A polyvariant type analysis for Erlang

Sven-Olof Nyström

Department of Information Technology,
Uppsala University, Sweden
svenolof@csd.uu.se

Abstract. This paper presents a type analysis for the programming language Erlang. The analysis computes interprocedural control-flow and data-flow information, and should be applicable to any higher-order functional programming language with call-by-value semantics. The analysis uses a novel method for polyvariance, *static limiting*, where an approximation of the call graph is analyzed to determine whether a function should be treated as polyvariant or monovariant. A general framework for polyvariant analysis is presented. This framework is used for experimental investigations to evaluate the cost and potential benefits of polyvariant analysis and to compare different apporoaches to polyvariant analysis. The experimental results show that static limiting gives the same or better precision as the other polyvariant analyses, while having more predictable analysis times. However, the experiments show only small improvements in precision for the various polyvariant analyses.

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 [11]). 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.

One disadvantage with dynamic typing is that its implementation usually requires a large number of run-time type checks. Every primitive operation must check that its arguments are of the intended type. Absence of static type information also complicates the implementation of various compiler optimizations.

This paper presents a data flow analysis which determines an approximation of the possible types of variables and expressions in the program. In the paper we focus on one aspect of the analysis, that it is *polyvariant* (context-dependent). It has long been recognized [13, 14] that an interprocedural program analysis will

obtain better precision if it distinguishes different calls to a function. We present a polyvariant analysis that uses a novel technique, *static limiting*, to control the cost of polyvariance.

The work presented here can be seen as a generalization of monovariant analysis techniques such as 0CFA [14] or set-based analysis [8]. Shivers also proposed a simple polyvariant analysis, kCFA, which distinguished calls based on the k most recent call sites.

Cartwright and Fagan's soft-typing system [5] tried to apply type inference to dynamically typed languages, in particular Scheme. The system was intended to serve two purposes, to help the programmer by detecting possible programming errors and to help the compiler by providing type information. Marlow and Wadler's soft-typing system for Erlang [10] had a different goal. Instead of typing all Erlang programs, their system refused programs it could not type. Thus, it defined, in effect, a new programming language consisting of those Erlang programs that it could type.

Other work on type analysis for functional programming languages include polyvariant analyses by Ashley and Dybvig [3] and Wright and Jagannathan [16].

2 Polyvariant analysis

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 *monovariant* or *context-insensitive* analysis) suffers from the problem that if a function is called with 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 a function where the result depends on its argument (perhaps it simply returns the argument). If there is one call site where the type of the argument is unknown, the result will also be unknown. Thus, a monovariant analysis may propagate a low-precision result where a polyvariant analysis would confine it to one part of the program.

Generally speaking, a polyvariant analysis will be more expensive since some functions are analyzed many times. In theory, a monovariant analysis might take longer to compute since time is spent propagating an over-approximation.

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'. Within this simple framework it is possible to express analyses with a wide range of precision and efficiency.

One choice is to set $\mathit{Context} = \{c_0\}$, i.e, use a minimal set of contexts, and to define Call as

$$\mathsf{Call}(f, l, c_0) = c_0,$$

for all f and l. This will give us a monovariant analysis.

One simple way to get a polyvariant analysis is to set $\mathit{Context} = \mathit{Lab} \cup \{\text{start}\}$, i.e., let the contexts be the set of labels plus one additional element to be used in the beginning of the analysis. With the definition

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

this would give us a very limited form of polymorphism.

The reason for polyvariant analysis is that we want to keep unrelated calls separate, but there must be some limit on the number of contexts generated, to guarantee termination, and to make sure that the number of contexts does not become excessive for large programs. In the following section, we will discuss a number of techniques for controlling the number of contexts.

2.1 k-limiting

This is an adaption of the technique proposed by Shivers [14]. There are some differences due to the differences in programming languages (for example, Shivers assumes that the program is in continuation-passing style).

In a computation, each invocation of a function can be identified by a sequence of call sites (labels). Thus, by letting the set of contexts consist of arbitrary sequences of labels we would be able to treat each invocation of a function separately. The problem is that the set of contexts would be infinite, and thus the analysis would not terminate! Shivers' solution to this problem is to set a maximum length (k) to sequences of call sites. This gives us

$$\mathit{Context} = \{l_1 \dots l_n \mid n \leq k\}$$

and

$$\mathsf{Call}(f, l, l_1 \dots l_n) = \begin{cases} l l_1 \dots l_n, & \text{if } n < k \\ l l_1 \dots l_{n-1}, & \text{if } n = k \end{cases}$$

By limiting the size of a context to k, we can guarantee that the set of contexts is finite. However, since the number of contexts is exponential in k, we are forced to settle for rather small values of k. Also, it is unclear that the strategy of keeping the most recent part of the call stack and forgetting the earlier parts is the right one. For example, if a function contains a single recursive call (at label l), there will be a context $l \dots l$ corresponding to the situation after at least k recursive calls. If the function is called from many different parts of the program, we will always arrive at the same context after considering k recursive calls.

2.2 Dynamic detection of recursion

In a function with a recursive call, the arguments given in the recursive call tend to have the same type as the arguments in the first call to the function. Thus,

it would make sense to evaluate the two calls in the same context. If we let the contexts include, for each call site, information on which function is called

$$\mathit{Context} = \{c \in (\mathit{Lab} \times \mathit{Function})^* \mid \text{each } f \in \mathit{Function} \text{ occurs at most once in } c\}$$

and define Call to look for the previous call to the same function

$$\mathsf{Call}(f, l, c) = \begin{cases} \langle f, l\rangle\, c, & \text{if } f \text{ does not occur in } c \\ c', & \text{if } f \text{ occurs in } c \text{ and } c' = \langle f, \ldots\rangle \ldots \text{ is a suffix of } c \end{cases}$$

we guarantee that the set of contexts is finite.

2.3 Static detection of recursion

What about mutually recursive functions? There are some situations where it is natural to express an algorithm as a set of mutually recursive functions. A state machine can be implemented with one function for each state, and a tail-calls between functions for each transition. Recursive-descent parsers will typically contain systems of mutually recursive functions (depending on the grammar, of course). Similarly, any tool that traverses syntax trees is likely to contain systems of mutually recursive functions.

A system of mutually recursive functions might induce a large number of contexts. Suppose, for example, that there is a system of n mutually recursive functions, and that each function definition contains a call to each of the other functions. Now, any of the $n!$ possible orderings of the functions may occur in a context, giving an analysis with exponential complexity.

The problem can be avoided by building a syntactic call graph of program-defined functions, such that there is an edge from function f_1 to function f_2 if the body of f_1 contains an call to f_2. Compute the strongly connected components (sccs). Each set of mutually recursive functions will form an scc. With SCC equal to the set of strongly connected components,

$$\mathit{Context} = \{c \in (\mathit{Lab} \times \mathit{SCC})^* \mid \text{each } s \in \mathit{SCC} \text{ occurs at most once in } c\}$$

and letting Call check if a recursive call is to the same scc or not, we improve the behavior in the case of mutual recursion. In the definition below, let s_f be the strongly connected component containing f.

$$\mathsf{Call}(f, l, c) = \begin{cases} c, & \text{if } c = \langle s_f, l\rangle\, c', \text{ for some } c' \\ \langle s_f, l\rangle\, c, & \text{otherwise} \end{cases}$$

By using static detection of recursion, we eliminate polyvariance within sets of mutually recursive functions. A similar situation can be found in programming languages such as ML, where syntactic restrictions guarantee that calls between mutually recursive functions can not be polymorphic. Note that even though static detection of recursion removes one source of exponential growth, the number of contexts may still be exponential in the size of the program, for example, if there is a chain of functions f_0, f_1, $\ldots$ f_n such that each function has two calls to the next function in the chain.

2.4 Static limiting

Note that the strongly connected components form a directed acyclic graph (DAG), with sccs as nodes, and an edge $s_0 \xrightarrow{l} s_1$ between two nodes if there some function f contains a call at label l to a function g, and f is a member of s_0 and g of s_1.

If we consider computations starting with a function f, the contexts created by an analysis correspond to the set of paths in the DAG starting at s_f. Since the number of paths starting at any given node in the graph can be computed statically, we can set a limit to the number of contexts generated. This brings us to the idea of static limiting.

In the following, all definitions are with respect to a directed acyclic graph $G = \langle N, E \rangle$. A *path* q in G is a sequence $q = s_0 l_0 s_1 l_1 \ldots s_{n-1} l_{n-1} s_n$ such that that $s_i \xrightarrow{l_i} s_{i+1}$, for $i < n$. Let the *weight* of a node s be the number of paths in G starting with s. Say that a node s is *polyvariant*, if its weight is less than a threshold p, monovariant otherwise. Also, an edge $s_0 \xrightarrow{l} s_1$ is polyvariant if its destination s_1 is polyvariant. Similarly, say that a function is polyvariant if it is a member of a polyvariant node.

We define the contexts as paths in the subgraph of polyvariant nodes;

$$\begin{aligned} \mathit{Context} = \{ s_0 l_0 s_1 l_1 \ldots l_{n-1} s_n \mid & \text{ where } s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} \ldots \xrightarrow{l_{n-1}} s_n \\ & \text{ and all } s_k \text{ are polyvariant, for } k \leq n \} \end{aligned}$$

$$\mathsf{Call}(f, l, c) = \begin{cases} s_f, & \text{if } f \text{ is monovariant} \\ c, & \text{if } c = s_f \ldots \\ s_f l c, & \text{otherwise} \end{cases}$$

The use of polyvariant analysis implies that the analysis problem grows larger, as some parts of the program are analyzed under different contexts. How much does the analysis problem grow? It turns out that the resulting analysis problem grows with at most a constant factor.

The expanded DAG $X(G) = \langle N_X, E_X \rangle$ is constructed with

$$\begin{aligned} N_X &= \{ q \in \mathrm{path}(G) \mid \text{ each edge of } q \text{ is polyvariant} \} \\ E_X &= \{ \langle q, l, q' \rangle \mid l \text{ is polyvariant and } q' = qls, \text{ for node } s \text{ and edge } l \text{ in } N \} \\ &\quad \bigcup \{ \langle s, l, s' \rangle \mid l \text{ is a monovariant edge } s \xrightarrow{l} s' \text{ in } G \} \end{aligned}$$

Let the *size* of a graph be the total number of nodes and edges.

Theorem 1. *For any directed acyclic graph G,* $\mathrm{size}(X(G)) \leq 2 * p * \mathrm{size}(G)$.

Proof. The proof is by induction on the size of G.

Suppose $\mathrm{size}(G) = 0$. Since there are no nodes in G, it follows immediately that $X(G)$ has no nodes and thus no edges. Since $\mathrm{size}(X(G)) = 0$, the theorem follows immediately.

Suppose $\mathrm{size}(G) > 0$. G must contain at least one node. Since G is a DAG there must be one node r which is the successor of no other node. We have three cases.

1. r has no successors.
2. There is an edge $r \xrightarrow{l} s$ such that s is monovariant.
3. There is an edge $r \xrightarrow{l} s$ such that s is polyvariant.

Case 1 is straight-forward. Consider the graph G' obtained by removing the node r from G. Since r occurs in only one path of G, we see that there is only a single node of $X(G)$ that is not a node of $X(G')$. All edges of $X(G)$ are also edges of $X(G')$. Thus $\text{size}(X(G)) = \text{size}(X(G')) + 1 \leq 2p * \text{size}(G') + 1 = 2p * (\text{size}(G) - 1) + 1 \leq 2p * \text{size}(G)$.

Case 2. Consider the graph G' obtained by removing the edge l from G. Since l is monovariant, the nodes of $X(G')$ are exactly the nodes of $X(G)$. The only edge of $X(G')$ that is not a node of $X(G)$ is $\langle r, l, s\rangle$. The theorem follows by an application of the induction hypothesis.

Case 3. Again, let G' be the graph obtained by removing l. Since s is polyvariant, there are at most p paths starting with s. Thus, there are at most p paths starting with $rls\ldots$ in G. Since r has no predecessors, all paths containing l are of this form. It follows that there are at most p nodes of $X(G)$ that are not nodes of $X(G')$. The edges of $X(G)$ that do not occur in $X(G')$ are all of the form $\langle q, l', q'\rangle$, where q' is a path containing l. Since there are at most p such paths, $\text{size}(X(G)) \leq \text{size}(X(G')) + 2p$. By the induction hypothesis, $\text{size}(X(G')) + 2p \leq 2p * \text{size}(G') + 2p = 2p * (\text{size}(G) - 1) + 2p = 2p * \text{size}(G)$.

3 The analysis

To make the analysis more uniform, 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 assume two nullary constructors 'true' and 'false'. We also assume a set of predefined 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

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

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.

3.1 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 polyvariant, 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 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.

Let Work, the set of analysis tasks, be the set of pairs $\langle f, c \rangle$, where f is a program defined functions and c is a context. The analysis will maintain a work list, containing a subset of Work.

3.2 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.

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 re-analyzed 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.

Lookup(X):
1: Add current task to X.depend
2: Return X.value

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

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 Add(t, X)

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

Fig. 1. Implementation of set abstraction.

3.3 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.

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

Expressions involving a constructor simply build a corresponding term.

Analyze($C[E_1, \ldots, E_n], \mathcal{E}, c$):
1: let $t_k =$ 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.

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

Analyze(`if` E_1 `then` E_2 `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 $\mathsf{Res}(f, c')$

$\mathsf{Analyze}(\mathtt{fun}^l\ (x_1, \ldots, x_n) \to E_0, \mathcal{E}, c)$:
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 $\mathsf{ApplyRes}(l, c)$

Fig. 2. Analyzing expressions.

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 Section 3.4 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) $\mathtt{fun}^l\ (x_1, \ldots x_n) \to E_0$, 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 values 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.

The main loop maintains a work list of all function-context pairs that need to be analyzed. Since the arguments are passed in the store (associated with the context), the task only needs to contain function and context.

MainLoop:

1: if WorkList is empty,
2: terminate analysis
3: if WorkList is not empty,
4: remove a pair of a program-defined function context $\langle f, c\rangle$ from WorkList,
5: let $\langle f, c\rangle$ be the current task, and
6: Analyze(f, c)
7: continue MainLoop

AnalyzeProgram:

1: Let n be the arity of f_e.
2: Add(any, Arg(f, k, c)), for $k \leq n$.
3: Put $\langle f_e, c_0\rangle$ in WorkList.
4: Execute MainLoop.

Analyze$(f(x_1, \ldots, x_n) \to E, c)$:

1: Create environment $\mathcal{E}$ mapping each of X_k to the term Arg(f, k, c), for $k \leq n$
2: let $t =$ Analyze$(E, \mathcal{E}, c)$
3: Add$(t,$ Res$(f, c))$

Fig. 3. Main loop of analysis.

3.4 Polyvariant analysis of higher-order functions

In the analysis presented in this paper, the analysis of calls to closures is monovariant. In this section we discuss the changes needed for a polyvariant analysis of closures.

Note that in Section 2, when reasoning about contexts and call graphs, a 'function' is assumed to be a top-level function. We must extend this concept to include closures (which can be identified by their label). The mechanism for dynamic detection of recursion (Section 2.2) will discover if a closure calls itself recursively.

To implement static detection of recursion and static limiting, we need access to a call graph. However, with higher-order functions, we must run the analysis to construct the call graph! One way around this problem is to first use a simple flow analysis to construct the call graph (for example, Shivers' 0cfa).

In the call graph, the nodes will be (top-level) functions and (labels of) closures. For each higher-order function call, there is an edge to each closure that may be called. Now, consider a top-level function containing one or more closures. A call occuring in one of the closures corresponds to one or more edges from the closure. A call in the function body, outside the closures, corresponds to an edge from the function. Computing strongly connected components and determining whether a function or closure should be monovariant or polyvariant is done as previously (Sections 2.3 and 2.4).

We add FunDest(l, c) and CVar(l, k, c) to the set of analysis variables *Var*, for labels l, contexts c, and $k \geq 0$. We also introduce a term FunCall(c). The variable FunDest(l, c) is associated with fun appearing at label l, evaluated in context c. Each term FunCall(c') in FunDest(l, c) represents a request to evaluate the body of the fun in the environment c'. In this evaluation, the values associated with the free variables of a closure will be stored in variables CVar(l, k, c') We must

Analyze($\texttt{fun}^l\ (x_1, \ldots x_n) \to E_0, \mathcal{E}, c$):
1: For each c' such that Contains(FunCall(c'), FunDest(l, c)),
2: Add($\mathcal{E}(y_k)$, CVar(l, k, c')), for $k \leq m$,
3: unless $\langle l, c' \rangle$ has been analyzed before, add $\langle l, c' \rangle$ to work list.
4: return FunTerm(l, c).

Analyze(l, c):
1: Let $\texttt{fun}^l\ (x_1, \ldots x_n) \to E_0$ be the fun expression labeled l.
2: Create environment $\mathcal{E}$ mapping each of x_k to the term Arg(f, k, c), for $k \leq n$, and y_k to CVar(l, k, c), for $k \leq m$
3: let t = Analyze($E_0, \mathcal{E}, c$)
4: Add(t, FunRes(f, c))

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

Fig. 4. Polyvariant analysis of higher-order functions.

also introduce a new class of analysis tasks, associated with the evaluation of a closure. We will write those $\langle l, c \rangle$.

In the polyvariant analysis a closure should be analyzed once for each context in which it is called. Thus, the analysis of a fun expression looks for terms FunCall(c'), indicating a call to the fun in context c. The analysis rules are given in Figure 4. We assume that the free variables of the fun expression are $y_1, \ldots, y_m$.

4 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.

4.1 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. 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.

4.2 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.

4.3 Meta-call

Erlang's meta-call takes three arguments; an atom which gives the module being called, an atom giving the name of the function and a list which is sent as an argument list to the function. In other words, the destination of a meta-call is computed on the fly. This makes it impossible to compute the call graph before the analysis begins, as is required by static limiting. The solution is simply to build the graph with the call destinations available. To avoid problems with infinite recursion, the analysis also implements dynamic detection of recursion.

5 Experimental Results

Table 1 lists the benchmarks used in the measurements. For each benchmark the number of lines is shown (not including comments and blank lines). Barnes solves the n-body problem. It is packaged as a compiler benchmark, so all data is given in the program (the program was modified to only export one function). B2i is a module in the Hipe compiler which translates BEAM code to an internal representation. Eddie is a high availability clustering tool. The eddie+ benchmark was originally intended as a compiler benchmark and consists of an http parser and a set of support modules. Eweb is a tool for Erlang literate programming. The hipe+ benchmark consists of modules from the Sparc backend of the Hipe compiler [12]. Igor is a tool for merging and renaming Erlang modules, downloaded from the Erlang user contributions list. Othello is a Othello-playing program, downloaded from the Erlang user contributions list. Scan is a lexical analyzer for XML, downloaded from the Erlang user contributions list.

In the evaluation, we considered the following settings for polyvariance.

Name	Included modules	Lines
barnes	barnes	180
b2i	hipe_beam_to_icode	1115
eddie	http_parse	335
eddie+	http_parse, http_fields, lists	
	srv_parse, srv_table	1610
igor	igor	1556
igor+	igor, lists	2062
hipe+	hipe_rtl2sparc, gb_trees, hipe_consttab, hipe_gensym	
	hipe_rtl.erl, hipe_sparc, hipe_sparc_registers, lists	4312
othello	othello	173
othello+	othello, othello_board, othello_adt	932
scan	xmerl_scan	2118

Table 1. Benchmark programs used in the evaluation.

- Static limiting with the parameter p (as in Section 2.4) set to 10, 100, and 1000 (sl-10, sl-100, and sl-1000).
- Monovariant analysis (0cfa) and Shiver's polyvariant analysis with k set to 1 (1cfa).
- Dynamic detection of recursion (ddr).
- Dynamic detection of recursion with the size of contexts limited to size 1 (ddr-1).
- Both dynamic and static detection of recursion (dsdr).
- Dynamic and static detection of recursion with contexts limited to 1.

The measurements were done on a dual processor Intel Xeon 2.4 GHz machine with 1 GB of RAM and 512 KB of cache per processor, running Linux. The benchmarks were run under the BEAM byte code interpreter.

Name	sl-10	sl-100	sl-1000	0cfa	1cfa	ddr	ddr-1	dsdr	dsdr-1
barnes	0.6	0.6	0.5	0.3	1.1	0.6	0.6	0.6	0.6
b2i	43.0	50.7	50.9	995.7	*	168.2	108.5	105.4	9.1
eddie	1.5	1.6	1.6	0.7	2.9	5.1	1.4	1.4	1.1
eddiepl	6.0	6.5	6.5	0.9	385.4	23.7	150.5	6.5	151.9
hipepl	20.9	71.3	72.2	2.0	13.6	72.3	12.1	71.7	12.3
igor	6.5	13.9	29.3	1.3	3.7	169.7	3.4	154.6	3.3
igorl	11.5	34.4	122.5	1.6	8.6	753.7	5.0	604.9	4.8
eweb	0.9	1.2	2.3	0.4	2.6	2.3	0.6	2.3	0.6
othello	0.2	0.2	0.4	0.2	0.3	0.5	0.3	0.4	0.2
othello+	8.1	15.3	21.3	5.1	7.8	741.3	7.1	515.5	6.8
scan	32.8	52.3	75.7	2.1	22.6	+	39.8	4118.8	28.9

Table 2. Analysis time for all combinations of benchmarks and polyvariance. All times are in seconds. * - terminated after running for more than one hour, + - ran out of memory.

The timings for all benchmarks is given in Table 2. 0cfa is always the fastest, except in the b2c benchmark. In b2c, 1cfa was run for more than one hour without terminating, and 0cfa was the slowest of the rest. A partial explanation to this rather peculiar behavior is that the module contains a function `mk_label` which is called from more than 20 locations in the module. This function takes an environment as an argument and returns a new environment, and as many functions pass around environments, one might hypothesize that computing transitive closure is easier when using static limiting as the graph becomes more tree-like. To test this hypothesis, 0cfa was modified to allow polyvariant analysis of `mk_label`. This version of the analysis terminated in 37 seconds.

The timings for the polyvariant cfa-based analyses vary greatly, note for example in the case of eddie+ how the limited analyses (ddr-1, and dsdr-1) are between 6 and 20 times slower than their unlimited counterparts. The main module of the eddie benchmark contains a recursive-descent parser, using explicit state (in essence, a continuation) to avoid deep recursion. A partial explanation to the slower performance is that the limited analyses will see more possible intermediate states, and each possible intermediate state will trigger more work for the analysis.

For all other benchmarks the limited versions of ddr and dsdr are faster than their unlimited variants. In the case of the igor, othello and scan benchmarks, the unlimited ddr and dsdr fare particularly badly, and either fail to terminate or require more than 50 times as much time as 0cfa.

In contrast, the analyses based on static limiting have quite predictable execution times. sl-10 is at most 16 times slower than 0cfa, sl-100 is never more than 25 time slower, and sl-1000 is at most 76 times slower than 0cfa. It is interesting to relate this result with Theorem 1, which guarantees that the analysis problem solved by an sl-p analysis will at most be $p*2$ times larger than the corresponding problem solved by 0cfa. The theorem does not make any guarantees regarding analysis time, but the experimental results suggest that a similar property holds for analysis time.

Next we turn to the estimates of precision shown in Table 3. The precision of each analysis was estimated by looking at each function parameter and return value of the exported function of the main modules of each benchmark. These were divided into two categories, *known* and *unknown*. A type was considered to be unknown if it was the union of two or more top-level constructors, or if it was `any`, or if it was a tuple of unknown length. The figures indicate the percentage of parameters and return values with known type.

The difference in precision between monovariant and polyvariant analysis is never very large. For example, in the igor benchmark, monovariant analysis is able to determine the types of 36% of all function parameters and results, while all polyvariant analyses determine the types of 42%. Why is the improvement so small? The fact that the underlying analysis is quite precise means that the advantage of separating calls is smaller. (Also, since the monovariant analysis sets a rather high baseline the room for improvement is smaller.) Most of the values that remain unknown may have been passed from some unknown module,

Benchmark	sl-10	sl-100	sl-1000	0cfa	1cfa	ddr	ddr-1	dsdr	dsdr-1	Positions.
barnes	88.1	88.1	88.1	54.2	88.1	88.1	88.1	88.1	88.1	59
b2i	40.1	40.1	40.1	34.9	*	40.1	40.1	40.1	40.1	212
eddie	32.9	32.9	32.9	32.9	32.9	32.9	34.2	32.9	32.9	161
eddie+	34.2	49.1	49.1	32.9	33.5	49.1	35.4	49.1	34.2	161
hipepl	68.5	68.5	68.5	59.1	68.5	68.5	68.5	68.5	68.5	127
eweb	30.5	30.5	30.5	24.5	26.0	30.5	30.5	30.5	30.5	200
igor	42.0	42.0	42.0	36.5	36.7	42.0	42.0	42.0	42.0	529
igor+	51.0	51.0	51.0	41.4	43.1	51.0	51.0	51.0	51.0	529
othello	22.0	22.0	22.0	16.5	22.0	22.0	22.0	22.0	22.0	109
othello+	42.2	42.2	42.2	31.2	42.2	43.1	42.2	43.1	42.2	109
scan	30.9	33.4	30.9	28.2	30.9	-	30.9	30.9	30.9	482

Table 3. Estimates of precision. Numbers indicate percentage of function parameters and return values in main module whose type could be determined by the analysis. The final column indicates the total number of function parameters and return values examined.

or are due to limitations of the underlying analysis. Also note that we treat a union of two types, for example two different tuples, as an unknown. The fact that the programs analyzed make very little use of polymorphism might also contribute.

It is also worth noting that the difference in precision between different polyvariant analyses is either small or non-existent, so if one wants a polyvariant analysis, one might as well choose the fastest one, sl-10.

6 Related work

Ashley and Dybvig [3] describe an polyvariant analysis where the new context is dependent on the types of the arguments in the call. Thus, two calls to a function will be separated if at least one of their arguments differ in type.

Wright and Jagannathan [16] describe an approach similar to Shivers's kCFA and evaluates its efficiency in two optimizations; elimination of run-time checks and inlining. Their analysis treats a function as polyvariant if it is defined in a surrounding let-expression. Thus, a recursive call is never polyvariant. This syntactic restriction resembles the technique described in Section 2.3 which identifies strongly connected components in the call graph. Emami et al.[6] describe a polyvariant pointer alias analysis which limits the size of contexts through a mechanism which resembles dynamic detection of recursion.

Experimental investigations comparing monovariant and polyvariant pointer analysis for C have shown mixed results. Liang and Harrold [9] compare their polyvariant analysis with a monovariant analysis [1] and find that the polyvariant analysis is comparable in precision but faster. Foster at al. [7] compare polyvariant and monovariant versions of Andersen's and Steensgaard's [15] analyses and find that polyvariance gives a large improvement for Steensgaard's analysis but hardly any improvement at all for Andersen's analysis.

7 Conclusions

We have presented a technique for efficient polyvariant type analysis, static limiting. It is simple in that its implementation only requires small modifications to a monovariant analysis, and robust as it never increases analysis time with more than a constant factor. Two rather surprising results from the experiments are that different polyvariant analyses show approximately the same gain in precision, and that this gain is usually quite small. Still, the use of a polyvariance can be motivated, as it can be implemented with (relatively) little programming effort and often gives a significant improvement in precision. It is difficult to see how a monovariant analysis could ever give the same precision.

References

1. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.
2. Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.
3. J. Michael Ashley and R. Kent Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM TOPLAS*, 20(4):845–868, July 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. M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural Points-to analysis in the presence of function pointers. *SIGPLAN Notices*, 29(6):242–256, 1994. PLDI.
7. J. Foster, M. Fahndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Static Analysis Symposium*, number 1824 in LNCS, pages 175–198, 2000.
8. N. Heintze. Set-based analysis of ML programs. In *ACM Conference on Lisp and Functional Programming*, pages 306–317, 1994.
9. D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In Oscar Nierstrasz and Michel Lemoine, editors, *ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, pages 199–215. Springer-Verlag / ACM Press, 1999.
10. Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. *ACM SIGPLAN Notices*, 32(8):136–149, August 1997.
11. Peter Norvig. *Paradigms of artificial intelligence programming: case studies in Common LISP*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1992.
12. M. Pettersson, K. Sagonas, and E. Johansson. The HiPE/x86 Erlang compiler: System description and performance evaluation. In Zhenjiang Hu and Mario Rodríguez-Artalejo, editors, *Proceedings of the Sixth International Symposium on Functional and Logic Programming*, number 2441 in LNCS, pages 228–244. Springer, September 2002.
13. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.

14. O. Shivers. Control flow analysis in scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, 1988.
15. B. Steensgaard. Points-to analysis in almost linear time. In *POPL'96*, pages 32–41, Jan 1996.
16. Andrew K. Wright and Suresh Jagannathan. Polymorphic splitting: an effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems*, 20(1):166–207, January 1998.