Typing Erlang

John Hughes
Karol Ostrovsky

Erlang workshop 2004
Wanted

• A type-checker for Erlang

• Usable on existing code without too much effort
Hasn’t it already been done?

- Marlow and Wadler, ICFP 1997
  - Type-inference for Erlang
  - No need for any programmer annotations
  - Discovered recursive datatypes automatically
  - Subtyping and ”lacks” predicates to handle multiple return types

```prolog
lookup(Tree, Key) = Value | fail
lookup:: A lacks fail => tree(A) -> A | fail
```

A must not include fail, to avoid confusion
But...

- The types inferred were large
- Type inference was slow
- Type errors were hard to understand

So no-one uses it!
I’m not telling you anything about my code! Figure it out for yourself!
An Analogy

Your blurilewurile is boomziwacked
An Analogy

Why don’t you tell me what’s wrong in *my* terms?
An Alternative

Here’s how my program works

I think you made a mistake just here
Hindley-Milner Typing (ML, Haskell...) 

These are the datatypes I’m using 
Then these are the types of your functions

A reasonably small burden of annotations
An Indirect Benefit of Marlow and Wadler

- Multiple return types are now often avoided
  - Compare
    
    \[
    \text{lookup(Tree,Key)} = \text{Value} \mid \text{fail} \\
    \text{lookup :: A lacks fail} \Rightarrow \text{tree(A)} - \rightarrow A \mid \text{fail}
    \]
  
  - With
    
    \[
    \text{keysearch(Key, N, TupleList)} - \rightarrow \\
    \{\text{value, tuple()}\} \mid \text{false}
    \]

- So the type-checking problem is now easier!
Plan for an Erlang typechecker

- prog.erl
- prog.type
- prog.beam

Types stored in the BEAM file used to typecheck other modules
Erlang Datatype Declarations

-\texttt{data(maybe(A) = \{value,A\} | false)}.

- Atoms can be declared to belong to a new datatype
- So can tuples tagged with an atom at the front
- Atoms can be used with several arities
  
  -\texttt{data(as(A) = \{a, A\} | \{a\} | a)}.
A Problem

• Atoms can be used in more than one type!

- \texttt{data(maybe(A) = \{value,A\} | false)}.

- \texttt{data(bool() = true | false)}.

What is the type of false?
A Solution

• false is *overloaded* – must be resolved when a whole function is typechecked.

• Function types can be stated if necessary to resolve overloading.

```c
-type(odd(integer()) -> bool()).
```
Inspiration from Haskell

- Function types are inferred *when possible*

- Stating function types enables a *more powerful* type system!
  
  - Type *checking* is easier than type *inference*
Look up Revisited

- Functions like `lookup` have not disappeared altogether

```plaintext
lookup(Key,[ ]) -> false;
lookup(Key, [{Key,Value} | Rest]) -> Value;
lookup(Key,[ _ | Rest]) -> lookup(Key,Rest).
```

- Inferred type

```plaintext
lookup(K,[{K,bool()})] -> bool()
```
Lookup with a Type Declaration

-\text{type}(\text{lookup}(K,\{(K,V)\}) \rightarrow V \mid \text{bool}())

lookup(Key, [ ]) \rightarrow \text{false};
lookup(Key, [{Key,Value} \mid \text{Rest}]) \rightarrow \text{Value};
lookup(Key, [ _ \mid \text{Rest}]) \rightarrow \text{lookup}(Key,\text{Rest}).

- Easy to check these are in $V \mid \text{bool}()$.  

- Cf. Bidirectional type checking
Refining Case Analysis

Normally an argument has the *same* type in each case

Cf. "learning by testing" in languages with dependent types
Some Problems

- **lists:keysearch(e,2,[{a,b,4},{d,e,5}]).**
  - Returns `{value, {d,e,5}}`
  - The 2 specifies which tuple component is the key field.
  - Type of the key depends on the value 2!

- **list_to_tuple([a,b,c]).**
  - Returns `{a,b,c}`
  - The type of the result depends on the value of the argument
Some More Problems

- **apply**(lists,append,[[1,2],[3,4]]).
  - Returns [1,2,3,4]
  - The module and function name are *atoms*!
  - The argument list must have the right length.
  - The list elements may have different types!

- **spawn**(lists,append,[[1,2],[3,4]]).
  - Used to start every Erlang process!
OTP Behaviours

- `gen_server:start_link({local, ch3}, ch3, [], [])`
  - `ch3` names a call-back module, which must export `init`, `handle_call` etc.
  - Callback functions invoked via `apply` must have types which make the `gen_server` well-typed.
Supervisors in OTP

start_link() ->
    supervisor:start_link(ch_sup, []).
init(_Args) ->
    {ok, [{one_for_one, 1, 60},
           [{ch3, {ch3, start_link, []}, permanent,
             brutal_kill, worker, [ch3]}]}].

Initial call to start the child

Parameter module
Applications in OTP

- Type checker needs to know the contents of the application resource file

```
{application, ch_app,
    [{description, "Channel allocator"}],
    {vsn, "1"},
    {modules, [ch_app, ch_sup, ch3]},
    {registered, [ch3]},
    {applications, [kernel, stdlib, sasl]},
    {mod, {ch_app,[]} }} ]}.
```

Started by calling
```
ch_app:start(normal,[])```

Dependent Types

- Types in Erlang depend on values \((\text{dependent types})\)

- Values aren’t known until run-time!

- Bad news for type checking!
Observation

- The initial call is usually constant

```prolog
start_link() ->
    supervisor:start_link(ch_sup, []).
init(_Args) ->
    {ok, [{'one_for_one', 1, 60},
        [{ch3, {ch3, start_link, []}, permanent, brutal_kill, worker, [ch3]}
        ]}
```
Observation

lists:keysearch(e,2,[\{a,b,4\},\{d,e,5\}]).

• The position of the key is usually constant
Observation

spawn(lists,append,[[1,2],[3,4]]).

• The module and function are often not constant – but they are constants passed from elsewhere!
How can you write a correct program,

if you don’t know the values
of the ”dependent” parameters?
Our idea

• Combine *partial evaluation* and type inference
Partial Evaluation

\[
\text{power}(0, X) \rightarrow 1;
\]
\[
\text{power}(N, X) \text{ when } N > 0 \rightarrow X \times \text{power}(N-1, X).
\]

... power(3, Y+Z) ...

Known ("static")
Partial Evaluation

\[
\text{power}(0, X) \rightarrow 1; \\
\text{power}(N, X) \text{ when } N > 0 \rightarrow X \times \text{power}(N - 1, X).
\]

\[
... \text{power}(3, Y + Z) ...
\]

\[
\text{power3}(X) \rightarrow X \times \text{power}(2, X).
\]

\[
... \text{power3}(Y + Z) ...
\]
Partial Evaluation

\[
\text{power}(0,X) \rightarrow 1;
\]
\[
\text{power}(N,X) \text{ when } N > 0 \rightarrow X \times \text{power}(N-1,X).
\]

\[
\ldots \text{power}(3,Y+Z) \ldots
\]

\[
\text{power3}(X) \rightarrow X \times \text{power2}(X).
\]
\[
\text{power2}(X) \rightarrow X \times \text{power}(1,X).
\]

\[
\ldots \text{power3}(Y+Z) \ldots
\]
Partial Evaluation

\[
\text{power}(0, X) \rightarrow 1; \\
\text{power}(N, X) \text{ when } N > 0 \rightarrow X \times \text{power}(N-1, X).
\]

\[
\ldots \text{power}(3, Y+Z) \ldots
\]

\[
\text{power3}(X) \rightarrow X \times \text{power2}(X). \\
\text{power2}(X) \rightarrow X \times \text{power1}(X). \\
\text{power1}(X) \rightarrow X \times \text{power}(0, X).
\]

\[
\ldots \text{power3}(Y+Z) \ldots
\]
Partial Evaluation

\[
\begin{align*}
\text{power}(0,X) & \rightarrow 1; \\
\text{power}(N,X) \text{ when } N>0 & \rightarrow X \times \text{power}(N-1,X). \\
\text{... power}(3,Y+Z) ... \\
\text{power3}(X) & \rightarrow X \times \text{power2}(X). \\
\text{power2}(X) & \rightarrow X \times \text{power1}(X). \\
\text{power1}(X) & \rightarrow X \times \text{power0}(X). \\
\text{power0}(X) & \rightarrow 1. \\
\text{... power3}(Y+Z) ... 
\end{align*}
\]
Our idea

• Combine *partial evaluation* and type inference
• Compute the ”dependent values” during type-inference
• Infer types from *specialised* versions of the code
Example

\[
\text{keysearch}(\text{Key}, \ N, [H|T]) \\
\quad \text{when } \text{element}(N, H) == \text{Key} \rightarrow \{\text{value}, H\} \\
\text{keysearch}(\text{Key}, \ N, [H|T]) \rightarrow \text{keysearch}(\text{Key}, \ N, \ T) \\
\text{keysearch}(\text{Key}, \ N, []) \rightarrow \text{false}.
\]

- Specialise with \(N=2\)

\[
\text{keysearch2}(\text{Key}, \ [H|T]) \\
\quad \text{when } \text{element2}(H) == \text{Key} \rightarrow \{\text{value}, H\} \\
\text{keysearch2}(\text{Key}, \ [H|T]) \rightarrow \text{keysearch2}(\text{Key}, \ T) \\
\text{keysearch2}(\text{Key}, \ []) \rightarrow \text{false}.
\]
What does a Partial Evaluator Compute?

• Conventionally – everything it can!
  – Everything depending only on known values
  – Code explosion!
  – Not input/output

• For type-checking – everything it must!
  – Only values which affect types
  – (Hopefully) small code expansion
  – Including reading application resource files, etc.
A Promising Approach

• Looks very promising for e.g. generic servers

• Demands *mixing* partial evaluation and type inference
  – E.g. `length(tuple_to_list(T))`

  Type of `T` `{int(),bool()}`...

  – Just like *type specialisation* (Hughes 1996)

...known value [N,B]

...known result 2
A Tough Nut: Concurrency

Is the protocol followed?

Not just types, but also which messages are sent
A Tough Nut: Concurrency

- Kobayasi

Is the protocol followed?

Similar to UBF(B)

Client

Server

Abstract client

Abstract server

FSMs
But...

- Servers which talk to many clients?
  - Many protocol instances to keep track of
- Clients which talk to many servers?
  - Can protocols be confused?
- Aliases for the same Pid?
  - Sending to one changes the state of the other
- Partial evaluation of concurrent programs?
  - Hitherto only static number of processes
  (Marinescu and Goldberg 1997)
Summary

• Typing Erlang is an exciting problem!
  – Draws on Hindley-Milner, bidirectional typing, partial evaluation, type specialisation, concurrency theory...

• Mixing values and types is a powerful idea

• Concurrency is a tough nut to crack
• Lots more work to do!