A soft-typing system for Erlang

Sven-Olof Nyström
Department of Information Technology,
Uppsala University, Sweden
svenolof@csd.uu.se

ABSTRACT

This paper presents a soft-typing system for the programming language Erlang. The system is based on two concepts; a (forward) data flow analysis that determines upper approximations of the possible values of expressions and other constructs, and a specification language that allows the programmer to specify the interface of a module. We examine the programming language Erlang and point to various aspects of the language that make it hard to type. We present experimental result of applying the soft-typing system to some previously written programs.

1. INTRODUCTION

Erlang is a functional and concurrent programming language, developed at Ericsson [2] and intended for telecom applications. Erlang is dynamically typed, i.e., no type declarations are required (or allowed), and there is no requirement that an Erlang program should be examined by a type checker before it is run. Instead, each value carries dynamic type information.

One of the primary advantages with dynamic typing is that the language design is simplified. Also, it is often argued that dynamic typing helps rapid development, especially for prototyping and testing (see for example [13]). Another advantage is that it is possible to write general routines for writing and reading data of any type. This is particularly useful for Erlang's intended applications, as it allows communication over untyped channels.

However, most functional programming languages use static typing. Types are either given in explicit declarations, or computed automatically using type inference. Among the advantages with static typing is that many errors are discovered at compile time, instead of during testing. The type system might also discover errors that would have been difficult to catch by testing the program. Static type declarations can serve as documentation, and as they are always checked by the compiler, they are guaranteed to be up to date.

Erlang Workshop '03 29/08/2003, Uppsala, Sweden

A static type system added to an existing dynamically typed programming language is sometimes referred to as a *soft-typing system*. Generally speaking, a soft typing system might serve two purposes; it can produce type information to help compiler optimizations, and it can be used, just like a static type system, to help the programmer find bugs and inconsistencies in the program.

When used as a development tool, a soft typing system can either be used when programs are written from scratch, or applied to existing programs. Any soft typing system will be sensitive to the choice of data representation and control structures, so it is much easier to develop a system that works well on programs written for the system. In Section 7 we relate some experiences when applying the system to existing programs.

This paper presents a soft-typing system for the programming language Erlang. The system is based on two concepts; a (forward) data flow analysis that determines upper approximations of the possible values of expressions and other constructs, and a specification language that allows the programmer to specify the interface of a module. The soft-typing system checks one Erlang module at the time, uses the specification module as a starting point, and checks for each specified function that the function computes the value given by the specification.

The specification language allows parametric type and function specifications, Type specifications may be *local* to a module, *public*, i.e., accessible to other modules, or *abstract*, i.e., other modules may refer to the type but only the defining module has access to the representation.

The specification language also allows specification of process communication and process dictionaries. However, the checking of the specifications is not complete in the sense that the system may fail to detect programs that violate the specifications.

One advantage with basing the type system on a data flow analysis is that it is (relatively) easy to extend the system to handle constructs that would undoubtedly be very difficult to capture in a regular type system. For example, handling Erlang's meta-call (where the destination of a call is computed dynamically) is straight-forward in a flow analysis. As type information is only propagated in the direction of data flow, the spread of incorrect type information is limited. This makes it practical to allow the analysis to proceed, even if some inconsistencies are detected. Thus, the system can provide useful information, even for programs that are not completely type-correct.

2. RELATED WORK

The soft typing systems designed by Cartwright and Fagan [5] and Wright [16] target Scheme and Lisp. The main objective is to to provide type information for the compiler, but the soft typing systems also provide debugging information. They handle *all* programs, i.e., even programs with type errors will be annotated with type information. They use a generalization of Hindley-Milner type inference, relying on the presence of identifiable constructors. The system was intended to help the programmer by detecting possible programming errors and to help the compiler by providing type information.

Flanagan et. al [8] presented a tool for static debugging of Scheme programs. The system uses set-based analysis [15, 9] to identify potential run-time errors.

Lindgren [11] developed a soft-typing system for Erlang based on a constraint solver by Aiken [1]. The constraint solver could not represent types corresponding to individual atoms, a rather severe restriction as atoms often are used as constructors. Lindgren concludes that the constraint solver used is not suitable for Erlang.

Marlow and Wadler [12] have presented a soft-typing system for Erlang. While the previously described soft-typing systems could handle any legal program, Marlow and Wadler's simply refused to type programs where matching or case expressions might fail. Thus, it defined, in effect, a new programming language consisting of those Erlang programs that it could type. As in the earlier soft-typing systems by Cartwright&Fagan and Wright, the identification of data type constructors is crucial. Two types of constructors are used; tuples {`T1, ..., Tn`} and {`c, T1, ..., Tn`} where `c` is an atom. The type system is based on sub-typing, so that, for example, {`c, T2`} may be a subtype of {`T1, T2`}. Unlike the system presented in this paper, their system can deduce function types from function definitions. However, the system fails to deduce the expected type for some very simple function definitions. For example, consider the following function definition:

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

Their system uses pattern match compilation to transform the above into a system of simpler case-expressions. Given that the arguments of `and` are `X` and `Y`, the body of `and` will be an expression

```
case Y of
    false -> false
```

From this the type system concludes that the type of the second arguments must be `false`. Given this, the system then (correctly) finds that the first argument may be any value. Thus, the type for the function is

```
and(any(), false) -> true + false.
```

One may wonder whether a simple modification of Marlow and Wadler's system could improve the situation. Note, however, that the type deduced by Marlow and Wadler's system is technically correct. Before considering modifications to their system one should explain in formal terms why the above type should be rejected.

Arts and Armstrong developed a system called the specification web [3], which given declaration files generates a collection of html pages, *the specification web*, which provides cross-references between function definitions and their uses, type definitions, and so on. Their system assumes that each exported function in a module is declared.

Huch [10] presents an approach for the verification of Erlang programs using abstract interpretation and model checking. He defines a core fragment of Erlang, defines its operational semantics, an abstraction of process state which reduces the set of process states to a finite set, and an 'abstract' operational semantics over abstract states. To determine the behavior of a program is now a matter of enumerating the abstract states reachable in the abstract operational semantics. One difficulty with this approach is that even very small program are likely to generate an excessive number of abstract states.

3. THE ERLANG PROGRAMMING LANGUAGE

Since this paper is intended for an Erlang workshop, we assume familiarity with the Erlang programming language. In this section we discuss various features of Erlang which influenced the design of the soft-typing system.

3.1 Constructors

In statically typed programming languages such as SML, each data constructor is associated with a data type. One can introduce a new data type with associated constructors by writing, say,

```
datatype tree =
   NODE of tree * tree
 | LEAF of data
```

which introduces a type `tree` with constructors `NODE` and `LEAF`. Now, the type system will assume that each use of of these constructors follows the definition above, Thus, given an expression

```
NODE(x, y)
```

the type system infers not only that the returned value is of type `tree`, but also that the types of the arguments, `x` and `y`, are of type `tree`.

In contrast, Erlang has only two ways to build complex objects; tuples and lists. To express trees (as in the SML example above), an Erlang program might use terms of the forms

```
{node, Left, Right}
```

and

```
{leaf, Data}.
```

Many Erlang programs follows the convention illustrated above—use tuples with a specific atoms in first position as type constructors.

There is of course nothing that guarantees that the 'data constructor' `{node, _, _}` will not be used for some unrelated purpose in some other part of the program. Still, programs that follow the convention above are relatively easy to type. The soft-typing system presented in this paper handles such programs, even when the same constructor is used for different purposes in different parts of the program. Marlow and Wadler's type system also seems to handle them

(at least when a constructor is only used for one purpose); it seems that the reason they used a sub-typing system was so they could say that the type representing the set of values of the form `{leaf, Data}` is a subtype of the type of binary tuples.

There are, unfortunately, many programs that don't follow the convention. For example, when tracing a process, the operations of that process is represented as a sequence of messages

```
{trace, Pid, Operation, Data}
```

where '`Pid`' identifies the process, '`Operation`' the operation performed, and '`Data`' describes what the operation is applied to. Among the trace messages we find

```
{trace, Pid, 'receive', Message}
{trace, Pid, call, {M,F,A}}
{trace, Pid, link, Pid2}
```

In the case of a call, the data field is a trinary tuple indicating module, name and arity of the receiving process. For the link operation, the data field is the pid of another process. Thus, we have a data type (trace messages) where the type of one field depends on the value stored in another field.

As another example of an Erlang program which uses a data representation which may be hard to type, consider the function `flatten` from the `lists` standard library.

```
flatten(List) ->
    flatten(List, [], []).

flatten([H|T], Cont, Tail) when list(H) ->
    flatten(H, [T|Cont], Tail);
flatten([H|T], Cont, Tail) ->
    [H|flatten(T, Cont, Tail)];
flatten([], [H|Cont], Tail) ->
    flatten(H, Cont, Tail);
flatten([], [], Tail) ->
    Tail.
```

Flatten assumes as input a deeply nested list, i.e., a list which may contain other deeply nested lists or arbitrary terms (which are not lists). To distinguish between the two cases, the first clause of `flatten/3` contains a guard `list(H)`. Now, the result of a call to flatten is a list consisting of terms that are not lists. If a soft-typing system is expected to type this function, it must be able to reason about the effects of the guard, i.e., that when the guard succeeds, `H` is always a list, and when it fails, `H` is *not* a list.

3.2 Records

Erlang is unique among the functional programming languages in its use of a C-style preprocessor, macros and header files.

The preprocessor allows among other things record definitions such as the following.

```
-record(node, { left, right}).
-record(leaf, {data}).
```

The expression `#node{left = Left, right = Right}` refers to a record with fields `left` and `right` given by variables `Left` and `Right`, respectively. If `X` is a node, the expression `X#node.left` extracts the first field. An expression `X#node{left = NewLeft}` creates a new node where the first field has been changed.

Internally, a record with n fields is represented as an with $n+1$-tuple where the first position is the name of the record, and one can assume that many applications take advantage of this. A soft-typing system for Erlang must of course handle records, even allowing for programs that access the internal representation. Reasoning about preprocessing, header files and macro expansion in the soft-typing system would have introduced complications, so we chose to let the soft-typing operate on code that had been passed through the preprocessor.

Now, the use of records should in principle be completely unproblematic as they are expanded into other Erlang constructs. However, we must make sure that the soft-typing system can reason about the code that the preprocessor generates for records.

Given the source code expression

```
#node{left = Left, right = Right},
```

the preprocessor will produce a call `element(2,X)` which extracts the second element of `X`, if `X`, is a tuple. For expressions that update the field of a record, a call to `setelement/3` is generated. Thus, the analysis must 'know' the semantics of the built-in functions `element/2` and `setelement/3`. On the other hand, there is also code that uses tuples to represent arrays and lets an index run through all positions of a tuple. To summarize, a tuple is sometimes used as a data constructor and a type system for Erlang must allow for calls to built-in functions `element(2, X)` and `element(3, X)` to return values of different types. In other situations, the tuples are used as constant arrays, whose size may be unknown, and the type system should also allow for this possibility.

3.3 Processes and process dictionaries

Here we will argue that regardless of underlying type system, it is difficult (or impossible) to find a reasonable typing of Erlang processes.

Clearly, the type of an Erlang process should include information about the messages it receives and handles—we want to be able to tell whether a send `P ! {message, ...}` is type-correct. Since a process is created with a reference to a function in, say, `P = spawn(m, f, [X1, ..., Xn])`, it follows that functions must also be typed with what they receive.

What makes reasoning about process types in Erlang awkward is the way "internal" communication is expressed. If a function `f` contains communication with some process `P`, it is typically expressed using the following pattern:

```
f(...) ->
    P ! {query, ..., self()},
    receive
        {answer, P, Data} -> ...
    end,
```

i.e., the function sends a message `{query, ...}` to the process `P`, containing a reference to the current process, and then waits for a reply. The response to this message will of course arrive in the mailbox of the current process. (Examples of code following this pattern can be found in, for example, the `io` standard library, any application written using the `gen_server` library. Even the simplest example of process communication given in the Erlang textbook by

Armstrong et al. [2], the `counter` process, page 72 follows this pattern.)

In the above example, we must include in the type of the function `f` information that it may receive messages of the form `{anwer, ...}` (since any process executing `f` may receive such messages). This is unpleasant, as the communication with `P` is part of the implementation of `f`, and not intended to be part of the external interface. Further, the same information must be added to the type of any function that calls `f`. To consider a concrete example, the function

```
hello() ->
    io:format("Hello, world!~n", []).
```

will have a rather complicated type, as the call to `format` will result in internal process communications.

It seems that we would not do the programmer any favor by forcing him to deal with the internals of various libraries. In the system presented in this paper, we have chosen a simpler approach which allows the programmer to declare, for a module, or a function, the expected messages in this particular part of the program.

Note that a type system might distinguish between messages that may be sent to the process and messages that a process will handle, see for example [6] but the conclusions of the discussion above still hold.

Erlang's process dictionaries pose similar problems. Each Erlang process maintains a mapping from keys to values (both keys and values may be arbitrary terms) which may be accessed by the built-in function `get` and `put`. Safe typing of process dictionaries requires that the type system guarantees that different accesses to the process dictionary are consistent. If the type system relies on specifications of the external interface, as the soft-typing system described in this paper, every access to the process dictionary, direct or indirect, must be recorded in the specification of a function. Adding a call to `put` in one function could require changes to a large number of specification modules. Since this is obviously impractical and unreasonable, we chose the simpler (and unsafe) approach to allow declarations of process dictionaries in code that accesses the dictionary directly.

4. THE SPECIFICATION LANGUAGE

The syntax of the specification language is influenced by the specifications used by Marlow and Wadler [12] and Arts and Armstrong [3].

As a first example, consider a simple function definition.

```
foo(X, Y) ->
       R = X + Y,
       R.
```

One possible specification for this function might be:
`foo(int(), int()) -> int().`

As a second example, consider the definition of the function `append/2`.

```
-module(append).

append([X | L1], L2) ->
    [X | append(L1, L2)];
append([], L) ->
    L.
```

Most programmers would probably argue that the following specification of `append` is the correct one:

```
-module(append).

-type list(X) = [] | [X | list(X)].

append(list(X), list(X)) ->
    list(X).
```

However, it is worth noting that other specifications are consistent with the function definition, for example, if append is called with the empty list as first argument and a floating-point value as second it will always return a floating-point value. Thus `append` is a correct implementation of the following specification: `append([], float()) -> float().`

4.1 Function specifications

Function specification are of the form

`f(t1, ..., t_n) -> t_0.`

where each of `t0`, `t1`, ..., `tn` are type expressions. The meaning of the specification is: When the function is called with arguments according to `t1`, ..., `tn` the result will be according to `t0`.

4.2 Type expressions

A type expression can be one of the following:

1. a primitive type, `int()`, `float()`, `pid()`, or `atom()`.
2. An atomic value, such as `foo`, `true`, `false`, `42`, or `3.14`.
3. The universal type: `any()`.
4. The empty type: `none()`.
5. A union, for example `int() + float() + true + false`.
6. Complex types, i.e., lists and tuples, for example `{foo, int(), float()}` or `[1, 2, 3.14]`.
7. Function types, i.e., `fun(int()) -> int() end`.
8. In parametric definitions, we also allow type variables, written with initial capital letters, say `X`, `Key`, or `Table`.
9. A reference to a defined type, for example `list(int())`.

4.3 Type definitions

In its simplest form, a type definition simply gives a shorthand for a more complex expression.

```
- type bool() =
    true + false.
```

The type `bool()` is only accessible locally. It is synonymous to `true + false`.

A type definition can also be recursive.

```
- type intlist() =
    [] + [int() | intlist()].
```

The defined type `intlist()` can be referenced in function specifications:

```
append(intlist(), intlist()) ->
    intlist().

map(fun int() -> intlist() end, intlist()) ->
    intlist().
```

4.4 Parametric specifications

Many Erlang functions are polymorphic, that is, they are designed to work with many different data types. To specify such functions, we allow function specifications with type variables.

```
- type list(X) =
    [] + [X | list(X)].

- type tree(K, V) =
    nil + {K, V, tree(K, V), tree(K, V)}.
```

Examples of parametric function specifications:

```
append(list(X), list(X)) ->
    list(X).

lookup(K, tree(K, V)) ->
    not_found + {found, V}.

map(fun(X) -> Y end, list(X)) ->
    list(Y).
```

4.5 Abstract types

Type definitions with the '`type`' keyword are only visible within the specification module where they are given. Suppose that an Erlang module defines an abstract data type, i.e., a data structure with a set of operations to create and operate on the structure, where the intention is that no other code should access the data structure directly. In the specification language, this can be expressed using the keyword `abstype`.

```
- abstype tree(K, V) =
    nil + {K, V, tree(K, V), tree(K, V)}.
```

Abstract type can be referenced from other modules. However, during type checking, a reference to an abstract type defined in an other module, say,

```
m:tree(int(), list(int())),
```

is treated as a data type constructor. It is of course possible to write a module that examines the internal representation of an abstract type, but this module will not be accepted by the type system.

Thus, an abstract type has two faces; to the module where it is defined, it is just another defined type, but to other modules it is a data type constructor, i.e., a type which cannot be decomposed into other types.

4.6 Public types

Public types are just like ordinary (`type`) type definitions, except that they are accessible from all modules. It may be inconvenient to repeat the definition of the `list` type in every module. A better approach may be to give a single definition, for example, in the `lists` library, and make it public.

```
-public_type list(X) =
    [X | list(X)] + [].
```

A public type definition can be referred to in any specification module by the syntax `lists:list(int())`.

4.7 Unsafe extensions

It turns out that specifying the interaction of an Erlang process is rather difficult. The specification language allows specification such as

```
+ mbox = increment + stop + {pid(), int()}.
```

or

```
+ mbox(loop/1) = increment + {pid(), int()} + stop.
```

The first declaration reads: the messages received in the mailbox while executing in the current module are either the atom `increment`, the atom `stop`, or a tuple of a pid and an integer). The second declaration refers to the situation when executing the function `loop/1`.

Unfortunately, the system cannot verify that the above declarations always hold. Still, declarations such as these are useful in conveying the programmer's intentions and in showing what happens when the process receives the messages given in the declarations.

It is also possible to declare the contents of the process dictionary. A declaration from one of the specification files of the analysis:

```
+ dict = (max_contexts  -> int();
          weight_table  -> weight_table();
          strata_table  -> strata_table()).
```

The declaration gives the types of values associated with the keys `max_contexts`, `weight_table`, and `strata_table`. The expressions `weight_table()` and `strata_table()` refer to types defined elsewhere.

5. THE SOFT-TYPING SYSTEM

This section deals with the inner workings of the soft-typing system. Readers mainly interested in the use of the system can skip this section.

Generally speaking, to verify that a function behaves according to specification, the following basic steps must be performed:

1. Generate function arguments according to specification.

2. Use data flow analysis to determine (approximation of) result of function call.

3. Check whether the result matches the result type given in the specification.

For an Erlang module `foo.erl`, we assume that the specifications are written in a separate file `foo.spec`. The specification module should contain specifications of functions that the module exports and definitions of various data types.

The dataflow analysis is based on analysis techniques such as 0CFA [15] or set-based analysis [9]. The generator and the matcher operate on the same representation of type information.

We assume that all external modules are specified, thus, the matcher checks that the arguments of the external call are of the specified type and the generator gives the return value.

5.1 Programs

In the presentation of the analysis, we assume that all data-types (for example atoms, integers, floating-point numbers, lists and tuples) are expressed using a set of type constructors, $C \in \mathit{Con}$, where each constructor has a given arity. We also assume a set of pre-defined functions $p \in \mathit{Pre}$ and a set of program-defined functions $f \in \mathit{Function}$ and a set of labels, Lab.

Let a *program* be a set of definitions of the form

$$f(x_1, \ldots, x_n) \to E,$$

where expressions are defined according to

$$\begin{array}{rl} E ::= & x \mid C[E_1, \ldots, E_n] \mid \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \\ & \mid f(E_1, \ldots, E_n) \mid \texttt{fun}^l\ (x_1, \ldots, x_n) \to E_1 \\ & \mid E_0(E_1, \ldots, E_n)^l \mid p(E_1, \ldots, E_n) \end{array}$$

We assume that there is a program-defined function $f_e \in \mathit{Function}$. The intention is that f_e will serve as an entry point in the analysis.

5.2 Specifications

A specification module consists of a set of function specifications of the form

$$f(T_1, \ldots, T_n) \to T_0,$$

and a set of type definitions

$$d(T_1, \ldots, T_n) = T_0$$

where type specifications are defined according to:

$$\begin{array}{rl} T ::= & x \mid \mathsf{any} \mid C[T_1, \ldots, T_n] \\ & \mid T_1 + \ldots + T_n \\ & \mid \texttt{fun}^l\ (T_1, \ldots, T_n) \to T_0 \mid d(T_1, \ldots, T_n) \end{array}$$

5.3 Basic structures

The state of the analysis is a store, mapping analysis variables to terms.

Analysis variables are used to store intermediate and final results. To make the analysis poly-variant, it is necessary to let analysis variables range over contexts. Thus, for contexts $c \in \mathit{Context}$, let Var be one of the following

1. $\mathsf{Arg}(f, k, c) \in \mathit{Var}$, where f is a program-defined function with arity $n \geq k$.
2. $\mathsf{Res}(f, c) \in \mathit{Var}$, where f is as above.
3. $\mathsf{FunArg}(l, k, c) \in \mathit{Var}$, where $l \in \mathit{Lab}$ is the label of a fun expression.
4. $\mathsf{FunRes}(l, c), \mathsf{ApplyRes}(l, c), \mathsf{IfRes}(l, c) \in \mathit{Var}$, where l is the label of a call to a higher-order function.

Let $t \in \mathit{Term}$, the set of terms, be the least set such that

1. $\mathit{Var} \subseteq \mathit{Term}$.
2. $\mathsf{any} \in \mathit{Term}$.
3. $C[t_1 \ldots t_n] \in \mathit{Term}$, where constructor C has arity n and $t_1, \ldots, t_n \in \mathit{Term}$.
4. $\mathsf{FunTerm}(l, c) \in \mathit{Term}$, where l is the label of a fun expression.

The implementation divides the type checking problem into a set of tasks. Among the tasks are analyze a function f in context c, generate analysis information according to a type definition, and check that generated analysis information matches a type definition. The set of tasks is listed in Section 5.8. Let Work, the set of tasks. The analysis will maintain a work list, containing a subset of Work.

A flow analysis computes, for each variable and subexpression in the program, an approximation of the set of possible values. An analysis which simply associates values to different parts of the program (i.e., a *mono-variant* or *context-insensitive* analysis) suffers from the problem that if a function is called from different parts of the program, the analysis will set the result of the different calls to be union of all calls to the function. For polymorphic functions, this will of course give lower precision, but functions that are not polymorphic may also be affected. Consider for example the function `append`. If there is one call site where the type of the second argument is unknown, the result will also be unknown. Thus, a mono-variant analysis may propagate a low-precision result to all call sites where a poly-variant analysis would confine it to one part of the program.

To allow functions to be typed as polymorphic, we use a mechanism for poly-variant analysis described in more detail elsewhere [14]

The mechanism we use involves a set of *contexts*, $c \in \mathit{Context}$, and a function

$$\mathsf{Call}(f, l, c) = c'$$

which, given a function f, a call site l and a context c returns a new context. We also assume an initial context, c_0. The idea is that if a call to function f occurs at label l in context c, the body of f will be analyzed in context c'. To guarantee termination, the set of contexts must be finite.

5.4 Implementation of set abstraction

The *store* will associate with each analysis variable X the following:

1. $X.\mathsf{value} \subseteq \mathit{Term}$, a set of terms which are not analysis variables.
2. $X.\mathsf{link} \subseteq \mathit{Var}$, a set of variables.
3. $X.\mathsf{depend} \subseteq \mathit{Work}$, a set of analysis tasks.

When the analysis is finished, the relevant information for each variable is collected in $X.\mathsf{value}$. For example, for a function f, $\mathsf{Arg}(f, 1, c).\mathsf{value}$ gives an approximation of the values that may be passed in the first argument of f.

For each variable X we also store $X.\mathsf{link}$, a set of variables such that $X \subseteq Y$, for each $Y \in X.\mathsf{link}$, and $X.\mathsf{depend}$, a set of analysis tasks whose result may depend on the value of X. Thus, if the value of X changes, it may be necessary to re-analyze any member of $X.\mathsf{depend}$.

We define the following operations on the store.

1. $\mathsf{Lookup}(X)$. Determine the current value of X.
2. $\mathsf{Add}(t, X)$. Add the term t to the value of X.
3. $\mathsf{Add}(X, Y)$. Add the value of X to Y, i.e., make X a subset of Y.

$\mathsf{Lookup}(X)$:
1: Add current task to X.depend
2: Return X.value

$\mathsf{Add}(t, X)$:
1: Test if t is contained in X.value
2: If not,
3: set X.value to X.value $\cup \{t\}$,
4: put all tasks in X.depend on work list,
5: for each variable $Y \in X$.link, do $\mathsf{Add}(t, Y)$.

$\mathsf{Add}(X, Y)$:
1: if X is a member of Y.link,
2: do nothing
3: if X is *not* a member of Y.link,
4: add X to Y.link,
5: let $t = Y$.value and
6: do $\mathsf{Add}(t, X)$

$\mathsf{Contains}(t_1, t_2)$:
1: Return true if
2: $t_1 = t_2$,
3: $t_2 = \mathsf{any}$, or
4: t_2 is a variable X, and $\mathsf{Contains}(t_1, t')$ holds, for some $t' \in \mathsf{Lookup}(X)$.
5: Return false otherwise.

Figure 1: Implementation of set abstraction.

We assume that during any point in the analysis, it is possible to determine the current analysis task (an element of *Work*). By dividing the analysis problem into a set of separate tasks, it is possible to devise a work-list oriented strategy where a portion of the program only needs to be reanalyzed when a value on which it depends on has changed. The purpose of the link field is to represent inclusion relations explicitly. The implementation of the operations is given in Figure 1.

5.5 Analyzing Erlang expressions

Analysis of an expression takes

1. expression to be analyzed
2. an environment mapping program variables to terms and
3. current context
4. a store

and returns

1. a term and
2. an updated store.

When analyzing expressions consisting of a single variable, simply look up the value of the variable in the current environment.

$\mathsf{Analyze}(x, \mathcal{E}, c)$:
1: return $\mathcal{E}(c)$

Expressions involving a constructor simply build a corresponding term.

$\mathsf{Analyze}(C[E_1, \ldots, E_n], \mathcal{E}, c)$:
1: let $t_k = \mathsf{Analyze}(E_k, \mathcal{E}, c)$, for $k \leq n$
2: construct the term $C[t_1, \ldots, t_n]$ and
3: return it as the result of the analysis.

$\mathsf{Analyze}(x, \mathcal{E}, c)$:
1: return $\mathcal{E}(c)$

$\mathsf{Analyze}(\texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3, \mathcal{E}, c)$:
1: let $t_1 = \mathsf{Analyze}(E_1, \mathcal{E}, c)$
2: if $\mathsf{Contains}(\text{true}, t_1)$ holds,
3: let $t_2 = \mathsf{Analyze}(E_2, \mathcal{E}, c)$
4: $\mathsf{Add}(t_2, \mathsf{IfRes}(l, c))$
5: if $\mathsf{Contains}(\text{false}, t_1)$ holds,
6: let $t_3 = \mathsf{Analyze}(E_3, \mathcal{E}, c)$
7: $\mathsf{Add}(t_3, \mathsf{IfRes}(l, c))$
8: return $\mathsf{IfRes}(l, c)$.

$\mathsf{Analyze}(f(E_1, \ldots, E_n)^l, \mathcal{E}, c)$:
1: let $t_k = \mathsf{Analyze}(E_k, \mathcal{E}, c)$ for $k \leq n$
2: let $c' = \mathsf{Call}(f, l, c)$,
3: $\mathsf{Add}(t_k, \mathsf{Arg}(f, k, c'))$, for $k \leq n$
4: unless $\langle f, c' \rangle$ has been analyzed before, add $\langle f, c' \rangle$ to work list
5: return $\mathsf{Res}(f, c')$

$\mathsf{Analyze}(\texttt{fun}^l\ (x_1, \ldots, x_n) \to E_0, \mathcal{E}, c)$:
1: create a new environment $\mathcal{E}_1$ by extending old environment $\mathcal{E}$ with bindings $x_k \mapsto \mathsf{FunArg}(l, k, c)$, for $k \leq n$
2: let $t = \mathsf{Analyze}(E_0, \mathcal{E}_1, c)$
3: $\mathsf{Add}(t, \mathsf{FunRes}(l, c))$
4: return $\mathsf{FunTerm}(l, c)$

$\mathsf{Analyze}(E_0(E_1, \ldots, E_n)^l, \mathcal{E}, c)$:
1: let $t_k = \mathsf{Analyze}(E_k, \mathcal{E}, c)$ for $0 \leq k \leq n$
2: for each l', such that $\mathsf{Contains}(\mathsf{FunTerm}(l', c'), t_0)$,
3: $\mathsf{Add}(t_k, \mathsf{FunArg}(l', k, c'))$, for $1 \leq k \leq n$
4: $\mathsf{Add}(\mathsf{FunRes}(l', c'), \mathsf{ApplyRes}(l, c))$
5: Return $\mathsf{ApplyRes}(l, c)$

Figure 2: Analyzing expressions.

The analysis of complex expressions is given in Figure 2.

In the analysis of calls to program-defined functions, we use the function Call to compute a new context. When a call is analyzed for the first time, a new analysis task consisting of the called function and the new context is added to the work list.

In the analysis presented here, closures are *not* polymorphic. A polymorphic analysis would be more complex, and as most Erlang applications make very little use of higher-order functions the added complexity and cost of an analysis that could treat closure applications polymorphically cannot be justified. See [14] for a detailed discussion.

Analyzing calls to higher-order functions is similar to analyzing calls to user-defined functions, but slightly complicated by the fact that we need the analysis to determine the destination of the call. For a fun-expression (a closure) $\texttt{fun}^l\ (x_1, \ldots x_n) \rightarrow E_0$, we use analysis variables $\mathsf{FunArg}(l, 1, c)$ through $\mathsf{FunArg}(l, n, c)$ to represent the arguments, i.e., the set of possible values that may be passed as arguments to the function. In a similar way, the set of values that may be returned by the function is stored in the variable $\mathsf{FunRes}(l, c)$.

However, we will still distinguish between different instances of a closure. Thus, a closure is represented by a term of the form $\mathsf{FunTerm}(l, c)$, to distinguish between closures created at the same program point but in different contexts.

5.6 Generation

From a given type specification we generate type information in the internal representation used by the data flow analysis.

In this section, we will only consider simple type definitions. Generally speaking, generation assumes

1. a type expression,
2. an environment (mapping type variables to terms), and
3. a context,

and returns

1. a term, and an
2. updated store

The type of a type variable is obtained from the environment.

$\mathsf{Gen}(x, \mathcal{A}, c)$:
1: return $\mathcal{A}(c)$

Handling of the universal type and constructors is straightforward.

$\mathsf{Gen}(\mathsf{any}, \mathcal{A}, c)$:
1: return the term any

$\mathsf{Gen}(C[C]T_1, \ldots, T_n, \mathcal{A}, c)$:
1: let $t_k = \mathsf{Gen}(T_k, \mathcal{A}, c)$, for $k \leq n$
2: return $C[C]t_1, \ldots, t_n$

If the type expression is a function type, generating type information consists of three parts; create a fun term indicating a function object, match the arguments against the arguments given in the function type, and return the function term.

$\mathsf{Gen}(\texttt{fun}^l\ (T_1, \ldots, T_n) \rightarrow T_0, \mathcal{A}, c)$:
1: $\mathsf{Match}(T_k, \mathsf{FunArg}(l, k, c),)$, for $1 \leq k \leq n$
2: If any of the matchings fail, report bad function argument at current fnid.
3: let $t_0 = \mathsf{Gen}(T_0, \mathcal{A}, c)$
4: $\mathsf{Add}(t, \mathsf{FunRes}(l, c))$
5: return $\mathsf{FunTerm}(l, c)$

The representation of a union type is easy, since there is an internal representation of unions.

$\mathsf{Gen}(T_1 + \ldots + T_n, \mathcal{A}, c)$:
1: let $t_k = \mathsf{Gen}(T_k, \mathcal{A}, c)$, for $1 \leq k \leq n$
2: return $t_1 + \ldots + t_n$

As mentioned above, we assume that d is a local type definition.

$\mathsf{Gen}(d(T_1, \ldots, T_n)^l, \mathcal{A}, c)$:
1: let $t_k = \mathsf{Gen}(T_k, \mathcal{A}, c)$, for $k \leq n$
2: let $c' = \mathsf{Call}(d, l, c)$,
3: $\mathsf{Add}(t_k\mathsf{Arg}(f, k, c'),)$, for $k \leq n$
4: if $\langle \mathsf{Gen}, f, c \rangle$ is not already active,
5: make $\langle \mathsf{Gen}, f, c \rangle$ active, and
6: put $\langle \mathsf{Gen}, f, c \rangle$ on work list
7: Return $\mathsf{Res}(f, c')$

5.7 Matching

Matching a term against a type expression assumes

1. a type expression,
2. a term,
3. an environment (mapping type variables to (analysis) variables,
4. a context, and
5. a store

and returns an updated store.

We use an analysis variable $\mathsf{MatchRes}(c)$ to pass information about the success of the matching done in context c, if the matching fails, a term fail is added to $\mathsf{MatchRes}(c)$.

Type variables can be encountered in four situations; at the top-level (when checking a function with a polymorphic specification), when analysing a function call to a specified function, and during matching or generation of a type specification containg a defined type.

Suppose that we are checking a function with a polymorphic specification, for example the function `append`

```
append(list(A),list(A)) -> list(A).
```

Here, the type variable `A` is (implicitly) universally quantified, i.e., the specification implies that the implementation should operate on lists of any type. Thus, a function that only operated on lists of integers should be rejected by the type system.

To be able to check that the program makes no assumptions on type variables introduced in polymorphic specifications, we introduce a new class of constructors, *parameters*,

one for each type variable. A parameter can only be matched against it self (or an unbound variable). Thus, the type system can check that no assumptions are made about types passed as parameters.

The other three uses of type variables are handled by binding the type variables to fresh analysis variables.

In the matching of a type variable against a term (below), we distinguish between the two cases—either x is bound to an analysis variable in which case matching always suceeds, or x is bound to a parameter in which case matching succeeds if and only if the term t is that parameter.

$\mathsf{Match}(x, t, \mathcal{A})c$:
1: if $\mathcal{A}(x)$ is an analysis variable X,
2: $\quad \mathsf{Add}(t, X)$
3: otherwise, if $\mathsf{Contains}(t, \mathcal{A}(x))$ does
4: not hold, report that matching failed by
5: $\quad \mathsf{Add}(\mathsf{fail}, \mathsf{MatchRes}(c))$

Matching the universal type is of course straight-forward.

$\mathsf{Match}(\mathsf{any}, t, \mathcal{A})c$:
1: (nothing needs to be done)

When matching a type expression with a constructor, we check that the term is of the same form and then proceed recursively with the sub-terms.

$\mathsf{Match}(C[C]T_1, \ldots, T_n, t, \mathcal{A})c$:
1: if t is an analysis variable,
2: $\quad$ let $t' = \mathsf{Lookup}(t)$,
3: $\quad$ do $\mathsf{Match}(C[C]T_1, \ldots, T_n, t', \mathcal{A})c$
4: otherwise, if t is a union $t_1 + \ldots + t_n$,
5: $\quad$ do $\mathsf{Match}(C[C]T_1, \ldots, T_n, t_k, \mathcal{A})c$, for $k \leq n$
6: otherwise, if t is of the form $C[C]t_1, \ldots, t_n$,
7: $\quad$ do $\mathsf{Match}(C[C]T_k t_k \mathcal{A} c, ,)$ for $k \leq n$
8: otherwise, report that matching failed by
9: $\quad \mathsf{Add}(\mathsf{fail}, \mathsf{MatchRes}(c))$

To match a term against a union of type specifications we assume two functions Extract and Remove which given a term and a specification produce new terms. Keep in mind that each term represents a set of possible values. Now, assume that S_t is the set of values represented by the term t, and that S_T is the set of values matched by the type T. Now, Extract and Remove can be specified as follows:

1. If $t' = \mathsf{Extract}(T, t, Env, c)$, then t' represents a superset of all values represented by t that are also matched by specification T, i.e., $S_{t'} \supseteq S_t \cap S_T$, where $S_{t'}$ is the set of values represented by t'.

2. If $t' = \mathsf{Remove} T t Env c$, then t' represents a superset of all values represented by t that are *not* matched by specification T, i.e., $S_{t'} \supseteq S_t \setminus S_T$.

Computing Extract and Remove excatly is a difficult problem, so we will settle for rather crude approximations. Given Extract and Remove, the matching of unions is straightforward.

$\mathsf{Match}(T_1 + \ldots + T_n, t, \mathcal{A})c$:
1: let $t_1 = \mathsf{Extract} T_1 t \mathcal{A} c$
2: let $t_r = \mathsf{Remove} T_1 t \mathcal{A} c$
3: $\mathsf{Match}(T_1, t_1, \mathcal{A})c$
4: $\mathsf{Match}(T_2 + \ldots + T_n, t_r, \mathcal{A})c$

$\mathsf{Extract}(T, t, \mathcal{A}, c)$:
1: $\mathsf{Extract}(\mathsf{any}, t, n) = t$
2: $\mathsf{Extract}(\mathsf{none}, t, n) = \mathsf{none}$
3: $\mathsf{Extract}(C[\ldots], V, n) =$
4: $\quad \{\mathsf{Extract}(C[...], t, n-1) \mid t \in V.\mathsf{value}\}$
5: $\mathsf{Extract}(C[T_1, \ldots, T_m], C[t_1, \ldots, t_m], n) =$
6: $\quad C[t'_1, \ldots, t'_m]$,
7: $\quad$ where $tk' = \mathsf{Extract}(T_k, t_k, n)$, for $k \leq m$.
8: $\mathsf{Extract}(C[\ldots], C'[\ldots], n) = \mathsf{none}$, if $C \neq C'$
9: $\mathsf{Extract}(T_1 + T_2, t, n) = t_1 + t_2$
10: $\quad$ where $t_1 = \mathsf{Extract}(T)1, T, n)$
11: $\quad$ and $t_2 = \mathsf{Extract}(T_2, T, n)$
12: $\mathsf{Extract}(\mathtt{fun}^l\ (\ldots) \to \ldots, t, n) = t$, if t represents a function type
13: $\mathsf{Extract}(\mathtt{fun}^l\ (\ldots) \to \ldots, t, n) = \mathsf{none}$, otherwise
14: $\mathsf{Extract}(d(T_1, \ldots, T_m), t, 0) = t$
15: $\mathsf{Extract}(d(T_1, \ldots, T_m), t, n) = \mathsf{Extract}(T', t, n-1)$
16: $\quad$ where $d(X_1, \ldots, X_m) = T$,
17: $\quad$ and T' is the result of replacing each occurrence of
18: $\quad X_k$ in T with T_k, for $k \leq m$ and $n > 0$

$\mathsf{Remove}(T, t, \mathcal{A}, c)$:
1: $\mathsf{Remove}(\mathsf{any}, t, n) = \mathsf{none}$
2: $\mathsf{Remove}(\mathsf{none}, t, n) = t$
3: $\mathsf{Remove}(T = C[...], V, n) =$
4: $\quad \{\mathsf{Remove}(T, t, n-1) | t \in V.\mathsf{value}\}$
5: $\mathsf{Remove}(C[T_1, \ldots T_m], C[t_1, \ldots, t_m], n) =$
6: $\quad \{C[t'_1, \ldots, t'_m]\ |$,
7: $\quad$ where for some k, $t'_k = \mathsf{Remove}(T_k, t_k, n)$,
8: $\quad$ and $t_k \neq \mathsf{none}$
9: $\quad$ and $t'_l = t_l, l \neq k\}$
10: $\mathsf{Remove}(C[\ldots], t = C'[\ldots], n) = t$
11: $\mathsf{Remove}(T_1 + T_2, t, n) = \mathsf{Remove}(T_1, \mathsf{Remove}(T_2, t, n), n)$
12: $\mathsf{Remove}(\mathtt{fun}^l\ \ldots \to \ldots, t, n) = t$,
13: $\mathsf{Remove}(d(T_1, \ldots, T_m), t, 0) = t$
14: $\mathsf{Remove}(d(T_1, \ldots, T_m), t, n) = \mathsf{Remove}(T', t, n-1)$
15: $\quad$ where $d(X_1, \ldots, X_m) = T$,
16: $\quad$ and T' is the result of replacing each occurrence of
17: $\quad X_k$ in T with T_k, for $k \leq m$ and $n > 0$

Figure 3: Implementation of extract and remove operations.

Figure 3 shows a simplified version of the extract and remove operations used in the analysis. When matching a type expression $\texttt{fun}^l\ (T_1, \ldots, T_n) \to T_0$ with a term t, we must check that t is indeed a closure, and then verify that t returns a value of type T_0 when called with arguments of type T_1 through T_n.

Match($\texttt{fun}^l\ (T_1, \ldots, T_n) \to T_0, t, \mathcal{A})c$:
1: If t is an analysis variable,
2: call Match($\texttt{fun}^l\ (T_1, \ldots, T_n) \to T_0, t', \mathcal{A})c$
3: for each $t' \in t$.value.
4: if t is not an analysis variable,
5: if t is a term FunTerm(l', c'), for some l' and c',
6: let t_k = Gen($T_k, \mathcal{A}, c$), for $1 \le k \le n$
7: Add(t_k, FunArg(l', k, c')), for $k \le n$
8: do Match(T_0, FunRes(l, c), $\mathcal{A}$)
9: if t is any other term, report that matching failed by Add($fail$, MatchRes(c)).

References to defined types are treated in a manner similar to the treatment of function calls in the analysis. A new context is created, a new task is created for the matching of the body of d (i.e., the right-hand side of the definition of d) against the "result". Since the direction of data flow is backwards compared to the evaluation of a function definition, the term t is added to the result of the call and the formal arguments T_1 through T_n are matched against the actual arguments t_1 through t_n.

the arguments to the defined type are passed

Match($d(T_1, \ldots, T_n)^l, t, \mathcal{A})c$:
1: let c' = Call(d, l, c),
2: Add(MatchRes(c'), MatchRes(c))
3: if $\langle$Match$, d, c'\rangle$ is not already active,
4: make $\langle$Match$, d, c'\rangle$ active, and
5: put $\langle$Match$, d, c'\rangle$ on work list
6: Add(Res(d, c'), t)
7: let t_k = Arg(d, k, c'), for $k \le n$
8: Match($T_k, t_k, \mathcal{A})c$, for $k \le n$

5.8 The main loop

The soft-typing system is based on a work list algorithm—the problems of analysis, matching and generation are represented as tasks in the work list. The use of a work list allows a fairly straight-forward handling of recursive function and type definitions. Similarly, the treatment of parametric types resembles the passing of arguments to functions.

The main loop maintains a work list of tasks. A task is one of the following.

1. $\langle$Analyze$, f, c\rangle$, analyze function f in context c,
2. $\langle$Gen$, d, c\rangle$, generate type information accoring to type definition d,
3. $\langle$Match$, d, c\rangle$, match against defined type d,
4. $\langle$Check$, f, c\rangle$, check that function f behavesaccording to specification,
5. $\langle$UseSpec$, f, c\rangle$, use specification of f when computing the result of a call to f.

. Since the relevant arguments and results are passed in the store (associated with the context), the task only needs to contain function and context.

The main loop will simply remove a task from the worklist and preform the corresponding operation until the work list is empty.

AnalyzeProgram:
1: For each function f mentioned in the specification, put $\langle$Check$, f, c_0\rangle$ in WorkList.
2: Execute MainLoop.

MainLoop:
1: if WorkList is empty,
2: terminate analysis
3: if WorkList is not empty,
4: remove a task from WorkList,
5: if the task is $\langle$Analyze$, f, c\rangle$, do Analyze(f, c)
6: if the task is $\langle$Gen$, d, c\rangle$, do Gen(d, c)
7: if the task is $\langle$Match$, d, c\rangle$, do Match(d, c)
8: if the task is $\langle$Check$, f, c\rangle$, do CheckSpec(f, c)
9: if the task is $\langle$UseSpec$, f, c\rangle$, do UseSpec(f, c)
10: continue MainLoop untl WorkList is empty

To compute the effects of a call to a function f in context c, we build an environment from information in the store. It is assumed that the type of argument k is stored in analysis variable Arg(f, k, c). Similarly, the result of the call is passed back to the caller in analysis variable Res(f, c).

Analyze(f, c):
1: Let the definition of f be $f(x_1, \ldots, x_n) \to E$
2: Create environment $\mathcal{E}$ mapping each of X_k to the term Arg(f, k, c), for $k \le n$
3: let t = Analyze($E, \mathcal{E}, c$)
4: Add(t, Res(f, c))

To generate a term from a type expression $d(t_1, \ldots, t_n)$, where d is a defined type, an analysis task is created for the body of the definition. As in the analysis of function calls, the arguments and results are passed in the store.

Gen(d, c):
1: Let the definition of d be $d(T_1, \ldots, T_n) = T_0$
2: Let $x_1, \ldots, x_m$ be the free variables of $T_0, \ldots, T_n$
3: Create type environment $\mathcal{A}$ mapping each of x_k to a term Univ(k)c, for $0 \le k \le m$.
4: Match(T_k, Arg(d, k, c), $\mathcal{A}$)c, for $1 \le k \le n$
5: let t = Gen($T_0, \mathcal{A}, c$)
6: Add(t, Res(d, c))

Matching a defined type to a given term is similar, but data flows in the opposite direction. We assume that the term is given in the analysis variable t = Res(d, c), which in other situations is associated with the result.

Match(d, c):
1: Let the definition of d be $d(T_1, \ldots, T_n) = T_0$
2: Let $x_1, \ldots, x_m$ be the free variables of $T_0, \ldots, T_n$
3: Create type environment $\mathcal{A}$ mapping each of x_k to a term Univ(k)c, for $0 \le k \le m$.
4: Let t = Res(d, c)
5: let Match($T_0, t, \mathcal{A}$)c
6: Let t_k = Gen($\mathcal{A}, T_k, c$), for $1 \le k \le n$
7: and Add(t_k, Arg(d, k, c))

The initial task in the work list will be a checking of a specification. For a function f, we assume that there is both a function definition and a specification. We generate terms for the arguments according to the argument types in the specification, then use the flow analysis to determine the result of a function call, and then match the result against the result type in the specification.

$\mathsf{CheckSpec}(f, c)$:

1: Let the specification of f be $f(T_1, \ldots, T_n) \to T_0$ and the
2: definition $f(x_1, \ldots, x_n) \to E$
3: Let $y_1, \ldots, y_m$ be the free variables of $T_0, \ldots, T_n$
4: Create type environment $\mathcal{A}$ mapping each of x_k to a term $\mathsf{Parameter}(k, c)$, for $0 \leq k \leq m$.
5: Let $t_k = \mathsf{Gen}(\mathcal{A}, T_k, c)$, for $k \leq n$,
6: and create environment $\mathcal{E}$ mapping each of x_k to t_k.
7: Let $t = \mathsf{Analyze}(E, \mathcal{E}, c)$
8: If $\mathsf{Match}(T_0, t, \mathcal{A})c$ fails, report a warning to the user.

If a function f is called, and there is no definition of the function, i.e., there is no Erlang code, but there is a specification, we use the specification to comute the expected result. We first match the argument types in the specification against the arguments in the call. Next we generate the result of teh function call according to the result type in the specification.

$\mathsf{UseSpec}(f, x)$:

1: Let the specification of f be $f(T_1, \ldots, T_n) \to T_0$
2: Let $x_1, \ldots, x_m$ be the free variables of $T_0, \ldots, T_n$
3: Create type environment $\mathcal{A}$ mapping each of x_k to a term $\mathsf{Univ}(k)c$, for $0 \leq k \leq m$.
4: For $1 \leq k \leq n$ do $\mathsf{Match}(T_k, \mathsf{Arg}(f, k, c), \mathcal{A})c$,
5: Let $t = \mathsf{Gen}(T_0, \mathcal{A}, c)$ and
6: $\mathsf{Add}(t, \mathsf{Res}(d, c))$.

6. THE IMPLEMENTATION

The analysis is written in Erlang. As Erlang is (apart from the concurrency primitives) a pure functional programming language and lacks arrays and hash tables, the store is represented as a balanced binary search tree.

6.1 Modules

The analysis is applied to a single Erlang module. All exported functions are entry points, and their arguments are assumed to be the universal type (any). The analysis has information about the return types of built-in functions and functions in the standard library `math`. For some benchmarks, the analysis is provided with the source code of some external modules. Calls to other modules are assumed to return the universal type.

6.2 Core Erlang

Even though Erlang may on the surface appear to be a simple language, it is not completely straight-forward to write a front end which handles all aspects of the Erlang language. To avoid dealing with these details, the analysis instead operates on the Core Erlang intermediate code [4]. The translation is performed using the front end of the OTP distribution.

In the translation to Core Erlang, all primitive operations (for example, arithmetic) will appear as function calls. Also, the translation adds a clause with a call to 'exit' (which generates an exception) to each case statement, thus making make the exceptions thrown when a case expression fails to find a matching clause explicit. This means that even fairly simple Erlang functions may contain several calls to built-in functions. The computation of weights in static limiting treats these calls as any other calls, the result is that each function will be assigned a greater weight. This is not unreasonable, as the weight is intended to reflect the cost of analyzing a function polymorphically, and functions containing many built-in calls will be more expensive to analyze. Note that, as the weight of each function is greater, the choice of the parameter p is affected.

7. EXPERIENCES

In this section we study the performance of the type system when applied to Erlang modules that were not written to conform to the type system. All measurements were made on an Intel Xeon 2.4 GHz with 1 GB of RAM and 512 KB cache, running Linux. The size of each module is given in number of lines of code, excluding blank lines and comments.

7.1 The lists module

We first consider the standard module `lists` that defines various lists operations. The module contains 595 lines of code. Many functions, for example `append/1` (which appends a list of lists), `append/2`, `map/2` and `foldl/3` resemble those found in the standard libraries of other functional programming languages. The `lists` module also defines operations on deeply nested lists (for example `flatten/1` and `flatlength/1`), and operations on lists of tuples (for example `keysearch/3` and `keysort/2`). Operations on lists of tuples typically takes an integer as argument, indicating on which element in the tuples to index. Thus, `keysearch/3` looks for tuples with a given element in the position given by the index, and `keysort` sorts a list of tuples with respect to the values stored in a given position in the tupels. Instead of having an explicit string representation, Erlang represents strings as lists of character codes. The function `concat/1` takes a deeply nested list of objects and returns a string consisting of the concatenation of the string representations of the objects. The soft-typing system gives 30 warnings when checking the list module.

The type definitions are straight-forward.

```
-type list(X) = [] + [X | list(X)].
-type deeplist(X) = list(deeplist(X) + X).
```

Many functions are analyzed without warnings. This holds for typical functions on lists such as `append/1`, `append/2`, `reverse/1 map/2` and `foldl/3`. For example, the system derives the following information for `append/1`:

```
Function lists:append/1:

'{result,{fnid,lists|...}}' = A
'{arg,1,{fnid|...}}' = B

where
A = [parameter('A')|A] + []
B = [A|B] + []
```

As one example of a function which correctly generates a warning, consider `nth/2`, which takes an integer and a list,

and returns the element of the list indicated with the integer (where the first element has index one):

```
nth(int(), list(A)) -> A.

nth(1, [H|_]) -> H;
nth(N, [_|T]) when N > 1 ->
    nth(N - 1, T).
```

The system warns that the function may throw an exception. This is correct, since `nth/2` may indeed throw an exception when the index is out of range. The same holds for all functions that index on lists or tuples (`keysearch/3` is one example of the latter).

Surprisingly, checking `flatten/1` gives no false alarms. The derived information states (correctly) that flatten takes a deeply nested list of some data type and returns a list of that data type.

```
Function lists:flatten/1:

'{result,{fnid,lists|...}}' = A
'{arg,1,{fnid|...}}' = B

where
A = [parameter('X')|A] + []
B = [C|B] + []
C = parameter('X') + [C|B] + []
```

Higher-order functions such as `map/2` and `foldl/3` pose no problems.

The function `concat/1` takes a list of Erlang terms and builds a string representation of the terms.

```
concat(List) ->
    flatmap(fun thing_to_list/1, List).

thing_to_list(X) when integer(X) ->
    integer_to_list(X);
thing_to_list(X) when float(X)   ->
    float_to_list(X);
thing_to_list(X) when atom(X)    ->
    atom_to_list(X);
thing_to_list(X) when list(X)    ->
    X. %Assumed to be a string
```

Its type is simply

```
concat(deeplist(int() + float() + atom() +
       string())) -> string().
```

Typing this function gives two warnings that are due to limitations of the analysis:

```
Function lists:thing_to_list/1
Bad argument 1 in call to erlang:atom_to_list/1
Bad argument 1 in call to erlang:float_to_list/1
Exception:
{'EXIT',{function_clause,atom() + float(any())}}
```

The reason for these warnings is that the analysis does not handle guard tests `float(X)` and `atom(X)` (but for the tests `integer(X)` and `list(X)` the analysis correctly infers that if the guard succeeds, `X` must be an integer (or list), and if it fail `X` must be something else). Extending the analysis to handle guards testing for floating-point values and atoms should be straight-forward,

7.2 Othello

A program downloaded from the Erlang user's contributions directory which plays the game Othello. It consists of three modules; `othello`, the main module which (among other things) implements alpha-beta search, `othello_adt` which implements the board as an abstract data type, takes care of evaluation, computing the of possible moves and so on. The module `othello_board` interfaces with a GUI library.

Specification files were written for the three modules. We will only consider the checking of two modules; `othello` and `othello_adt`, as checking `othello_board` would require writing a specification file for the `pxw` graphics library.

As examples of type declarations in the specification files, consider the representation of the board. The board is represented as a pair, where the second element is a 64-element tuple, representing the contents of the board, and the first element is a list of all positions which are empty and adjoin an occupied position. (`ordsets` is a library module which represents sets as lists.)

```
-type boolean() = true + false.
-type list(X) = [] + [X | list(X)].

-public_type color() = grey + white + black.

-public_type player_color() = white + black.

-type squares() =
    tuple(list(color())).

-abstype board() =
    {ordsets:set(int()), squares()}.
```

In the above, `color()` is public, as other modules in the program also rely on the same representation of colors. The type `squares()` represents the contents of the board. The type constructor `tuple(...)` represents, when applied to a list type, the set of all tuples with the same length and type as the lists types. The type `squares()` is the set of all tuples where the elements are the atoms `grey`, `white` and `black`. The type `board()` is declared as abstract as other modules do not access the internal representation of boards. The type `player_color()` represents the color of a player.

Next we consider some of the function specifications.

```
new(t) ->
    board().

all_pos(board()) ->
    list({int(), color()}).

evaluate_board(player_color(), board()) ->
    int().

inv(black+white) ->
    player_color().

is_draw(int(), player_color(), board()) ->
    boolean().

possible_draws(player_color(), board()) ->
```

```
    list(int()).

set(int(), player_color(), board()) ->
    board().
```

The function `new/1` creates a new board. (For some reason, it expects the atom `t` as argument.) `all_pos()` takes a board and returns a list of pairs, consisting of a position and the color at that position. As one might expect, the function `evaluate_board/2` evaluates the board from a player's point of view. `is_draw/3` checks if a particular move is legal, `possible_draw/2` returns the set of possible moves, and `set/3` returns the updated board after one of the players has put a piece on a given position.

Checking the module `othello_adt.erl` against the specification took 11 seconds. This is rather long, considering that the module is only 345 lines of code. Other modules of similar size took less than one second to check. At the time of writing, the author does no have a satisfactory explanation to why checking module `othello_adt.erl` requires so much time.

The type system gives 5 warnings. Three of the warnings are due to the use of the catch/throw mechanism of Erlang to break out of a recursion. For example, the function which checks if a move is legal throws an exceptions when it discovers that the requirements for a legal move are satisfied. The analysis makes no attempts to track exceptions, and conservatively assumes that a catch expression may return any value, thus we have two warnings because a result obtained through a catch is too general, and one warning because a function throws an exception.

One function expects a pair of two integers in the interval $1 \ldots 8$, representing row and column of the board. Because of the way these integers are computed, and since the analysis does not reason about integer ranges, the analysis fails to determine that the integers lie in the desired interval and warns that the function might throw an exception (as it might if called with integers outside the interval).

Finally, the system warns that the function `set/3` may return the result `invalid_position`. (It seems that this could only happen if there is a bug in the program.) The warning is completely correct, but as the functions in module `othello` that call `set/3` do not handle the result `invalid_position` it seemed appropriate to exclude the result from the return type.

The module `othello` has a much simpler interface. The complete specification file is shown below.

```
-module(othello).
-type init() = first_time + restart.

start1(any(), othello_adt:player_color(),
       othello_adt:player_color(), int(),
       init()) ->
    any().


new_game(othello_adt:player_color(),
         othello_adt:player_color(), int(),
         init()) ->
    any().
```

Checking module `othello`, 173 lines of code (excluding blank lines and comments) takes 0.1 seconds. The systems gives 7 warnings. 5 warnings are due to the use of catch-throw in the implementation of the alpha-beta search algorithm. One warning is because the system fails to determine that when the following function is called (with arguments `Value`, `Alpha`, `Beta` and `NoDs` of integer type) one clause will always be selected.

```
cutoff(...,Value,...,Alpha,Beta,...,Nods)
    when Value >= Beta ->
    ...
cutoff(...,Value,...,Alpha,Beta,...,Nods)
    when Alpha < Value, Value < Beta ->
    ...
cutoff(...,Value,...,Alpha,Beta,...,Nods)
    when Value == Alpha, NoDs < 13 ->
    ...
cutoff(...,Value,...,Alpha,Beta,...,Nods)
    when Value =< Alpha ->
    ...
```

(Variables not relevant for the case analysis were omitted.) To analyze this function correctly would require some rather advanced reasoning about integer ranges.

The final warning occurs in the expression

```
case random:uniform(2) of
    1 -> ...;
    2 -> ...
end
```

The library function `random:uniform/1` will indeed return either 1 or 2, but the analysis does not have such detailed knowledge about the function, nor can it reason in such detail about integer ranges,

7.3 Lines

Lines is a module downloaded from the Erlang user contributions directory. The module implements an abstract data type which maintains an abstract sequence of lines, i.e., objects indexed by their relative position in the structure. The module allows insertion and removal of a line anywhere in the sequence, and as the sequence is represented as a binary tree, operations should usually be logarithmic in the depth of the tree.

The lines module is 164 lines of code, excluding blank lines and comments. Checking the module required 0.6 seconds. Even though the module implements a relatively simple data structure, the soft-typing system generates 17 warnings.

Consider, for example `insert/3`, which performs insertion at a given position and tries to re-balance the tree. Type-checking this function gives 6 warnings. Three warnings are due to the handling of if-expressions. Due to what appears to be a bug in the Core Erlang front end, a guard `true` is translated into a call `erlang:'=:='(true, true)`. The analysis knows that the former expression will always succeeds, but assumes conservatively that the latter expression may either succeed or fail. Thus the soft-typing system issues warnings that the if-expression may throw an exception.

One warning is due to the fact that insert may give an exception if the index is out of range.

The remaining two warnings are due to the fact that the analysis assumes that an index may be out of range, even though a preceeding test ensures that this is not the case.

7.4 Gb-trees

The standard library module `gb_trees` implements balanced binary search trees. As it was originally developed as part of the soft-typing system, the author had expected that it would pass through the type checker without any warnings at all. However, the type checker gives 9 warnings.

The tree type is abstract. We give the specification of the tree type and the function `insert/3`,

```
-abstype tree(K, V) = {int(), tree1(K, V)}.

-type tree1(K, V) =
    nil + {K, V, tree1(K, V), tree1(K, V)}.

insert(K, V, tree(K, V)) ->
        tree(K, V).
```

To allow an efficient implementation of the balancing algorithm each tree stores a count of the number of nodes in the tree.

The `gb_trees` module consists of 202 lines of code, checking the module took 0.4 seconds. As mentioned, the system will report 9 warnings. Among them, four warnings concern exceptions that may actually take place. For example, `insert/3` assumes that the key being inserted is not present in the tree, `update/3` assumes that it *is* present, and so does `delete/2`. The function `take_smallest/1` assumes that the tree is non-empty. In other words, the functions are written to throw exceptions when called with the wrong data, and the type checker warns us that this might indeed happen.

The other five warnings concern the passing of values between different functions in the module. For example, the function `insert/3` calls an internal function `insert_1` which will either return a one-element tuple containing a balanced tree, or a tuple of three elements containing an unbalanced tree plus some additional information. Now, the properties of the algorithm guarantees that `insert_1` when applied to a complete tree will always return a balanced tree, so the call to `insert_1` is written so that an exception will be thrown if this is not the case. As the type system does not understand the algorithm, it warns that there may be an exception. Similarly, there is one function call in the balancing algorithm which should always return a tuple where the second element is an empty list. The program is deliberately written so that an exception will be thrown if this is not the case. The remaining three warnings are similar; we know that a particular intermediate value should be of a particular form, so we check that this is indeed the case—there is no point in handling any other situation gracefully as it must be due to a bug.

7.5 Counter

As an example of a program with process communication, we consider the simple counter process from [2].

```
start() ->
    spawn(counter, loop, [0]).

increment(Counter) ->
    Counter ! increment.

value(Counter) ->
    Counter ! {self(), value},
    receive
        {Counter, Value} ->
            Value
    end.

stop(Counter) ->
    Counter ! stop.

loop(Val) ->
    receive
        increment ->
            loop(Val+ 1);
        {From, value} ->
            From ! {self(), Val},
            loop(Val);
        stop ->
            true;
        _ ->
            loop(Val)
    end.
```

The function `start()` spawns a process which executes the function `loop()`. Three functions handle communication with the process; `increment/1`, `value/1` and `stop/1`.

The specification file is as follows:

```
-module(counter).

start() ->
    pid().

increment(pid()) ->
    increment.

+mbox(value/1) = {pid(), int()}.

value(pid()) ->
    int().

stop(pid()) ->
    stop.

+mbox(loop/1) =
    increment + {pid(), value} + stop.

loop(int()) ->
    true.
```

First, the specification of `start/1` indicates that it will return a process identifier. The function `increment` is specified to take a pid as argument and return the atom `increment`. Similarly, the functions `value` and `stop` take a pid as argument and return an integer and the atom `stop`, respectively. Since `value` contains a receive expression in which it expects a message consisting of a tuple of a pid and an integer, we indicate this in the specification. Similarly, we indicate the messages that the function `loop` receives.

Checking this module gives no warnings. Still, there are many ways to use the module (without breaking the specification) that may give strange results. Consider, for example

```
f() ->
    P = counter:start(),
    self() ! {P, foo},
    V = counter:value(P).
```

This function function creates a counter process, and then asks for its current value. However, before it asks the question it forges a reply from the process. Now, the function `value` will pick up the first message in the mailbox and return the value `foo`.

It is unlikely that the above code would occur in practise, but the example shows that the process types do not guarantee safety.

8. DISCUSSION

There are a number of aspects of the Erlang programming language that are hard to combine with static typing; for example, the fact that all complex data structures are constructed using lists and tuples makes it difficult to distinguish objects of different types.

Statically typed programming languages are of course designed to facilitate static typing. Further, since the type system is integrated in the implementation, a programmer is forced to obey the discipline of the type system—any program that does not type won't compile. In a language with dynamic typing this discipline is missing—it is o perfectly possible to run a program that exhibits some inconsistent use of types For example, the Othello program discussed in Section 7 has a function that either returns an integer if an operation is successful or the atom `false` otherwise. In Erlang, this is completely unproblematic, but in, say, SML such a program would have to be rewritten.

It is also worth noting that static and dynamically typed programming languages show difference in programming style; in a dynamically typed programming language one writes code so that the presence of a bug will cause it to crash as soon as possible—the sooner one discovers that an input or an intermediate result is incorrect the better. For example, if we expect that a function should return a non-empty list, make sure that any other result gives an exception. Paradoxically, making the program crash sooner makes it more robust, as it gives us more opportunities to detect bugs in the program. The use of this technique compensates to a certain degree for the lack of early detection of errors provided by static typing. (Also, it seems that implementations of dynamically typed programming languages do a better job in providing useful debugging information when the program throws an exception.) The difference in programming style makes it difficult to add static typing to a program written for dynamic typing—precisely what we do in a language with dynamic typing to make the programs more robust also make them harder to type.

Another problem is that adding static types to a program written for a language with dynamic typing requires some understanding of the program, especially if one is to modify the program to fit the type system. (On other hand, sometimes the type system helped reveal the inner workings of programs.)

One argument against adding static types in retrospect is that if the programs have already been debugged and tested, the chances that the type system will discover any remaining bugs are slim. In the experiments reported in Section 7, and in other attempts to to type previously written programs, the author never found any bugs due to type errors. (However, applying the type system to simple toy programs that had not been tested sometimes revealed accidental type errors.)

One of the main motivations for using a static type system is that it discovers bugs. However, as careful testing tends to reveal these bugs, it seems that best way to use a static type system is to apply it before the program is tested. Also, it may happen that a program has to be re-written to fit the type system. This also supports the view that static type systems should be applied early in the development process, perhaps as an integrated part of the compiler. Findler et al. [7] appear to have drawn similar conclusions. They describe a development system for Scheme, a dynamically typed programming language, with an integrated static type system.

9. CONCLUSION

We have presented a soft-typing system for Erlang. The system is based on two ideas—use a specification language to give the interface of each module, and use a data flow analysis to verify that the implementation of the module matches the specification.

As we saw in the experimental section, the system can reason about substantial programs and produce useful results. It is worth noting that even though the programs were debugged and tested, the type system still produced warnings. The warnings typically concerned programming constructs that the data flow analysis could not analyze precisely (and where one would not expect any other static typing system to give better results).

One disappointing result is that the experiments did not uncover a single bug in programs that had already been tested and debugged. Perhaps this is an indication that careful testing tends to reveal most type errors.

10. REFERENCES

[1] Alexander Aiken. Illyria demo, May 1995.

[2] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.

[3] Thomas Arts and Joe Armstrong. A practical type system for Erlang. In *Erlang User Conference*, September 1998.

[4] R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S. Nyström, M. Pettersson, and R. Virding. Core Erlang 1.0 language specification. Technical Report 030, Information Technology Department, Uppsala University, November 2000.

[5] Robert Cartwright and Mike Fagan. Soft typing. In *PLDI*, pages 278–292, 1991.

[6] Fabien Dagnat and Marc Pantel. Static analysis of communications for erlang. In *Proceedings of 8th International Erlang/OTP User Conference*, 2002.

[7] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. Drscheme: a programming environment for Scheme. *Journal of Functional Programmin*, 12(2):59–182, March 2002.

[8] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *PLDI*, volume 31 of *ACM SIGPLAN Notices*, pages 23–32, 1996.

[9] N. Heintze. Set-based analysis of ML programs. In *ACM Conference on Lisp and Functional Programming*, pages 306–317, 1994.

[10] Frank Huch. Verification of Erlang programs using abstract interpretation and model checking. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, pages 261–272, September 1999.

[11] Anders Lindgren. A prototype of a soft type system for Erlang. Master's thesis, Uppsala University, April 1996.

[12] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. *ACM SIGPLAN Notices*, 32(8):136–149, August 1997.

[13] Peter Norvig. *Paradigms of artificial intelligence programming: case studies in Common LISP*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1992.

[14] Sven-Olof Nyström. A polyvariant type analysis for Erlang. In preparation.

[15] O. Shivers. Control flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, 1988.

[16] Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. *ACM Transactions on Programming Languages and Systems*, 19(1):87–152, January 1997.