Escape Analysis: Correctness Proof, Implementation and Experimental Results

Bruno BLANCHET

ENS and INRIA Rocquencourt*

Abstract

We describe an escape analysis [32, 14], used to determine whether the lifetime of data exceeds its static scope.

We give a new correctness proof starting directly from a semantics. Contrary to previous proofs, it takes into account all the features of functional languages, including imperative features and polymorphism. The analysis has been designed so that it can be implemented under the small complexity bound of $\mathcal{O}(n \log^2 n)$ where n is the size of the analyzed program. We have included it in the Caml Special Light compiler (an implementation of ML), and applied it to very large programs. We plan to apply these techniques to the Java programming language.

Escape analysis has been applied to stack allocation. We improve the optimization technique by determining minimal lifetime for stack allocated data, and using inlining. We manage to stack allocate 25% of data in the theorem prover Coq. We analyzed the effect of this optimization, and noticed that its main effect is to improve data locality, which is important for efficiency [8].

1 Introduction

Today, most functional languages use a garbage collector (GC) to manage memory. The GC automatically frees the room taken by unreferenced data when the program runs. However, this mechanism is time consuming. Therefore, stack allocation may be an interesting optimization for such languages, as it allows to statically deallocate data, without calling the GC. Appel claims that "garbage collection can be faster than stack allocation" [3] but in fact we shall see that this is only true when there is much more memory than really needed. However, stack allocating data is only possible if its lifetime does not exceed its static scope. The goal of escape analysis is precisely to determine, thanks to abstract interpretation [10, 11], which data can be stack allocated.

1.1 Related Work

Escape analysis on lists has been introduced by Park and Goldberg [32], and Deutsch [14] has much improved the complexity of their analysis, reducing it to $\mathcal{O}(n \log^2 n)$, with exactly the same results for first-order expressions (there is an unavoidable loss of precision in the higher-order case). He has also suggested many extensions.

Mohnen [30, 29] describes a similar analysis, but its complexity is quadratic and the analyzed language is first order and does not contain imperative operations.

Hughes [24] already introduces integer levels to represent the escaping part of data. He does not perform stack allocation, but keeps in memory addresses of data to be deallocated in order to avoid using the GC. The work closest to Hughes' is [25] by Inoue, Seki and Yagi, who only free the top of lists, but give experimental results.

Alias analysis [13], reference counting [22, 19], storage use analysis [34] which is similar to [26, 18, 12, 35] can be applied to stack allocation though at a much higher cost.

Another allocation optimization has been suggested in [38, 5, 2]: region allocation. All objects are allocated in heap regions whose size is not known statically in general, but for which we know when they can be deallocated. Regions can therefore be deallocated without GC. This analysis solves a more general problem than ours, but at the cost of a much increased complexity. In fact, on many programs, opportunities for stack allocation outnumber opportunities for region allocation, as noticed in [5].

[16] uses annotated types to describe escape information. The results are not as precise as ours and it only gives inference rules and no algorithm to compute annotated types.

1.2 Overview

Deutsch [14] proved that his analysis gave exactly the same results as Park and Goldberg's for first-order purely functional programs. Here, in Sections 2 to 4, we give a direct correctness proof from the semantics, and we extend it to all the extensions Deutsch suggested: imperative operations, pairs, polymorphism, approximate treatment of higher-order functions. We also extend the analysis to all inductive types (not only lists). The details of the proofs have been omitted because of their length (more than 20 pages).

Section 5 describes our implementation which is within the complexity bound given by Deutsch : $\mathcal{O}(n \log^2 n)$. It is based on Caml Special Light (CSL), and improves [14] thanks to intermodular analysis, taking the shortest possible lifetime for variables, preserving as often as possible tail call optimizations. We also introduce inlining to increase stack allocation opportunities.

Section 6 is an experimental study of the effect of the optimization. The comparison between SparcStation 5, Alpha and Pentium Pro shows that the improvement depends on the characteristics of the processor. The main improvement is not on GC time but on data locality. We manage to get 5 to 20% speedup on many programs, and our analysis can be applied to the largest applications thanks to its very good efficiency.

*E-mail: Bruno.Blanchet@ens.fr. Address: 45, rue d'Ulm. F-75005 PARIS.

POPL 98 San Diego CA USA

$e =$ `v`
$\mid$ `let x =`e_1 `in` e_2
$\mid$ `let rec f(x) =`e_1 `in` e_2
$\mid$ $e_1\ e_2$ $\mid$ `fun x` $\rightarrow e$
$\mid$ `true` $\mid$ `false` $\mid 0 \mid 1 \mid 2 \mid \ldots$
$\mid$ $e_1 :: e_2$ $\mid$ `[]` $\mid$ `hd` $\mid$ `tl` $\mid$ `null`
$\mid$ `ref` $\mid$ `!` $\mid$ `:=`
$\mid$ (e_1,e_2) $\mid$ `fst` $\mid$ `snd`
$\mid$ $e_1;e_2$
$\mid$ `if` e_1 `then` e_2 `else` e_3

Figure 1: Syntax

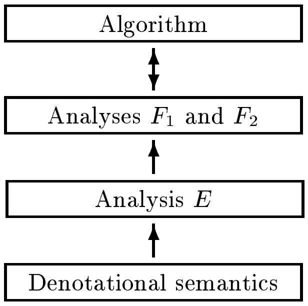


Figure 2: The exact and abstract semantics

1.3 Notations

$\{x_1 \mapsto y_1, ..., x_n \mapsto y_n\}$ is the function which maps x_i to y_i for $i \in [1,n]$. $f[x_1 \mapsto y_1, ..., x_n \mapsto y_n]$ is the extension of f which maps x_i to y_i for $i \in [1,n]$. If f was already defined at some of these values, the new value replaces the old one. $f_{|E}$ is the restriction of f to E. $\mathrm{Dom}(f)$ is the definition domain of f.

S^* is the set of lists whose elements are in S. $[]$ is the empty list. $[p_1, \ldots, p_n]$ is the list of elements $p_1, \ldots, p_n$. $p_1 : l$ is the list l at the head of which p_1 has been concatenated. $l(i)$ is the ith element of l.

$FV(M)$ is the set of free variables of M.

In a lattice, the join is $\sqcup$ and the meet is $\sqcap$.

2 Analyzed Language

2.1 Syntax

The analysis can handle all features of a functional language as CSL, but for the theoretical description, we shall restrict ourselves to a small subset of CSL. We shall use the following data types:

$$\tau = \tau_1 \rightarrow \tau_2 \mid \tau\ \texttt{list} \mid \tau_1 * \tau_2 \mid \tau\ \texttt{ref} \mid \texttt{bool} \mid \texttt{int} \mid \texttt{unit}$$

The syntax of the language is summarized on Figure 1.

We can easily generalize pairs to records and tuples, `let rec` to mutually recursive functions. We also extend the analysis to polymorphism and inductive types (see Sections 4.1 and 4.2).

Several semantics are associated to this language, beginning with an exact denotational semantics, and then approximating it by abstract semantics (Figure 2).

$x \in \mathit{Var}$	Variables
$\ell \in \mathit{Loc}$	Locations
$v \in \mathit{Val} = \{\mathrm{true}, \mathrm{false}, \mathrm{NIL}\} \cup \mathbb{Z} \cup \mathit{Loc}$	Values
$\quad \mathit{SV} = \mathit{Val} \cup (\mathit{Val} \times \mathit{Val}) \cup (\mathit{Env} \rightarrow \mathit{Exp}) \times \mathit{Env} \times \mathit{Var}$	
$s \in \mathit{Store} = (\mathit{Loc} \times \mathit{SV})^*$	Stores
$\quad \mathit{Exp} = \mathit{Store} \rightarrow (\mathit{Val} \times \mathit{Store})_\perp$	Expressions
$e \in \mathit{Env} = \mathit{Var} \rightarrow (\mathit{Store} \times \mathit{Val})$	Environments

$s[\ell \mapsto v] = (\ell, v) : s$
$s(\ell)$ is such that $(\ell, s(\ell))$ is the first element of s of the form $(\ell, \_)$
$e[\![x]\!] = v$ if $e(x) = (s, v)$
up is the injection from $\mathit{Val} \times \mathit{Store}$ to $(\mathit{Val} \times \mathit{Store})_\perp$
$$\text{let } [x] = h \text{ in } g = \begin{cases} \perp & \text{if } h = \perp, \\ g[x'/x] & \text{if } h = \mathrm{up}(x'). \end{cases}$$
$\mathrm{newloc}(s, x) = \mathrm{up}(\ell, s[\ell \mapsto x])$ where $\ell \notin \mathrm{Dom}(s)$

Figure 3: Notations

2.2 Semantics

We use a denotational semantics, with store and non-termination denoted by $\perp$ (Figures 3 and 4).

We use two unusual definitions, which are useful to define the correctness of the analysis: a store is an association list to remember the history of its changes, but we also consider a store as a function which maps every location to its contents: $\mathit{Store} = \mathit{Loc} \rightarrow \mathit{SV}$. This is similar to Hoare's definition of execution traces [21] as sequences of tests and assignments. In environments, we memorize not only the value but also the store in which it has been created.

3 Escape Analysis on Paths

The following analysis can be applied as it is to any functional language, even untyped. It is also very precise, so it can be used as a basis for several less precise analyses which can be derived by abstract interpretation. Its drawback is that it is too complex to be directly implemented. We shall therefore perform a second step of approximations in Section 4.

Abstract values associated with data are access paths which represent the part of data which is useful (Figure 5):

$$\mathit{Path} = \mathrm{l}.\mathit{Path} | \mathrm{r}.\mathit{Path} | \mathrm{app}.\mathit{Path} | \top | \perp$$

where $\top$ means that the whole data is used; $\perp$ means that nothing is used; l means that the left part of a pair, or the head of a list, or the contents of a reference is used; r means that the right part of a pair or the tail of a list is used. app means that the data is a function which is applied to an argument.

We define path restrictions by:

- $\mathrm{app}.c_{|\rightarrow} = c$, $c_{|\rightarrow} = \perp$ if c is not of the form $\mathrm{app}.c'$.
- $\top_{|l} = \top$, $\mathrm{l}.c_{|l} = c$, $c_{|l} = \perp$ otherwise. We define $|r$ in the same way.

The definition of the correctness of the analysis is summarized on Figure 6. Here are some explanations.

Contexts associated to data are sets of paths: $\mathit{Ctx} \subseteq \mathcal{P}(\mathit{Path})$. An empty context would intuitively correspond to an unevaluated expression. Here, because of call-by-value, expressions are evaluated even if the result is not needed, and this evaluation may cause escapement because of imperative operations. That is why we use non-empty contexts. $(\mathit{Ctx}, \subseteq)$ is a sup-semilattice. If $\phi : \mathit{Path} \rightarrow \mathit{Path}$

$[\![M]\!] : \mathit{Env} \rightarrow \mathit{Exp}$

$[\![x]\!]es = \mathrm{up}(e[\![x]\!], s)$

$[\![M\ N]\!]es = \mathrm{let}\ [(f', s')] = [\![M]\!]es\ \mathrm{in}\ \ \mathrm{let}\ [(v, s'')] = [\![N]\!]es'\ \mathrm{in}\ \ \mathrm{let}\ (m, e', x) = s'(f')\ \mathrm{in}\ \ m(e'[x \mapsto v])s''$

$[\![\mathtt{fun}\ x \rightarrow M]\!]es = \mathrm{newloc}(s, ([\![M]\!], e_{|FV(\mathtt{fun}\ x \rightarrow M)}, x))^a$

$[\![\mathtt{let}\ x = M\ \mathtt{in}\ N]\!]es = \mathrm{let}\ [(v, s')] = [\![M]\!]es\ \mathrm{in}\ [\![N]\!](e[x \mapsto (s', v)])s'$

$[\![\mathtt{let\ rec}\ f(x) = M\ \mathtt{in}\ N]\!]es = [\![N]\!](e[f \mapsto (s', \ell)])s'$ where $\ell \notin \mathrm{Dom}(s)$, $s' = s[\ell \mapsto (\phi, e_{|FV(\mathtt{fun}\ x \rightarrow M)}, x)]$ and

$\phi = \mathrm{lfp}(\lambda\phi_1.\lambda e_1.\lambda s_1.[\![M]\!](e_1[f \mapsto (s_1', \ell)])s_1')$ where s_1' is the store s_1 where $s_1(\ell)$ has been replaced by $(\phi_1, e_1, x)^b$

$[\![\mathtt{if}\ M\ \mathtt{then}\ N\ \mathtt{else}\ P]\!]es = \mathrm{let}\ [(x, s')] = [\![M]\!]es\ \mathrm{in}\ \ \mathrm{if}\ x\ \mathrm{then}\ [\![N]\!]es'\ \mathrm{else}\ [\![P]\!]es'$

$[\![M{::}N]\!]es = \mathrm{let}\ [(x, s')] = [\![M]\!]es\ \mathrm{in}\ \ \mathrm{let}\ [(y, s'')] = [\![N]\!]es'\ \mathrm{in}\ \ \mathrm{newloc}(s'', (x, y))$

$[\![(M, N)]\!]es = \mathrm{let}\ [(\ell_1, s')] = [\![M]\!]es\ \mathrm{in}\ \ \mathrm{let}\ [(\ell_2, s'')] = [\![N]\!]es'\ \mathrm{in}\ \ \mathrm{newloc}(s'', (\ell_1, \ell_2))$

$[\![M; N]\!]es = \mathrm{let}\ [(\ell, s')] = [\![M]\!]es\ \mathrm{in}\ [\![N]\!]es'$

Initial store: $s_0 = \{\mathrm{ref} \mapsto (\lambda e.\lambda s'.\,\mathrm{newloc}(s', e[\![x]\!]), \emptyset, x), \mathrm{deref} \mapsto (\lambda e.\lambda s'.\,\mathrm{up}(s'(e[\![x]\!]), s'), \emptyset, x),$
$\mathrm{affect} \mapsto (\lambda e.\lambda s'.\,\mathrm{newloc}(s', (\lambda e'.\lambda s''.\,\mathrm{up}(\mathrm{NIL}, s''[e'[\![m]\!] \mapsto e'[\![n]\!]]), e, n)), \emptyset, m),$
$\mathrm{fst} \mapsto (\lambda e.\lambda s'.\,\mathrm{let}\ (h, t) = s'(e[\![x]\!])\ \mathrm{in}\ \ \mathrm{up}(h, s'), \emptyset, x), \mathrm{snd} \mapsto (\lambda e.\lambda s'.\,\mathrm{let}\ (h, t) = s'(e[\![x]\!])\ \mathrm{in}\ \ \mathrm{up}(t, s'), \emptyset, x),$
$\mathrm{null} \mapsto (\lambda e.\lambda s'.(e[\![x]\!] = \mathrm{NIL}), \emptyset, x)\}$

Initial environment: $e_0 = \{\mathtt{ref} \mapsto (s_0, \mathrm{ref}), \mathtt{!} \mapsto (s_0, \mathrm{deref}), \mathtt{:=} \mapsto (s_0, \mathrm{affect}), \mathtt{true} \mapsto (s_0, \mathrm{true}), \mathtt{false} \mapsto (s_0, \mathrm{false}),$
$n \mapsto (s_0, n)$ if $n \in \mathbb{Z}$, $\mathtt{[]} \mapsto (s_0, \mathrm{NIL}), \mathtt{hd} \mapsto (s_0, \mathrm{fst}), \mathtt{tl} \mapsto (s_0, \mathrm{snd}), \mathtt{null} \mapsto (s_0, \mathrm{null}),$
$\mathtt{fst} \mapsto (s_0, \mathrm{fst}), \mathtt{snd} \mapsto (s_0, \mathrm{snd})\}$.

[a] We create minimal closures, so we only store in the environment of the closure the free variables of $\mathtt{fun}\ x \rightarrow M$.

[b] We do not write $s_1' = s_1[\ell \mapsto (\phi_1, e_1, x)]$ because we do not want to consider theses changes as assignments.

Figure 4: Denotational semantics

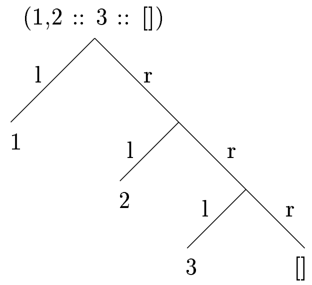


Figure 5: Paths. For example, 2 is represented by path r.l.⊤

maps c to one of the paths $\mathrm{l}.c, \mathrm{r}.c, \mathrm{app}.c, c_{|l}, c_{|r}, c_{|\rightarrow}$, we define $\phi^C : \mathit{Ctx} \rightarrow \mathit{Ctx}$ by $\phi^C(c) = \{\phi(p) | p \in c\}$. We will use the same notations for ϕ and ϕ^C. The path c and the context $\{c\}$ will be considered identical.

Intuitively, s' is after s if s' can be obtained after s during the execution of a program.

$a(s)$ is the set of locations of right hand sides of assignments from the store s_0 at the beginning of the program to the store s. To be able to define this from the store s only, the stores must keep the history of their modifications. That is why we have encoded stores as association lists.

$\mathrm{loca}(s, v, c, e_p)$ is the set of locations which escape when we keep the locations designated by context c in the value v in the store s. The environment e_p indicates which parameters will be given to the function v (it is empty if v is not a function). (1) makes sure that if we estimate the escapement in a given store, the analysis will remain correct in a future store. (3) is the set of locations accessible from ℓ in the store s, except that we exclude locations in $a(s)$ to balance the effect of imperative operations. Intuitively, we follow a path in two different ways: for data structures, we simply follow pointers (4, 5), whereas for functions, we apply them to a parameter taken in the parameter environment (6).

$\mathrm{dest}((s, v), c)$ is the set of locations in the value v represented by the context c. If v is not functional, the union on e_p and the subtraction of the locations of e_p in (9) are useless.

In Definitions (10) and (11), we mark the part ($\mathrm{loca}(s, v, c, e_p)$ or A) of the result represented by the context c and the stored locations which we consider as escaping, and we search which locations of the variables are marked.

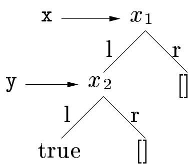
f is ρ-transitive if it takes into account locations that escape through intermediate variables. For example, in the expression :

```
let x = M in let y = N in P
```

if $f = E[\![P]\!]\rho$ is the analysis of P, and if we keep the part represented by context c of the result, the escapement of y will be represented by $c_1 = fcy$. This escapement of y will cause the escapement $c_2 = \rho[\![y]\!]c_1x$ of x. On the other hand, what escapes from x is what escapes through assignments during the evaluation of N, $\rho[\![y]\!]\bot x$, union what escapes through the result, fcx. ρ-transitivity asserts that this escapement takes into account the escapement c_2 through y.

Example 3.1

```
let x = (true :: []) :: [] in
let y = hd x in
  y
```



Let e and s be the environment and the store when $\mathtt{y}$ is evaluated. Let x_1, $x_2 = y$ be the locations such that $e[\![\mathtt{x}]\!] = x_1$, $e[\![\mathtt{y}]\!] = x_2$, $s(x_1) = (x_2, \mathrm{NIL})$, $s(x_2) = (\mathrm{true}, \mathrm{NIL})$. We compute $\phi\top\mathtt{x}$ such that $\mathrm{corr}_v(e, s, y, \phi)$:

$\mathrm{loca}(s, y, \top, []) = \{x_2\}$, $\mathrm{dest}((s, x_1), \mathrm{l}.\top) = \{x_2\}$ and $\mathrm{dest}((s, x_1), \mathrm{l.l}.\top) = \emptyset$ therefore $\phi\top\mathtt{x} = \mathrm{l}.\top$.

Theorem 3.2 *Consider an expression* `let` x `=` M `in` N *or* `let rec` $x(y)$ `=` M `in` N. *Assume that the location* ℓ *at the top of* x *is new when* x *is defined.*

Let s *and* s' *be respectively the stores at the beginning and at the end of an evaluation of* N, *and* r *the result of* N. *If* $\ell \notin ((a(s') - a(s)) \cup \mathrm{loca}(s', r, \top, [])) \cap \mathrm{loc}(s) = (a(s') \cup r\downarrow_{s'}) \cap \mathrm{loc}(s) - a(s)$, *then* ℓ *can be stack allocated.*

Proof sketch: In this situation, ℓ is only accessible through x in the store s'. Indeed, $\ell \notin a(s)$ and $\ell \in \mathrm{loc}(s)$ so $\ell \notin a(s') \cup r\downarrow_{s'}$.

Domains

$c \in Ctx = \mathcal{P}(Path) - \{\emptyset\}$ Contexts

$i \in Ind = \mathbb{N}$ Parameter indices

$\phi \in Exp^{\#} = Val^{\#} = Ctx \to (Var \cup Ind) \to Ctx$

$\rho \in Env^{\#} = Var \to Val^{\#}$ Abstract environment

$E[\![M]\!] : Env^{\#} \to Exp^{\#}$ Abstract semantic function

$e_p \in Env_p = (Store \times Val)^*$ Parameter environment

Accessible locations

$\ell \downarrow_s = \{$locations accessible from location ℓ in store $s\}$

$v \downarrow_s = \emptyset$ if $v \in \{\text{true}, \text{false}, \text{NIL}\} \cup \mathbb{Z}$

$e \downarrow_s = \cup\{e[\![x]\!] \downarrow_s \mid x \in \text{Dom}(e)\}$

$\text{loc}(s) = \text{Dom}(s)$ (locations existing in the store s)

$\text{loc}(e_p) = \cup_{i, e_p(i)=(v_p, s_p)} v_p \downarrow_{s_p}$

$\text{loc}(s, e_p) = \text{loc}(s) \cup \text{loc}(e_p)$

Store order

$sAs' \Leftrightarrow s' = s[\ell \mapsto v]$ where $\ell \in Loc$

if $\ell \in \text{Dom}(s)$, ℓ is mutable (ie has been created by a reference)

$v \in \text{val}(s) \cup \text{val}(s) \times \text{val}(s) \cup (Env \to Exp) \times (Var \to \text{val}(s)) \times Var$

$\text{val}(s) = \text{loc}(s) \cup \{\text{true}, \text{false}, \text{NIL}\} \cup \mathbb{Z}$

s' after s if sA^*s' (A^* is the reflexive and transitive closure of A)

Stored locations

$a : Store \to \mathcal{P}(Loc)$

$a([]) = \emptyset$

$a(s[\ell \mapsto v]) = a(s) \cup \ell \downarrow_{s'}$ if $\ell \in \text{loc}(s)$

(a destructive assignment)

$a(s[\ell \mapsto v]) = a(s)$ otherwise (an allocation)

Context concretization

$\text{loca} : Store \times Val \times Ctx \times Env_p \to \mathcal{P}(Loc)$

If c is a path,

$\text{loca}(s, v, c, e_p) = \cup\{\text{loca}_0(s', v, c, e_p) | s' \text{ after } s\}$ (1)

If $c = \bot$ or $v \in \{\text{true}, \text{false}, \text{NIL}\} \cup \mathbb{Z}, \text{loca}_0(s, v, c, []) = \emptyset$ (2)

If $c = \top, \text{loca}_0(s, \ell, c, []) = \ell \downarrow_s - a(s)$ (3)

If $c = \text{l}.c_1,\ s(\ell) = (\ell_1, \ell_2)$ (pairs or lists) or $s(\ell) = \ell_1$ (references), $\text{loca}_0(s, \ell, c, e_p) = \text{loca}(s, \ell_1, c_1, e_p)$ (4)

If $c = \text{r}.c_2,\ s(\ell) = (\ell_1, \ell_2),\ \text{loca}_0(s, \ell, c, e_p) = \text{loca}(s, \ell_2, c_2, e_p)$ (5)

If $c = \text{app}.c_1,\ s(\ell) = (m, e', x),\ m(e'[x \mapsto p_1])s = \text{up}(\ell'', s')$,

$\text{loca}_0(s, \ell, c, (s, p_1) : e_p) = ((a(s') - a(s)) \cup \text{loca}(s', \ell'', c_1, e_p)) \cap \text{loc}(s, e_p)$ (6)

If the store s is not the one indicated for the parameter p_1 in the parameter environment e_p, the result is $\emptyset$.

If $c = \text{app}.c_1,\ s(\ell) = (m, e', x)$, and $m(e'[x \mapsto p_1])s = \bot, \text{loca}_0(s, \ell, c, (s', p_1) : e_p) = \emptyset$ (7)

If the path c does not correspond to the value ℓ (the path begins with r for a reference, with l or r for a closure, with app for something else than a closure), $\text{loca}(s, \ell, c, e_p) = \emptyset$

If c is a context, $\text{loca}(s, \ell, c, e_p) = \cup\{\text{loca}(s, \ell, c_1, e_p) | c_1 \in c\}$ (8)

$\text{dest} : (Store \times Val) \times Ctx \to \mathcal{P}(Loc)$

$\text{dest}((s, v), c) = \cup\{\text{loca}(s, v, c, e_p) - \text{loc}(e_p) | e_p \in Env_p\}$ (9)

Correctness

$\text{corr}_v : Env \times Store \times Val \times Val^{\#} \to \{\text{true}, \text{false}\}$

$\text{corr}_v(e, s, v, \phi) \Leftrightarrow \forall c \in Ctx, \forall e_p \in Env_p, \text{loca}(s, v, c, e_p) \subseteq \cup_{x \in \text{Dom}(e)} \text{dest}(e(x), \phi c x) \cup \cup_{i \in Ind} \text{dest}(e_p(i), \phi c i)$ (10)

$\text{corr}_e : Env \times Store \times Exp \times Exp^{\#} \to \{\text{true}, \text{false}\}$

$\text{corr}_e(e, s, f, \phi) \Leftrightarrow \forall c \in Ctx, \forall e_p \in Env_p, A \subseteq \cup_{x \in \text{Dom}(e)} \text{dest}(e(x), \phi c x) \cup \cup_{i \in Ind} \text{dest}(e_p(i), \phi c i)$ (11)

$A = \emptyset$ if $f\ s = \bot$

$A = ((a(s') - a(s)) \cup \text{loca}(s', v, c, e_p)) \cap \text{loc}(s, e_p)$ if $f\ s = \text{up}(v, s')$.

$f \in Val^{\#} = Exp^{\#}$ is ρ-transitive if, for all y in the lexical scope of x, $fcx \sqcup \rho[\![y]\!] \bot x \supseteq \rho[\![y]\!](fcy)x$

$\text{corr} : (Env \to Exp) \times (Env^{\#} \to Exp^{\#}) \to \{\text{true}, \text{false}\}$

$\text{corr}(f, \phi) \Leftrightarrow \forall e, \forall s, \forall \rho, (\forall y \in \text{Dom}(e), \text{corr}_v(e, s, e[\![y]\!], \rho[\![y]\!])$ and $\rho[\![y]\!]$ is ρ-transitive$) \Rightarrow \text{corr}_e(e, s, fe, \phi\rho)$ (12)

and $\phi\rho$ is ρ-transitive ($\phi\rho c$ is defined on $\text{Dom}(e) \cup Ind$).

Figure 6: Definition of the correctness of the analysis

$$
\begin{aligned}
&E\llbracket M\rrbracket : \mathit{Env}^{\#} \to \mathit{Exp}^{\#}\\
&E\llbracket y\rrbracket\rho c = \rho\llbracket y\rrbracket c\\
&E\llbracket M\ N\rrbracket\rho c = \{i \mapsto \phi(i+1) \text{ if } i \in \mathit{Ind}, x \mapsto \phi(x)C \sqcup \psi(x) \text{ if}\\
&\quad x \in \mathit{Var}\} \text{ where } \phi = E\llbracket M\rrbracket\rho(\mathrm{app}.c) \text{ and } \psi = E\llbracket N\rrbracket\rho(\phi(1))\\
&E\llbracket \texttt{fun}\ y \to M\rrbracket\rho c =\\
&\begin{cases} \bigsqcup_{z\in FV(\texttt{fun}\ y\to M)} \rho\llbracket z\rrbracket\top & \text{if } c = \top,\\ \{\} & \text{if } c = \bot,\\ \{1 \mapsto \phi(y), i \mapsto \phi(i-1) \text{ if } i > 1, & \\ x \mapsto \phi(x) \text{ if } x \in \mathit{Var}\} & \text{otherwise.}\end{cases}\\
&\quad \text{where } \phi = E\llbracket M\rrbracket\rho[y \mapsto \rho_\top](c_{|\to})_{|\overline{\{y\}}}\\
&\quad \text{and } \rho_\top(c) = \{i \mapsto \top, y \mapsto c\}.\\
&E\llbracket \texttt{let}\ y = M\ \texttt{in}\ N\rrbracket\rho c =\\
&\quad E\llbracket N\rrbracket\rho[y \mapsto E\llbracket M\rrbracket\rho \sqcup \lambda c.\{y \mapsto c\}]c_{|\overline{\{y\}}} \sqcup E\llbracket M\rrbracket\rho\bot_{|\mathit{Var}}\\
&E\llbracket \texttt{let rec}\ f(y) = M\ \texttt{in}\ N\rrbracket\rho c = E\llbracket N\rrbracket\\
&\quad (\mathrm{lfp}(\lambda\rho_1.\rho[f \mapsto E\llbracket \texttt{fun}\ y \to M\rrbracket\rho_1 \sqcup \lambda c.\{f \mapsto c\}])c_{|\overline{\{f\}}}\\
&E\llbracket \texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ P\rrbracket\rho c =\\
&\quad E\llbracket M\rrbracket\rho\bot_{|\mathit{Var}} \sqcup E\llbracket N\rrbracket\rho c \sqcup E\llbracket P\rrbracket\rho c\\
&E\llbracket M{::}N\rrbracket\rho c = E\llbracket M\rrbracket\rho(c_{|l}) \sqcup E\llbracket N\rrbracket\rho(c_{|r})\\
&E\llbracket (M,N)\rrbracket\rho c = E\llbracket M\rrbracket\rho(c_{|l}) \sqcup E\llbracket N\rrbracket\rho(c_{|r})\\
&E\llbracket M;N\rrbracket\rho c = E\llbracket M\rrbracket\rho\bot_{|\mathit{Var}} \sqcup E\llbracket N\rrbracket\rho c\\
&\rho\llbracket \texttt{ref}\rrbracket c = \begin{cases}\{\} & \text{if } c = \bot \text{ or } \top,\\ \{1 \mapsto c_{|\to|r}, i \mapsto \top \text{ if } i > 1\} & \text{otherwise.}\end{cases}\\
&\rho\llbracket {:=}\rrbracket c = \begin{cases}\{\} & \text{if } c = \bot, \mathrm{app}.\bot \text{ or } \top\\ \{1 \mapsto \top\} & \text{if } c = \mathrm{app}.\top,\\ \{2 \mapsto \top\} & \text{if } c = \mathrm{app.app}.\bot \text{ or app.app}.\top.\end{cases}\\
&\rho\llbracket a\rrbracket c = \{\} \text{ where } a \in \{\texttt{true}, \texttt{false}, \texttt{[]}\} \cup \mathbb{Z}.\\
&\rho\llbracket f\rrbracket c = \begin{cases}\{\} & \text{if } c = \bot \text{ or } \top,\\ \{1 \mapsto \mathrm{l}.c_{|\to}, i \mapsto \top \text{ if } i > 1\} & \text{otherwise.}\end{cases}\\
&\quad \text{where } f \text{ is } \texttt{fst}, \texttt{hd} \text{ or } \texttt{!}.\\
&\rho\llbracket f\rrbracket c = \begin{cases}\{\} & \text{if } c = \bot \text{ or } \top,\\ \{1 \mapsto \mathrm{r}.c_{|\to}, i \mapsto \top \text{ if } i > 1\} & \text{otherwise.}\end{cases}\\
&\quad \text{if } f \text{ is } \texttt{snd} \text{ or } \texttt{tl}.\\
&\rho\llbracket \texttt{null}\rrbracket c = \{\}\\
&E\llbracket M\rrbracket\rho c x = \cup\{E\llbracket M\rrbracket\rho c' x | c' \in c\} \text{ if } c \text{ is a context.}
\end{aligned}
$$

Figure 7: The equations of escape analysis

ℓ is not accessible through the result r as $\ell \notin r\downarrow_{s'}$, and ℓ is not accessible through other variables in the environment because it has not been stored. When we get rid of x, ℓ is no longer accessible and can be deallocated. □

We denote by $f = \{x \mapsto f(x), ...\}$ a function $f : (\mathit{Var} \cup \mathit{Ind}) \to \mathit{Ctx}$ defined pointwise. f evaluates to $\bot$ when it is not explicitly defined. $\{\}$ is therefore the constant function equal to $\bot$. The analysis E is defined on Figure 7, first for paths, then we extend it to contexts by taking unions (last formula of Figure 7).

Example 3.3

```
let l =
   let a = 1 :: 2 :: 3 :: [] in
   let g = fun x -> (x,x) in
   let rec map f l =
      if null l
      then []
      else (f (hd l)) :: (map f (tl l))
   in map g a
in ...
```

Let $M =$ `let rec map = ...  in map g a`. Let ρ_0 be the environment containing predefined variables.

$$
\begin{aligned}
&\rho = \rho_0[\texttt{a} \mapsto (E\llbracket \texttt{1::2::3::[]}\rrbracket\rho_0 \sqcup \lambda c.\{\texttt{a} \mapsto c\}),\\
&\qquad \texttt{g} \mapsto (E\llbracket \texttt{fun x -> (x,x)}\rrbracket\rho_0 \sqcup \lambda c.\{\texttt{g} \mapsto c\})]\\
&E\llbracket M\rrbracket\rho c = E\llbracket \texttt{map g a}\rrbracket(\mathrm{lfp}(\lambda\rho_1.\rho[\texttt{map} \mapsto\\
&\quad E\llbracket \texttt{fun f l -> if ...}\rrbracket\rho_1 \sqcup \lambda c.\{\texttt{map} \mapsto c\}]))c
\end{aligned}
$$

Let $\rho_2 = \rho_1[\texttt{f} \mapsto \rho_\top, \texttt{l} \mapsto \rho_\top]$, $c' = \mathrm{app.app}.c_{|\to|\to|r}$.

$$
\begin{aligned}
&E\llbracket \texttt{fun f l -> if ...}\rrbracket\rho_1 c =\\
&\quad \{1 \mapsto (\mathrm{app}.c_{|\to|\to|l}) \sqcup \rho_2\llbracket \texttt{map}\rrbracket c'\texttt{f} \sqcup \rho_2\llbracket \texttt{map}\rrbracket c' 1,\\
&\quad\ 2 \mapsto \{\mathrm{l}.\top\} \sqcup \rho_2\llbracket \texttt{map}\rrbracket c'\texttt{l} \sqcup \mathrm{r}.\rho_2\llbracket \texttt{map}\rrbracket c' 2,\\
&\quad\ \texttt{map} \mapsto c' \sqcup \rho_2\llbracket \texttt{map}\rrbracket c'\texttt{map},\\
&\quad\ x \mapsto \emptyset \text{ if } x \in \mathit{Var} - \{\texttt{map}\}\}
\end{aligned}
$$

The fixed point is:

$$
\begin{aligned}
&\rho_1\llbracket \texttt{map}\rrbracket c = \{\texttt{map} \mapsto c, 1 \mapsto \mathrm{app}.c_{|\to|\to|r^*|l}, 2 \mapsto \mathrm{r}^*.\mathrm{l}.\top\}\\
&E\llbracket M\rrbracket\rho\{\top\}\texttt{a} = \rho_1\llbracket \texttt{map}\rrbracket\{\mathrm{app.app}.\top\}2 = \mathrm{r}^*.\mathrm{l}.\top\\
&E\llbracket M\rrbracket\rho\{\top\}\texttt{g} = \rho_1\llbracket \texttt{map}\rrbracket\{\mathrm{app.app}.\top\}1 = \{\mathrm{app}.\top\}
\end{aligned}
$$

The top of `a` and the closure `g` can therefore be stack allocated.

Theorem 3.4 *The analysis E is correct* $\mathrm{corr}(\llbracket M\rrbracket, E\llbracket M\rrbracket)$, *and e is coherent with ρ on predefined variables:* $\mathrm{corr}_v(e, s, e\llbracket y\rrbracket, \rho\llbracket y\rrbracket)$.

Proof sketch: The proof, omitted, is by induction on the syntax.

In the `let` $y = M$ `in` N case, we have to show that $\mathrm{dest}(e'\llbracket y\rrbracket, E\llbracket N\rrbracket\rho' cy) \cap \mathrm{loc}(s) \subseteq \cup_{x\in\mathrm{Dom}(e)} \mathrm{dest}(e(x), E\llbracket N\rrbracket\rho' cx \sqcup E\llbracket M\rrbracket\rho\bot x)$ where $e' = e[y \mapsto (s', v)]$ and $\rho' = \rho[y \mapsto E\llbracket M\rrbracket\rho \sqcup \lambda c.\{y \mapsto c\}]$ are respectively the semantic and abstract environments in the analysis of N. This expresses that the escapement through the bound variable y is already taken into account by other variables. It comes from ρ-transitivity.

In the `let rec` case, we do a fixed point induction, and use ρ-transitivity as in the `let` case. □

In the case:

```
let rec f(x) = ... z := f ...
in f(3)
```

$\mathrm{dest}((s, \texttt{f}), \mathrm{app}.\top)$ contains all the locations of `f`, so $E\llbracket \texttt{f(3)}\rrbracket\rho\top = \mathrm{app}.\top$ would be correct wrt. $\mathrm{corr}_e(e, s, \llbracket \texttt{f(3)}\rrbracket e, E\llbracket \texttt{f(3)}\rrbracket\rho)$, so we cannot use this criterion directly to decide that `f` does not escape : ρ-transitivity is necessary.

However, we can prove the following result:

Theorem 3.5 (Correctness of E) *Consider the expression* `let x = M in N` *or* `let rec x(y) = M in N`. *Assume that the creation of the location ℓ at the top of x is the last operation in M. Also assume that $\top \notin E\llbracket N\rrbracket\rho\, \mathit{Path}\, x$. Then ℓ can be stack allocated.*

This arises from Theorem 3.4 and ρ-transitivity.

Proof sketch: Let s_x and s'_x be respectively the stores at the beginning and at the end of an evaluation of N.

First, we show that if $c \in E\llbracket N\rrbracket\rho\, \mathit{Path}\, x$, $\ell \notin \mathrm{dest}((s_x, \ell), c)$. If x is defined by `let`, we show that if $c \neq \top$, $\ell \notin \mathrm{dest}((s_x, \ell), c)$ using the definition of dest. If x is defined with `let rec`, x itself can reference ℓ. By ρ-transitivity, $E\llbracket N\rrbracket\rho\, \mathit{Path}\, x \sqcup \rho\llbracket x\rrbracket\bot x \supseteq \rho\llbracket x\rrbracket(E\llbracket N\rrbracket\rho\, \mathit{Path}\, x)x$ with $\rho\llbracket x\rrbracket = \mathrm{lfp}(\lambda\rho_x.E\llbracket \texttt{fun}\ y \to M\rrbracket\rho[x \mapsto \rho_x] \sqcup \lambda c.\{x \mapsto c\})$. Then $\rho\llbracket x\rrbracket\bot x = \bot$, so $\top \notin$

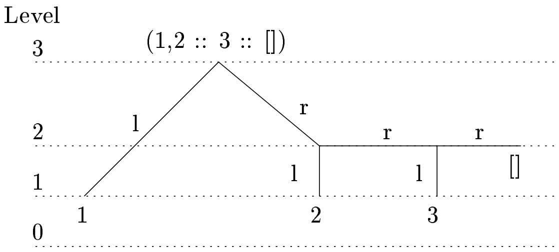


Figure 8: Type levels

$\rho[\![x]\!](E[\![N]\!]\rho\, Path\, x)x$. Let $c \in E[\![N]\!]\rho\, Path\, x$. $\top \notin \rho[\![x]\!]cx = \mathrm{lfp}(\lambda\rho_x.E[\![\texttt{fun}\ y \to M]\!]\rho[x \mapsto \rho_x] \sqcup \lambda c.\{x \mapsto c\})cx$.

We consider imaginary executions where we only iterate n times in the fixed point computation, to show by induction that $\ell \notin \mathrm{dest}((s_x^n, \ell), c)$, if $c \neq \top$. So we still have $\ell \notin \mathrm{dest}((s_x, \ell), c)$ if $c \in E[\![N]\!]\rho\, Path\, x$.

Then $\ell \notin \mathrm{dest}((s_x, \ell), E[\![N]\!]\rho\, Path\, x)$. Let r be the result of N. By correctness of the analysis (Theorem 3.4), $((a(s'_x) - a(s_x)) \cup \mathrm{loca}(s'_x, r, \top, [])) \cap \mathrm{loc}(s_x) \subseteq \cup_{x \in Var}\, \mathrm{dest}(e(x), E[\![N]\!]\rho\, Path\, x)$. Then $\ell \notin (a(s'_x) - a(s_x)) \cup r \downarrow_{s'_x}$, as $\ell \in \mathrm{loc}(s_x)$ and ℓ is not in the variables other than x, as it has been created at the end of the computation of x. Therefore, $\ell \notin ((a(s'_x) - a(s_x)) \cup \mathrm{loca}(s'_x, r, \top, [])) \cap \mathrm{loc}(s_x)$ and by Theorem 3.2, ℓ can be stack allocated. $\square$

4 Escape Analysis on Numerical Contexts

We now represent escapement by integer levels, and we define a translation from sets of paths to integers.

We define type levels by

$\top_2[\tau] = 1$ if $\tau = \texttt{bool}$, $\texttt{int}$ or $\texttt{unit}$

$\top_2[\tau_1 \to \tau_2] = \top_2[\tau_2]$

$\top_2[\tau_1 * \tau_2] = 1 + \max(\top_2[\tau_1], \top_2[\tau_2])$

$\top_2[\tau\ \texttt{list}] = 1 + \top_2[\tau]$

$\top_2[\tau\ \texttt{ref}] = 1 + \top_2[\tau]$

$\top_1[\tau] = 1$ if τ contains a functional type $\tau_1 \to \tau_2$

$\top_1[\tau] = 0$ otherwise

The level of a path is defined by

$\alpha_2^\tau(\top) = \top_2[\tau]$ if τ is not functional, ∞ otherwise

$\alpha_2^\tau(\bot) = 0$

$\alpha_2^{\tau_1 \to \tau_2}(\mathrm{app}.c) = \alpha_2^{\tau_2}(c)$

$\alpha_2^{\tau_1 * \tau_2}(\mathrm{l}.c) = \alpha_2^{\tau_1}(c)$

$\alpha_2^{\tau\ \texttt{list}}(\mathrm{l}.c) = \alpha_2^{\tau}(c)$

$\alpha_2^{\tau\ \texttt{ref}}(\mathrm{l}.c) = \alpha_2^{\tau}(c)$

$\alpha_2^{\tau_1 * \tau_2}(\mathrm{r}.c) = \alpha_2^{\tau_2}(c)$

$\alpha_2^{\tau\ \texttt{list}}(\mathrm{r}.c) = \alpha_2^{\tau\ \texttt{list}}(c)$

$\alpha_2^\tau(c) = 0$ in all other cases (forbidden paths)

We define the level of a context by taking the upper bound:

$$\alpha_2^\tau(c) = \bigsqcup_{c' \in c} \alpha_2^\tau(c')$$

The concretization function is

$$\gamma_2^\tau(c') = \{c \mid \alpha_2^\tau(c) \leq c'\}$$

$(\alpha_2^\tau, \gamma_2^\tau)$ is a semi-dual Galois connection:

$$(Ctx, \subseteq) \underset{\alpha_2^\tau}{\overset{\gamma_2^\tau}{\leftrightarrows}} (\mathbb{N} \cup \{\infty\}, \leq)$$

We write in exponent of α_2 or γ_2 an expression instead of a type to mean the type of this expression. We underline a part of a type to symbolize $\top_2$ of this part. For example, $\underline{\tau_1}\ \texttt{list}$ means $\top_2[\tau_1]$.

Let $Ctx_1 = \{0, 1\}$ and $Ctx_2 = \mathbb{N}$. We define the abstraction $\alpha_1^\tau(c) = (1, 0)$ if $\alpha_2^\tau(c) = \infty$, $\alpha_1^\tau(c) = (0, \alpha_2^\tau(c))$ otherwise. $\gamma_1^\tau(c_1, c_2) = \sqcup\{c \in C(\tau) \mid \alpha_1^\tau(c) \leq (c_1, c_2)\}$ where pairs are ordered lexicographically. $(\alpha_1^\tau, \gamma_1^\tau)$ is a semi-dual Galois connection:

$$(Ctx, \subseteq) \underset{\alpha_1^\tau}{\overset{\gamma_1^\tau}{\leftrightarrows}} (Ctx_1 \times Ctx_2, \leq)$$

$Val_i = Ctx_i \to (Var \cup Ind) \to Ctx_i$ and $Env_i = Var \to Val_i$. We define two analyses $F_i[\![M]\!] : Env_i \to Val_i$ on Figure 9. In these definitions, $x \in Var$ and $i \in Ind$. A function evaluates to 0 when it is not explicitly defined. We write $\top_{1x} = \top_1[\tau]$ if x is of type τ, and $\top_{1,x}$ for clarity when x is a numeric parameter index. We use the same notations for $\top_2$.

The analysis F_1 takes into account variables which escape through closures, whereas F_2 gives more precise information for the other variables. One analysis is not enough because the level of a variable in a closure may be higher than the level of the closure, so the corresponding escape function would not be inferior and our fast algorithm would not work (see Section 5.1).

Example 4.1 For the `map : (int -> int) -> int list -> int list` function (Example 3.3),

$F_1[\![\texttt{fun f l -> if ...}]\!]\rho_1 c =$

$\{1 \mapsto c \sqcup \rho_1[\![\texttt{map}]\!]c\,\texttt{f} \sqcup \rho_1[\![\texttt{map}]\!]c\,1,$

$2 \mapsto c \sqcup \rho_1[\![\texttt{map}]\!]c\,\texttt{l} \sqcup \rho_1[\![\texttt{map}]\!]c\,2,$

$\texttt{map} \mapsto c \sqcup \rho_1[\![\texttt{map}]\!]c\,\texttt{map}\}$

$F_2[\![\texttt{fun f l -> if ...}]\!]\rho_2 c =$

$\{1 \mapsto (c \sqcap (\texttt{int} \to \underline{\texttt{int}})) \sqcup \rho_2[\![\texttt{map}]\!]c\,\texttt{f} \sqcup \rho_2[\![\texttt{map}]\!]c\,1,$

$2 \mapsto \underline{\texttt{int}}\ \texttt{list} \sqcup \rho_2[\![\texttt{map}]\!]c\,\texttt{l} \sqcup \rho_2[\![\texttt{map}]\!]c\,2,$

$\texttt{map} \mapsto c \sqcup \rho_2[\![\texttt{map}]\!]c\,\texttt{map}\}$

Then,

$\rho_1[\![\texttt{map}]\!]c = \{\texttt{map} \mapsto c, 1 \mapsto c, 2 \mapsto c\}$

$\rho_2[\![\texttt{map}]\!]c = \{\texttt{map} \mapsto c, 1 \mapsto c \sqcap (\texttt{int} \to \underline{\texttt{int}}), 2 \mapsto \underline{\texttt{int}}\ \texttt{list}\}$

Definition 4.2 The analyses $f_1 : Ctx_1 \to Ctx_1$ and $f_2 : Ctx_2 \to Ctx_2$ which give escape information about a variable of type τ' in a term of type τ are said to be correct wrt. the analysis $f : Ctx \to Ctx$ if

$$C(\tau[\tau'], f, f_1, f_2) \Leftrightarrow \begin{cases} \text{If } f_1 1 = 0,\ f\, Path \subseteq \gamma_2^{\tau'}(0) \\ \text{If } f_1 0 = 0,\ f(\gamma_2^\tau(c)) \subseteq \gamma_2^{\tau'}(f_2 c) \end{cases}$$

The first line handles the case when a closure escapes in the result (the context passed to analysis f_1 is 1). In this case, the analysis f_1 is just a rough approximation which says that all free variables escape, so if f_1 gives back 0, we know that the variable is not free and nothing escapes from it. The second line handles the case when no closure escapes in the result. In this case, the analysis f_1 gives back 0 if we can prove that no closure escapes from the variable, and when $f_1 0 = 0$, the analysis f_2 gives the precise escape information.

Theorem 4.3
If $\forall y, \forall x, C(\tau_y[\tau_x], \lambda c.\rho[\![y]\!]cx, \lambda c.\rho_1[\![y]\!]cx, \lambda c.\rho_2[\![y]\!]cx)$, $\forall x, C(\tau_M[\tau_x], \lambda c.E[\![M]\!]\rho cx, \lambda c.F_1[\![M]\!]\rho_1 cx, \lambda c.F_2[\![M]\!]\rho_2 cx)$.

$F_1[\![y]\!]\rho_1 c = \rho_1[\![y]\!]c$
$F_1[\![M\ N]\!]\rho_1 c = \{i \mapsto \phi(i+1), x \mapsto \phi(x) \sqcup \psi(x)\}$
where $\phi = F_1[\![M]\!]\rho_1 c$ and $\psi = F_1[\![N]\!]\rho_1(\phi(1) \sqcup c)$.
$F_1[\![\texttt{fun}\ y \rightarrow M]\!]\rho_1 c = \{1 \mapsto \phi(y), i \mapsto \phi(i-1)$ if $i > 1$,
$x \mapsto \phi(x)\}$ where $\phi = F_1[\![M]\!]\rho_1[y \mapsto \rho_{1\top}]c$
and $\rho_{1\top} c = \{i \mapsto \top_{1i} \sqcup c, y \mapsto c\}$.
$F_1[\![\texttt{let}\ y = M\ \texttt{in}\ N]\!]\rho_1 c =$
$F_1[\![N]\!]\rho_1[y \mapsto F_1[\![M]\!]\rho_1 \sqcup \lambda c.\{y \mapsto c\}]c \sqcup F_1[\![M]\!]\rho_1 c_{|\,Var}$
$F_1[\![\texttt{let rec}\ f(y) = M\ \texttt{in}\ N]\!]\rho_1 c = F_1[\![N]\!]\rho_3 c \sqcup \rho_3(f)c_{|\,Var}$
with $\rho_3 = \mathrm{lfp}(\lambda\rho_3.\rho_1[f \mapsto F_1[\![\texttt{fun}\ y \rightarrow M]\!]\rho_3 \sqcup \lambda c.\{f \mapsto c\}])$
$F_1[\![\texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ P]\!]\rho_1 c =$
$F_1[\![M]\!]\rho_1 c_{|\,Var} \sqcup F_1[\![N]\!]\rho_1 c \sqcup F_1[\![P]\!]\rho_1 c$
$F_1[\![M{::}N]\!]\rho_1 c = F_1[\![M]\!]\rho_1 c \sqcup F_1[\![N]\!]\rho_1 c$
$F_1[\![(M,N)]\!]\rho_1 c = F_1[\![M]\!]\rho_1 c \sqcup F_1[\![N]\!]\rho_1 c$
$F_1[\![M;N]\!]\rho_1 c = F_1[\![M]\!]\rho_1 c_{|\,Var} \sqcup F_1[\![N]\!]\rho_1 c$
$\rho_1[\![:=]\!]c = \{1 \mapsto c, 2 \mapsto \top_{1,2} \sqcup c\}$
$\rho_1[\![a]\!]c = \{\}$ where $a \in \{\texttt{true}, \texttt{false}, \texttt{[]}\} \cup \mathbb{Z}$.
$\rho_1[\![f]\!]c = \{1 \mapsto c, i \mapsto \top_{1i} \sqcup c$ if $i > 1\}$
where f is `hd`, `tl`, `fst`, `snd`, `ref` or `!`.
$\rho_1[\![\texttt{null}]\!]c = \{1 \mapsto c\}$

$F_2[\![y]\!]\rho_2 c = \rho_2[\![y]\!]c$
$F_2[\![M\ N]\!]\rho_2 c = \{i \mapsto \phi(i+1), x \mapsto \phi(x) \sqcup \psi(x)\}$
where $\phi = F_2[\![M]\!]\rho_2 c$ and $\psi = F_2[\![N]\!]\rho_2(\phi(1))$.
$F_2[\![\texttt{fun}\ y \rightarrow M]\!]\rho_2 c = \{1 \mapsto \phi(y), i \mapsto \phi(i-1)$ if $i > 1$,
$x \mapsto \phi(x)\}$ where $\phi = F_2[\![M]\!]\rho_2[y \mapsto \rho_{2\top}]c$
and $\rho_{2\top} c = \{i \mapsto \top_{2i}, y \mapsto c\}$.
$F_2[\![\texttt{let}\ y = M\ \texttt{in}\ N]\!]\rho_2 c =$
$F_2[\![N]\!]\rho_2[y \mapsto F_2[\![M]\!]\rho_2 \sqcup \lambda c.\{y \mapsto c\}]c \sqcup F_2[\![M]\!]\rho_2 0_{|\,Var}$
$F_2[\![\texttt{let rec}\ f(y) = M\ \texttt{in}\ N]\!]\rho_2 c = F_2[\![N]\!]$
$(\mathrm{lfp}(\lambda\rho_4.\rho_2[f \mapsto F_2[\![\texttt{fun}\ y \rightarrow M]\!]\rho_4 \sqcup \lambda c.\{f \mapsto c\}])c$
$F_2[\![\texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ P]\!]\rho_2 c =$
$F_2[\![M]\!]\rho_2 0_{|\,Var} \sqcup F_2[\![N]\!]\rho_2 c \sqcup F_2[\![P]\!]\rho_2 c$
$F_2[\![M{::}N]\!]\rho_2 c = F_2[\![M]\!]\rho_2(c \sqcap \top_{2M}) \sqcup F_2[\![N]\!]\rho_2 c$
$F_2[\![(M,N)]\!]\rho_2 c = F_2[\![M]\!]\rho_2(c \sqcap \top_{2M}) \sqcup F_2[\![N]\!]\rho_2(c \sqcap \top_{2N})$
$F_2[\![M;N]\!]\rho_2 c = F_2[\![M]\!]\rho_2 0_{|\,Var} \sqcup F_2[\![N]\!]\rho_2 c$
$\rho_2[\![\texttt{ref}]\!]c = \{1 \mapsto c \sqcap \top_{2,1}, i \mapsto \top_{2i}$ if $i > 1\}$
$\rho_2[\![:=]\!]c = \{2 \mapsto \top_{2,2}\}$
$\rho_2[\![a]\!]c = \{\}$ where $a \in \{\texttt{true}, \texttt{false}, \texttt{[]}\} \cup \mathbb{Z}$.
$\rho_2[\![f]\!]c = \{1 \mapsto c, i \mapsto \top_{2i}$ if $i > 1\}$
where f is `hd`, `tl`, `fst`, `snd` or `!`.
$\rho_2[\![\texttt{null}]\!]c = \{\}$

Figure 9: Analyses F_1 and F_2

The initial environment satisfies the above hypothesis, hence the conclusion is true and F_1 and F_2 are correct.

Proof sketch: By induction on M. In the function case, we notice that when evaluating $F_1[\![M]\!]\rho_1 cx$, the context passed to subexpressions of M is always at least c, so by induction

$$\forall x \in Var, F_1[\![M]\!]\rho_1 cx \geq \sqcup_{z \in FV(M)} \rho_1[\![z]\!]cx$$

If $F_1[\![\texttt{fun}\ y \rightarrow M]\!]\rho_1 1x = 0$, $\forall z \in FV(\texttt{fun}\ y \rightarrow M), \rho_1[\![z]\!]1x = 0$ so $\rho[\![z]\!]Path\ x \subseteq \gamma_2^x(0)$ therefore $E[\![\texttt{fun}\ y \rightarrow M]\!]\rho\ Path\ x \subseteq \gamma_2^x(0)$. The other cases are easier. □

Theorem 4.4 (Correctness of F_1 and F_2) *Consider the expression* `let x = M in N` *or* `let rec x(y) = M in N`*. Assume that the last operation of M is the allocation of the location ℓ at the top of x. Assume furthermore that*

$$\begin{cases} \textit{if neither } N \textit{ nor } x \textit{ are functional,} \\ \quad F_1[\![N]\!]\rho_1 0x = 0 \textit{ and } F_2[\![N]\!]\rho_2 \top_{2N} x < \top_{2x} \\ \textit{if } N \textit{ is functional, } F_1[\![N]\!]\rho_1 1x = 0 \\ \textit{else } F_1[\![N]\!]\rho_1 0x = 0 \end{cases} \tag{13}$$

Then ℓ can be stack allocated.

Proof sketch: Condition (13) combined with Theorem 4.3 shows that $\top \notin E[\![N]\!]\rho\ Path\ x$. Then Theorem 3.5 yields the conclusion. □

4.1 Polymorphism

We first analyze all expressions as if they were monomorphic, ie type variables are supposed to be atomic : $\top_2[\beta] = 1, \alpha_2^\beta(c_1.\bot) = 0, \alpha_2^\beta(c_1.\top) = 1$ if β is a type variable. We can then infer an approximate analysis for the instantiations of this expression. This can lead to a loss of precision wrt. the direct analysis of the instantiation but, surprisingly, this may also be more precise than the direct analysis. For example,

```
phi = fun x -> fun y ->
        let z = (x,y) in
          a := snd z;
          fst z
```

Here, the analysis gives $F_2[\![\texttt{phi}]\!]\rho c1 = c \sqcup \top_2[\tau]$ if `y` is of type τ. Assume that we analyze the polymorphic version $\tau = \beta$. Then $F_2[\![\texttt{phi}]\!]\rho c1 = c \sqcup 1$ and the instantiation does not change this formula (because we instantiate wrt. the type of `x`, without considering the type of `y`). However, the direct analysis with $\tau =$ `int list` for example gives $F_2[\![\texttt{phi}]\!]\rho c1 = c \sqcup 2$ which is less precise.

Therefore, the analysis by instantiation and the direct analysis are not comparable : our analysis is not polymorphically invariant [1]. We can however prove the correctness of instantiation wrt. the analysis E.

Definition 4.5 Let σ be a given substitution on type variables. The instantiation and generalization functions are respectively:

$$I_2^\tau(c') = \begin{cases} \sqcup\{\top_2[\sigma\tau'] \mid \tau' \in S(\tau), \top_2[\tau'] \leq c'\} & \text{if } c' > 0 \\ 0 & \text{if } c' = 0 \end{cases}$$

$$G_2^\tau(c) = \begin{cases} \sqcup\{\top_2[\tau'] \mid \tau' \in S(\tau), \top_2[\sigma\tau'] \leq c\} \sqcup 1 & \text{if } c > 0 \\ 0 & \text{if } c = 0 \end{cases}$$

where τ is the type associated to the contexts we instantiate or generalize, and $S(\tau)$ is the set of subexpressions of τ, defined as usual except that for the arrow type, the subexpressions are only the subexpressions of the result. For the analysis F_1, the instantiation is $I_1^\tau(c,b) = 1$ if σ substitutes some type variables of τ by function types and $b \neq 0$, and $I_1^\tau(c,b) = c$ otherwise. The generalization is $G_1^\tau(c) = c$.

Theorem 4.6 (Instantiation) *Let $f = \lambda c.E[\![M]\!]\rho cx$.*

$$\begin{aligned} &C(\tau[\tau_x], f, \lambda c.(c \sqcap a_1) \sqcup b_1, \lambda c.(c \sqcap a_2) \sqcup b_2) \\ &\quad \Rightarrow C(\sigma\tau[\sigma\tau_x], f, \lambda c.(c \sqcap a_1) \sqcup I_1^{\tau_x}(b_1, b_2), \\ &\qquad \lambda c.(c \sqcap I_2^{\tau_x}(a_2)) \sqcup I_2^{\tau_x}(b_2)) \end{aligned}$$

This result is useful as every manipulated function is of the form $\lambda c.(c \sqcap a) \sqcup b$ (because this form is stable under composition, meet and join). This proves the correctness of the instantiation functions I_2^τ and I_1^τ.

Proof sketch: First, f can be put under the form $f(c) = \bigsqcup_{i \in I} c_i.c_{|c_i'} \sqcup c_1$ where $c_i \in (\mathrm{app}|\mathrm{l}|\mathrm{r})^*$, $c_i' \in (\rightarrow |\mathrm{l}|\mathrm{r})^*$, $c_1 \in Ctx$ and the join may be infinite. To prove the cases $f(c) = c_1$, and $f(c) = c_1.c_{|c_2}$, we show that $\alpha_2^{\sigma\tau}(c) \leq I_2^\tau \circ \alpha_2^\tau(c)$ when $\alpha_2^{\sigma\tau}(c) \neq \infty$. This means that $I_2^\tau(c)$ represents at least as many paths in type $\sigma\tau$ as c in type τ. □

Example 4.7 The `map` function has type $(\alpha \to \beta) \to \alpha$ `list` $\to \beta$ `list`. The analysis for the polymorphic version is

$\rho_1[\![\texttt{map}]\!]c = \{\texttt{map} \mapsto c, 1 \mapsto c, 2 \mapsto c\}$

$\rho_2[\![\texttt{map}]\!]c = \{\texttt{map} \mapsto c, 1 \mapsto c \sqcap (\texttt{int} \to \underline{\texttt{int}}), 2 \mapsto \underline{\texttt{int}}\ \texttt{list}\}$

When instantiating $\alpha = \tau_1, \beta = \tau_2$, where τ_1 and τ_2 are not functional types, we find

$$\rho_1[\![\texttt{map}]\!]c = \{\texttt{map} \mapsto c, 1 \mapsto c, 2 \mapsto c\}$$
$$\rho_2[\![\texttt{map}]\!]c = \{\texttt{map} \mapsto c, 1 \mapsto c \sqcap (\tau_1 \to \underline{\tau_2}), 2 \mapsto \underline{\tau_1}\ \texttt{list}\}$$

Theorem 4.8 (Generalization)

$$C(\tau[\tau_x], f, f_1, f_2) \Rightarrow C(\sigma\tau[\tau_x], f, f_1, f_2 \circ G_2^\tau)$$

This shows the correctness of the generalization functions $G_1 = \mathrm{id}$ *and* G_2. *This is useful for the case* $f = \lambda c.E[\![y]\!]\rho c x$ *when we instantiate a variable* y *of type* τ *into the type* $\sigma\tau$. *We shall write*

$$F_1[\![\Gamma, y : \tau \vdash y : \sigma\tau]\!]\rho_1 = \rho_1$$
$$F_2[\![\Gamma, y : \tau \vdash y : \sigma\tau]\!]\rho_2 c x = \rho_2(G_2^\tau(c))x$$

Proof sketch: The key point is that $\alpha_2^\tau \leq G_2^\tau \circ \alpha_2^{\sigma\tau}$. $\square$

4.2 Inductive types

We now extend the definition of type levels $\top_2[\tau]$ to inductive types. This level must satisfy the following property: if a value of type τ_1 is inside a value of type τ_2,

$$\top_2[\tau_1] \leq \top_2[\tau_2] \tag{14}$$

Then, in the case of recursive data types, all the locations of the same strongly connected component of the type must have the same level. Between distinct strongly connected components, we add 1 to the level.

Example 4.9

```
type term =
    Var of int
  | Prop of head * term list
and head =
    { name: string;
      mutable props: (term * term) list }
```

$\top_2[\texttt{term}] = \top_2[\texttt{term list}] = \top_2[\texttt{term} * \texttt{term}] = \top_2[(\texttt{term} * \texttt{term})\texttt{list}] = \top_2[\texttt{head}] = 2$ and $\top_2[\texttt{head list}] = 1 + \top_2[\texttt{head}] = 3$ because this type is not is the strongly connected component.

This definition is compatible with the previous definition of $\top_2$ for the `list` type:

```
type 'a list = Nil | Cons of 'a * 'a list
```

If τ is an abstract type, $\top_2[\tau] = 1$ as nothing can be extracted from τ.

However, constraint (14) cannot always be satisfied: if we define non-monotonic recursive types, for example:

α `t = None | Some of` α `*` α `list t`

the cycle check is not enough, as we use infinitely many different types α `list` ... `list t`. In this case, the type is called not level preserving, it is handled as functional types in the preceding analysis and its level is 1. Anyway, such a type has no practical use, as we cannot write an iterator on it.

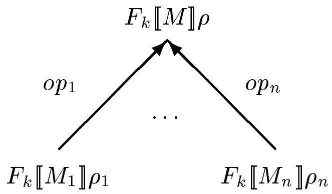


Figure 10: Tree used to represent the equations

5 Implementation

The analyzer is implemented in Caml Special Light (an implementation of the ML programming language). It is relatively small (less than 5000 lines). It works in two passes. The first pass transforms the abstract syntax tree of the analyzed program into a tree labeled with escapement information. The second pass takes this tree and performs the optimizations. Escapement information is stored in a file to allow reusing it for intermodular analysis.

5.1 Computing the escape information

Deutsch's technique [14] can be applied here, except that, as explained in the following, we must label the tree with operations $\lambda c.(c \sqcap f) \sqcup i$ (which can be represented by a pair of contexts) instead of just contexts, to take into account imperative operations. Because of the presence of higher order functions, we also have two analyses F_1 and F_2 instead of just one similar to F_2 in Deutsch's paper.

Semantic equations are represented by a tree whose nodes are occurrences of $F_k[\![M]\!]\rho$ and edges are such that $F_k[\![M]\!]\rho = \sqcup_i F_k[\![M_i]\!]\rho_i \circ op_i$ if $F_k[\![M]\!]\rho$ is linked by edges labeled op_i to $F_k[\![M_i]\!]\rho_i$ (Figure 10). The definition of F_1 and F_2 shows that all the operations op_i are of the form $op = \lambda c.(c \sqcap f) \sqcup i$. We represent them by pairs (f, i).

The composition is defined on the associated pairs:

$$(f_1, i_1) \circ (f_2, i_2) = (f_1 \sqcap f_2, (i_2 \sqcap f_1) \sqcup i_1)$$

If some of these values are unknown (when we compute a fixed point), we create new unknowns to represent the composed operation and emit equations representing the above formula.

The operation $eval(n)$ computes the composition of operations labeling the edges on the path from the node n to the root, doing path compression [37]. We maintain the relation: $F_k[\![M]\!]\rho = \sqcup_{x \in FV(M), n \in \delta(x)} \rho[\![x]\!] \circ eval(n)$ when analyzing M, where $\delta(x)$ is the set of nodes representing occurrences of x. This enables us to compute F_k.

We solve the equations with a generalization of Dijkstra's shortest paths algorithm given by Knuth [28]. It gives the least fixed point of an equations system $Y = \sqcup_i g_i(X_1, \ldots, X_k)$ where g_i are inferior functions: $g_i(x_1, \ldots, x_k) \leq \min(x_1, \ldots, x_k)$. However, the instantiation function I_2^τ (Definition 4.5) is not inferior. To be able to use Knuth algorithm, we first split the system in strongly connected components. We solve each component separately with Knuth's algorithm. Inside a component, we approximate the instantiation by the constant function equal to the level of the type. This is less precise, but correct. Between components, we use the more precise instantiation, which is the most frequent case.

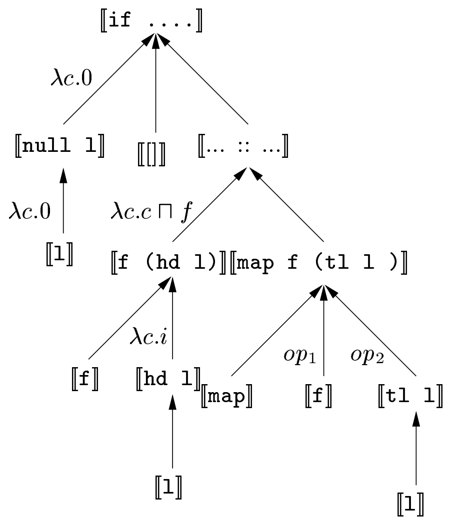


Figure 11: Tree built by analysis F_2 on Example 3.3, without path compression. If f is of type $\tau_1 \to \tau_2$, $f = \top_2[\tau_2]$ and $i = \top_2[\tau_1]$. op_1 and op_2 are the unknowns representing the escapement of the parameters of the `map` function. The identity label is omitted.

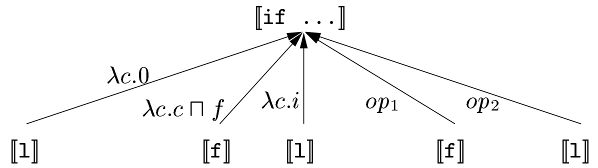


Figure 12: Same as Figure 11, after path compression. We only kept the nodes representing variables for simplicity.

5.2 Program Transformation

We use an expression `letstack` $x = M$ `in` N equivalent to `let` $x = M$ `in` N except that the outer constructor of x will be stack allocated, and deallocated at the end of the execution of N. `letstack`$'$ $x = M$ `in` N also stack allocates the outer constructor of x, but deallocates it before the tail call of N, which enables us to preserve tail call optimizations.

Example 5.1

```
let kb_completion =
  let rec kbrec j rules =
  let rec process failures (k,l) eqs =
   ...
   let enter_rule(left,right) =
     letstack' left_reducible rule =... in
     letstack' right_reduce rule =... in
       ...
       kbrec (j+1) (new_rule::irreds) []
       (k,l) (eqs @ eqs' @ failures)
   in
  ...
```

`left_reducible` and `right_reduce` are deallocated before the recursive call to `kbrec`, which avoids a useless stack growth, and enables us to code the `kbrec` tail call as a jump.

The expression $C[$`let` $x = M$ `in` $N]$ is changed into `letstack` $x = M$ `in` $C[N]$ and $C[M]$ is changed into `letstack` $x = M$ `in` $C[x]$ if the top of x does not escape from $C[N]$ or $C[x]$, ie using Theorem 4.4, if condition (13) is satisfied, which is determined by escape analysis. To perform this transformation, the head of M must be an allocator: a type constructor, or a standard primitive doing an allocation. This transformation must not change the evaluation order, so this limits the size of the context $C[]$ (it must not contain a `fun` expression for example). This may also lead to add more `let`s. For example, we can transform $M_0 M_1 \ldots M_n$ into

$$\begin{aligned} &\texttt{let } x_0 = M_0 \texttt{ in} \\ &\texttt{let } x_n = M_n \texttt{ in} \\ &\ldots \\ &\texttt{let } x_{i+1} = M_{i+1} \texttt{ in} \\ &\texttt{letstack } x_i = M_i \texttt{ in} \\ &x_0 M_1 \ldots M_{i-1} x_i \ldots x_n \end{aligned}$$

We put `let`s for M_j $(j > i)$ because M_j is evaluated before M_i if $j > i$ (CSL evaluates expressions from right to left).

We choose the context $C[]$ as small as possible to reduce the time during which x is in the stack. The idea is as follows: the program transformation takes as a parameter an expression to transform and gives back the transformed expression, and the set of `let`s to put outside of the current expression. These `let`s are collected in a tree, whose edges are labeled with context transformers defining the escapement of the `let` in the current containing expression. These transformers have been computed for each expression by the escape analysis. We perform path compression in this tree using Tarjan's algorithm [37], to update the context transformers as we go up in the abstract syntax tree. As soon as the last `let` (in evaluation order) does not escape any more of the current expression, it is extracted from the tree, and written in the program.

Example 5.2

```
let rec tak (x,y,z) =
  if x>y then
    tak
     (letstack %t1=(x-1, y, z) in tak %t1,
      letstack %t2=(y-1, z, x) in tak %t2,
      letstack %t3=(z-1, x, y) in tak %t3)
  else
      z
```

`%t1`, `%t2` and `%t3` are deallocated each before the allocation of the next, which reduces the stack size (by a factor 3).

According to command line options, we avoid putting a `letstack` which forbids a recursive tail call optimization or any tail call optimization. In this case, only `letstack'` is possible, if the escapement is correct. Indeed, if we put a `letstack` in tail position in a recursive loop, the size of the stack increases at each loop iteration, whereas with tail call optimization, it would not have increased. This can make the program fail, and so must be avoided [9].

To increase opportunities for stack allocation, we perform automatic inlining of small functions which allocate data (only when this effectively allows more stack allocation). For example,

```
let f x = [x];; hd(f 3)
```

becomes

```
hd (let x = 3 in [x])
```

and then

```
let x = 3 in letstack %t1 = [x] in hd %t1.
```

5.3 Complexity

Let n be the size of the program. The number of generated equations e and unknowns u and the time r to compute all right-hand sides of the equations are in $\mathcal{O}(n \log n)$ [14], which gives a $\mathcal{O}(e \times \log u + r) = \mathcal{O}(n \log n \log(n \log n)) = \mathcal{O}(n \log^2 n)$ solving time [28] (the computation of strongly connected components is linear, and therefore dominated by the solving time).

The computation of type levels is dominated by the size of types (more precisely, the sum of the sizes of types declarations that are used). The program transformation performs $\mathcal{O}(n)$ path compressions in a $\mathcal{O}(n)$ forest, so by Theorem 1 of [37], its complexity is $\mathcal{O}(n \log n)$. Finally, the complexity of the analysis is $\mathcal{O}(n \log^2 n + t)$ where t is the type levels compute time.

If inlining is activated, the complexity of course increases. If I is the maximum size of an inlined function (fixed by the user), the size of the resulting program is at most $n' = \mathcal{O}(nI)$, and the analysis time is $\mathcal{O}(nI + n' \log n' + n \log^2 n)$ (we just post-transform the inlined program).

6 Experimental Results

6.1 Compiler Benchmarks

The compiler is given with benchmark programs (Figure 14) which we have tested in several configurations (Figure 15) and on several machines (Figure 13).

taku	Integer computation, allocating tuples
reynolds2	Binary tree search, allocating closures
reynolds3	Binary tree search, allocating pairs
boyer	Terms treatments
kb	Knuth-Bendix's completion algorithm
nucleic	Floating point computations

Figure 14: Tested programs

All	Preserves all tail call optimizations
Rec.	Preserves only recursive tail call opt.
None	Preserves no tail call optimization

Figure 15: Program versions

Inlining does not change anything except on kb and nucleic, so we will only test one version for the other programs. The inlined versions of kb and nucleic will be named kb-inl and nucleic-inl. In the following tables, for each information X, we have given $(X - X_{sta})/X$ where the value is X without stackallocation, and X_{sta} with stackallocation. The last column of the tables gives X. On Figures 21 to 24, the "Total time" curve represents the speedup percentage; the "GC time" curve represents the part of the speedup percentage due to the GC, and the "let" curve represents the part due to the `let`s alone (there is no stack allocation in this case. We only put `let`s where there are `letstack`s in the version with stack allocation, and inlining has been performed as if there were stack allocation). The GC parameters have their default values: GC ratio 30%, minor heap size 32768 words. Uncertainty on time measures is about 1 to 2%, although we have repeated each test 30 times to improve precision.

Not preserving tail recursion optimization enables more stackallocation (Figure 16), but it causes a stack growth (Figure 17) which may cause the failure of the program (even if it does not happen in these benchs). On Sparc 5 and Alpha, it improves speedups, whereas on Pentium Pro, preserving recursive tail call optimizations is often better (nucleic, reynolds3). Stack allocation brings more speedup when the allocated data is larger (because more GC time is then saved up and the cache behavior is improved for a greater number of accesses). Then, a size limit can exist above which stack allocation is more interesting than tail call optimization. This limit depends on complex factors (cache and GC behavior), but it may explain that on Pentium Pro, reynolds2 slightly benefits from not preserving tail call optimizations, whereas reynolds3 loses, as the closures allocated by reynolds2 are larger than the pairs allocated by reynolds3.

Inlining increases the effect of stack allocation, but alone slows down execution on Sparc 5 (probably because of a larger code transfer between memory and chip) which explains the bad results on the "let" curve for nucleic-inl. Its effect is negligible on kb, as we cannot inline enough functions.

Optimal `letstack` moves and `letstack'` also increase stack allocation opportunities. For example, if we preserve recursive tail call optimizations, and disable these improvements, stack allocation becomes negligible for taku. Optimizing the lifetime of data also enables to reduce the stack

Memory size decrease (%)				Size
	All	Rec.	None	(Mwords)
taku	74	74	99	12.7
reynolds2	49	49	99	10.4
reynolds3	9	9	99	31.4
boyer	0	0	16	1.2
kb	1	1	46	8.3
kb-inl	1	1	46	8.3
nucleic	11	11	13	3.8
nucleic-inl	31	43	45	3.8

Figure 16: Tested on Sparc 5

Stack size increase (%)				Size
	All	Rec.	None	(bytes)
taku	25	25	29	996
reynolds2	69	71	71	692
reynolds3	45	45	107	668
boyer	0	0	1	1772
kb	58	58	165	85812
kb-inl	74	74	181	85812
nucleic	0	0	40	2236
nucleic-inl	3	3	44	2236

Figure 17: Tested on Sparc 5

Total speedup (%)				Time
	All	Rec.	None	(ms)
taku	25	23	26	488
reynolds2	1	0	1	6072
reynolds3	2	0	-6	3070
boyer	0	0	5	523
kb	0	0	2	1332
kb-inl	0	0	2	1339
nucleic	6	6	0	696
nucleic-inl	6	10	5	692

Figure 18: Tested on Pentium Pro

Tested computer	Primary cache	Secondary cache	SPECint95
Alpha DEC 3000/300	8kb(I) + 8kb(D)	512kb(I+D)	N/A
Pentium Pro 200MHz	8kb(I) + 8kb(D)	256kb(I+D)	8.20
Sparc 5 110MHz	16kb(I) + 8kb(D)	None	1.59

Figure 13: Machines characteristics

size (on taku, it is 25% larger instead of 75% if we take the longest lifetime).

GC speedup on total time (%)				Time
	All	Rec.	None	(ms)
taku	3	2	3	2241
reynolds2	0	0	0	13707
reynolds3	0	0	2	8087
boyer	1	1	6	1876
kb	0	0	2	4937
kb-inl	0	0	3	4917
nucleic	0	1	1	1942
nucleic-inl	1	2	2	1912

GC speedup (%)				GC time
	All	Rec.	None	(ms)
taku	83	72	100	81
reynolds2	56	42	100	112
reynolds3	29	20	100	235
boyer	2	2	9	1425
kb	-1	0	6	1543
kb-inl	0	2	9	1558
nucleic	1	15	17	162
nucleic-inl	20	27	25	155

Figure 19: Tested on Sparc 5

Tests concerning the GC give two results: the speedup does not depend on the GC parameters (GC ratio, minor heap size), and the speedup due to the GC represents a minor part of the total speedup (Figure 19), which confirms [27]. This can be explained as we mainly stack allocate short lived data, which have no cost for the GC, since they are not scanned by minor GCs.

So most of the speedup is due to a better data locality, as only the data on top of the stack are used in general. For example, without stack allocation taku allocates 12Mwords, which are located in a 32kwords minor heap, whereas with stack allocation, it uses only a less than 1kword stack.

The Alpha is the machine which benefits the most from stack allocation. taku gives the best results, but kb, reynolds2, reynolds3 and nucleic (with inlining) are also subject to important improvements.

6.2 Coq

Coq is a theorem prover that we have tested by making it compile its own benchmarks. In all these tests, `letstack'`, optimal `let` moves and inlining are allowed.

Speedup (%)			Time
	Rec.	None	(s)
Sparc 5	-2.1	3.0	5713
Pentium Pro	3.0	4.3	2339

GC speedup on total time (%)			GC time
	Rec.	None	(s)
Sparc 5	1.2	3.8	2942
Pentium Pro	1.7	2.9	1342

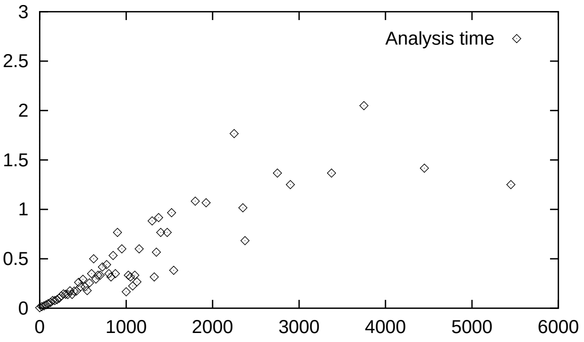


Figure 20: Analysis time (in seconds) as a function of the size of the expression (number of nodes in the syntax tree)

During the bench, Coq allocates 5,25 gigawords, and puts 17% of them on the stack when recursive tail call optimizations are preserved, and 25% when they are not. Inlining of small functions enables us to stack allocate much more data (we could not stack allocate more than 11% of data without inlining).

6.3 Analysis speed

The analysis speed enables us to use it to compile large programs, of which Coq is a good example featuring 65000 lines. The analysis takes 14% of the compile time on Sparc 5 with inlining. The compilation is 19% longer because the compilation of the transformed code is a bit longer than the one of the initial code, especially with inlining. However Coq does not contain only ML code, so there is a part of the work which is not subject to the optimization. This shows nevertheless that the analysis is not too costly. Figure 20 shows that it is nearly linear. We can analyze each recursive declaration independently, which reduces the value of n in the complexity $\mathcal{O}(n \log n)$.

Let C be the time to compile the compiler itself without stack allocation, A the escape analysis time, C_a the compile time when enabling stack allocation, and C_{ao} the compile time with stack allocation on an optimized (stack-allocating) compiler. The analysis cost is given in the following table.

	$\frac{A}{C}$(%)	$\frac{C_a - C}{C}$(%)	$\frac{C_{ao} - C}{C}$(%)
Alpha	17	19	17
Sparc 5	16	20	19
Pentium Pro	19	21	19

7 Conclusion

We have implemented an escape analysis modified from Park and Goldberg's [32], but we have much improved its complexity thanks to Deutsch's method [14]. We also have extended it to inductive types, imperative operations and (with approximations) higher order functions.

The results given by this analysis are quite satisfactory. Experience has shown this analysis can be applied even to

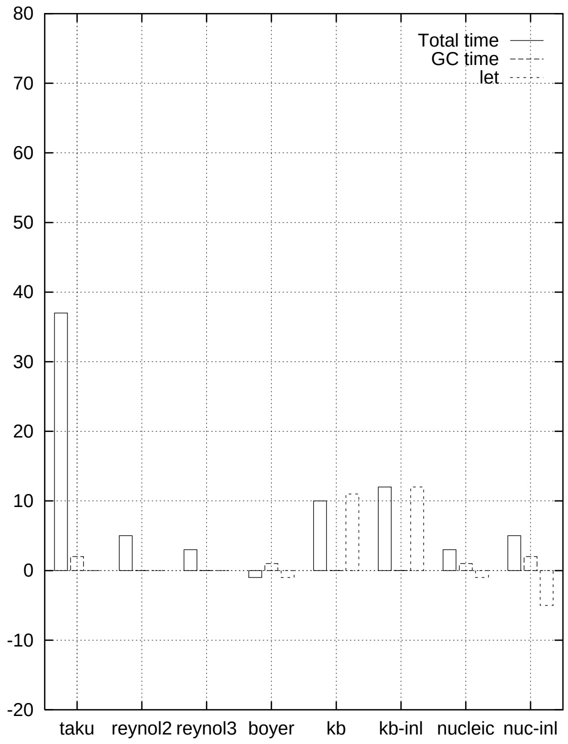


Figure 21: Tested on Sparc 5, preserving recursive tail call optimization.

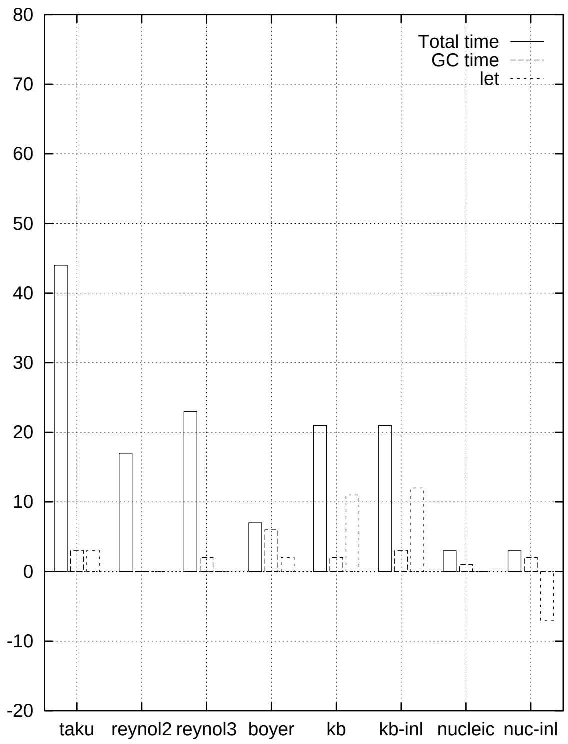


Figure 22: Tested on Sparc 5, preserving no tail call optimization

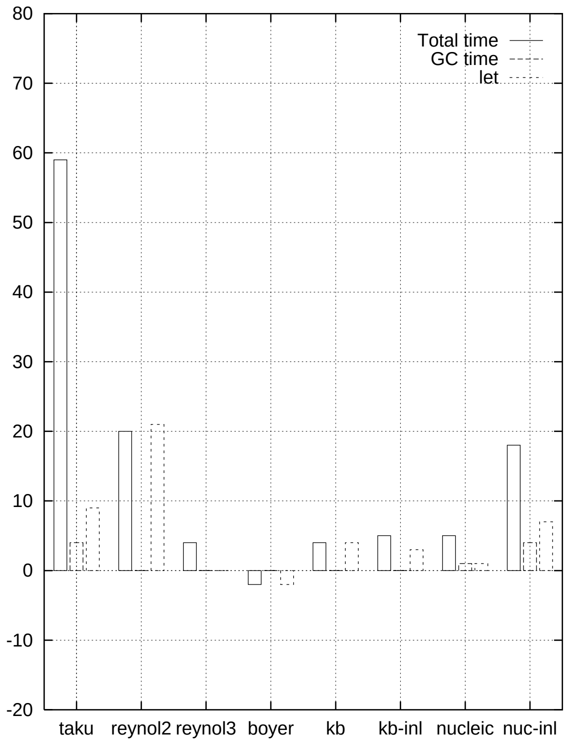


Figure 23: Tested on Alpha, preserving recursive tail call optimization

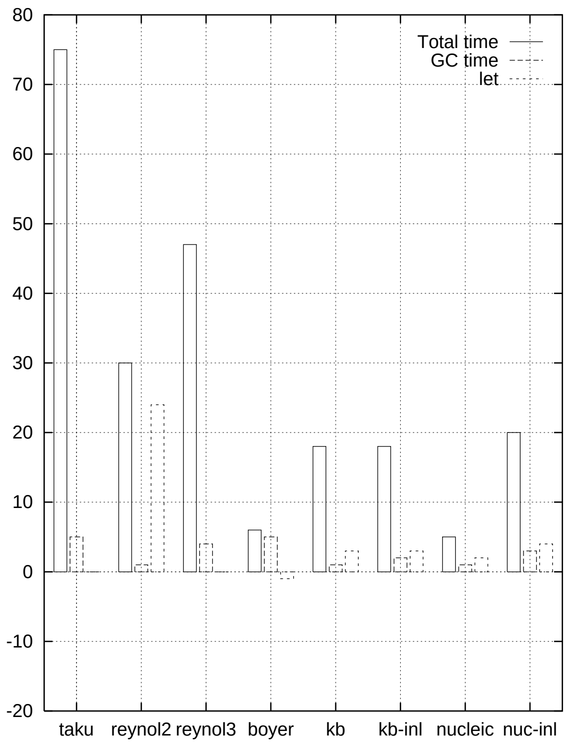


Figure 24: Tested on Alpha, preserving no tail call optimization

large programs with at most 20% compile time overhead, thanks to its reasonable cost in $\mathcal{O}(n \log^2 n)$ (assuming type levels are precomputed). It also demonstrates the benefits of stack allocation, as we get important speedups: often 5 to 20%, sometimes more in particularly good cases. Improvements are less important for the theorem prover Coq (4%).

Execution speedups are explained to a very limited extend by a decrease of the GC workload. In fact, the observed speedups are essentially accounted for by an improvement in data locality. The best results are obtained when not preserving tail recursion optimization on Sparc 5 and Alpha. This solution is unsafe, as it may cause stack overflow, but this does not happen in the considered examples. On the other hand, on Pentium Pro processors, it is better to keep tail recursion optimization, as it balances the improvement of stack allocation.

Inlining of small functions to increase stack allocation opportunities and a better adaptation of the GC to stack allocation have slightly improved the results. However, the quality of the GC of CSL [15] reduces speedup possibilities.

Acknowledgements

Many thanks to Alain Deutsch for his help during this work.

References

[1] Abramsky, S. Strictness analysis and polymorphic invariance. In *Proc. Programs as Data Objects* (1986), vol. 217 of *Lecture Notes on Computer Science*, Springer Verlag.

[2] Aiken, A., Fähndrich, M., and Levien, R. Better static memory management: Improving region-based analysis of higher-order languages. In *PLDI95* (San Diego, California, June 1995).

[3] Appel, A. W. Garbage Collection can be faster than Stack Allocation. *Information Processing Letters 25*, 4 (Jan. 1987), 275 – 279.

[4] Appel, A. W., and Shao, Z. An Empirical and Analytic Study of Stack vs. Heap Cost for Languages with Closures. *Journal of Functional Programming 1*, 1 (Jan. 1993), 1 – 27.

[5] Birkedal, L., Tofte, M., and Vejlstrup, M. From Region Inference to von Neumann Machines via Region Representation Inference. In *23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Jan. 1996).

[6] Blanchet, B. Garbage collection statique. DEA report, INRIA, Rocquencourt, Sept. 1996.

[7] Burn, G. A relationship between abstract interpretation and projection analysis. In *Principles of Programming Languages* (Jan 1990), ACM, pp. 151–156.

[8] Carr, S., McKinley, K. S., and Tseng, C.-W. Compiler optimizations for improving data locality. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct. 1994), pp. 252 – 262.

[9] Chase, D. R. Safety considerations for storage allocation optimizations. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation* (22-24 June 1988), ACM Press, pp. 1 – 10.

[10] Cousot, P., and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Annual ACM Symposium on Principles of Programming Languages* (Jan. 1977), pp. 238 – 252.

[11] Cousot, P., and Cousot, R. Systematic design of program analysis frameworks. In *Sixth Annual ACM Symposium on Principles of Programming Languages* (Jan. 1979), pp. 269 – 282.

[12] Deutsch, A. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Seventeenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Jan. 1990), pp. 157 – 168.

[13] Deutsch, A. Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation* (20-24 June 1994), ACM Press, pp. 230 – 241.

[14] Deutsch, A. On the Complexity of Escape Analysis. In *24th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Jan. 1997).

[15] Doligez, D. *Conception, réalisation et certification d'un glaneur de cellules concurrent.* PhD thesis, Université Paris VII, May 1995.

[16] Hannan, J. A Type-based Analysis for Stack Allocation in Functional Languages. In *Proceedings of the Second International Static Analysis Symposium (SAS '95)* (Sept. 1995).

[17] Harper, R., Milner, R., and Tofte, M. The definition of Standard ML. LFCS Report Series ECS-LFCS-89-81, Laboratory for Foundations of Computer Science, Dept. of Computer Science, University of Edinburgh, May 1989.

[18] Harrison, W. The interprocedural analysis and automatic parallelisation of Scheme programs. *Lisp and Symbolic Computation 2* (1989), 176 – 396.

[19] Hederman, L. Compile Time Garbage Collection Using Reference Count Analysis. Tech. Rep. Rice COMP TR88-75, Rice University, Houston, Texas, Aug. 1988.

[20] Hicks, J. Experiences with Compiler-Directed Storage Reclamation. In *Conference on Functional Programming Languages and Computer Architecture (FPCA'93)* (June 1993), ACM Press.

[21] Hoare, C. Some properties of predicate transformers. *Journal of the ACM 25*, 3 (1978), 461 – 480.

[22] Hudak, P. A Semantic Model of Reference Counting and its Abstraction (Detailed Summary). In *Proceedings of the 1986 ACM Conference on LISP and functional programming* (Aug. 1986), pp. 351 – 363.

[23] Hughes, J. Backward Analysis of Functional Programs. In *Partial Evaluation and Mixed Computation* (1988), D. Bjørner, A. P. Ershov, and N. D. Jones, Eds., Elsevier Science Publishers B.V. (North Holland).

[24] Hughes, S. Compile-Time Garbage Collection for Higher-Order Functional Languages. *J. Logic Computat. 2*, 4 (1992), 483 – 509.

[25] Inoue, K., Seki, H., and Yagi, H. Analysis of Functional Programs to Detect Run-Time Garbage Cells. *ACM Transactions on Programming Languages and Systems 10*, 4 (Oct. 1988), 555 – 578.

[26] Jones, N. D., and Muchnick, S. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Nineth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (1982), pp. 66 – 74.

[27] Jones, S. B., and White, M. Is compile time garbage collection worth the effort. In *Workshop on Functional Programming* (Glasgow, 1990), Springer.

[28] Knuth, D. E. A Generalization of Dijkstra's Algorithm. *Information Processing Letters 6*, 1 (Feb. 1977), 1 – 5.

[29] Mohnen, M. Efficient closure utilisation by higher-order inheritance analysis. In *Static Analysis Symposium (SAS'95)* (1995).

[30] Mohnen, M. Efficient compile-time garbage collection for arbitrary data structure. In *PLILP'95* (1995).

[31] Neyrinck, A., Panangaden, P., and Demers, A. Computation of aliases and support sets. In *Fourteenth Annual ACM Symp. on Principles of Programming Languages* (Jan. 1987), ACM Press, pp. 274–283.

[32] Park, Y. G., and Goldberg, B. Escape Analysis on Lists. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation* (17-19 July 1992), vol. 27, pp. 116 – 127.

[33] Ruggieri, C., and Murtagh, T. P. Lifetime Analysis of Dynamically Allocated Objects. In *Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Jan. 1988), pp. 285 – 293.

[34] Serrano, M., and Feeley, M. Storage Use Analysis and its Applications. In *1996 ACM SIGPLAN International Conference on Functional Programming* (May 1996).

[35] Shivers, O. Control flow analysis in Scheme. *Conference on Programming Language, Design and Implementation* (June 1988), 164 – 174.

[36] Tarjan, R. Finding dominators in Directed Graphs. *SIAM Journal of Computing 3*, 1 (Mar. 1974), 62 – 89.

[37] Tarjan, R. E. Applications of Path Compression on Balanced Trees. *Journal of the Association for Computing Machinery 26*, 4 (Oct. 1979), 690 – 715.

[38] Tofte, M., and Talpin, J.-P. A theory of Stack Allocation in Polymorphically Typed Languages. Tech. Rep. 93/15, Departement of Computer Science, Copenhagen University, 9 July 1993.