Storage Use Analysis and its Applications

Manuel Serrano[1,2] *and Marc Feeley*[1]
{Serrano,Feeley}@IRO.UMontreal.CA

[1] Université de Montréal C.P. 6128, succ. centre-ville, Montréal Canada H3C 3J7
[2] INRIA B.P. 105, Rocquencourt, 78153 Le Chesnay Cedex, France

Abstract

In this paper we present a new program analysis method which we call *Storage Use Analysis*. This analysis deduces how objects are used by the program and allows the optimization of their allocation. This analysis can be applied to both statically typed languages (e.g. ML) and latently typed languages (e.g. Scheme). It handles side-effects, higher order functions, separate compilation and does not require CPS transformation. We show the application of our analysis to two important optimizations: stack allocation and unboxing. The first optimization replaces some heap allocations by stack allocations for user and system data storage (e.g. lists, vectors, procedures). The second optimization avoids boxing some objects. This analysis and associated optimizations have been implemented in the Bigloo Scheme/ML compiler. Experimental results show that for many allocation intensive programs we get a significant speedup. In particular, numerically intensive programs are almost 20 times faster because floating point numbers are unboxed and no longer heap allocated.

1 Introduction

Modern strict functional languages such as Scheme and ML are still often much less efficient than traditional imperative languages such as Fortran and C. Few compilers for functional languages are able to produce executable programs whose efficiency is close to that of imperative ones [12]. To a large extent, this inefficiency is due to poor use of memory. Because read and write operations are much more expensive than arithmetic operations and control operations on modern computers, memory access is a major performance issue. Consequently, an efficient system must allocate as few objects as possible and must choose very carefully the location where the objects are allocated. Let's discuss these two points further.

- High allocation rate:

  For languages like Scheme and ML polymorphism is difficult to implement efficiently. With these languages, functions which accept several kinds of arguments are legal, such as an "identity" function which accepts characters, fixnums and flonums. This feature is hard to handle efficiently (fixnums and flonums are not generally of the same size and cannot be stored in the same kind of hardware registers). The traditional solution is to "box" values, i.e. use pointers to values in memory rather that direct values. This *uniform representation* [19] is inefficient because it requires memory allocation for all objects. In this paper, we present an algorithm which allows *mixed representation*. With this framework, values can be directly represented without requiring indirections. There are two benefits: memory allocation is less frequent and values are accessed more efficiently.

- Location of allocation:

  Some programs make heavy use of objects with dynamic extent (nested lifetime). Stack based language implementations[1] which do not exploit this characteristic will pay a higher cost for allocation than is required. Instructions are already present in the program to allocate and deallocate activation frames. Objects with dynamic extent could be allocated (and deallocated) at no cost in the frames. In order to automatically find when objects can be allocated in the stack we have designed a conservative analysis which determines if objects have dynamic extent.

The two optimizations presented in this paper (mixed representation and stack allocation) use the same analysis but in different ways. Section 2 first develops this analysis for a small source language and then the language is extended to obtain a language with data storage, side effects and modules. The stack allocation decision algorithm is presented in Section 3 and the mixed representation in Section 4. We have implemented these two optimizations in the Bigloo Scheme/ML compiler, and have measured the gain in performance on benchmark programs. The experimental results are presented in Section 5.

2 Storage Use Analysis (SUA)

In this section, we will present Storage Use Analysis (SUA) by first describing the analysis for a simple first-order language with only fixnum and flonum values and then we will extend it by adding several data types (higher-order functions, lists and vectors).

2.1 The input language Λ

The input language for the first version of our analysis is a simple language resembling Lisp (functions are second class citizens and closures do not exist), with only immediate values (fixnum and flonum), and without any data storage. Λ's grammar is shown below:

[1] Some systems like Sml/NJ [2] allocate activation frames in the heap.

Syntactic categories

v	$\in$	VarId	(Variables identifier)
f	$\in$	FunId	(Functions identifier)
Λ	$\in$	Exp	(Expressions)
k	$\in$	Cnst	(Constant values)
Π	$\in$	Prgm	(Program)
Γ	$\in$	Def	(Definition)

Concrete syntax

$$
\begin{array}{lll}
\Pi & ::= & \Gamma \ldots \Gamma \\
\Gamma & ::= & (\texttt{define}\ (f\ v \ldots v)\ \Lambda) \\
\Lambda & ::= & k \\
 & | & v \\
 & | & (\texttt{labels}\ ((f\ (v \ldots v)\ \Lambda) \ldots)\ \Lambda) \\
 & | & (\texttt{if}\ \Lambda\ \Lambda\ \Lambda\ ) \\
 & | & (\texttt{set!}\ v\ \Lambda) \\
 & | & (f\ \Lambda \ldots \Lambda) \\
 & | & (+\ \Lambda\ \Lambda)
\end{array}
$$

Note that since functions are first-order Λ is *not* a functional language. A program is composed of several global function definitions; local functions are introduced by the `labels` special form. The language includes side effects on variables (the `set!` form).

2.2 The first-order SUA

For the sake of simplicity, we will consider the last function definition as the entry point of the program (equivalent to the C `main` function). So running a Λ program means calling the last function defined, with no arguments.

$\mathcal{S}ua^0$(Π)=
  repeat
    $\mathcal{S}ua^0_{app}$($\Pi{\downarrow}_{main}$)
  until no approximation set changed in this iteration

$\mathcal{S}ua^0_{app}$(f, a_1, ..., a_n)=
  $\forall i \in [1..n]$
    let x=$\mathcal{S}ua^0_{ast}$(a_i)
      $\mathcal{A}_{var}$($f{\downarrow}_{arg_i}$) $\leftarrow$ $\mathcal{A}_{var}$($f{\downarrow}_{arg_i}$)$\bigcup x$,
  if f not yet processed in this iteration
    then let x=$\mathcal{S}ua^0_{ast}$($f{\downarrow}_{body}$)
      $\mathcal{A}_{var}$($f{\downarrow}_{res}$) $\leftarrow$ $\mathcal{A}_{var}$($f{\downarrow}_{res}$)$\bigcup x$,
  $\mathcal{A}_{var}$($f{\downarrow}_{res}$)

$\mathcal{S}ua^0_{ast}$(*atree*)=
  case *atree*
    $[\![$ k $]\!]$:
      {$\mathcal{T}$(k)}
    $[\![$ *var* $]\!]$:
      $\mathcal{A}_{var}$(*var*)
    $[\![$ (`if` *atree* $atree_t$ $atree_f$) $]\!]$:
      $\mathcal{S}ua^0_{ast}$(*atree*),
      $\mathcal{S}ua^0_{ast}$($atree_t$)$\bigcup \mathcal{S}ua^0_{ast}$($atree_f$)
    $[\![$ (`set!` *var* *val*) $]\!]$:
      let x=$\mathcal{S}ua^0_{ast}$(*val*)
        $\mathcal{A}_{var}$(*var*) $\leftarrow$ $\mathcal{A}_{var}$(*var*)$\bigcup x$,
        $\emptyset$
    $[\![$ (`labels` ((f_1 (v_1 ... v_n) $atree_1$) ...) *atree*) $]\!]$:
      $\mathcal{S}ua^0_{ast}$(*atree*)
    $[\![$ (+ a_1 a_2) $]\!]$:
      {`fixnum`, `flonum`}
    $[\![$ (f a_1 ... a_n) $]\!]$:
      $\mathcal{S}ua^0_{app}$(f, a_1, ..., a_n)
  end

Algorithm 2.1: A first-order analysis

The SUA algorithm shown in algorithm 2.1 computes type information about variables and function results (which will both be called variables). The result of the analysis is an "approximation set" for each variable, which indicates the type of values that can be bound to the variable. Since the only data types are fixnum and flonum, an approximation set is a subset of the set $\mathcal{A}^0 = \{\texttt{fixnum}, \texttt{flonum}\}$. Note that because the analysis is conservative an approximation set is a superset of the true set of types that can be bound to the variable.

Note that the SUA algorithm requires α-converted programs. It is written in an intuitive pseudo language which uses the following notation:

$\mathcal{T}(k)$	The type of a constant (`fixnum` or `flonum`).
$\Pi\downarrow_{main}$	The program entry point.
$\mathcal{A}_{var}(v)$	The approximation set of variable v.
$f\downarrow_{body}$	The body of function f.
$f\downarrow_{arg_i}$	The i^{th} formal parameter of function f.
$f\downarrow_{res}$	The artificial variable representing the result of function f.

The algorithm performs a fix point iteration. Each iteration is a depth first traversal of the entire call graph initiated by the program's entry point. The fix point iteration stops when an iteration does not add any new information. This process is guaranteed to stop because there is a finite number of variables, a finite number of possible approximation sets, and no element is ever removed from a variable's approximation set. Let's study SUA's behavior with the program:

```
(define (id x) x)
(define (plus a b) (+ a b))
(define (foo) (plus (id 4) (id 5.0)))
```

The analysis collects approximation sets for `x`, `a` and `b` and the result of `id`, `plus` and `foo`. The traversal of the call graph starts with the body of `foo` which leads to a call of the function `id` with the value `4`. We are collecting type information so this call to `id` assigns the approximation set {`fixnum`} to `x`. Since `id` returns `x`, the approximation set {`fixnum`} is also assigned to the result of `id`. After processing this first call to `id`, the analysis examines the second call `(id 5.0)`. This time, the approximation set for `id`'s argument is {`flonum`} so the analysis assigns the approximation set {`fixnum`,`flonum`} to `x`. In a given iteration, the depth first traversal of the call graph will not visit a function's body more than once, so `id`'s body is not processed again and the approximation set of `id`'s result is not changed (this is done in the next iteration). After the second call to `id`, the call to `plus` is processed. Since at this point the approximation set of `id`'s result is {`fixnum`}, the analysis assigns the approximation set {`fixnum`} to `a` and `b`.

During the second iteration, when the analysis processes the body of `id`, `flonum` is added to the approximation set of `id`'s result. This approximation set is propagated to `a` and `b` and the fix point is reached in 3 iterations. SUA concludes that all variables and function return values of this program can be a `fixnum` or a `flonum`.

Implementation note: *The set of variables and function results is finite and known at compile time. This property is important because it allows efficient implementation of $\mathcal{A}_{var}$ table using efficient set representations (e.g. using bit-vectors).*

2.3 The first-order SUA with modules

We now extend Λ to support modules. Rather than add new constructions to the language we will assume that all global functions are *exported* (i.e. that they are visible in other modules).

From the compiler's point of view, the fact that a function is exported means that its actual parameters may be unknown because it can be invoked outside the current module. To handle this we have to introduce a representative for unknown values. As customary [8] this is noted "top" ($\top$). Approximation sets are now subsets of $\mathcal{A}^1 = \{\top, \texttt{fixnum}, \texttt{flonum}\}$. When a variable's approximation set contains $\top$, it means that any value may be bound to the variable. Algorithm 2.2 contains the updated SUA algorithm[2]. It makes use of the new notation:

$\Pi\downarrow_{export}$ The set of Π's exported functions.

In this new version of the analysis, the traversal of the call graph is initiated by all exported functions. The formal parameters of all exported functions are initially approximated by $\{\top\}$. The function $\mathcal{A}_{\text{spread-}\top}$ will be useful later on to spread $\top$ into data storage approximations.

$\mathcal{S}ua^1(\ \Pi\ )=$
  $\forall f\in\Pi\downarrow_{export}$
    $\mathcal{S}ua^1_{export}(\ f\ )$

$\mathcal{S}ua^1_{export}(\ f\ )=$
  let n=f's arity
    $\forall i\in[1..n]$
      $\mathcal{A}_{\text{var}}(\ f\downarrow_{arg_i}\ ) \leftarrow \mathcal{A}_{\text{var}}(\ f\downarrow_{arg_i}\ )\bigcup\{\top\}$,
    if f not yet processed in this iteration
      then let $x=\mathcal{S}ua^1_{ast}(\ f\downarrow_{body}\ )$
        $\mathcal{A}_{\text{var}}(\ f\downarrow_{res}\ ) \leftarrow \mathcal{A}_{\text{var}}(\ f\downarrow_{res}\ )\bigcup x$,
    $\mathcal{A}_{\text{spread-}\top}(\ \mathcal{A}_{\text{var}}(\ f\downarrow_{res}\ )\ )$

$\mathcal{S}ua^1_{app}(\ f,\ a_1,\ \ldots,\ a_n\ )=$
  if f is imported
    then $\{\top\}$
    else $\forall i\in[1..n]$
        let $x=\mathcal{S}ua^1_{ast}(\ a_i\ )$
          $\mathcal{A}_{\text{var}}(\ f\downarrow_{arg_i}\ ) \leftarrow \mathcal{A}_{\text{var}}(\ f\downarrow_{arg_i}\ )\bigcup x$,
      if f not yet processed in this iteration
        then let $x=\mathcal{S}ua^1_{ast}(\ f\downarrow_{body}\ )$
          $\mathcal{A}_{\text{var}}(\ f\downarrow_{res}\ ) \leftarrow \mathcal{A}_{\text{var}}(\ f\downarrow_{res}\ )\bigcup x$,
      $\mathcal{A}_{\text{var}}(\ f\downarrow_{res}\ )$

$\mathcal{A}_{\text{spread-}\top}(\ a\ )=\ a$

Algorithm 2.2: A first-order analysis with modules

2.4 The higher-order SUA with modules

We now extend the analysis to accept as input a higher-order functional language. Three new constructions are added to Λ: `make-closure` (to create closures), `closure-ref` (to access a closure's free variables) and `closure-call` (to invoke a closure). Here are the modifications to Λ's grammar:

Λ ::= ...
  | `(make-closure` $f\ v\ \ldots v$`)`
  | `(closure-ref` $v\ k$`)`
  | `(closure-call` $\Lambda\ v\ \ldots v$`)`

The usefulness of these constructs rests in the ability to easily translate Scheme programs into Λ. The translation of the following program:

```
(define (curry-plus x) (lambda (y) (+ x y)))
(define (add a b) ((curry-plus a) b))
(define (main) (add 3.4 5.6))
```

is:

```
(define (curry-plus x) (make-closure₁ f x))
(define (f p y) (+ (closure-ref p 0) y))
(define (add a b) (closure-call (curry-plus a) b))
(define (main) (add 3.4 5.6))
```

The translation required to map Scheme, ML or other higher-order functional languages to Λ is the so-called λ-lifting transformation [16].

$\mathcal{S}ua^2_{ast}(\ atree\ )=$
  case *atree*
    ⋮
    $[\![$ `(closure-call` $e\ a_1\ \ldots\ a_n$`)` $]\!]$:
      $\bigcup_{f\in\mathcal{S}ua^2_{ast}(\ e\ )} \mathcal{S}ua^2_{clocall}(\ f,\ a_1,\ \ldots,\ a_n\ )$
    $[\![$ `(make-closure`$_i$ $f\ v_1\ \ldots$`)` $]\!]$:
      let $a_1=\mathcal{S}ua^2_{ast}(\ v_1\ ),\ \ldots$
        $\mathcal{A}_{\text{clo}}(\ i\ ) \leftarrow \widehat{\texttt{make-closure}}(\ f,\ a_1,\ \ldots\ )$,
        $\{\texttt{clo}_i\}$
    $[\![$ `(closure-ref` $f\ k$`)` $]\!]$:
      $\bigcup_{\texttt{clo}_i\in\mathcal{S}ua^2_{ast}(\ f\ )} \widehat{\texttt{closure-ref}}(\ \mathcal{A}_{\text{clo}}(\ i\ ),\ k\ )$
  end

$\mathcal{S}ua^2_{clocall}(\ e,\ a_1,\ \ldots\ )=$
  case e
    $\texttt{clo}_i$:
      $\mathcal{S}ua^2_{app}(\ \widehat{\texttt{closure-function}}(\ \mathcal{A}_{\text{clo}}(\ i\ )\ ),\ a_1,\ \ldots\ )$
    else:
      $\mathcal{S}ua^2_{failure}()$
  end

$\mathcal{S}ua^2_{app}(\ f,\ a_1,\ \ldots,\ a_n\ )=$
  if $n\ =\ f$'s arity
    then $\mathcal{S}ua^1_{app}(\ f,\ a_1,\ \ldots,\ a_n\ )$
    else $\mathcal{S}ua^2_{failure}()$

$\mathcal{S}ua^2_{failure}()=\ \emptyset$

$\mathcal{A}_{\text{spread-}\top}(\ a\ )=$
  $\forall\texttt{clo}_i\in a$
    $\mathcal{S}ua^2_{export}(\ \widehat{\texttt{closure-function}}(\ \mathcal{A}_{\text{clo}}(\ i\ )\ )\ )$,
  a

Algorithm 2.3: A higher-order analysis with modules

Closures introduced by `make-closure` are for now the only data structures of our language. SUA is modified to compute information about types *and* data storage by adding "closure approximations". A closure approximation is a tuple containing a closure function and a closure environment (a list of approximation sets, one for each free variable). Closure approximations are created by the function

[2] function $\mathcal{S}ua^1_{ast}$ is not defined here because it has the same definition as $\mathcal{S}ua^0_{ast}$ (with references to $\mathcal{S}ua^0_{ast}$ replaced with $\mathcal{S}ua^1_{ast}$). This kind of misuse will be used in the remainder of the paper to avoid redundancy.

$\widehat{\texttt{make-closure}}$. The function associated with closure approximation a is obtained with $\widehat{\texttt{closure-function}}(a)$ and $\widehat{\texttt{closure-ref}}(a, i)$ returns the approximation set associated with the i^{th} free variable of closure approximation a.

Closure approximations are stored in a closure approximation table named $\mathcal{A}_{\text{clo}}$. There is a one-to-one correspondence between entries in this table and `make-closure`s in the program. To add a closure approximation to an approximation set, $\texttt{clo}_i$ is added to the set, where i is the entry's index in $\mathcal{A}_{\text{clo}}$. In this version of our algorithm, approximation sets are subsets of $\mathcal{A}^2 = \{\top, \texttt{fixnum}, \texttt{flonum}, \texttt{clo}_1, \ldots, \texttt{clo}_k\}$, where k is the number of `make-closures` in the program.

A new problem arises with functional values. In latently typed languages `closure-call` can lead to two possible errors: the object given to `closure-call` is not a closure or the number of arguments is incompatible with the function called. We have to deal with these possible errors in the SUA. For the sake of simplicity we suppose that `closure-ref` is always correct. This is reasonable since these constructions are inserted by the program which is in charge of the λ-lifting and not by the user. The treatment of errors is straightforward: errors just produce empty approximation sets. This means that an error leaves all the approximation sets as they are. This is sound because at run time, if an error occurs, the program is interrupted, so errors *do not* return values.

Our handling of closures has been guided by their special nature: they are immutable data (since we are using flat closures, mutable free variables are stored in cells) and they are always accessed via the `closure-ref` procedure which requires a constant index as second argument. It is thus possible to distinguish the free variables.

Algorithm 2.3 presents the extensions to SUA needed to accept the higher-order version of Λ (we assume function $\mathcal{S}ua^1_{app}$ in the algorithm uses the new version of the graph traversal function, i.e. $\mathcal{S}ua^2_{ast}$). The $\mathcal{A}_{\text{spread-}\top}$ function also requires a slight modification: if a closure can be returned by an exported function, this closure is also exported as it can be invoked with unknown actual parameters. The new $\mathcal{A}_{\text{spread-}\top}$ handles this.

Let's study SUA on the example of the curried addition `curry-plus`. Assuming no exported functions, the iteration process starts by traversing `main`. The call to `add` assigns the approximation set $\{\texttt{flonum}\}$ to `a` and `b`. The call `curry-plus` in turn assigns the approximation set $\{\texttt{flonum}\}$ to `x`, and the function's result is assigned the approximation set $\{\texttt{clo}_1\}$ after storing a closure approximation over one flonum (i.e. $\widehat{\texttt{make-closure}}(\texttt{f}, \{\texttt{flonum}\})$) in $\mathcal{A}_{\text{clo}}(1)$. The first argument of the `closure-call` has the approximation set $\{\texttt{clo}_1\}$, so SUA continues by analyzing `f`'s body with the approximation set $\{\texttt{clo}_1\}$ for `p`. The `closure-ref` thus returns the approximation set $\{\texttt{flonum}\}$.

2.5 The higher-order SUA with modules and lists

SUA can be easily extended to accept other data types. In this section, we present how lists are added to the analysis. Lists (in Lisp and Scheme) differ from closures because they are mutable data. Lists are built out of pairs. The two fields of a pair can be distinguished in our approximation scheme (just like all the free variables of a closure are distinguished).

```
Λ  ::=   ...
       | (cons Λ Λ)
       | (car Λ)
       | (cdr Λ)
       | (set-car! Λ Λ)
       | (set-cdr! Λ Λ)
```

The handling of pairs in the SUA is very similar to closures. The SUA extended for pairs is shown in Algorithm 2.4 (the cases for `cdr` and `set-cdr!` are left out because of their obvious symmetry with `car` and `set-car!`). $\mathcal{A}_{\text{cons}}$ is a table similar to $\mathcal{A}_{\text{clo}}$ but for pair approximations. A pair approximation is a tuple of two approximation sets (one for each field of the pair) and is created with the function $\widehat{\texttt{cons}}$. $\widehat{\texttt{car}}(a)$ and $\widehat{\texttt{cdr}}(a)$ respectively return the approximation set associated with the `car` and `cdr` field of pair approximation a to which is added the special approximation *obj* (as explained in section 4.5, *obj* denotes the generic Scheme object type and is needed to prevent unboxed pairs). Approximation sets are now subsets of $\mathcal{A}^3 = \{\top, \texttt{fixnum}, \texttt{flonum}, \texttt{clo}_1, \ldots, \texttt{clo}_k, \texttt{cons}_1, \ldots, \texttt{cons}_c, obj\}$, where c is the number of calls to cons in the program.

The main change is in the $\mathcal{A}_{\text{spread-}\top}$ function. Even when closures are exported, the values they hold cannot be changed because closures are immutable data storage. Because pairs are mutable they may be altered when exported (i.e. the fields of the pair can be changed using the `set-car!` and `set-cdr!` functions). The new $\mathcal{A}_{\text{spread-}\top}$ function handles this. When pairs are exported, $\top$ is added to the approximation set of each field. Note that a pair is spread at most once per iteration by $\mathcal{A}_{\text{spread-}\top}$. This is necessary to handle cyclic approximations.

```
Sua³_ast( atree )=
   case atree
        ⋮
     ⟦ (cons_i a d) ⟧:
        A_cons( i ) ← cons^( Sua³_ast( a ), Sua³_ast( d ) ),
        {cons_i}
     ⟦ (car p) ⟧:
               ⋃                 car^( A_cons( i ) )
        cons_i∈Sua³_ast( p )
     ⟦ (set-car! p x) ⟧:
        let x′=Sua³_ast( x )
          ∀cons_i∈Sua³_ast( p )
            A_cons( i ) ← cons^( car^( A_cons( i ) )⋃x′,
                                 cdr^( A_cons( i ) ) ),
          ∅
   end

A_spread-⊤( a )=
   ∀cons_i∈a
     if cons_i not already spread in this iteration
       then let a′=car^( A_cons( i ) ),
              let d′=cdr^( A_cons( i ) )
                A_spread-⊤( a′ ),
                A_spread-⊤( d′ ),
                A_cons( i ) ← cons^( a′⋃{⊤}, d′⋃{⊤} ),
   ...,
   a
```

Algorithm 2.4: A higher-order analysis with modules and lists

Let's study SUA on the following Scheme program (the program is presented in Scheme rather than in Λ so that it is easier to read):

```
1:  (define lst (let ((p1 (cons₁ 1 0)))
2:                (let ((p2 (cons₂ 2 p1)))
3:                  p2)))
4:  (define (length l)
5:     (if (pair? l) (+ 1 (length (cdr l))) 0))
6:  (length lst)
```

Types used by this program are: `fixnum`, pairs (i.e. $\mathtt{cons}_1$ and $\mathtt{cons}_2$) and the special *obj* type (we omit `boolean`, needed for the `pair?` predicate, because it does not appear in a variable's approximation set). Here is the state of the tables at the end of the analysis:

$$
\begin{array}{lcl}
\mathcal{A}_{\text{var}}(\mathtt{p1}) & = & \{\mathtt{cons}_1\} \\
\mathcal{A}_{\text{var}}(\mathtt{p2}) & = & \{\mathtt{cons}_2\} \\
\mathcal{A}_{\text{var}}(\mathtt{lst}) & = & \{\mathtt{cons}_2\} \\
\mathcal{A}_{\text{var}}(\mathtt{l}) & = & \{\mathtt{fixnum}, \mathtt{cons}_1, \mathtt{cons}_2, obj\} \\
\mathcal{A}_{\text{cons}}(1) & = & \widehat{\mathtt{cons}}(\{\mathtt{fixnum}\}, \{\mathtt{fixnum}\}) \\
\mathcal{A}_{\text{cons}}(2) & = & \widehat{\mathtt{cons}}(\{\mathtt{fixnum}\}, \{\mathtt{cons}_1\})
\end{array}
$$

The invocation of `length` at line *6* has added $\mathtt{cons}_2$ to `l`'s approximation set. Because of the call to `cdr`, the recursive call at line *5* has added $\widehat{\mathtt{cdr}}(\mathcal{A}_{\text{cons}}(2))$, that is $\{\mathtt{cons}_1, obj\}$, in one iteration and $\{\mathtt{fixnum}, obj\}$ in the next iteration.

2.6 The higher-order SUA with modules, lists and vectors

We conclude this section by adding vectors.

$$
\begin{array}{lcl}
\Lambda & ::= & \ldots \\
 & | & (\mathtt{make\text{-}vect}\ \Lambda\ \Lambda) \\
 & | & (\mathtt{vref}\ \Lambda\ \Lambda) \\
 & | & (\mathtt{vset!}\ \Lambda\ \Lambda\ \Lambda)
\end{array}
$$

```
Sua⁴_ast( atree )=
   case atree
        ⋮
     ⟦ (make-vect_i len filler) ⟧:
       Sua⁴_ast( len ),
       A_vect( i ) ← make-vect^( Sua⁴_ast( filler ) ),
       {vect_i}
     ⟦ (vref v o) ⟧:
       Sua⁴_ast( o ),
             ⋃                    vref^( A_vect( i ) )
       vect_i∈Sua⁴_ast( v )
     ⟦ (vset! v o x) ⟧:
       Sua⁴_ast( o ),
       let x'=Sua⁴_ast( x )
         ∀vect_i∈Sua⁴_ast( v )
           A_vect( i ) ←
                make-vect^( vref^( A_vect( i ) )⋃x' ),
         ∅
   end

A_spread-⊤( a )=
   ∀vect_i∈a
     if vect_i not already spread in this iteration
       then let r'=vref^( A_vect( i ) )
              A_spread-⊤( r' ),
              A_vect( i ) ← make-vect^( r'⋃{⊤} ),
   ...,
   a
```

Algorithm 2.5: A higher-order analysis with modules, lists and vectors

Vectors differ from closures and lists in that they are mutable and because it is not possible, *a priori*, to know, at compile time, which part of a vector is addressed when using vector accessors. SUA computes information about types and data storage but it does not discover the exact value of a fixnum. So for vectors the SUA merges all possible values contained in a vector into a single approximation set (e.g. if a vector is composed of a character and a fixnum, SUA will indicate that each entry is a "character or a fixnum"). Algorithm 2.5 presents the modification to our previous analysis to support vectors.

$\mathcal{A}_{\text{vect}}$ is a table similar to $\mathcal{A}_{\text{cons}}$ but for vector approximations. Vector approximations are created by the function $\widehat{\mathtt{make\text{-}vect}}$ and $\widehat{\mathtt{vref}}(a)$ return the approximation set associated with the vector approximation a. Approximation sets are now subsets of $\mathcal{A}^4 = \{\top, \mathtt{fixnum}, \mathtt{flonum}, \mathtt{clo}_1, \ldots, \mathtt{clo}_k, \mathtt{cons}_1, \ldots, \mathtt{cons}_c, \mathtt{vect}_1, \ldots, \mathtt{vect}_v, obj\}$, where v is the number of calls to `make-vector` in the program.

Just like for pairs, vector exportations have to be handled carefully. This kind of object is mutable so when exported vectors can hold any value.

2.7 Related work

The SUA is an extension of Shivers' 0cfa (0th order control flow analysis) [29, 28]. We have generalized his analysis to different data storage. Shivers' analysis only handles closures, our analysis also handles lists and vectors.

In a previous paper [26] we have presented an algorithm which is close to the present $\mathcal{S}ua^1$. The goal of that work was to study the impact of control flow analysis on function compilation. The analyses presented here are more general because closures are only considered as *one* special data storage. Efficient closure compilation is not the focus here. We study the problems of unboxed representation and stack allocation.

Ayers has studied similar improvements to Shivers' 0cfa. In his PhD thesis, he presents extensions for lists, vectors, etc. Our work has been realized concurrently with his [24, 3]. The large difference between our analyses comes from the formalism. Ayers uses Galois connection while we chose a more algorithmic approach.

Jagannathan and Wright describe in [15] a control-flow analysis and an application which removes type checks. Their analysis gives more precise type information than ours because it does not merge types for polymorphic programs. More precise type information is valuable to remove type checks but is not more valuable for a *mixed representation.* As explained in section 4, we use unboxed representation for monomorphic program parts which are efficiently detected by our analysis.

2.8 Extensions

Our input language Λ, is still much simpler than a full programming language such as Scheme or ML. Some important constructions are missing. We present here how to add them to SUA.

- Variable arity functions: this construction can be added to SUA by splitting functional application in two separate cases. Each time a function is applied (in a direct call or in a `closure-call` construction), the analysis handles the last parameter of variable arity functions specially. In Scheme, this parameter is bound to the list of the optional actual parameters. In SUA, this means that the approximation set of the last formal parameter is the approximation set of the list of the optional actual parameters.

- The Scheme special form `apply`: this construction is an alternative way to apply functions. Rather than apply a function to its n actual parameters, it is applied to a list of length n which holds the actual values. Since our analysis is able to distinguish individual elements of a list, the `apply` form can be efficiently handled: each formal parameter is assigned the corresponding approximation set from the list's approximation set.

- `call/cc`: the analysis does not treat `call/cc` specially. This library function takes closures as arguments. These closures therefore escape because `call/cc` is managed as any imported function. `call/cc`'s result is simply approximated by the set $\{\top\}$.

- Multiple values can be easily added by the addition of a treatment similar to the one for vectors.

- Scheme and ML global variables: global variables can be managed using a global environment. A subtle problem with global variables can arise when the source language allows references to global variables before their declaration. For example, the Bigloo Scheme compiler considers this program as legal:

  ```
  (module foo (static x))

  (define (foo) (print x))
  (foo)
  (define x 8)
  ```

  Before its declaration `x` holds a special value (*uninitialized*, which stands for the lack of initialization) which has to be stored in `x`'s approximation set. This implies that no optimization can be applied to `x` because it holds at least two types: the type of the *uninitialized* value and the first value used on the declaration site.

  In order to give a unique type to global variables, before the SUA analysis, we perform a simple conservative analysis to determine the set of variables which are always defined (then initialized) before being referenced. This analysis is straightforward, because it consists of a simple abstract tree traversal. These variables do not hold the special *uninitialized* value in their approximation set.

3 Stack allocation

The storage allocation optimizations discussed here assume an area of memory managed by a garbage collector and an area of memory managed as a stack. The stack is scanned by the collector to find the root pointers. Activation records are allocated on the stack when entering a procedure and they are removed from the stack upon procedure exit. Within a procedure activation the allocation of additional storage from the stack is permitted; this storage is freed when the procedure exits.

We present a conservative optimization based on SUA which automatically replaces heap allocations by stack allocations when it is legal to do so. This optimization is valuable if stack allocations and deallocations are fast with respect to heap allocations. For the sake of simplicity we add a `let` form to Λ:

$$\Lambda \ ::= \ \ldots \ \mid \ (\texttt{let}\ ((v\ \Lambda))\ \Lambda)$$

This form has no impact on the SUA algorithm (it can be seen as a macro over function application). For our stack allocation optimization, we assume that in Λ source programs all allocations (the result of `make-closure`, `cons` and `make-vect`) are bound to local variables using `let` forms. Consequently, each allocation has a unique name.

Here are three examples that present interesting situations for our optimization.

```
(define (foo)
   (let ((x (cons1 1 2)))
      (car (id x))))

(define (id z) z)
```

In this first example, the pair bound to `x` can be stack allocated since it is never used outside of `x`'s `let` extent.

```
1:  (define (bar)
2:     (let ((x (cons1 1 2)))
3:        (let ((y (cons2 3 x)))
4:           (cdr y))))
```

In this second example, the pair allocated at line *2* is live when `bar` exits (since it is the result of `bar`) while the one at line *3* is dead outside `x`'s scope. Only the second pair can be stack allocated.

```
(define (hux)
   (let ((p0 (cons1 1 2)))
      (let ((p1 (cons2 3 4)))
         (let ((p2 (gee p0 p1)))
            p0))))

(define (gee a b) (set-cdr! a b))
```

Finally, in this example, no pairs can be stack allocated because they are all live when `hux` returns.

3.1 When is it legal to stack allocate?

We present in this section the condition an allocation must satisfy to be done on the stack rather than in the heap. For now we are not concerned with preserving the tail-recursive property of the program (at the end of this section we discuss modifications of our optimization to make it suitable for languages like Scheme which have to implement tail-recursive calls without consuming stack space).

An allocation can be done on the stack if the data storage allocated is not live at the end of the procedure that allocates it. Data storage is live at the end of a function if it appears (directly or indirectly) in the result of the function that allocated it or if it appears (directly or indirectly) in a global variable. Compile time computation of the liveness property requires information about data storage which is provided by SUA. This information is: the set of allocations a variable may be bound to, the set of allocations possibly contained in an allocation, and the set of allocations possibly returned by a function.

3.2 Stack allocation decision algorithm

Each allocation is marked with a stamp. The "current stamp" is incremented each time a `let` form is encountered. When a function definition for f is reached, the current stamp is saved in h and then f's body is processed. Each allocation in the approximation set of f's result that is stamped with a more recent value than h escapes from f and so cannot be stack allocated. In addition all allocations which are accessible from a global variable cannot be stack allocated.

The first part of the algorithm (algorithm 3.1) dispatches between two function types: exported functions and static functions. These two kinds of function differ. For the first one, no returned (or pointed by) value can be stack allocated (since the function is exported the result value usage is not known to the compiler). For the second one, only data storage allocated by the function cannot be stack allocated.

```
*H*: 0    (0 stands for an initial stamp value)

Stack_prog( Π )=
   ∀f∈Π
     if f∈Π↓export
       then Stack_export( f )
       else Stack_static( f )

Stack_static( f )=
   let h=*H*
     Stack( f↓body ),
     spread_unstackable( A_out( f ), h, *H* )

Stack_export( f )=
   Stack( f↓body ),
   spread_unstackable( A_out( f ), -1, -1 )

Stack( atree )=
   case atree
         ⋮
     ⟦ (closure-call e a1 ... an) ⟧:
       Stack( e ),
       ∀i∈[1..n]
         Stack( ai )
     ⟦ (set! var val) ⟧:
       Stack( val )
     ⟦ (labels ((f1 ...) ... (fn ...)) atree) ⟧:
       ∀i∈[1..n]
         Stack_static( fi ),
       Stack( atree )
     ⟦ (let ((var val)) atree) ⟧:
       *H* ← *H* + 1,
       Stack( val ),
       Stack( atree )
     ⟦ (f a1 ... an) ⟧:
       if f is an allocator
         then mark!( atree, *H* ),
       ∀i∈[1..n]
         Stack( ai )
   end
```

Algorithm 3.1: The "stackability" algorithm

Our stack algorithm uses a $\text{spread}_{unstackable}$ function. This function is similar to $\mathcal{A}_{\text{spread-}\top}$. It follows a data storage chain to mark as "unstackable" all allocations which are younger (marked as younger) than the value of the second argument.

The algorithm's main part is the function *Stack*. It dispatches on the abstract syntax tree. Before calling $\mathcal{S}tack_{prog}$ all allocations which have been passed as argument to $\mathcal{A}_{\text{spread-}\top}$ have to be marked as *unstackable*. These allocations escape from the current module. The compiler is not able to discover the exact usage of these allocations and thus, it cannot make any assumption about their lifetime. Once the algorithm has completed, allocations which have not been marked as *unstackable* can be stack allocated (we will introduce new constraints to safely allocate data storage in stack in the next section).

```
spread_unstackable( a, min, max )=
   if a not yet processed for the values min and max
     then case a
         pair_i:
           if mark( a ) > min and mark( a ) ≤ max
             then mark-unstackable!( a ),
           ∀a'∈car^( A_pair( i ) )
             spread_unstackable( a', min, max ),
           ∀a'∈cdr^( A_pair( i ) )
             spread_unstackable( a', min, max )
         vect_i:
           ...
         clo_i:
           ...
       end
```

Algorithm 3.2: Spreading "unstackability"

Let's study the algorithm's behavior on our previous `bar` function example. The function allocates two pairs $\texttt{cons}_1$ and $\texttt{cons}_2$. SUA proves that `y` points to $\texttt{cons}_2$ (which points to $\texttt{cons}_1$) and `x` points to $\texttt{cons}_1$. The $\texttt{cons}_1$ pair is pointed to by the result of `bar`. So, the algorithm concludes that this pair cannot be stack allocated.

3.3 Extension for proper tail-recursion implementations and safety considerations

Some languages like Scheme require that executions of an iterative computation take constant space. Let's consider the following two functions:

```
(define (foo1 x y)              (define (foo2 x y)
   (if (= y 0)                     (if (= y 0)
       (display x)                     (display x)
       (foo1 (cons 1 2)                (foo2 (cons x x)
             (- y 1))))                      (- y 1))))
```

The two functions differ only in their recursive call. In `foo1`, only one allocated pair of the recursive call is live at a time; in `foo2`, allocated pairs are linked together and they are all live at any given point. The common intuitive idea of the tail-recursive property imposes an implementation to require only one free pair to run `foo1`. Our algorithm presented in algorithm 3.1 provides rough data storage lifetime. It is not able to distinguish that pairs allocated in `foo1` cannot be stack allocated while pairs allocated in `foo2` can be.

The problem is more general than tail-recursion. As revealed by Chase in [7], there is a general safety problem for stack allocation optimizers. Sometime, allocating an object in the stack rather than in the heap extends its lifetime. For instance, in `foo1`, a garbage collector is free to reclaim previous allocated pairs but if these pairs are stack allocated they will be all freed at the same time and required space to run this program is no longer constant. Stack allocation can convert a running program into one that fails. In his paper, Chase, presents "safety conditions for stack allocation" in order to decide the replacement of heap allocations by stack allocations in presence of loops or recursions. His method and our work are complementary.

3.4 Related work

Kranz presents in [17] the strategy used by Orbit to realize stack allocations. His method is less precise than ours. In Orbit only closures can be stack allocated and only if they are passed as an argument or applied. These conditions are very restrictive.

In [9], B. Goldberg and G. Park present a method for optimizing the allocation of closures in memory. Their method is based on what they call an *escape analysis*, an application of abstract interpretation to higher-order functional languages. Escape analysis determines, at compile time, if any arguments to a function have a greater lifetime than the function call itself. The language studied does not contain side effects and the only data storage used are closures and lists. List management is very rough because their analysis is not able to distinguish the elements of a list. Separate compilation is not studied in that paper.

Ruggieri and Murtagh present in [23] a data storage allocation framework called *sub-heap allocation*. This framework consists of partitioning the heap into sub-heaps, one associated with each active procedure. The contents of the sub-heap associated with a procedure is exactly the objects whose lifetime are guaranteed to be contained by the lifetime of the procedure but not by any younger procedure. The paper presents an algorithm to compute lifetime analysis in order to divide the heap with an input source language which contains no higher-order functions nor side effects.

Ayers also presents sub-heap optimization in [3]. Our lifetime analysis is similar to his but we do not use it for the same goal. We decided to stack allocate rather than sub-heap allocate for two reasons:

- Safety considerations presented by Chase [7] are very difficult to satisfy with the sub-heap allocation framework because it tends to enlarge object lifetime. Optimized objects are not freed when leaving the function they have been created but when leaving the function which is the upper bound of their lifetime.

- Sub-heap allocation is difficult to implement efficiently. This framework needs allocated memory to hold several objects which share a lifetime upper bound. If all objects are freed at the same time, they are allocated at different moments. This has two negative incidences:

  - Sub-heap size is difficult to estimate. Sub-heaps will probably need linking machinery to be extended which slows down the allocation process.
  - Sub-heaps must be allocated empty (i.e. a sub-heap cannot be filled up at the moment of its creation). Included in a runtime with automatic memory management, uninitialized blocks of memory are annoying.

Tofte and Talpin present in [31] a way of implementing λ calculus based languages using regions for memory management. At runtime the store consists of a stack of regions. All values are put into regions with the intended goal to avoid garbage collection in the runtime system. The allocation of new regions and the bindings of values to regions rely on a typing system and so this technique cannot be applied to dynamically typed languages.

In [4], Banerjee and Schmidt present a static criterion to detect stackability of environments for a call-by-value λ-calculus. The presented analysis does not include higher-order nor imperative features. Thus their approach and ours are hard to compare.

Other approaches to stack allocation have been proposed by Hudak in [14].

4 Data representation

Approximations computed by our SUA can be used to remove some runtime type checks. A type check can be removed when SUA proves that all the values possibly contained by the argument of the test is (or is not) of the tested type. This idea has been presented by Shivers [29, chapter 9] in his *type-recovery*. The goal is to speed up program execution of latently typed languages. In the same way, Henglein in [13] and Ayers in [3, chapter 6] have presented frameworks to remove useless tagging/untagging operations. Heinglein uses type inference while Ayers uses an extended control flow analysis close to our SUA. The intended goal is more than compile-time type check reductions. Appel claims "the use of tag bits leads to inefficiency" [1], Steenkiste and Henessy evaluate, in [30], at 25% the cost of type checking and tagging operations for "classical" Lisp applications. We think this time figure is an upper bound of the real cost. Classical data flow optimization (such as *copy propagation*) removes most type checks and, for smart runtime design, tagging and untagging operations could require only a logical mask in the most frequent cases. On modern computers, applying a mask costs one cycle and these operations are much cheaper than memory fetches. They have a very small impact on global performance (for more details see [25]).

We think a much more important source of inefficiency for language like Scheme or ML come from uniform data representation. Tag handling is cheap but uniform representation is very expensive.

4.1 Uniform representation

Using uniform data representation, all objects have exactly the same size (usually one word, i.e. pointer size). Objects that do not fit naturally in one word, such as long floating-point numbers, have to be boxed (allocated in the heap and handled through a pointer). This scheme makes it possible to assume a default size, common to all objects, and default calling conventions, common to all functions.

Polymorphism leads to the use of the uniform representation because an object can belong to several different types at the same time and the actual type cannot be known at compile-time. Polymorphic functions (e.g. the identity function) can be applied to arguments of any type. Therefore, when compiling these functions, the compiler knows neither the size of the argument nor the correct calling convention.

4.2 The uniform representation is inefficient

We claim uniform representation results in a serious loss of efficiency and we present two arguments for this assertion.

Objects that do not fit in one word have to be boxed. Long floating-point numbers dramatically illustrate this. For floating-point intensive programs, boxing numbers can slow down applications by a large factor. This problem has such an important impact that many ML and Lisp implementations use *ad hoc* methods to reduce the creation of number handles (descriptions of these methods for modern implementations can be found in [20, 12]). Mainly, they consist of local optimizations to avoid boxing numbers for intermediate results.

Another negative impact of floating-point boxing is register allocation. When flonums are allocated in memory, every floating-point operation, requires a memory fetch for each operant. These operations are expensive and much less efficient than a solution where the numbers are held in registers.

Tagging optimization is not boxing optimization in the sense that it removes tagging/untagging operations but such optimized programs must still satisfy the polymorphism constraint. They are still obliged to box numbers (even if no tag is written on the handle or stored in the allocated memory).

Small objects (i.e. characters) are also inefficiently managed by uniform representations as memory is wasted.

4.3 Mixed representation

Mixed representation is a representation where all objects are not required to be of the same size. It mixes boxed objects and unboxed objects. Initial efforts using mixed representation are Leroy [18, 19] and Peyton-Jones and Launchbury [21], and more recently Shao and Appel [27]. Their works are complementary because Peyton-Jones and Launchbury introduce a (non-strict) language where boxing operations are explicit and introduce several source-to-source optimizations for this language while Leroy and Shao and Appel present a translation of ML to this mixed language. In this paper, we will focus on the translation of source languages (like ML or Scheme) to mixed languages, comparing our work to Leroy's.

Leroy's translation only uses type information. It mixes specialized representations when the static types are monomorphic and uniform representation when the static types are polymorphic. Coercions between the two representation styles are performed when a polymorphic object is used with a more specific type. As Leroy presents, "in the case of a polymorphic function, for instance, coercions take place just before the function call and just after the function result". This solution is very elegant because the translation's quality does not suffer from separate compilation (type informations are propagated across ML modules) but is has some disadvantages. Every time a polymorphic function is used, objects have to be boxed. Some data accessors *are* polymorphic. For instance, vector accessors are polymorphic where the same function is used to access a vector of fixnums or a vector of flonums. Vectors are a too important kind of data storage to be out of the scope of this transformation. In other words, Leroy's translation requires some *ad hoc* treatments for some special functions. The second restriction to Leroy's work is about the input languages of its translation. Programs have to be statically type checked, and as a consequence Leroy's optimization is not applicable to the Lisp family.

4.3.1 Untagging vs. unboxing

Optimizing tagging/untagging operations as in [13, 3] does not require the same analysis as mixed representation. Constraints about untagged representations are weaker than for unboxed representations. An object can be untagged as soon as its type is never required at runtime, without any polymorphism consideration. For instance, with untagged representation, a vector can hold in its first slot an untagged floating point number and in its second slot a tagged one. This is not possible with unboxed representation because these two kind of objects do not have the same size. Untagging optimization consists of type analysis (possibly using type system as Henglein, control flow analysis as Ayers or any other data flow analysis) while unboxing optimization requires type *and* polymorphism analyses.

4.3.2 Other polymorphism implementation improvements

In [10], Goubault presents an optimization for latently typed languages. Like our efforts, its source language is not required to be statically type checked. Goubault uses data flow equations to choose unboxed representation. However it is difficult to compare his work with our because the method employed is very different than ours and the paper contains neither measurements nor examples.

Harper and Morrisett present in [11] a new scheme to implement polymorphism. The key idea is to separate values and types for polymorphic functions and to defer the selection of the code to execute until types are known (e.g. at runtime). Unfortunatelly this work addresses statically typed languages and cannot be applied to languages such as Scheme.

4.4 SUA and mixed representation

In this section, we use the SUA approximation to introduce unboxed representations. This is done in two stages.

4.4.1 Type election

A first stage after the SUA analysis is the *type election* which gives types to all variables and function results. This pass obviously uses SUA approximations. It does not perform any data flow analysis to choose better type. Let us study type election on the following Scheme program (`-fx` and `=fx` are the fixnum subtraction and equality test procedures):

```
(define (bcopy! dst src size)
   (let loop ((i (-fx size 1)))
      (if (=fx i -1)
          0
          (let ((c (string-ref src i)))
             (string-set! dst i c)
             (loop (-fx i 1))))))
(define (copy-string src)
   (let* ((len (string-length src))
          (new (make-string len)))
      (bcopy! new src len)
      new))

(copy-string "foo")
```

SUA shows that variables `new`, `dst` and `src` are strings, variables `len` and `i` are only fixnums and variable `c` is a character. SUA is able to compute these type approximations because the types of the library functions (`string-length`, `make-string`, `string-ref` and `string-set!`) are known by the compiler. Each one of the variables contains one type in its approximation set. Hence type approximations also are the results of type election. If a variable contains more than one type in its approximation set, then it is given the special *obj* type.

SUA merges all possible values in single sets. Hence, if a variable contains one unique type in its approximation, this variable can only take place in a monomorphic program. For instance, if SUA shows that the formal parameter of the identity function can only be a fixnum it means that this function has only been given fixnums as argument, no matter the polymorphism of this function. This is the main advantage of our method compared to Leroy's one. SUA isolates monomorphic parts of polymorphic programs, thus our method allows us to use unboxed representation when Leroy's fails.

4.4.2 Type conversion

The second stage is called *type conversion.* It introduces conversions between boxed representation and unboxed representation in the abstract syntax tree. Objects can be boxed or unboxed. One type exists for these two states. The boxed state is denoted by the *obj* type. Conversion introduction is straightforward because the abstract syntax tree is fully annotated. Here is an example that illustrates type conversion:

```
(define (id x) x)
(define (foo y)
   (id (+fl 1.0 (id y))))
```

Let's assume that `foo` is exported, hence, `y` and `foo`'s result have type *obj*. Identity `id` is invoked with a flonum (result of `+fl` invocation) and an *obj*, so formal `x` and the function result are typed as *obj*. Conversions are then inserted.

```
(define (id x) x)
(define (foo y)
   (id (float-box (+fl 1.0 (float-unbox (id y))))))
```

$$
\begin{array}{l}
\mathcal{C}(\ \Pi\ )= \\
\quad \forall f \in \Pi \\
\qquad \mathcal{C}_{\mathbf{ast}}(\ f\!\downarrow_{body},\ \mathcal{T}(\ f\ )\ ) \\
\\
\mathcal{C}_{\mathbf{ast}}(\ atree,\ \tau\ )= \\
\quad \mathbf{case}\ atree \\
\qquad [\![\ k\ ]\!]: \\
\qquad\quad convert!(\ k,\ \mathcal{T}(\ k\ ),\ \tau\ ) \\
\qquad [\![\ v\ ]\!]: \\
\qquad\quad convert!(\ v,\ \mathcal{T}(\ v\ ),\ \tau\ ) \\
\qquad [\![\ (\mathtt{if}\ atree\ atree_t\ atree_f)\ ]\!]: \\
\qquad\quad \mathbf{let}\ atree{=}\mathcal{C}_{\mathbf{ast}}(\ atree,\ boolean\ ), \\
\qquad\quad \mathbf{let}\ atree_t{=}\mathcal{C}_{\mathbf{ast}}(\ atree_t,\ \tau\ ), \\
\qquad\quad \mathbf{let}\ atree_f{=}\mathcal{C}_{\mathbf{ast}}(\ atree_f,\ \tau\ ) \\
\qquad\qquad (\mathtt{if}\ atree\ atree_t\ atree_f) \\
\qquad\qquad \vdots \\
\qquad [\![\ (f\ a_1\ \ldots)\ ]\!]: \\
\qquad\quad \mathbf{let}\ atree{=}[\![\ (f\ \mathcal{C}_{\mathbf{ast}}(\ a_1,\ \mathcal{T}(\ f\!\downarrow_{arg_1}\ )\ )\ \ldots)\ ]\!] \\
\qquad\qquad convert!(\ atree,\ \mathcal{T}(\ f\ ),\ \tau\ ) \\
\quad \mathbf{end}
\end{array}
$$

Algorithm 4.1: Type conversion introduction

Algorithm 4.1 presents a fragment of the complete type conversion algorithm. Function $\mathcal{T}$ is a function that returns the type of an expression. Function *convert*! takes three arguments: an abstract syntax tree, a *from* type, and a *to* type. It introduces boxing operations required by the translation. Function *convert*! is source language dependent. For latently typed languages with boxing operations, it introduces runtime type checks to ensure the soundness of the translation. For instance, when introducing conversions from *obj* to character, the Scheme *convert*! version also introduces a type check using the `char?` predicate. No type checks are introduced for statically type checked languages.

4.5 Unboxed data storage

In this section we discuss the unboxed representation of the three Λ data types presented in section 2.

4.5.1 Unboxed pairs

Pairs have a special status: they are widely used (many library functions exist to manage them) and in Scheme they are heterogeneous data structures (i.e. elements of a list can be of different types). For these reasons, we have decided to make pairs hold boxed values. If pairs were allowed to hold unboxed values, they would not have a fixed size and library functions which have to be applicable to all pairs would be inefficient and difficult to write. To prevent pairs from holding unboxed objects, we simply force $\widehat{\mathtt{car}}$ and $\widehat{\mathtt{cdr}}$ to have the *obj* type in their approximation sets.

4.5.2 Unboxed vectors

Vectors are widely used in all programming languages. We think vectors are not used in the same way as pairs. Even if Scheme vectors are heterogeneous (each vector slot can be of a different type), we think they are mostly used as homogeneous data storage and thus have allowed unboxed values in vectors. Mixed vectors (vector holding boxed *and* unboxed objects) are forbidden because this would prevent the efficient implementation of vector indexing functions. If a vector only contains elements of a given type, it will be transformed into an unboxed vector. If a vector contains at least two elements of different types, it will be a boxed vector. As shown in section 2.6, SUA merges all possible values held by a vector in a single approximation so it is easy to check if all its elements are of the same type.

4.5.3 Unboxed procedures

Because closure creation and access to the free variables are handled by the compiler, each closure creation can be treated independently. Closures can hold boxed *and* unboxed values (because SUA distinguishes approximated values in the closures free variables) but unboxed closures are not allowed.

5 Experimental results

SUA has been implemented in the new release of the Bigloo Scheme/ML compiler. Both stack optimization and unboxed representation are implemented. Hence, we have been able to make experimental analyses and performance measurements.

Experimental results obtained by running some Scheme benchmarks on a DEC Alpha (DEC 3000/300 (150 MHz), running OSF/1 v3.0, with 160 MBytes of memory) are given in Figure 1. The times given are user+system time, including garbage collection time. Bigloo1.7 is the current distributed version of the system, Bigloo1.8 is the new version including the unboxed representation and the stack allocation optimization. Both versions of Bigloo use Boehm's garbage collector release 4.7 [6]. This collector allows ambiguous pointers and uses a traditional mark & sweep algorithm. Gsc is the Gambit-C compiler version 2.3a, S2c is

Bartlett's Scheme-to-C compiler version 15mar93jfb [5] and Gcc is the popular Gnu C compiler version 2.6.3, used at optimized level 2. Here is a short description of the Scheme test programs we used:

Nucleic (3496 lines) :	Floating-point arithmetic.
Fft (127 lines) :	Floating-point arithmetic, loops.
Bcopy (43 lines) :	Strings, chars, fixnum, loops.
Ttak (20 lines) :	Function calls (with tuples).
Beval (548 lines) :	Functionals, conditional.
Boyer (606 lines) :	Term processing, functionals.
Maze (800 lines) :	Arrays, fixnum, iterations.
Slatex (2821 lines) :	IO, strings, lists.
Mbrot (46 lines) :	Floating-point arithmetic, loops.

	Compiler				
Test	Bigloo1.8	Bigloo1.7	Gsc	S2c	Gcc
Nucleic	9.0 s	47.2 s	10.3 s	∘	*4.1* s
Fft	1.3 s	22.7 s	5.6 s	39.7 s	*1.1* s
Bcopy	*9.9* s	12.2 s	14.5 s	12.2 s	*9.9* s
Ttak	2.9 s	12.0 s	4.8 s	57.2 s	*1.9* s
Beval	6.7 s	*6.5* s	6.9 s	14.8 s	•
Boyer	*3.4* s	*3.4* s	3.8 s	4.1 s	•
Maze	*6.2* s	7.7 s	8.0 s	18.7 s	•
Slatex	*7.8* s	*7.8* s	23.9 s	22.9 s	•
Mbrot	*1.0* s	20.1 s	9.2 s	35.6 s	*1.0* s

Figure 1: Runtime statistics on DEC Alpha

Significant speed up occurs with the numerical benchmarks **Nucleic**, **Fft** and **Mbrot**. On **Mbrot** and **Fft** (which is a translation of a C routine from [22], not the Lisp version from the Gabriel suite) Bigloo's performance is very close to Gcc. **Fft** and **Mbrot** are efficiently compiled by Bigloo; no floating point values get boxed. **Fft** makes use of vectors of floats which are optimized as described in Section 4.5.2.

Nucleic computes 7 million floating point values. Our unboxing optimization allows Bigloo to only allocate 13608 flonums in the heap. The difference in performance between the Bigloo and Gcc versions is mainly due to the use of structures to hold 3 D points. In the C version, all the structures are explicitly allocated on the stack. The Scheme version does not allow our stack optimization to be frequently applied. Hence, profiling the Bigloo executable shows that even though many heap allocations are avoided, 30% of the execution time is still spent in the garbage collector.

Ttak is written in a ML style using tuples to pass arguments. Our stack optimization avoids heap allocation entirely and the speedup is thus important. The impact of stack allocation depends on the program tested. The most important speedup is observed for **Ttak** (75% of memory is allocated on the stack, which leads to a speed up factor of 5).

Figure 2 presents dynamic statistics on the amount of memory allocated by the programs. For each program tested and for each compiler, the amount of heap memory allocated is given. The total amount of memory allocated on the stack for Bigloo1.8 is also given.

In accordance with the execution time speedup, the main reduction of heap allocation is observed on numerical programs (**Nucleic**, **Fft** and **Mbrot**). Except for the **Ttak** program, stack allocations are not widely applied. This poor result may come from the style of our programs. The natural Scheme style is to write "allocating" functions which return fresh allocations as in:

	Memory allocated		
Test	Bigloo1.7	Bigloo1.8 (heap)	Bigloo1.8 (stack)
Nucleic	747589 k	127523 k	5655 k
Fft	425432 k	174 k	0 k
Bcopy	140 k	98 k	0 k
Ttak	301148 k	0 k	301148 k
Beval	32947 k	32903 k	0 k
Boyer	14369 k	14318 k	0 k
Maze	17563 k	15902 k	1 k
Slatex	67607 k	67431 k	2 k
Mbrot	265589 k	27 k	0 k

Figure 2: Allocation statistics on DEC Alpha

```
(define (foo x) (car (gee x)))
(define (gee x) (cons x x))
```

The pair built in `gee` cannot be stack allocated by our method.

The worst case complexity of the SUA algorithm is high. The maximum number of iterations to reach the fix point is the product of the maximum size of approximation sets and the maximum number of approximation sets (i.e. n^2 for a program of size n). Each iteration has a $\mathcal{O}(n^2)$ complexity (the call graph traversal is $\mathcal{O}(n)$ and operations performed on the tree's nodes are $\mathcal{O}(n)$). The overall complexity is thus $\mathcal{O}(n^4)$. In spite of this complexity, our analysis is relatively fast in practice. Figure 3 presents statistics on compilation time. For each program tested, we have measured the time required by SUA, the compilation time until the C code production and the global compilation time including the C compilation. In the worst case (**Slatex**), the time required by SUA is only 20% of the overall compilation.

	Compilation time				
Test	SUA	Bigloo1.8	δ	Bigloo1.8+cc	δ
Nucleic	7.1 s	22.9 s	*0.31*	538 s	*0.01*
Fft	0.1 s	0.8 s	*0.13*	2.9 s	*0.03*
Bcopy	0.1 s	0.5 s	*0.20*	1.5 s	*0.07*
Ttak	0.1 s	0.7 s	*0.14*	2.6 s	*0.04*
Beval	1.8 s	3.9 s	*0.46*	17.8 s	*0.11*
Boyer	0.1 s	0.9 s	*0.11*	3.5 s	*0.03*
Maze	0.4 s	2.0 s	*0.20*	6.9 s	*0.06*
Slatex	15.7 s	22.6 s	*0.69*	79.7 s	*0.20*
Mbrot	0.0 s	0.5 s	*0*	1.8 s	*0*

Figure 3: Compilation statistics on DEC Alpha

6 Conclusion

We have presented in this paper a new static analysis method called *Storage Use Analysis* (SUA) which extends Shivers' 0cfa to modules and general data storage. This analysis allows two important optimizations: unboxed representation and stack allocation. None of these optimizations require type information, so both statically typed languages like ML and latently typed languages like Scheme can use them. Experimental results demonstrate important speedups for numerical applications where a speedup factor of 20 has been measured for some programs.

Acknowledgments

Many thanks to Pierre Weis and Alain Deutsch for early discussions and to Xavier Leroy and Joel F. Bartlett for their helpful feedbacks on this work.

References

[1] A. Appel. Runtime Tags Aren't Necessary. Technical Report CS-TR-142-88, Princeton University, 1989.

[2] A. Appel. *Compiling with continuations.* Cambridge University Press, 1992.

[3] A. Ayers. *Abstract Analysis and Optimization of Scheme.* PhD thesis, Massachusetts Institute of Technology, September 1993.

[4] A. Banerjee and D. Schmidt. Stackability in the Simple-Typed Call-By-Value Lambda Calculus. In *1fst Static Analysis Symposium*, pages 131–146, Namur, Belgium, September 1994.

[5] J.F. Bartlett. Scheme->C a Portable Scheme-to-C Compiler. Research Report 89 1, DEC Western Research Laboratory, Palo Alto, CA, January 1989.

[6] H.J. Boehm. Space efficient conservative garbage collection. In *Conference on Programming Language Design and Implementation*, number 28, 6 in SIGPLAN NOTICES, pages 197–206, 1991.

[7] D. Chase. Safety considerations for storage allocation optimizations. In *Conference on Programming Language Design and Implementation*, Atlanta, Georgia, USA, June 1988.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, USA, January 1977.

[9] B. Goldberg and G. Park. Higher order escape analysis: Optimizing stack allocation in functional program implementations. In *European Symposium on Programming*, number 432 in Lecture Notes on Computer Science, pages 152–160, May 1990.

[10] J. Goubault. Generalized Boxings, Congruences and Partial Inlining. In *1fst Static Analysis Symposium*, pages 147–161, Namur, Belgium, September 1994.

[11] R. Harper and G. Morrisset. Compiling polymorphism using intensional type analysis. In *22 Annual* ACM *Symposium on Principles of Programming Languages*, pages 130–141, New York, NY, USA, January 1995.

[12] P. Hartel et al. Pseudoknot: a Float-Intensive Benchmark for Functional Compilers. *Journal of Functional Programming*, To appear, 1996.

[13] F. Henglein. Global Tagging Optimization by Type Inference. In *Conference on Lisp and Functional Programming*, 1992.

[14] P. Hudak. A semantic model of reference counting and its abstraction. In *Abstract Interpretation of Declarative Languages*, pages 45–62. Ellis Horwood, 1987.

[15] S. Jagannathan and A. Wright. Effective Flow Analysis for Avoiding Run-Time Checks. In *2nd Static Analysis Symposium*, Lecture Notes on Computer Science, pages 207–224, Glasgow, Scotland, September 1995.

[16] T. Johnson. *Lambda Lifting:* Transforming Programs to Recursive Equations. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 190–203, 1985.

[17] D.A. Kranz. *ORBIT: An Optimizing Compiler For Scheme.* PhD thesis, Yale university, February 1988.

[18] X. Leroy. Efficient data representation in polymorphic languages. In P. Deransart and J.Małuszyński, editors, *Int. Symp. on Programming Language Implementation and Logique Programming*, volume 456 of *Lecture Notes on Computer Science.* Springer-Verlag, 1990. Also available as INRIA research report 1264.

[19] X. Leroy. Unboxed objects and polymorphic typing. In *Symposium on Principles of Programming Languages*, pages 177–188, Albuquerque, New Mexico, January 1992.

[20] R. MacLachlan. The Python Compiler for CMU Common Lisp. In *Proceedings of the 1992* ACM *Conference on Lisp and Functional Programming*, pages 235–246, San Francisco, CA, USA, June 1992.

[21] S. Peyton Jones and J. Launchbury. Unboxed Values as First Class Citizens in a Non-Strict Functional Language. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 636–666, Cambridge, MA, USA, August 1991.

[22] W. Press, B. Flannery, S. Teukolsky, and Vetterling W. *Numerical Recipes in C.* Cambridge University Press, 1988.

[23] C. Ruggieri and T. Murtagh. Lifetime Analysis of Dynamically Allocated Object. In *Symposium on Principles of Programming Languages*, pages 285–293, 1988.

[24] M. Serrano. De l'utilisation des analyses de flot de contrôle dans la compilation des langages fonctionnels. In Pierre Lescanne, editor, *Actes des journées du GDR de Programmation*, September 1993.

[25] M. Serrano. *Vers une compilation portable* et *performante des langages fonctionnels.* Thèse de doctorat d'université, Université Pierre et Marie Curie (Paris VI), Paris, France, December 1994.

[26] M. Serrano. Control Flow Analysis: a Functional Languages Compilation Paradigm. In *10th Symposium on Applied Computing*, Nashville, Tennessee, USA, February 1995.

[27] Z. Shao and A. Appel. A Type-Based Compiler for Standard ML. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.

[28] O. Shivers. Control flow analysis in scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988.

[29] O. Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* CMU-CS-91-145, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 1991.

[30] P.A. Steenkiste and J. Hennessy. Tags and Type Checking in LISP: Hardware and Software Approaches. In *Architectural support for programming languages and operating systems*, pages 50–59, Palo Alto. CA US, 1987.

[31] M. Tofte and J-P. Talpin. Implementation of the Typed Call-by-Value λ-calculus using a Stack of Regions. In *21st* ACM SIGPLAN-SIGACT *Symposium on Principles of Programming Languages*, pages 188–201, Portland, Oregon, USA, January 1994.