception itself. When this happens, the set of possible types of values the function could return is empty; it doesn't actually return. This error propagates to the function that called it, giving us the "no local return" error.

The second error is somewhat more understandable. It says that calling `some_op(5, 'you')` breaks what Dialyzer detected would be the types required to make the function work, which are two numbers (`number()` and `number()`). Granted the notation is a bit foreign, but we'll explore it in more detail soon enough.

The third error is yet again a "no local return." The first one was because `some_op/2` would fail; this one is because the `+` call will fail. This is what the fourth and last error is about. The plus operator (actually the function `erlang:'+'/2`) can't add the number 5 to the atom `you`.

What about *discrep2.erl*? Here's what it looks like:

```
-module(discrep2).
-export([run/0]).

run() ->
    Tup = money(5, you),
    some_op(count(Tup), account(Tup)).

money(Num, Name) -> {give, Num, Name}.
count({give, Num, _}) -> Num.
account({give, _, X}) -> X.

some_op(A, B) -> A + B.
```

If you run Dialyzer on that file again, you'll get similar errors to those found in *discrep1.erl*:

```
$ dialyzer discrep2.erl
  Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis...
discrep2.erl:4: Function run/0 has no local return
```

```
discrep2.erl:6: The call discrep2:some_op(5,'you') will never return
since it differs in the 2nd argument from the success typing arguments:
(number(),number())
discrep2.erl:12: Function some_op/2 has no local return
discrep2.erl:12: The call erlang:'+'(A::5,B::'you') will never return
since it differs in the 2nd argument from the success typing arguments:
(number(),number())
 done in 0m0.69s
done (warnings were emitted)
```

During its analysis, Dialyzer can see the types right through the `count/1` and `account/1` functions. It infers the types of each of the elements of the tuple, and then figures out the values they pass. It can then find the errors again, without any problems.

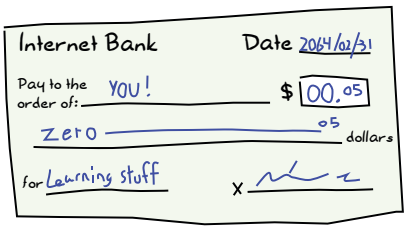
Let's push it a bit further, with *discrep3.erl*:

```
-module(discrep3).
-export([run/0]).

run() ->
    Tup = money(5, you),
    some_op(item(count, Tup), item(account, Tup)).

money(Num, Name) -> {give, Num, Name}.

item(count, {give, X, _}) -> X;
item(account, {give, _, X}) -> X.

some_op(A, B) -> A + B.
```

This version introduces a new level of indirection. Instead of having a function clearly defined for the `count` and the `account` values, this one works with atoms and switches to different function clauses. If we run Dialyzer on it, we get this:

```
$ dialyzer discrep3.erl
  Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis... done in 0m0.70s
done (passed successfully)
```

Uh-oh—somehow the new change to the file made things complex enough that Dialyzer got lost in our type definitions. The error is still there though. We'll get back to understanding why Dialyzer doesn't find the errors in this file and how to fix it in "Typing Functions" on page 556, but for now, there are a few more ways to run Dialyzer that we need to explore.



If we wanted to run Dialyzer over, say, our Process Quest release, we could do it as follows:

```
$ cd processquest/apps
$ ls
processquest-1.0.0  processquest-1.1.0  regis-1.0.0
regis-1.1.0  sockserv-1.0.0  sockserv-1.0.1
```

So, we have a bunch of libraries. Dialyzer wouldn't like it if we had many modules with the same names, so we'll need to specify directories manually:

```
$ dialyzer -r processquest-1.1.0/src regis-1.1.0/src sockserv-1.0.1/src
  Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis...
dialyzer: Analysis failed with error:
No .beam files to analyze (no --src specified?)
```

Oh, right—by default, Dialyzer will look for *.beam* files. We need to add the `--src` flag to tell Dialyzer to use *.erl* files for its analysis:

```
$ dialyzer -r processquest-1.1.0/src regis-1.1.0/src sockserv-1.0.1/src --src
  Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis... done in 0m2.32s
done (passed successfully)
```

Notice that we added the *src* directory to all requests. We could have done the same search without it, but then Dialyzer would have complained about a bunch of errors related to EUnit tests, based on how some of the assertion macros work with regard to the code analysis—we do not really care about these. Plus, if you sometimes test for failures and make your software crash on purpose inside of tests, Dialyzer will pick up on that, and you might not want it to.

Typing About Types of Types

As seen with *discrep3.erl*, Dialyzer will sometimes not be able to infer all the types in the way we intended. That's because Dialyzer cannot read our minds. To help Dialyzer in its task (and mostly help ourselves), we can declare types and annotate functions in order to both document them and help formalize the implicit expectations about types we put in our code.

Singleton Types

Erlang types can be as simple as, say, the number 42, noted `42` as a type (nothing different from usual), or specific atoms, such as `cat` or `molecule`. Those are called *singleton types*, as they refer to a value itself. Table 30-1 lists the singleton types.

Table 30-1: Erlang Singleton Types

Type	Description
`'some atom'`	Any atom can be its own singleton type
`42`	A given integer
`[]`	An empty list
`{}`	An empty tuple
`<<>>`	An empty binary

You can see that it could be annoying to program Erlang using only these types. There is no way to express things such as ages, much less "all the integers" for our programs, by using singleton types. And then, even if we had a way to specify many types at once, it would be irritating to express things such as "any integer" without writing them all by hand, which isn't possible anyway. Because of this, Erlang has union types and built-in types.

Union and Built-in Types

Union types allow you to describe more complex ideas, such as a type that has two atoms in it. *Built-in types* are predefined types, which are not necessarily possible to build by hand.

Union types and built-in types generally share a similar syntax, and they're noted with the form `TypeName()`. For example, the type for all possible integers would be noted as `integer()`. The reason parentheses are used is that they let us differentiate between, say the type `atom()` for all atoms, and `atom` for the specific `atom` atom. Moreover, to make code clearer, many Erlang programmers choose to quote all atoms in type declarations, giving us `'atom'` instead of `atom`. This makes it explicit that `'atom'` was meant to be a singleton type, rather than a built-in type where the programmer forgot the parentheses.

Table 30-2 lists the built-in types provided with the language. Note that they do not all have the same syntax as union types do. Some of them, like binaries and tuples, have a special syntax to make them friendlier to use.

Given the built-in types listed in the table, it becomes a bit easier to imagine how we would define types for our Erlang programs. Some of it is still missing though. Maybe things are too vague or not appropriate for our needs. Remember one of the `discrepN` modules' errors mentioning the type `number()`? That type is neither a singleton type nor a built-in type. It would be a union type, which means we could define it ourselves.

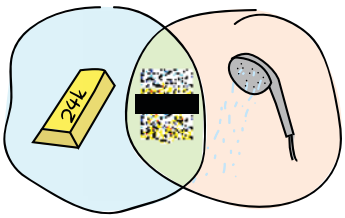


Table 30-2: Erlang Built-in Types

Type	Description
`any()`	Any Erlang term at all.
`none()`	A special type that means that no term or type is valid. Usually, when Dialyzer boils down the possible return values of a function to `none()`, it means the function should crash. It is synonymous with "this stuff won't work."
`pid()`	A process identifier.
`port()`	A port is the underlying representation of file descriptors (which we rarely see unless we dig deep inside the innards of Erlang libraries), sockets, or generally things that allow Erlang to communicate with the outside world, such as the `erlang:open_port/2` function. In the Erlang shell, they look like `#Port<0.638>`.
`reference()`	Unique values returned by `make_ref()` or `erlang:monitor/2`.
`atom()`	Atoms in general.
`binary()`	A blob of binary data.
`<<_:Integer>>`	A binary of a known size, where `Integer` is the size.
`<<_:_*Integer>>`	A binary that has a given unit size, but of unspecified length.
`<<_:Integer, _:_*OtherInteger>>`	A mix of both previous forms to specify that a binary can have a minimum length.
`integer()`	Any integer.
`N..M`	A range of integers. For example, if you wanted to represent a number of months in a year, you could define the range `1..12`. Note that Dialyzer reserves the right to expand this range into a bigger one.
`non_neg_integer()`	Integers that are greater or equal to 0.
`pos_integer()`	Integers greater than 0.
`neg_integer()`	Integers up to –1.
`float()`	Any floating-point number.
`fun()`	Any kind of function.
`fun(...) -> Type)`	An anonymous function of any arity that returns a given type. A given function that returns lists could be noted as `fun(...) -> list()`.
`fun(() -> Type)`	An anonymous function with no arguments, returning a term of a given type.
`fun((Type1, Type2, ..., TypeN) -> Type)`	An anonymous function taking a given number of arguments of a known type. For example, a function that handles an integer and a floating-point value could be declared as `fun((integer(), float()) -> any())`.

Table 30-2: Erlang Built-in Types *(continued)*

Type	Description
`[Type()]`	A list containing a given type. A list of integers could be defined as `[integer()]`. Alternatively, it can be written as `list(Type())`. Lists can sometimes be improper (like `[1, 2 | a]`). As such, Dialyzer has types declared for improper lists with `improper_list(TypeList, TypeEnd)`. The improper list `[1, 2 | a]` could be typed as `improper_list(integer(), atom())`, for example. Then, to make matters more complex, it is possible that you will not be sure whether or not the list will be proper. In such circumstances, the type `maybe_improper_list(TypeList, TypeEnd)` can be used.
`[Type(), ...]`	A special case of `[Type()]` that mentions that the list cannot be empty.
`tuple()`	Any tuple.
`{Type1, Type2, ..., TypeN}`	A tuple of a known size, with known types. For example, a binary tree node could be defined as `{'node', any(), any(), any(), any()}`, corresponding to `{'node', LeftTree, RightTree, Key, Value}`.

The notation to represent the union of types is the pipe (`|`). Basically, this lets us say that a given type *TypeName* is represented as the union of *Type1* | *Type2* | ... | *TypeN*. As such, the `number()` type, which includes integers and floating-point values, could be represented as `integer() | float()`. A Boolean value could be defined as `'true' | 'false'`. It is also possible to define types where only one other type is used. Although it looks like a union type, it is actually an *alias*.

In fact, many such aliases and type unions are predefined for you. Table 30-3 lists some of them.

Table 30-3: Predefined Union Types and Aliases

Type	Definition
`term()`	Equivalent to `any()`. It was added because other tools used `term()` before. Alternatively, the `_` variable can be used as an alias of both `term()` and `any()`.
`boolean()`	Defined as `'true' | 'false'`.
`byte()`	Defined as `0..255`. It's any valid byte in existence.
`char()`	Defined as `0..16#10ffff`, but it isn't clear whether this type refers to specific standards for characters. It's extremely general in its approach to avoid conflicts.
`number()`	`integer() | float()`
`maybe_improper_list()`	A quick alias for `maybe_improper_list(any(), any())` for improper lists in general.

(continued)

Table 30-3: Predefined Union Types and Aliases *(continued)*

Type	Definition
`maybe_improper_list(T)`	An alias for `maybe_improper_list(T, any())`, where `T` is any given type.
`string()`	Defined as `[char()]`, a list of characters. There is also `nonempty_string()`, defined as `[char(), ...]`. Sadly, there is so far no string type for binary strings only, but that's more because they're blobs of data that are to be interpreted in whatever type you choose.
`iolist()`	Defined as `maybe_improper_list(char() | binary() | iolist(), binary() | [])`. You can see that the iolist is itself defined in terms of iolists. Dialyzer does support recursive types, starting with R13B04. Before then, types like iolists could be defined only through some arduous gymnastics.
`module()`	A type that stands for module names, and is currently an alias of `atom()`.
`timeout()`	`non_neg_integer() | 'infinity'`, to represent the values accepted by the `after` part of a `receive` expression.
`node()`	An Erlang's node name, which is an atom.
`no_return()`	An alias of `none()` intended to be used in the return type of functions. It is particularly meant to annotate functions that loop (hopefully) forever, and thus never return.

Defining Types

Well, that makes a few types already. Table 30-2 mentions a type for a tree written as `{'node', any(), any(), any(), any()}`. Now let's see how we could declare it in a module. The syntax for type declaration in a module is as follows:

```
-type TypeName() :: TypeDefinition.
```

As such, our tree could be defined like this:

```
-type tree() :: {'node', tree(), tree(), any(), any()}.
```

Alternatively, we could define it with a special syntax that allows us to use variable names as type comments, like this:

```
-type tree() :: {'node', Left::tree(), Right::tree(), Key::any(), Value::any()}.
```

But that definition doesn't work, because it doesn't allow for a tree to be empty. A better tree definition can be built by thinking recursively, much as we did with our *tree.erl* module back in Chapter 5. In that module, an empty tree is defined as `{node, 'nil'}`. Whenever we hit such a node in a recursive function, we stop. A regular node (that is not empty) is noted as

{node, Key, Val, Left, Right}. Translating this into a type gives us a tree node of the following form:

```
-type tree() :: {'node', 'nil'}
              | {'node', Key::any(), Val::any(), Left::tree(), Right::tree()}.
```

That way, we have a tree that is either an empty node or a node with contents. Note that we could have used 'nil' instead of {'node', 'nil'}, and Dialyzer would have been fine with it. I just wanted to respect the way we had written our tree module.

There's another piece of Erlang code we might want to give types to, but that we haven't covered yet.

Types for Records

What about records? They have a somewhat convenient syntax to declare types. To see it, let's imagine a #user{} record. We want to store the user's name, some specific notes (to use our tree() type), the user's age, a list of friends, and a short biography.

```
-record(user, {name="" :: string(),
               notes :: tree(),
               age :: non_neg_integer(),
               friends=[] :: [#user{}],
               bio :: string() | binary()}).
```

The general record syntax for type declarations is Field :: Type, and if there's a default value to be given, it becomes Field = Default :: Type. In the #user{} record, we can see that the name needs to be a string, the notes must be a tree, and the age can be any integer from 0 to infinity (who knows how old people can get?).

An interesting field is friends. The [#user{}] type means that the user records can hold a list of zero or more other user records. It also tells us that a record can be used as a type by writing it as #RecordName{}. The last part tells us that the biography can be either a string or a binary.

Furthermore, to give a more uniform style to type declarations and definitions, people tend to add an alias such as -type Type() :: #Record{}.. We could also change the friends definition to use the user() type, ending up as follows:

```
-record(user, {name = "" :: string(),
               notes :: tree(),
               age :: non_neg_integer(),
               friends=[] :: [user()],
               bio :: string() | binary()}).

-type user() :: #user{}.
```

Here, we defined types for all fields of the record, but some of them have no default value. If we were to create a user record instance as `#user{age=5}`, there would be no type error. All record field definitions have an implicit `'undefined'` union added to them if no default value is provided for them. With earlier versions, the declaration would have caused type errors.

Typing Functions

While we could be defining types all day and night, filling files and files with them, and then printing the files, framing them, and feeling strongly accomplished, they won't be used automatically by Dialyzer's type inference engine. Dialyzer doesn't work from the types you declare to narrow down what is possible or impossible to execute.

Why the hell would we declare these types then? For documentation? Partially. There is an additional step to making Dialyzer understand the types we've declared. We need to pepper type signature declarations over all the functions we want augmented, bridging our type declarations with the functions inside modules.

So far, we have looked at things like "here is the syntax for this and that," but now it's time to get practical. A simple example of things needing to be typed could be card games. There are four suits: spades, clubs, hearts, and diamonds. Cards can then be numbered from 1 to 10 (where the ace is 1), and then be a Jack, Queen, or King.

Without Dialyzer, we would probably represent cards as `{Suit, CardValue}` so that we could have the ace of spades as `{spades, 1}`. Now, instead of just having this up in the air, we can define types to represent this:

```
-type suit() :: spades | clubs | hearts | diamonds.
-type value() :: 1..10 | j | q | k.
-type card() :: {suit(), value()}.
```

The `suit()` type is simply the union of the four atoms that can represent suits. The values can be any card from one to ten (`1..10`), or `j`, `q`, or `k` for the face cards. The `card()` type joins them together as a tuple.

These three types can now be used to represent cards in regular functions and give us some interesting guarantees. Take the following *cards.erl* module for example:

```
-module(cards).
-export([kind/1, main/0]).

-type suit() :: spades | clubs | hearts | diamonds.
-type value() :: 1..10 | j | q | k.
-type card() :: {suit(), value()}.
```

```
kind({_, A}) when A >= 1, A =< 10 -> number;
kind(_) -> face.

main() ->
    number = kind({spades, 7}),
    face   = kind({hearts, k}),
    number = kind({rubies, 4}),
    face   = kind({clubs, q}).
```

The `kind/1` function should return whether a card is a face card or a number card. You will notice that the suit is never checked. In the `main/0` function, you can see that the third call is made with the `rubies` suit, something we obviously didn't intend in our types, and likely not in the `kind/1` function:

```
$ erl
... <snip> ...
1> c(cards).
{ok,cards}
2> cards:main().
face
```

Everything works fine. That shouldn't be the case. Even running Dialyzer does not show any problems. Now let's add the following type signature to the `kind/1` function:

```
-spec kind(card()) -> 'face' | 'number'.
kind({_, A}) when A >= 1, A =< 10 -> number;
kind(_) -> face.
```

Then something more interesting will happen. But before we run Dialyzer, let's see how that works.

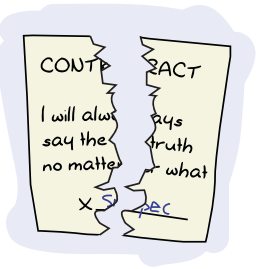
Type signatures are of the form `-spec` *`FunctionName`*`(`*`ArgumentTypes`*`) ->` *`ReturnTypes`*`.`. In the preceding specification, we say that the `kind/1` function accepts cards as arguments, according to the `card()` type we created. It also says the function returns either the atom `face` or `number`.

Running Dialyzer on the module yields the following:

```
$ dialyzer cards.erl
  Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis...
cards.erl:12: Function main/0 has no local return
cards.erl:15: The call cards:kind({'rubies',4}) breaks the contract (card()) -> 'face' | 'number'
 done in 0m0.80s
done (warnings were emitted)
```

Oh bloody fun. Calling `kind/1` with a "card" that has the `rubies` suit isn't valid according to our specifications.

In this case, Dialyzer respects the type signature we gave, and when it analyzes the `main/0` function, it figures out that there is a bad use of `kind/1` in there. This prompts the warning from line 15 (`number = kind({rubies, 4}),`). Dialyzer from there on assumes that the type signature is reliable, and that if the code is to be used according to that signature, it would logically not be valid. This breach in the contract propagates to the `main/0` function, but there isn't much that can be said at that level as to why it fails—just that it does.



NOTE *Dialyzer complained about this only once a type specification was defined. Before a type signature was added, Dialyzer couldn't assume that you planned to use* `kind/1` *only with* `card()` *arguments. With the signature in place, it can work with that as its own type definition.*

Here's a more interesting function to type, in *convert.erl*:

```
-module(convert).
-export([main/0]).

main() ->
    [_,_] = convert({a,b}),
    {_,_} = convert([a,b]),
    [_,_] = convert([a,b]),
    {_,_} = convert({a,b}).

convert(Tup) when is_tuple(Tup) -> tuple_to_list(Tup);
convert(L = [_|_]) -> list_to_tuple(L).
```

When reading the code, it is obvious that the two last calls to `convert/1` will fail. The function accepts a list and returns a tuple, or it accepts a tuple and returns a list. The two last calls to the function don't respect that, expecting a tuple from a tuple, and a list from a list. If we run Dialyzer on the code, though, it will find nothing.

That's because Dialyzer infers a type signature similar to the following:

```
-spec convert(list() | tuple()) -> list() | tuple().
```

Or to put it in words, the function accepts lists and tuples, and returns lists and tuples. This is true—sadly, a bit too true. The function isn't as permissive as the type signature would imply. This is one of the places where Dialyzer sits back and tries not to say too much without being 100 percent sure of the problems.

To help Dialyzer a bit, we can send in a fancier type declaration:

```
-spec convert(tuple()) -> list();
             (list()) -> tuple().
convert(Tup) when is_tuple(Tup) -> tuple_to_list(Tup);
convert(L = [_|_]) -> list_to_tuple(L).
```

Rather than putting `tuple()` and `list()` types together into a single union, this syntax allows us to declare type signatures with alternative clauses. If we call `convert/1` with a tuple, we expect a list, and the opposite in the other case.

With this more specific information, Dialyzer can now give more interesting results:

```
$ dialyzer convert.erl
  Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis...
convert.erl:4: Function main/0 has no local return
convert.erl:7: The pattern [_, _] can never match the type tuple()
 done in 0m0.90s
done (warnings were emitted)
```

Ah, this time it finds the error. Success! We can now use Dialyzer to tell us what we knew. Of course, putting it that way sounds useless, but when you type your functions correctly and make a tiny mistake that you forget to check, Dialyzer will have your back, which is definitely better than an error-logging system waking you up at night (or having your car keyed by your operations guy).

NOTE *Some people prefer the following syntax for a multiple-clause type signature:*

```
-spec convert(tuple()) -> list()
      ;      (list()) -> tuple().
```

This is the same as the syntax we used but puts the semicolon on another line because it might be more readable. There is no widely accepted standard at the time of this writing.

By using type definitions and specifications in this way, we're able to let Dialyzer find errors with our earlier `discrep` modules. Let's see how *discrep4.erl* does it:

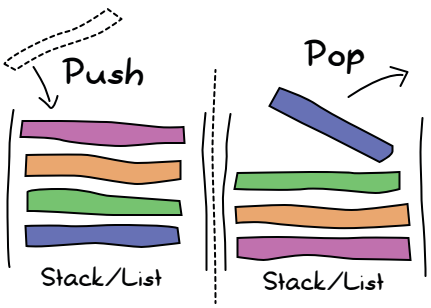
```
-module(discrep4).
-export([run/0]).
-type cents() :: integer().
-type account() :: atom().
-type transaction() :: {'give', cents(), account()}.
```

```
run() ->
    Tup = money(5, you),
    some_op(item(count,Tup), item(account,Tup)).

-spec money(cents(), account()) -> transaction().
money(Num, Name) -> {give, Num, Name}.

-spec item('count', transaction()) -> cents();
          ('account', transaction()) -> account().
item(count, {give, X, _}) -> X;
item(account, {give, _, X}) -> X.

some_op(A,B) -> A + B.
```

The especially useful definition here is about how `item/2` is typed using two alternative clauses. This will help Dialyzer to track the return values in relation to the input values and find type errors.

Typing Practice

Now we'll look at a queue module for FIFO operations. You should know what queues are, given Erlang's mailboxes are queues. The first element added will be the first one to be popped (unless we do selective receives). The module works like this:



To simulate a queue, we use two lists as stacks. One list stores the new elements, and one list lets us remove them from the queue. We always add to the same list and remove from the second one. When the list we remove from is empty, we reverse the list we add items to, and it becomes the new list to remove from. This generally guarantees better average performance than using a single list to do both tasks.

Here's our FIFO module, with a few type signatures added to check it with Dialyzer:

```
-module(fifo_types).
-export([new/0, push/2, pop/1, empty/1]).
-export([test/0]).
```

```
-spec new() -> {fifo, [], []}.
new() -> {fifo, [], []}.

-spec push({fifo, In::list(), Out::list()}, term()) -> {fifo, list(), list()}.
push({fifo, In, Out}, X) -> {fifo, [X|In], Out}.

-spec pop({fifo, In::list(), Out::list()}) -> {term(), {fifo, list(), list()}}.
pop({fifo, [], []}) -> erlang:error('empty fifo');
pop({fifo, In, []}) -> pop({fifo, [], lists:reverse(In)});
pop({fifo, In, [H|T]}) -> {H, {fifo, In, T}}.

-spec empty({fifo, [], []}) -> true;
           ({fifo, list(), list()}) -> false.
empty({fifo, [], []}) -> true;
empty({fifo, _, _}) -> false.

test() ->
    N = new(),
    {2, N2} = pop(push(push(new(), 2), 5)),
    {5, N3} = pop(N2),
    N = N3,
    true = empty(N3),
    false = empty(N2),
    pop({fifo, [a|b], [e]}).
```

This defines a queue as a tuple of the form `{fifo, list(), list()}`. You'll notice we didn't define a `fifo()` type, mostly to make it easy to create different clauses for empty queues and filled queues. The `empty(...)` type specification reflects that.

MUCH ADO ABOUT NONE()

You will notice that in the function `pop/1`, we do not specify the `none()` type, even though one of the function clauses calls `erlang:error/1`.

The type `none()` means a given function will not return. If the function might either fail or return a value, it is useless to type it as returning both a value and `none()`. The `none()` type is always assumed to be there, and as such, the union `Type() | none()` is the same as `Type()` alone.

`none()` is warranted whenever you're writing a function that always fails when called, such as if you were implementing `erlang:error/1` yourself.

All of our type specifications appear to make sense. Just to make sure, let's run Dialyzer and check the results:

```
$ dialyzer fifo_types.erl
  Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis...
```

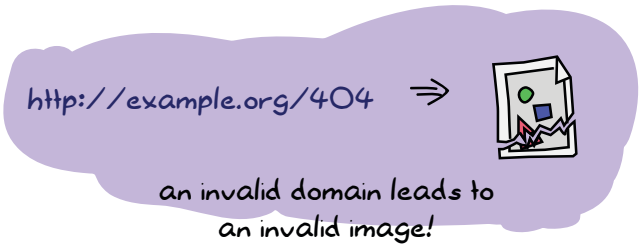
```
fifo_types.erl:16: Overloaded contract has overlapping domains; such contracts are currently
unsupported and are simply ignored
fifo_types.erl:21: Function test/0 has no local return
fifo_types.erl:28: The call
  fifo_types:pop({'fifo',nonempty_improper_list('a','b'),['e',...]})
   breaks the contract
   ({'fifo',In::[any()],Out::[any()]}) -> {term(),{'fifo',[any()],[any()]}}
 done in 0m0.96s
done (warnings were emitted)
```

So, we have a bunch of errors, and curses, they are not so easy to read. The second one, "Function test/0 has no local return," is at least something we know how to handle. If we just skip to the next one, it should disappear.

For now, let's focus on the first error—the one about contracts with overlapping domains. If we go into `fifo_types`, on line 16, we see this:

```
-spec empty({fifo, [], []}) -> true;
           ({fifo, list(), list()}) -> false.
empty({fifo, [], []}) -> true;
empty({fifo, _, _}) -> false.
```

So what are said overlapping domains? We need to refer to the mathematical concepts of *domain* and *image* (also *range*). To put it simply, the domain is the set of all possible input values to a function, and the image is the set of all possible output values of a function. "Overlapping domains" refers to two sets of input that overlap.



To find the source of the problem, we need to look at `list()`. `list()` is pretty much the same as `[any()]`, and both of these types also include empty lists. And there's your overlapping domain. When `list()` is specified as a type, it overlaps with `[]`. To fix this, we need to replace the type signature as follows:

```
-spec empty({fifo, [], []}) -> true;
           ({fifo, nonempty_list(), nonempty_list()}) -> false.
```

Alternatively, we could use this form:

```
-spec empty({fifo, [], []}) -> true;
           ({fifo, [any(), ...], [any(), ...]}) -> false.
```

Then running Dialyzer again will get rid of the warning.

Let's move on to the next error (which I broke into multiple lines):

```
fifo_types.erl:28:
The call fifo_types:pop({'fifo',nonempty_improper_list('a','b'),['e',...]})
breaks the contract
({'fifo',In::[any()],Out::[any()]}) -> {term(),{'fifo',[any()],[any()]}}
```

Translated into human, this means that on line 28, there's a call to `pop/1` that has inferred types breaking the one specified in the file:

```
pop({fifo, [a|b], [e]}).
```

That's the call. Now, the error message says that it identified an improper list (that happens to not be empty), which is entirely right—`[a|e]` is an improper list. It also mentions that it breaks a contract. We need to match the type definition that is broken between the following, coming from the error message:

```
{'fifo',nonempty_improper_list('a','b'),['e',...]}
{'fifo',In::[any()],Out::[any()]}
{term(),{'fifo',[any()],[any()]}}
```

The issue can be explained in one of three ways:

- The type signatures are right, the call is right, and the problem is the return value expected.
- The type signatures are right, the call is wrong, and the problem is the input value given.
- The call is right, but the type signatures are wrong.

We can eliminate the first one immediately. We're not actually doing anything with the return value. This leaves the second and third options. The decision boils down to whether or not we want improper lists to be used with our queues. This is a judgment call to be made by the writer of the library, and I can say without a doubt that I didn't intend improper lists to be used with this code. In fact, programmers very rarely want

improper lists. The winner is number 2: The call is wrong. To solve the problem, drop the call or fix it:

```
test() ->
    N = new(),
    {2, N2} = pop(push(push(new(), 2), 5)),
    ...
    pop({fifo, [a, b], [e]}).
```

And run Dialyzer again:

```
$ dialyzer fifo_types.erl
  Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis... done in 0m0.90s
done (passed successfully)
```

That makes more sense.

Exporting Types

Things have gone very well. We have types, we have signatures, and we have additional safety and verifications. So what would happen if we wanted to use our queue in another module? What about any other module we frequently use, such as `dict`, `gb_trees`, or ETS tables? How can we use Dialyzer to find type errors related to them?

We can use types coming from other modules. Doing so usually requires rummaging through documentation to find them. For example, the `ets` module's documentation (*http://www.erlang.org/doc/man/ets.html*) has the following entries under "DATA TYPES":

`continuation()`
Opaque continuation used by `select/1` and `select/3`.

`match_spec() = [{match_pattern(), [term()], [term()]}]`
A match specification, see above.

`match_pattern() = atom() | tuple()`
`tab() = atom() | tid()`
`tid()`
A table identifier, as returned by `new/2`.

Those are the data types exported by `ets`. If we had a type specification that accepted ETS tables and a key, and returned a matching entry, we could define it like this:

```
-spec match(ets:tab(), Key::any()) -> Entry::any().
```

And that's about it.

Exporting our own types works pretty much the same as with functions. All we need to do is add a module attribute of the form `-export_type([TypeName/Arity])`.. For example, we could have exported the `card()` type from our `cards` module by adding the following line:

```
-module(cards).
-export([kind/1, main/0]).

-type suit() :: spades | clubs | hearts | diamonds.
-type value() :: 1..10 | j | q | k.
-type card() :: {suit(), value()}.

-export_type([card/0]).
...
```

And from then on, if the module is visible to Dialyzer (either by adding it to the PLT or analyzing it at the same time as any other module), we can reference it from any other bit of code as `cards:card()` in type specifications.

Doing this will have one downside, though: Using a type like this doesn't forbid anyone using the `card` module from ripping the types apart and toying with them. Anyone could be writing pieces of code that match on the cards, a bit like `{Suit, _} = ...`. This isn't always a good idea, because it prevents us from being able to change the implementation of the `cards` module in the future. We would especially like to enforce this in modules that represent data structures, such as `dict` and `fifo_types` (if it exported types).

Dialyzer allows you to export types in a way that tells your users, "You know what? I'm fine with you using my types, but don't you dare look inside them!" It's a question of replacing a declaration like this:

```
-type fifo() :: {fifo, list(), list()}.
```

with this:

```
-opaque fifo() :: {fifo, list(), list()}.
```

Then you can still export it as `-export_type([fifo/0])`.

Declaring a type as `-opaque` means that only the module that defined the type has the right to look at how it's made and make modifications to it. This forbids other modules from pattern matching on the values other than the whole thing, guaranteeing (if they use Dialyzer) that they will never be bitten by a sudden change of implementation.

DON'T DRINK TOO MUCH KOOL-AID

Sometimes the implementation of opaque data types is either not strong enough to do what it should or is actually problematic (that is, buggy).

Dialyzer does not take the specification of a function into account until it has first inferred the success typing for the function.

This means that when your type looks rather generic without any -type information taken into account, Dialyzer might get confused by some opaque types. For example, when analyzing an opaque version of the `card()` data type, Dialyzer might see it as `{atom(), any()}` after inference. Modules using `card()` correctly might see Dialyzer complaining because they're breaking a type contract, even if they aren't. This is because the `card()` type itself doesn't contain enough information for Dialyzer to connect the dots and realize what's really going on.

Usually, if you see errors when using an opaque data type, tagging your tuple helps. Moving from a type of the form `-opaque card() :: {suit(), value()}.` to `-opaque card() :: {card, suit(), value()}.` might get Dialyzer to work fine with the opaque type.

The Dialyzer implementers are currently trying to make the implementation of opaque data types better and strengthen their inference. They are also trying to make user-provided specs more important and to trust them better during Dialyzer's analysis, but this is still a work in progress.

Typed Behaviors

Back in Chapter 14, we explored how to declare behaviors using the `behavior_info/1` function. The module exporting this function would give its name to the behavior, and a second module could implement callbacks by adding `-behavior(ModName).` as a module attribute.

The behavior definition of the `gen_server` module, for example, is as follows:

```
behavior_info(callbacks) ->
    [{init, 1}, {handle_call, 3}, {handle_cast, 2}, {handle_info, 2},
     {terminate, 2}, {code_change, 3}];
behavior_info(_Other) ->
    undefined.
```

The problem here is that there is no way for Dialyzer to check type definitions for that. In fact, there is no way for the behavior module to specify which kinds of types it expects the callback modules to implement, and thus there's no way for Dialyzer to do something about it.

Starting with R15B, the Erlang/OTP compiler was upgraded so that it now handles a new module attribute, named `-callback`. The `-callback` module attribute has a similar syntax to `spec`. When you specify function types with it, the `behavior_info/1` function will be declared automatically, and the

specifications are added to the module metadata in a way that lets Dialyzer do its work. For example, here's the declaration of the gen_server starting with R15B:

```
-callback init(Args :: term()) ->
    {ok, State :: term()} | {ok, State :: term(), timeout() | hibernate} |
    {stop, Reason :: term()} | ignore.
-callback handle_call(Request :: term(), From :: {pid(), Tag :: term()},
                      State :: term()) ->
    {reply, Reply :: term(), NewState :: term()} |
    {reply, Reply :: term(), NewState :: term(), timeout() | hibernate} |
    {noreply, NewState :: term()} |
    {noreply, NewState :: term(), timeout() | hibernate} |
    {stop, Reason :: term(), Reply :: term(), NewState :: term()} |
    {stop, Reason :: term(), NewState :: term()}.
-callback handle_cast(Request :: term(), State :: term()) ->
    {noreply, NewState :: term()} |
    {noreply, NewState :: term(), timeout() | hibernate} |
    {stop, Reason :: term(), NewState :: term()}.
-callback handle_info(Info :: timeout() | term(), State :: term()) ->
    {noreply, NewState :: term()} |
    {noreply, NewState :: term(), timeout() | hibernate} |
    {stop, Reason :: term(), NewState :: term()}.
-callback terminate(Reason :: (normal | shutdown | {shutdown, term()} | term()),
                              State :: term()) ->
    term().
-callback code_change(OldVsn :: (term() | {down, term()}), State :: term(),
                                Extra :: term()) ->
    {ok, NewState :: term()} | {error, Reason :: term()}.
```

And none of your code should break from the behavior changing things. Do realize, however, that a module cannot use both the `-callback` form and the `behavior_info/1` function at once—only one or the other. This means if you want to create custom behaviors, there is a rift between what can be used in versions of Erlang prior to R15 and what can be used in later versions.

The upside is that newer modules will have Dialyzer able to do some analysis to check for errors on the types of whatever is returned.

NOTE *During version R15B exclusively, Dialyzer would check types of callbacks only when the callback module included `-behaviour`, not when it included `-behavior`. This is a bug that was later resolved.*

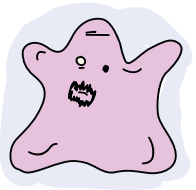
Polymorphic Types

Oh boy, what a section title. If you've never heard of *polymorphic types* (alternatively called *parameterized types*), this might sound a bit scary. It's fortunately not as complex as the name would lead you to believe.

The need for polymorphic types comes from the fact that when we're typing different data structures, we might find ourselves wanting to be

more specific about what they can store. For example, we may want our queue from earlier in the chapter to sometimes handle anything, sometimes handle only playing cards, or sometimes handle only integers. In the latter two cases, the issue is that we might want Dialyzer to be able to complain that we're trying to put floating-point numbers in our integer queue, or tarot cards in our playing cards queue.

This is impossible to enforce by strictly using types the way we have been so far.



A polymorphic type is a type that can be "configured" with other types. Luckily for us, we already know the syntax to use it. Earlier, I said we could define a list of integers as `[integer()]` or `list(integer())`—these are polymorphic types. A polymorphic type accepts a type as an argument.

To make our queue accept only integers or cards, we could have defined its type as follows:

```
-type queue(Type) :: {fifo, list(Type), list(Type)}.
-export_type([queue/1]).
```

When another module wishes to make use of the `fifo/1` type, it needs to parameterize that type. So a new deck of cards in the `cards` module could have had the following signature:

```
-spec new() -> fifo:queue(card()).
```

And Dialyzer would try to analyze the module to make sure that it submits and expects only cards from the queue it handles.

We Bought a Zoo

As a demonstration of the use of polymorphic type, let's say that we decided to buy a zoo to congratulate ourselves for being nearly finished with *Learn You Some Erlang*. In our zoo, we have two animals: a red panda and a squid (yes, it is a rather modest zoo, although that shouldn't keep us from setting the entry fee sky-high).

We've decided to automate the feeding of our animals, because we're programmers, and programmers like to automate stuff, often out of laziness. After doing a bit of research, we've found that red pandas can eat bamboo, some birds, eggs, and berries. We've also found that squids can fight with sperm whales, so we decided to feed them just that with our *zoo.erl* module:

```
-module(zoo).
-export([main/0]).

feeder(red_panda) ->
    fun() ->
        element(random:uniform(4), {bamboo, birds, eggs, berries})
    end;
```

```
feeder(squid) ->
    fun() -> sperm_whale end.

feed_red_panda(Generator) ->
    Food = Generator(),
    io:format("feeding ~p to the red panda~n", [Food]),
    Food.

feed_squid(Generator) ->
    Food = Generator(),
    io:format("throwing ~p in the squid's aquarium~n", [Food]),
    Food.

main() ->
    %% Random seeding
    <<A:32, B:32, C:32>> = crypto:rand_bytes(12),
    random:seed(A, B, C),
    %% The zoo buys a feeder for both the red panda and squid.
    FeederRP = feeder(red_panda),
    FeederSquid = feeder(squid),
    %% Time to feed them!
    %% This should not be right!
    feed_squid(FeederRP),
    feed_red_panda(FeederSquid).
```

This code makes use of `feeder/1`, which takes an animal name and returns a feeder (a function that returns food items). Feeding the red panda should be done with a red panda feeder, and feeding the squid should be done with a squid feeder. With function definitions such as `feed_red_panda/1` and `feed_squid/1`, there is no way to be alerted by the wrong use of a feeder. Even with runtime checks, it's impossible to do. As soon as we serve food, it's too late:

```
1> zoo:main().
throwing bamboo in the squid's aquarium
feeding sperm_whale to the red panda
sperm_whale
```

Oh no, our animals are not meant to eat that way! Maybe types can help. The following type specifications could be devised, using the power of polymorphic types:

```
-type red_panda() :: bamboo | birds | eggs | berries.
-type squid() :: sperm_whale.
-type food(A) :: fun(() -> A).

-spec feeder(red_panda) -> food(red_panda());
            (squid) -> food(squid()).
-spec feed_red_panda(food(red_panda())) -> red_panda().
-spec feed_squid(food(squid())) -> squid().
```

The food(A) type is the one of interest here. A is a free type, to be decided later. We then qualify the food type in feeder/1's type specification by declaring food(red_panda()) and food(squid()). The food type is then seen as fun(() -> red_panda()) and fun(() -> squid()), instead of some abstract function returning something unknown. If we add these specs to the file, and then run Dialyzer on it, the following happens:

```
$ dialyzer zoo.erl
  Checking whether the PLT /Users/ferd/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis...
zoo.erl:18: Function feed_red_panda/1 will never be called
zoo.erl:23: The contract zoo:feed_squid(food(squid())) -> squid() cannot be right because
 the inferred return for feed_squid(FeederRP::fun(() -> 'bamboo' | 'berries' | 'birds' |
'eggs'))
 on line 44 is 'bamboo' | 'berries' | 'birds' | 'eggs'
zoo.erl:29: Function main/0 has no local return
 done in 0m0.68s
done (warnings were emitted)
```

And the error is right. Hooray for polymorphic types!

Some Cautions

Although our example is pretty useful, minor changes in the code can have unexpected consequences in what Dialyzer is able to find. For example, suppose the main/0 function had the following code:

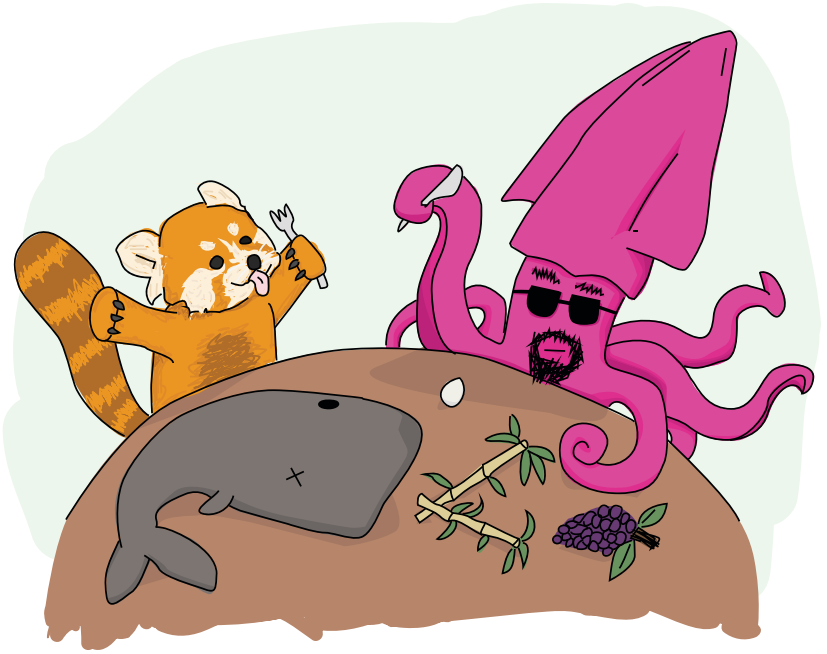
```
main() ->
    %% Random seeding
    <<A:32, B:32, C:32>> = crypto:rand_bytes(12),
    random:seed(A, B, C),
    %% The zoo buys a feeder for both the red panda and squid.
    FeederRP = feeder(red_panda),
    FeederSquid = feeder(squid),
    %% Time to feed them!
    feed_squid(FeederSquid),
    feed_red_panda(FeederRP),
    %% This should not be right!
    feed_squid(FeederRP),
    feed_red_panda(FeederSquid).
```

Things would not be the same. Before the functions are called with the wrong kind of feeder, they're first called with the right kind. As of R15B01, Dialyzer would not find an error with this code. The observed behavior is that as soon as a call to a given function succeeds within the function's body, Dialyzer will ignore later errors within the same unit of code.

Even if this is a bit sad for many static typing fans, we have been thoroughly warned. The following quote comes from the "Practical Type Inference Based on Success Typings" paper referenced earlier in the chapter:

> . . . A success typing is a type signature that over-approximates the set of types for which the function can evaluate to a value. The domain of the signature includes all possible values that the function could accept as parameters, and its range includes all possible return values for this domain. . . .
>
> . . . However weak this might seem to aficionados of static typing, success typings have the property that they capture the fact that if the function is used in a way not allowed by its success typing (e.g., by applying the function with parameters $\bar{p} \not\in \bar{\alpha}$) this application will definitely fail. This is precisely the property that a defect detection tool which never "cries wolf" needs. Also, success typings can be used for automatic program documentation because they will never fail to capture some possible—no matter how unintended—use of a function.

Again, keeping in mind that Dialyzer is optimistic in its approach is vital to working efficiently with it.



Who cares about food poisoning when supervisors have your back?

You're My Type

Dialyzer will often prove to be a true friend when programming in Erlang, although the frequent nagging might tempt you to just drop it. One thing to remember is that Dialyzer is practically never wrong, and you will likely make mistakes now and then. You might feel like some errors mean nothing, but contrary to many type systems, Dialyzer speaks out only when it knows it's right, and bugs in its code base are rare. Dialyzer might frustrate you and force you to be humble, but it is very unlikely to be the source of bad, unclean code.

That's All, Folks

So hey, that's about it for *Learn You Some Erlang for Great Good!* Thanks for reading it. There's not much more to say, but if you feel like getting a list of more topics to explore and some general words from me, you can read the book's afterword.

AFTERWORD

I see you chose to read the afterword after all. Good for you. Before I point you to a bunch of interesting topics that you might want to explore if you've decided Erlang is a development language that you want to learn more about, I would like a moment to talk about writing *Learn You Some Erlang*. It has been one hell of a ride. It took me three years of hard labor while studying and working full time, and juggling everyday life needs (if I had children, they would have died of neglect a while ago).

This book's site, coupled with some luck and some more work, allowed me to get jobs as an Erlang trainer, a course material writer, and a developer. It allowed me to travel around the world and meet a load of interesting people. It drained a lot of energy, and cost me a decent chunk of money and time, but it paid me back tenfold in most ways imaginable.

Then this book became reality, with more work needed. Even though I thanked these people in the thanks section of the book, I want to stress my appreciation of the Erlang community in general. They helped me learn stuff, reviewed pages and pages of material for free, fixed my typos, and helped me get better at writing English and writing in general. I also want to thank the whole team at No Starch Press, who put in even more time, bringing professional editing to *Learn You Some Erlang*. Finally, thanks again to Jenn (my girlfriend), who took the time to re-trace all my drawings so they would be suitable for print.

Other Erlang Applications

There are only so many topics I could cover without going over the top. This book is already large enough as it is. It has taken years to complete, and I'm tired and glad it's over (what am I gonna do with all that free time?), but there are still plenty of other topics I would have loved to include. Here's a quick list of applications you can look up in the documentation that ships with Erlang:

Tracing BIFs and DBG
: The Erlang VM is traceable inside and out. Got a bug or some stack trace you can't make sense of? Turn on a few trace flags, and the VM opens up to you. DBG, an application that comes with Erlang, takes these BIFs and builds an app on top of them. Messages, function calls, function returns, garbage collections, process spawning and dying, and so on are all traceable and observable. DBG also tends to work much better than any debugger for a concurrent language like Erlang. What's the best part about it? It's traceable within Erlang, so you can make Erlang programs that trace themselves! If you look into these functions and find them a bit hard to digest, you might be okay staying with the `sys` module's tracing functions. They work only on OTP behaviorized processes, but they're often good enough to get going.

Profiling
: Erlang comes with a bunch of different profiling tools to analyze your programs and find all kinds of bottlenecks. The `fprof` and `eprof` tools can be used for time profiling, `cprof` for function calls, `lcnt` for locks, `percept` for concurrency, and `cover` for code coverage. Most of them are built using the tracing BIFs of the language, funnily enough.

More introspection
: Unix `top`-like tools exist for Erlang, such as `etop`, part of the `observer` application. You can also use the Erlang debugger, but I recommend DBG instead of that one. The `observer` application also allows you to explore entire supervision trees for your nodes.

Documentation
: EDoc is a tool that lets you turn your Erlang modules into HTML documentation. It supports annotations and ways to declare specific pages that allow you to build small websites to document your code. It's similar to Javadoc for Java users.

GUIs
: The `wx` application is the new standard for multiplatform GUI writing with Erlang. I'm terrible at GUI stuff, so it's probably better for everyone that I didn't cover this app.

Erlang libraries
: Plenty of nice libraries come by default with Erlang: cryptography tools, web servers, web clients, all kinds of protocol implementations, and so on. You can get a general list of them at *http://www.erlang.org/doc/applications.html*.

Community Libraries

There are a ton of libraries coming from the Erlang community. I didn't cover them because they tend to change, and I didn't want to favor one over the other. Here's a quick list (links don't carry over very well in a book, so you can get the actual links at *http://learnyousomeerlang.com/conclusion*):

- Rebar and Sinan if you want to build systems
- Redbug for a friendlier approach to tracing
- Gproc for a very powerful and flexible process registry
- Mochiweb, Cowboy, and Yaws if you need web servers
- riak_core for a very powerful distribution library for Erlang
- lhttpc as a web client
- PropEr, QuickCheck, and Triq for kick-ass, property-based testing tools (you need to try one of them)
- Entop for a `top`-like tool
- A billion JSON libraries (mochijson2, jsx, ejson, and more)
- UX for Unicode handling and common Unicode-related algorithms pending their addition to the language (planned for R16B)
- Seresye and eXAT for some artificial intelligence (AI) tools
- Database client libraries
- Lager as a robust logging system that binds itself to Erlang's error logger
- Poolboy for some generic message-based pools

Many more libraries are out there. Community libraries could easily fill their own book.

Your Ideas Are Intriguing to Me and I Wish to Subscribe to Your Newsletter

I have a blog at *http://ferd.ca* where I discuss all kinds of stuff (or at least I want to) but inevitably come back to Erlang topics, due to using it all the time.

Is That It?

No, there's still an appendix and the index!

ON ERLANG'S SYNTAX

Many newcomers to Erlang manage to understand the syntax and program around it without ever getting comfortable with it. I've read and heard many complaints regarding the syntax and the "ant turd tokens" (a subjectively funny way to refer to ,, ;, and .), how annoying it is, and so on.

Erlang draws its syntax from Prolog. While this gives a reason for the current state of things, it doesn't magically make people like the syntax. I don't expect anyone to respond to this by saying, "Oh, it's Prolog, I get it. Makes complete sense!" As such, I'll suggest three ways to read Erlang code to possibly make it easier to understand.

The Template

The template way is my personal favorite. To understand it, you must first get rid of the concept of lines of code and think in expressions. An *expression* is any bit of Erlang code that returns something. In the shell, the

period (.) ends an expression. After writing `2 + 2`, you must add a period (and then press ENTER) to run the expression and return a value.

In modules, the period ends forms. *Forms* are module attributes and function declarations. Forms are not expressions, as they don't return anything. This is why they're terminated in a different manner than everything else. Given forms are not expressions, it could be argued that the shell's use of `.` to terminate expression is what is *not* standard here. Consequently, I suggest not caring about the shell for this method of reading Erlang.

The first rule is that the comma (`,`) separates expressions:

```
C = A+B, D = A+C
```

This is easy enough. However, it should be noted that `if ... end`, `case ... of ... end`, `begin ... end`, `fun() -> ... end`, and `try ... of ... catch ... end` are all expressions. As an example, it is possible to run this:

```
Var = if X > 0  -> valid;
         X =< 0 -> invalid
      end
```

And you'll get a single value out of the `if ... end`. This explains why we will sometimes see such language constructs followed by a comma. It just means there is another expression to evaluate after it.

The second rule is that the semicolon (`;`) has two roles. The first one is separating different function clauses:

```
fac(0) -> 1;
fac(N) -> N * fac(N-1).
```

The second one is separating different branches of expressions like `if ... end`, `case ... of ... end`, and others:

```
if X < 0  -> negative;
   X > 0  -> positive;
   X == 0 -> zero
end
```

It's probably the most confusing role because the last branch of the expression doesn't need to have the semicolon following it. This is because the `;` separates branches; it doesn't terminate them. Think in expressions, not lines. Some people find it easier to illustrate the role of a separator by writing the preceding expression in the following way, which is arguably more readable:

```
if X < 0  -> negative
 ; X > 0  -> positive
 ; X == 0 -> zero
end
```

This makes the role of separator more explicit. It goes in between branches and clauses, not after them.

Now, because the semicolon is used to separate expression branches and function clauses, it becomes possible to have an expression such as a `case` construct followed by `,` when followed by another expression, a `;` when in the last position of a function clause, or a `.` when at the last position of a function.

The line-based logic for terminating lines such as in C or Java must go out the window. Instead, see your code as a generic template you fill (hence the name the *template* way):

```
head1(Args) [Guard] ->
    Expression1, Expression2, ..., ExpressionN;
head2(Args) [Guard] ->
    Expression1, Expression2, ..., ExpressionN;
headN(Args) [Guard] ->
    Expression1, Expression2, ..., ExpressionN.
```

The rules make sense, but you need to get into a different reading mode. That's where the heavy lifting needs to be done—moving from lines and blocks toward a predefined template. If you think about it, things like `for (int i = 0; i >= x; i++) { ... }` (or even `for (...);`) have a weird syntax when compared to most other constructs in languages supporting them. We're just so used to seeing these constructs that we don't mind them anymore.

The English Sentence

The English sentence approach is about comparing Erlang code to English. Although this manner is not the one I prefer, I do realize that people have different ways to make sense of logical concepts, and this is one approach I've heard being praised many times.

Imagine you're writing a list of things. Well, no. Don't imagine it, read it.

```
I will need a few items on my trip:
  if it's sunny, sunscreen, water, a hat;
  if it's rainy, an umbrella, a raincoat;
  if it's windy, a kite, a shirt.
```

An Erlang translation can remain a bit similar:

```
trip_items(sunny) ->
    sunscreen, water, hat;
trip_items(rainy) ->
    umbrella, raincoat;
trip_items(windy) ->
    kite, shirt.
```

Here, just replace the items by expressions, and you have it. Expressions such as `if ... end` can be seen as nested lists.

And, Or, Done.

Another variant of the English sentence approach has been suggested to me on *#erlang*, and I think it's the most elegant one. The user simply reads the ant turd tokens as follows:

- `,` as *and*
- `;` as *or*
- `.` as *being done*

A function declaration can then be read as a series of nested logical statements and affirmations.

In Conclusion

Some people will just never like ant turd tokens or being unable to swap lines of code without changing the token at the end of the line. I guess there's not much to be done when it comes to style and preferences, but I still hope this appendix was useful. After all, the syntax is only intimidating; it's far from difficult.

INDEX

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

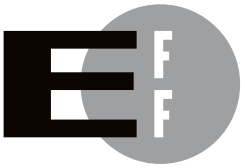
U

V

W

X

Z



The Electronic Frontier Foundation (EFF) is the leading organization defending civil liberties in the digital world. We defend free speech on the Internet, fight illegal surveillance, promote the rights of innovators to develop new digital technologies, and work to ensure that the rights and freedoms we enjoy are enhanced — rather than eroded — as our use of technology grows.

PRIVACY EFF has sued telecom giant AT&T for giving the NSA unfettered access to the private communications of millions of their customers. eff.org/nsa

FREE SPEECH EFF's Coders' Rights Project is defending the rights of programmers and security researchers to publish their findings without fear of legal challenges. eff.org/freespeech

INNOVATION EFF's Patent Busting Project challenges overbroad patents that threaten technological innovation. eff.org/patent

FAIR USE EFF is fighting prohibitive standards that would take away your right to receive and use over-the-air television broadcasts any way you choose. eff.org/IP/fairuse

TRANSPARENCY EFF has developed the Switzerland Network Testing Tool to give individuals the tools to test for covert traffic filtering. eff.org/transparency

INTERNATIONAL EFF is working to ensure that international treaties do not restrict our free speech, privacy or digital consumer rights. eff.org/global

EFF.ORG

ELECTRONIC FRONTIER FOUNDATION

Protecting Rights and Promoting Freedom on the Electronic Frontier

EFF is a member-supported organization. Join Now! www.eff.org/support

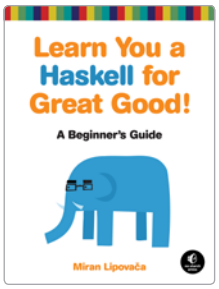
Learn You Some Erlang for Great Good! is set in New Baskerville, TheSansMono Condensed, Futura, and Dogma.

This book was printed and bound at Edwards Brothers Malloy in Ann Arbor, Michigan. The paper is 60# Williamsburg Smooth, which is certified by the Sustainable Forestry Initiative (SFI). The book uses a RepKover binding, which allows it to lie flat when open.

UPDATES

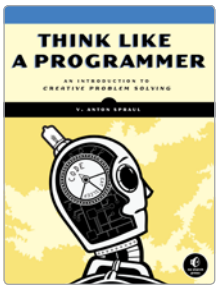
Visit *http://nostarch.com/erlang/* for updates, errata, and other information.

More no-nonsense books from NO STARCH PRESS

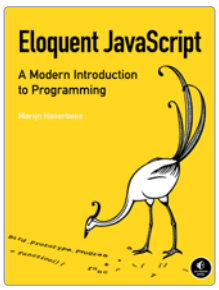


LEARN YOU A HASKELL FOR GREAT GOOD!
A Beginner's Guide
by MIRAN LIPOVAČA
APRIL 2011, 400 PP., $44.95
ISBN 978-1-59327-283-8

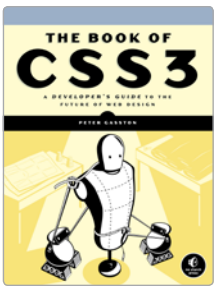
LAND OF LISP
Learn to Program in Lisp, One Game at a Time!
by CONRAD BARSKI, M.D.
OCTOBER 2010, 504 PP., $49.95
ISBN 978-1-59327-281-4



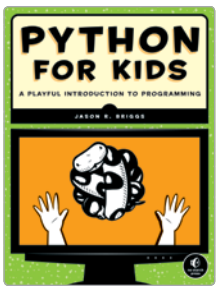
THINK LIKE A PROGRAMMER
An Introduction to Creative Problem Solving
by V. ANTON SPRAUL
AUGUST 2012, 256 PP., $34.95
ISBN 978-1-59327-424-5



ELOQUENT JAVASCRIPT
A Modern Introduction to Programming
by MARIJN HAVERBEKE
JANUARY 2011, 224 PP., $29.95
ISBN 978-1-59327-282-1



THE BOOK OF CSS3
A Developer's Guide to the Future of Web Design
by PETER GASSTON
MAY 2011, 304 PP., $34.95
ISBN 978-1-59327-286-9



PYTHON FOR KIDS
A Playful Introduction to Programming
by JASON R. BRIGGS
DECEMBER 2012, 344 PP., $34.95
ISBN 978-1-59327-407-8
full color

PHONE:
800.420.7240 OR
415.863.9900

EMAIL:
SALES@NOSTARCH.COM
WEB:
WWW.NOSTARCH.COM

Erlang to the People!

Erlang is the language of choice for programmers who want to write robust, concurrent applications, but its strange syntax and functional design can intimidate the uninitiated. Luckily, there's a new weapon in the battle against Erlang-phobia: *Learn You Some Erlang for Great Good!*

Erlang maestro Fred Hébert starts slow and eases you into the basics: You'll learn about Erlang's unorthodox syntax, its data structures, its type system (or lack thereof!), and basic functional programming techniques. Once you've wrapped your head around the simple stuff, you'll tackle the real meat-and-potatoes of the language: concurrency, distributed computing, hot code loading, and all the other dark magic that makes Erlang such a hot topic among today's savvy developers.

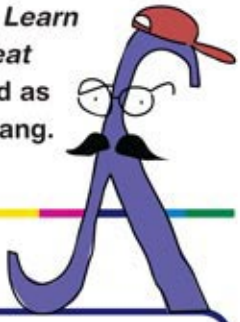
As you dive into Erlang's functional fantasy world, you'll learn about:

- Testing your applications with EUnit and Common Test
- Building and releasing your applications with the OTP framework
- Passing messages, raising errors, and starting/stopping processes over many nodes
- Storing and retrieving data using Mnesia and ETS
- Network programming with TCP, UDP, and the inet module
- The simple joys and potential pitfalls of writing distributed, concurrent applications

Packed with lighthearted illustrations and just the right mix of offbeat and practical example programs, *Learn You Some Erlang for Great Good!* is the perfect entry point into the sometimes-crazy, always-thrilling world of Erlang.

About the Author

Fred Hébert is a self-taught programmer who used to teach Erlang. He is currently working on a real-time bidding platform and was named Erlang User of the Year 2012. His online tutorial, *Learn You Some Erlang for Great Good!*, is widely regarded as the best way to learn Erlang.



no starch press
THE FINEST IN GEEK ENTERTAINMENT™
www.nostarch.com

PRICE: $49.95 ($52.95 CDN)
SHELVE IN:
PROGRAMMING LANGUAGES/ERLANG

ISBN: 978-1-59327-435-1
54995
9 781593 274351
6 89145 74351 7