Memory Consumption Analysis for a Functional and Imperative Language

Jérémie Salvucci, Emmanuel Chailloux

Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 330 (2016) 27–46

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Memory Consumption Analysis for a Functional and Imperative Language

Jérémie Salvucci[1] and Emmanuel Chailloux[2]

Sorbonne Universités,
UPMC Univ. Paris 06,
UMR 7606, LIP6,
4 Place Jussieu, F-75005 Paris, France

Abstract

The omnipresence of resource-constrained embedded systems makes them critical components. Programmers have to provide strong guarantees about their runtime behavior to make them reliable. Among these, giving an upper bound of live memory at runtime is mandatory to prevent heap overflows from happening. The paper proposes a semi-automatic technique to infer the space complexity of ML-like programs with explicit region management. It aims at combining existing formalisms to obtain the space complexity of imperative and purely functional programs in a consistent framework.

Keywords: ML, regions, static analysis, memory analysis.

1 Introduction

Deploying software in constrained environments requires strong guarantees about its runtime behavior. In memory-constrained embedded systems, dynamic allocation is often prohibited to keep execution time analyses doable and avoid heap overflows. We introduce a programming language and a resource consumption analysis to enable dynamic allocation while providing an upper bound of live memory at compile time.

In this paper, we propose a language *à la ML* mixing purely functional and imperative features with an explicit region mechanism. To retrieve information about a program memory interactions, we rely on a static type & effect system and manual memory management through region related primitives. The type system aims at ensuring the absence of memory-related errors at compile time. To

[1] jeremie.salvucci@lip6.fr
[2] emmanuel.chailloux@lip6.fr

http://dx.doi.org/10.1016/j.entcs.2016.12.013

perform this, programmers have to manage memory manually through a restricted set of primitives describe in section 3. The effect system helps generalize terms and discriminate between purely functional and imperative styles at the function level.

The analysis relies on the correctness of the type system to consider only error-free programs with respect to memory. It combines several existing resource consumption analyses depending on the style inferred by the effect system. For instance, a function which does not allocate memory do not require analysis whereas a function which allocates memory and performs side-effects needs careful handling. On pure functions, we apply automatic amortized analysis [7] adapted to the region mechanism. On imperative functions, regions offer spatial information for side-effects, we use invariants on iteration spaces provided by the programmer as annotations. Both analyses return a symbolic expression characterising the space complexity of the analyzed function for each region involved in the computation. The composition of these symbolic expressions with a careful handling of side-effects give the program memory consumption.

To allocate memory and to reclaim memory are orthogonal operations. Allocating memory does not require information about the current state of the memory graph. Whereas, reclaiming memory requires a global view of the heap to distinguish reachable from unreachable values. In this work, we use regions to gather enough information at compile time to prevent overpessimistic upper bounds by considering regions freed by the programmer in a sound way.

The main goal of this paper is to introduce a framework to combine various memory consumption analyses depending on the programming style used at function level to provide an upper bound of live memory at compile time considering reclaimed memory. In the remainder, related works are presented in section 2. We describe the language in section 3 with its type & effect system on which we base our analysis. Then, we show how to deal with purely functional and imperative features in sections 4, 5 as described above and section 6 composes them in a consistent framework. Then, we show how it works on an example in section 7. Finally, we conclude with a discussion about current limitations and further improvements.

2 Related works

Resource consumption analysis started in the late 70s with METRIC [14] targetting the best, worst and average execution times of programs written in a pure subset of Lisp. Based on recurrence relations, it can be adapted to memory consumption analysis. Contrary to time, memory can be reclaimed. Hence, new methods have emerged from both purely functional and imperative communities to obtain upper bounds on live memory.

Sized types [9] have been applied to the core part of HUME [13], a purely functional language with an eager evaluation mechanism. It infers linear space complexities and provides an upper bound on allocated memory without requiring the user intervention.

Automatic amortized analysis [6], based on Tarjan's work [11], has been used in

several projects. Among them, RAML [7], a pure ML language and RAJA [8], a subset of the Java language. It is able to infer polynomial bounds on live memory considering some side-effects [5] in the last version without any user annotation.

Recurrence relations have also been employed in the COSTA [1] project targetting the Java platform. Relying on a powerful solver, it is able to infer polynomial, logarithmic and exponential bounds on live memory thanks to a scope-based memory management mechanism.

Abstract interpretation [3] has been applied to Safe [10]. A pure first-order language equipped with an implicit region mechanism to manage memory. This technique does not restrain to particular complexity class. Unfortunately, it does not consider side-effects or higher-order functions.

Invariants over iteration spaces have been used in the JConsume [2] project targetting the Java language. It considers side-effects through an escape mechanism. It also relies on invariants provided by external tools or the programmer to extract program space complexity. Thanks to the escape mechanism, it can provide an upper bound on live memory.

In the following, we build on previous works to obtain an upper bound on live memory at compile time. We develop a language equipped with a type & and effect system and an explicit memory mechanism based on regions. The effect system allows us to mix automatic amortized analysis and invariants on iteration spaces depending on the programming style employed at the function level. Regions offer information about the heap state at compile time. We can then track side-effects and consider reclaimed memory to prevent inacurrate bounds.

3 Language & Analysis

In modern high-level languages, memory management is often performed by a garbage collector. Mainly ruled by dynamic criteria, predicting its behavior is a difficult problem. For instance, we need to know when it will be triggered and how much memory will be reclaimed.

To circumvent this problem, we develop a statically typed language *à la ML* equipped with a specific memory management mechanism: regions [12]. A region represents a set of data whose lifespan is similar. Originally developed to bring back lexical scope to heap-allocated values, they suffered memory leaks due to the stack discipline. A work about linear regions [4] shows that adding capabilities to the system makes it more general without involving such a stack discipline. A capability can be seen as a permission to operate on a region and as a witness that data within the related region are still necessary for the rest of the computations. This is a compile-time mechanism which allows us to consider reclaimed memory during resource consumption analysis (see section 6).

3.1 Syntax

From a programmer point of view, regions can be seen as a way to manage memory through the use of four primitives: *newrgn*, *@*, *aliasrgn* and *freergn*. They respectively allow the programmer to allocate new regions, specify where a value is allocated, remove temporarily the linear constraint (see typing rules) and safely free regions previously allocated. Correctness of these operations is ensured by the type system at compile-time through the use of capabilities. A capability can be seen as a permission to do some actions. The grammar in figure 1 shows that each expression whose evaluation turns into heap-allocated value is annotated with *@* to specify the region e_ρ where it is allocated at runtime. Each operation related to region-allocated values requires the associated capability of the relevant region. The type system checks that the right capability is owned. If not, this is considered as a forbidden operation and raises a type error.

The language syntax is presented in figure 1. The rule e is annotated with the nature of the corresponding expression. Hence, e_ρ denotes an expression whose evaluation should produce a region handler, e_c, e_t and e_f denote respectively the condition, the consequence and the alternative of a conditional expression. The two kinds of assignment, cumulative and non-cumulative, will be discussed in section 6.

expressions	description
$\langle e \rangle ::= () \mid \langle b \rangle \mid \langle n \rangle$	unit, booleans, integers
$\mid \langle x \rangle$	variables
$\mid$ fun $x \rightarrow \langle e \rangle$ @ $\langle e_\rho \rangle$	functions
$\mid \langle e_0 \rangle \langle e_1 \rangle$	function application
$\mid$ if $\langle e_c \rangle$ then $\langle e_t \rangle$ else $\langle e_f \rangle$	conditional
$\mid$ let $\langle x \rangle = \langle e_a \rangle$ in $\langle e_b \rangle$	variable binding
$\mid$ let rec $\langle f \rangle \langle x \rangle = \langle e_b \rangle$ @ $\langle e_\rho \rangle$ in $\langle e \rangle$	recursive binding (function only)
$\mid (\langle e_a \rangle, \langle e_b \rangle)$ @ $\langle e_\rho \rangle$	pair construction
$\mid \Pi_i \langle e \rangle$	pair projections
$\mid$ Nil @ $\langle e_\rho \rangle$	end of list
$\mid$ Cons $\langle e_h \rangle \langle e_t \rangle$ @ $\langle e_\rho \rangle$	new list head
$\mid$ ref $\langle e \rangle$ @ $\langle e_\rho \rangle$	reference
$\mid \langle e \rangle := \langle e \rangle \mid \langle e \rangle \mathrel{+}= \langle e \rangle$	assignment
$\mid\ !\langle e \rangle$	dereference
$\mid$ newrgn ()	new region primitive
$\mid$ aliasrgn $\langle e_\rho \rangle$ in $\langle e \rangle$	sharing a region handler
$\mid$ freergn $\langle e_\rho \rangle$	free region primitive

Fig. 1. Expressions

3.2 Type & effect system

The goal of this type system is twofold: gather information about the program memory behavior to prevent bad memory management and provide a topology of

Types	Description
$\langle \tau_s \rangle ::=$ unit | bool | int | α	type of singleton, booleans, integers type variables
$\langle \tau \rangle ::= \langle \tau_s \rangle$ | $(\langle \tau_a \rangle \xrightarrow[\phi]{C_{in}C_{out}} \langle \tau_b \rangle, \rho)$ | $(\langle \tau_a \rangle \times \langle \tau_b \rangle, \rho)$ | (list $\langle \tau \rangle$, ρ) | (ref $\langle \tau \rangle$, ρ) | hnd ρ	type of closures type of pairs type of lists type of references type of region handlers
$\langle \sigma_\rho \rangle ::= \langle \tau \rangle \mid \forall \rho.\ \langle \sigma_\rho \rangle$	region type schemes
$\langle \sigma \rangle ::= \langle \sigma_\rho \rangle \mid \forall \sigma.\ \langle \sigma \rangle$	type schemes

Fig. 2. Types

the heap available at compile-time. The typing judgement has the following shape

$$C; \Gamma \vdash e : \tau; \Gamma'; C'; \phi$$

where C is a set of capabilities, Γ a typing environment and ϕ is a set of effects. It reads as follows: "given a set of capabilities C and a typing environment Γ, the expression e has type τ, returns a set of capabilities C' and performs effects ϕ at runtime".

The effect system tracks three kinds of effects: alloc, read and write (see figure 3). They are mainly used for *let generalization* and the resource consumption analysis.

effect	description
$read\{r\}$	read a value allocated in region r
$write\{r\}$	update a reference allocated in region r
$alloc\{r\}$	allocate a new value in region r

Fig. 3. Tracked effects

In the language, there are two kinds of variables. Those bound to stack values and those bound to region-allocated values. The latest is dealt with the rule for region variables, RVar. To ensure correctness, two specific rules have been added for linear and non-linear region handlers, LRHVar and RHVar. As you can see, the type (see figure 2) of this expression contains a region name r. This name is qualified with a constraint q in the environment C. This qualifier can take two different values : 1 or +. Linearity, 1, ensures that you do not share a region handler whereas + allows you to weaken the linearity constraint. This rule checks

that the access to a value in region r is sound. For region allocated values, we have different rules. We need to distinguish region handlers from other values because of the linearity constraints. The role of the primitive *instantiate* is to replace type parameters with fresh type variables. Hence,

$$\frac{hnd\ r = instantiate(\sigma) \qquad C = C', r^1}{C; \Gamma, x : \sigma \vdash x : hnd\ r; \Gamma; C; \{read\ r\}}\ (\text{LRHVar})$$

The linear restriction on the capability r allows only one use of the hypothesis $x : \sigma$.

$$\frac{hnd\ r = instantiate(\sigma) \qquad C = C', r^+}{C; \Gamma, x : \sigma \vdash x : hnd\ r; \Gamma, x : hnd\ r; C; \{read\ r\}}\ (\text{RHVar})$$

When the linear restriction is weakened, its number of uses is unrestricted.

$$\frac{(\tau, r) = instantiate(\sigma) \qquad C = C', r^q}{C; \Gamma, x : \sigma \vdash x : hnd\ r; \Gamma, x : (\tau, r); C; \{read\ r\}}\ (\text{RVar})$$

In the previous rule, q means that the linearity constraint doesn't matter.

Typing a function requires planning for future calls. For instance, if some free variables bound to region-values are captured then the relevant set of capabilities has to be presented at each call site. To perform this verification, the arrow type is augmented with C_{in} and C_{out} respectively the set of required capabilities to evaluate the function body and the new set of capabilities once evaluation terminates. The predicate *unrestricted* checks that no region handler associated with a linear capability is captured. Moreover, we need to propagate the effects performed by the evaluation of e, ϕ_e. To do this, we add a latent effect to the arrow type.

$$\frac{\begin{array}{c} C; \Gamma \vdash e_\rho : hnd\ r; \Gamma'; C'; \phi_\rho \\ C_{in}; \Gamma', x : \tau_x \vdash e : \tau; \Gamma'; C_{out}; \phi_e \\ C' = C'', r^q \qquad unrestricted(C_{in}, \Gamma') \end{array}}{C; \Gamma \vdash fun\ x \to e\ @\ e_\rho : \tau_x \xrightarrow[\phi_e]{C_{in}\ |\ C_{out}} \tau; \Gamma'; C'; \phi_\rho \cup \{alloc\ r\}}\ (\text{Fun})$$

The application rule, APP, follows immediately the function rule. At each call site, we check that the operation is sound by checking that C entails C_{in}, the relevant set of capabilities to evaluate the function body. Here $\leq$ can be seen as a subtyping relation: C_v has to allow at least operations doable with C_{in}. Hence, we have $r^1 \leq r^+$ because linearity allows you to allocate and free a region [3]. This relation extends to sets of capability.

$$\frac{\begin{array}{c} C; \Gamma \vdash e_f : \tau_x \xrightarrow[\phi_e]{C_{in}\ |\ C_{out}} \tau; \Gamma_f; C_f; \phi_f \\ C_f; \Gamma_f \vdash e_v : \tau_x; \Gamma_v; C_v; \phi_v \qquad C_v \leq C_{in} \end{array}}{C; \Gamma \vdash e_f\ e_v : \tau; \Gamma_v; C_v \setminus (C_{in} \setminus C_{out}) \cup (C_{out} \setminus C_{in}); \phi_f \cup \phi_v \cup \phi_e}\ (\text{App})$$

[3] This relation can also be read as $\{alloc, free\} \leq \{alloc\}$

To introduce capabilities in the system, the primitive *newrgn* has to be used. It gives the permission to allocate, read, write values in the region. We can see that the capability is qualified with a linear property. At creation, we know that it has not been shared. This is an important criterion for reclaiming a region.

$$\frac{r \notin C}{C; \Gamma \vdash newrgn\ () : hnd\ r; \Gamma; C, r^1; \emptyset}\ (\textsc{New})$$

Sometimes using a region handler several times is necessary. For instance, when you need to pass several region handlers as arguments to a function. This is the case when you use a function that copies a list in two distinct regions. To perform this, *aliasrgn* can help. Leaving the scope of this primitive restores the linearity property we had before.

$$\frac{\begin{array}{c} C, r^1; \Gamma \vdash e_\rho : hnd\ r; \Gamma_\rho; C_\rho, r^1; \phi_\rho \\ C_\rho, r^+; \Gamma_\rho \vdash e : \tau; \Gamma'; C', r^+; \phi_e \end{array}}{C, r^1; \Gamma \vdash aliasrgn\ e_\rho\ in\ e : \tau; \Gamma'; C', r^1; \phi_\rho \cup \phi_e}\ (\textsc{Alias})$$

The most interesting rule for the analysis is Free. Here, linearity is the important part, it ensures that the region handler is not shared. Thus, the corresponding region can be freed in a sound way.

$$\frac{C; \Gamma \vdash e_\rho : hnd\ r; \Gamma'; C', r^1; \phi_\rho}{C; \Gamma \vdash freergn\ e_\rho : unit; \Gamma'; C'; \phi_\rho}\ (\textsc{Free})$$

We have a set of rules giving information about the memory behavior of our programs and providing guarantees that a well typed should not crash because of memory management. Moreover, regions give us an abstract view of the heap at compile-time. In section 6, we will see how this view can be useful to do a resource consumption analysis.

3.3 Cost model

To perform a resource consumption analysis, we need to model the runtime environment with respect to memory usage. Programs written in our language can allocate memory with the creation of five different kinds of values: closures, pairs, lists, references and region handlers. Thus, we introduce five constants representing the amount of allocated memory for a pair, a list node, an empty list, a reference and a region handler. For closures, we introduce a specific operator because the amount of memory used is proportional to the number of free variables. We assume that compilation schemes do not introduce additional heap memory allocations.

This allows us to manipulate symbolic expressions that will be instantiated according to the target system. Every amount of memory is a multiple of a memory word (see figure 4) which itself depends on the platform softwares are running on. The cost of a closure is represented with $\mathscr{C}_{cls}(n)$ where n is the size of the closure

Value	Constant	Size (Words)
Pair	$\mathscr{C}_{cpl}$	2
List node	$\mathscr{C}_{cons}$	2
Empty List	$\mathscr{C}_{nil}$	0
Reference	$\mathscr{C}_{ref}$	1
Region handler	$\mathscr{C}_{hnd}$	1

Fig. 4. Memory model

environment. The analysis relies on this model to predict amounts of live memory.

3.4 Analysis

The goal of this analysis is to provide an upper bound of live memory at compile-time to prevent heap overflows. The analysis consists in a mix of several existing resource consumption analyses. It returns the amount of allocated memory in each region involved by a function call. With region sizes and the region mechanism, we are able to consider reclaimed memory.

To analyze a program, we distinguish functions written with a purely functional style from those written with an imperative style. To do this, we rely on the language effect system. In this language, side effects happen through reference updates. Hence, if a function type is labelled with a *write* effect on a region r then this function is considered impure and is analyzed with the analysis based on invariants on iteration spaces. Whereas if the function only performs *alloc* or *read* effects then it is seen as pure and can be analyzed thanks to automatic amortized analysis.

If the side-effect is performed on a local region that do not escape then from a caller point of view, this function is pure. This is a property we rely on in the section 6.

In the next sections, we describe how each analysis works with pure and imperative programming styles and then we show how to compose these analyses.

4 Analysis of pure functions

Based on Tarjan's work [11], the goal is to apply amortized analysis using the potential method without requiring the user intervention. This analysis targets the cost of a sequence of operations considering interactions between these operations. A famous example is the complexity of a sequence of operations on a functional queue. A functional queue q is implemented as a pair of lists (see figure 5). When we push an element in q, we add it in front of the first component. When we take out an element of q, two cases are considered. First, the second component is non-empty, we remove the head and return it. Second, the second component is empty,

meaning either q is empty or only pushes have been performed until now, we reverse the first component in the second component. The option type used in the example is related to error handling.

```
type 'a option = None | Some of 'a

type 'a queue = 'a list * 'a list

let push x (xs, ys) = (x :: xs, ys)

let rec take q =
  match q with
  | ([], []) -> None
  | (xs, y :: ys') -> Some (y, (xs, ys'))
  | (xs, []) -> take ([], List.rev xs)
```

Fig. 5. Queues as paired lists in OCaml

$$push : a \rightarrow a\ queue \rightarrow a\ queue$$
$$take : a\ queue \rightarrow (a \times a\ queue)\ option$$

As we can see, the worst case execution time of the function *take* is linear in the length of the first component because of the call to *reverse*. Hence, pushing n elements and taking them back would be a quadratic sequence of operations. This situation arises because we do not consider the state of the queue.

The potential method introduces the notion of credit to solve this. It is represented by the function Φ in the following equations where $\mathscr{C}(P)$ is the program complexity and, c_i and $\hat{c}_i$ representing respectively the effective cost and the amortized cost of the i^{th} operation.

$$\mathscr{C}(P) = \sum_{i=1}^{n} \hat{c}_i \qquad \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

$$\mathscr{C}(P) = (\sum_{i=1}^{n} c_i) + \Phi(n) - \Phi(0)$$

Fig. 6. The potential method

In our case, the function Φ is twice the length of the first component of a queue. When we add an element to the queue we pay twice the size difference between the previous and the current state of the queue. When we start taking elements out of the queue, the call to *reverse* is already amortized by the accumulated credits. Hence, the worst case execution time of this sequence can be reduced to a linear complexity.

The crux of the analysis is the Φ function. Automating the analysis requires to infer this function. To do this, it is necessary to add some restrictions. In this paper, we are only interested in linear complexities. Then, the function Φ is a linear combination of the function parameters. To find this function, we extract a set of inequalities to minimize. In what follows, the potential method is applied only to memory consumption.

A resource judgement has the following shape and can be read as if there are $n + \Phi(\Gamma)$ memory cells available before the evaluation of e then heap overflows are prevented. At the end of the evaluation $n' + \Phi(\tau)$ memory cells are left.

$$\Gamma, n \vdash e : \tau, n'$$

Types are annotated with potentials c, d and k. We remove region information for readability but assume that they are still available when needed.

$$\begin{array}{ll} \langle \tau_s \rangle ::= & \text{unit} \\ \mid & \text{bool} \\ \mid & \text{int} \\ \mid & \alpha \end{array} \qquad \begin{array}{ll} \langle \tau \rangle ::= & \langle \tau_s \rangle \\ \mid & \langle \tau_a \rangle \xrightarrow[d]{c} \langle \tau_b \rangle \\ \mid & (\langle \tau_a \rangle \times \langle \tau_b \rangle, \text{k}) \\ \mid & (\text{list } \langle \tau \rangle, \text{k}) \\ \mid & (\text{ref } \langle \tau \rangle, \text{k}) \\ \mid & \text{hnd, k} \end{array}$$

Fig. 7. Annotated types for automatic amortized analysis

The analysis is directly adapted from earlier works on automatic amortized analysis. The main difference is the complexity computed on a per region basis to consider reclaimed memory. It is expressed as a set of syntax-directed rules describing memory consumption-related constraints of each language construct. Each type is decorated with a potential annotation. For instance, $List\ \alpha\ k$ represents the list type where each element has a potential k. Hence, the list potential is $k \times length(l)$. To consider region sizes, we need to adapt amortized analysis to the region mechanism. Instead of considering every allocations, we only count allocation made in a specific region. Typing rules are applied for a specific region r. When an expression contains the annotation, the corresponding amount of allocated memory counts only if it is in region r. When a function involves several regions, the analysis is performed once for each region. Hence, we obtain space complexities related to each region involved in the computations.

$$r\ \frac{n \geq m}{\Gamma, n \vdash b : Bool, m}\ (\text{AA-Bool}) \qquad r\ \frac{n \geq m}{\Gamma, n \vdash i : Int, m}\ (\text{AA-Int})$$

$$r\ \frac{n \geq m}{\Gamma, n \vdash x : \tau_s, m}\ (\text{AA-SVar}) \qquad r\ \frac{\Gamma(x) = (\tau, r) \qquad n \geq m}{\Gamma, n \vdash x : (\tau, r), m}\ (\text{AA-RVar})$$

Primitive values do not cost anything as they are allocated on the stack. Thus, their respective rules propagate constraints already known.

$$r\ \frac{\Gamma(\rho) = hnd\ r \qquad \Gamma, x : \tau_x, c \vdash e : \tau, d \qquad n \geq m}{\Gamma, n \vdash fun\ x \rightarrow\ e\ @\ \rho : (\tau_x, c) \rightarrow (\tau, d), m}\ (\text{AA-Fun})$$

If the closure is created in the current analyzed region r , then we need to consider the closure size as the amount of allocated memory.

The analysis being compositional, we cannot easily count the number of times a function is applied. This constrains us to add the following restriction: potential contains in a function closure has to be null. Thus, memory consumption can only be parameterized by the function parameters.

$$r\ \frac{\Gamma, n \vdash e_f : (\tau_x, n_f) \to (\tau, m_f), m' \qquad \Gamma, m' \vdash e_v : \tau_x, m \qquad m' - n_f + m_f \geq m}{\Gamma, n \vdash e_f\ e_v : \tau, m}\ \text{(AA-App)}$$

Function application requires enough accumulated potential to be performed.

$$r\ \frac{\Gamma, n \vdash e_c : Bool, m' \qquad \Gamma, m' \vdash e_t : \tau, m \qquad \Gamma, m' \vdash e_f : \tau, m}{\Gamma, n \vdash if\ e_c\ then\ e_t\ else\ e_f : \tau, m}\ \text{(AA-Cond)}$$

Conditional expressions propagate constraints generated by expressions e_c, e_t and e_f. They are not related to region-allocated memory. To be compositional, both branches return values of the same type and potential. With the $\geq$ relation, this will only compute the maximum amount of allocated memory.

$$r\ \frac{\Gamma, n \vdash e_a : \tau_a, m' \qquad \Gamma[x : \tau_a], m' \vdash e_b : \tau, m}{\Gamma, n \vdash let\ x = e_a\ in\ e_b : \tau, m}\ \text{(AA-Let)}$$

The rule for *let* expressions is similar to the conditional one. It only propagates constraints to the sub-expressions.

$$r\ \frac{\Gamma, n \vdash e_a : \tau_a, m' \qquad \Gamma, m' \vdash e_b : \tau_b, m'' \qquad \Gamma(\rho) = hnd\ r \qquad m'' \geq \mathscr{C}_{cpl} + m}{\Gamma, n \vdash (e_a, e_b)\ @\ \rho : \tau_a \times \tau_b, m}\ \text{(AA-Pair)}$$

Pairs are also allocated in regions. So, if the allocation is made in the region under analysis then we have to accumulate enough credits to create this value. This is represented by this constraint $m'' \geq \mathscr{C}_{cpl} + m$.

$$r\ \frac{\Gamma, n \vdash e : \tau_a \times \tau_b, m \qquad n \geq m}{\Gamma, n \vdash \Pi_i\ e : \tau_i, m}\ \text{(AA-Proj)}$$

$$r\ \frac{n \geq m}{\Gamma, n \vdash Nil : List\ a, m}\ \text{(AA-Nil)} \qquad r\ \frac{\Gamma, n \vdash e_h : a, m' \qquad \Gamma, m' \vdash e_t : List\ a\ k, m'' \qquad \Gamma(\rho) = hndr \qquad m'' \geq \mathscr{C}_{cons} + k + m}{\Gamma, n \vdash Cons\ e_h\ e_t\ @\ \rho : List\ a, m}\ \text{(AA-Cons)}$$

Data constructors without arguments are represented as integers. Hence, regions are not involved in their evaluation. However, *Cons* takes two arguments and is

therefore allocated in a region. The constraint $m'' \geq \mathscr{C}_{cons} + k + m$ characterizes this.

$$\frac{\begin{array}{c}\Gamma, n \vdash e : List\ a\ k, m' \\ \Gamma, m' \vdash e_{nil} : \tau, m \\ \Gamma, x : a, xs : List\ a\ k, m' + k \vdash e_{cons} : \tau, m\end{array}}{\begin{array}{l}\Gamma, n \vdash match\ e\ with \mid Nil \rightarrow e_{nil} \qquad\qquad : \tau, m \\ \qquad\qquad\qquad\qquad\quad \mid Cons\ x\ xs \rightarrow e_{cons}\end{array}}\ \text{(AA-Match)}$$

The pattern matching rule is crucial. Depending on the branch taken, different assumptions about data structure sizes can be made. For instance, if the *nil* branch is taken then we know that the list is empty. But if the *cons* branch is taken then we know that the list has at list one element with its potential. This rule combined with allocation sites drive the analysis.

$$\frac{n \geq \mathscr{C}_{rgn} + m}{\Gamma, n \vdash newrgn\ () : \tau, m}\ \text{(AA-New)}$$

Allocating a new region introduces a new handler to perform operations on this region. This handler is allocated in its own region and requires $\mathscr{C}_{rgn}$ memory words.

When we share a value, we need to share the potential accordingly. To carefully track this, we need the following structural rule which basically ensures that potential is not duplicated.

$$\frac{\Gamma, x : \tau_x, y : \tau_y, n \vdash e : \tau_e, m \qquad share(\tau | \tau_x, \tau_y)}{\Gamma, z : \tau, n \vdash e[z/x][z/y] : \tau_e, m}\ \text{(AA-Share)}$$

With this rule, the amount of potential available cannot be duplicated but is shared among different variables. For instance, $List\ a\ k_0$ means that each list element has k_0 credits. If this list is shared between variables $x : List\ a\ k_1$ and $y : List\ a\ k_2$ then we generate a constraint similar to $k_0 = k_1 + k_2$.

Example

Here, we are looking for the sizes of the regions involved in the computation of the function *duplicate* (see figure 9). This function duplicates twice the list passed as an argument.

$$duplicate : (List\ a\ r, hnd\ r_c, hnd\ r_1, hnd\ r_2) \rightarrow (List\ a\ r_1 \times List\ a\ r_2, r_c)$$

Fig. 8. Type of duplicate

From this type (figure 8), we can see that in the general case, the paired lists can be distributed in three distinct regions, r_c, r_1 and r_2.

```
let rec duplicate xs r r1 r2 =
  match xs with
  | Nil -> (Nil @ r1, Nil @ r2) @ r
  | Cons (x, xs') ->
    let (ys, zs) = duplicate xs' r r1 r2 in
    (Cons(x, ys) @ r1, Cons(x, zs) @ r2) @ r
```

Fig. 9. function duplicate

Duplicate clones xs twice. We can see that the list structures can reside in two different regions at most and that the pair can also be in its own region.

First, the effect analysis marks this function with $read\{r\}$ and $alloc\{r1, r2, r_c\}$ effects. Hence, automatic amortized analysis is employed. From previous explanations, the potential function will be of the form $a \times |xs|$ where a is the potential of each list element and $|xs|$ represents the size of the list xs parameter of the function *duplicate*. The analysis extracts a system of constraints from the program and tries to minimize it.

① $$\Gamma, n_1 \vdash (Nil, Nil) : List\ a\ k_1 \times List\ a\ k_2, m_1$$

② $$\frac{\Gamma_3, n_3 \vdash duplicate\ xs : List\ a\ k_1 \times List\ a\ k_2, m_3 \qquad \Gamma_4, n_4 \vdash (Cons\ x\ ys, Cons\ x\ zs) : List\ a\ k_3 \times List\ a\ k_4, m_4}{\Gamma, n \vdash \begin{array}{l} let\ (ys, zs) = duplicate\ xs \\ in\ (Cons\ x\ ys, Cons\ x\ zs) \end{array}}$$

---

③ $$\Gamma, n_0 \vdash match\ xs\ with \mid Nil \rightarrow (Nil, Nil) \mid Cons\ x\ xs \rightarrow \begin{array}{l} let\ (ys, zs) = duplicate\ xs \\ in\ (Cons\ x\ ys, Cons\ x\ zs) \end{array}$$

This function can allocate memory in different regions (depending on how it is called). We are interested in the more general case. What size regions r, r_1 and r_2 will be? To do this we apply the analysis to each region. Hence, we are going to get three different sets of relations to minimize.

Constraints related to region r_1:

① $$n_1 \geq m_1$$

② $$\begin{array}{l} \Gamma_4 = \Gamma, x : a, ys : List\ a\ k_3, zs : List\ a\ k_4 \\ n_4 \geq size(a \times List\ a\ k_3) + k_3 + m_4 \\ k_3 = k_1, k_4 = k_2 \end{array}$$

$$
\begin{array}{ll}
 & \Gamma = duplicate : List\ a\ q_0, c \rightarrow List\ a\ q_1 \times List\ a\ q_2, d \\
(3) & \quad xs : List\ a\ k_0 \\
 & n_3 - c + d \geq m_3 \\
 & k_0 = q_0, k_1 = q_1, k_2 = q_2
\end{array}
$$

When we solve this set of inequalities, we get $q_0 = 2, q_1 = 0$ and $q_2 = 0$ (assuming that a list node requires two words of memory in the runtime system). Thus, we can conclude that region r_1 grows of $2 \times |xs|$. Then, we apply the same method for r_c and r_2. In the end, we get three symbolic expressions characterizing the size of each region where allocations are performed.

5 Analysis of imperative functions

Programs written with an imperative style perform side-effects through the use of references. To consider them, we need to use a different method: invariants on iteration spaces. We rely on recent work done in the JConsume project [2] targeting Java programs at the bytecode level.

Analysis

We adapt this analysis to programs written in our language. We still rely on the user to provide invariants. They can be expressed using all classic arithmetical and logical operators. These can be directly provided by her or obtained through the use of external tools. They are written with the *with* syntax as in

```
fun x y -> x + y with x < y
```

Fig. 10. *With* syntax

Contrary to the original work, here we do not care about the notion of escape memory as it is already handled through the use of regions. As in the automatic amortized analysis, we are looking for the sizes of the different regions involved in the computations. To perform this, invariants characterize the number of iterations. Here, invariants are linear relations but the user could provide other classes as well. The advantage of linear invariants comes when we try to infer them.

Example

If we take an imperative version of the previous example, we obtain

The invariants we are interested in characterize size relations of data structures involved (see figure 12). Amounts of memory allocated in r_1 and r_2 are extracted from them. They are related to the length of the list xs. Expected invariants are in figure 12. The function *length* represents the projection of the list structure into the integer domain.

As side-effects are performed through references, we need to also keep track of the amount of memory available through references. Here, we notice that reference

```
let duplicate xs rc r1 r2 =
  let r = newrgn () in
  let ys = ref Nil @ r in
  let zs = ref Nil @ r in
  let rec loop es =
    match es with
    | Nil -> ()
    | Cons (e, es') ->
      ys += Cons (e, !ys) @ r1;
      zs += Cons (e, !zs) @ r2;
      loop es'
  in loop xs; (!ys, !zs) @ rc ;
```

Fig. 11. function duplicate

Region	Invariant
r_1	$length(xs) = length(es) + length(!ys)$
r_2	$length(xs) = length(es) + length(!zs)$
r_c	$\mathscr{C}_{cpl}$

Fig. 12. Invariants for duplicate

assignments are cumulative, meaning that data is added to previous data reachable through the reference. Hence, to determine the amount of memory reachable from them, we rely on the same invariants. In the end, we dereference ys and zs, so we propagate the amounts of memory reachable through these references.

From these, we obtain symbolic expressions characterizing the amounts of allocated memory in different regions. From the outside, this function is seen as pure.

Region	Amount
r_1	$\mathscr{C}_{cons} * length(xs)$
r_2	$\mathscr{C}_{cons} * length(xs)$
r_c	$\mathscr{C}_{cpl}$

Fig. 13. Amounts of memory

6 Composition of analyses

Previous analyses are concerned with space-complexity but a program mixing both purely functional and imperative features involves composition. Results provided by previous analyses do not track side-effects. This lack prevents propagation of size relations to make a sound analysis.

To track side-effects, it is necessary to manage references with some accuracy. For instance, if a reference is updated then we need to propagate new size information about the value being dereferenced to pursue the analysis. Unfortunately, space complexities are not directly related to the sizes of the data structures involved. The

append function is linear in term of the first argument but the returned list length is the sum of both argument lengths.

References require careful handling. For instance, a reference update may imply a size change. This new size has to be propagated to the rest of the program. To do this, we annotate each reference with a unique identifier. A reference is annotated with the region it lives in and the region referenced data lives in. To track effects, we refine the *write* effect with the label of the updated reference.

If the programmer employs the region mechanism with a fine granularity, it is possible to derive data structure sizes from space complexity. The function *append* is a good illustration of this principle. Its type shows that the two lists can reside in two different regions and that the resulting list lives in the same region than the second argument. The combination of the base case and the effect $alloc\ r_2$ entails that data is added to the second list. Thanks to this, we can deduce that the size of the resulting list is the sum of lists passed as arguments.

When size relations cannot be extracted automatically, the programmer has to provide them manually with the *with* syntax just like for the imperative analysis to run the analysis. Annotations can be provided through the use of classic arithmetical operators and the *size* operator. This *size* operator is a way to count the number of node of a data structure. Variables bound to integers can be used directly to refer to the integer itself.

To perform the composition, we assume the whole program available. The composition follows the control flow graph of the program. It is expressed as a set of rules whose shape is

$$\mathscr{C}; \sigma; \phi \vdash e : \sigma'; \phi'$$

where σ and ϕ represents respectively the sizes of regions allocated and amounts of memory reachable through each references visible in the current scope, e an expression of the language.

$$\frac{}{\mathscr{C}; \sigma; \phi \vdash n : \sigma; \phi}\ \text{(CSP-Int)} \qquad \frac{}{\mathscr{C}; \sigma; \phi \vdash b : \sigma; \phi}\ \text{(CSP-Bool)}$$

As in the previous analyses, primitives values are not allocated in regions. Hence, neither σ nor ϕ is modified.

$$\frac{}{\mathscr{C}; \sigma; \phi \vdash x : \sigma; \phi}\ \text{(CSP-SVar)} \qquad \frac{}{\mathscr{C}; \sigma; \phi \vdash x : \sigma; \phi}\ \text{(CSP-RVar)}$$

Using a variable does not affect regions.

$$\frac{\sigma' = update(\sigma, \mathscr{C}_{cls}(fv(e) \setminus \{x\}) + 1, \rho)}{\mathscr{C}; \sigma; \phi \vdash fun\ x \to e\ @\ \rho : \sigma'; \phi}\ \text{(CSP-Fun)}$$

Thanks to previous analyses, we already know the space-complexity of functions used by the program. The only thing we care about here is the size of the closure itself. One word for the code pointer and the rest for the closure environment. Here, the function *update* adds $\mathscr{C}_{cls}(fv(e) \setminus \{x\}) + 1$ memory words to the region represented by ρ.

$$\frac{\mathscr{C};\sigma;\phi \vdash e_a : \sigma_a;\phi_a \qquad \mathscr{C};\sigma_a;\phi_a \vdash e_b : \sigma_b;\phi_b \qquad \sigma';\phi' = instantiate(\mathscr{C}(e_a), size(e_b))}{\mathscr{C};\sigma;\phi \vdash e_a\ e_b : \sigma';\phi'}\ \text{(CSP-App)}$$

Composition really happens at call sites with the function *instantiate*. This is where symbolic expressions are merged and side-effects propagated.

$$\frac{\mathscr{C};\sigma;\phi \vdash e_c : \sigma_c;\phi_c \qquad \mathscr{C};\sigma;\phi \vdash e_t : \sigma_t;\phi_t \qquad \mathscr{C};\sigma;\phi \vdash e_f : \sigma_f;\phi_f \qquad \sigma';\phi' = max(\sigma_t,\sigma_f), max(\phi_t,\phi_f)}{\mathscr{C};\sigma;\phi \vdash if\ e_c\ then\ e_t\ else\ e_f : \sigma';\phi'}\ \text{(CSP-Cond)}$$

Conditional expressions introduce the use of the operator max. To keep the analysis sound, we need to consider the worst case. Here, it means the maximum of memory allocated in a region and the maximum amount of memory reachable from a reference.

$$\frac{\mathscr{C};\sigma;\phi \vdash e_a : \sigma_a;\phi_a \qquad \sigma_a;\phi_a \vdash e_b : \sigma';\phi'}{\mathscr{C};\sigma;\phi \vdash let\ x = e_a\ in\ e_b : \sigma';\phi'}\ \text{(CSP-Let)}$$

$$\frac{}{\mathscr{C};\sigma;\phi \vdash Nil : \sigma;\phi}\ \text{(CSP-Nil)} \qquad \frac{\mathscr{C};\sigma;\phi \vdash e_h : \sigma_h;\phi_h \qquad \mathscr{C};\sigma_h;\phi_h \vdash e_t : \sigma_t;\phi_t \qquad \sigma' = update(\sigma,\mathscr{C}_{cons},\rho)}{\mathscr{C};\sigma;\phi \vdash Cons\ e_h\ e_t\ @\ \rho : \sigma';\phi_t}\ \text{(CSP-Cons)}$$

$$\frac{\mathscr{C};\sigma;\phi \vdash e : \sigma_e;\phi_e \qquad \mathscr{C};\sigma_e;\phi_e \vdash e_{nil} : \sigma_{nil};\phi_{nil} \qquad \mathscr{C};\sigma_e;\phi_e \vdash e_{cons} : \sigma_{cons};\phi_{cons} \qquad \sigma';\phi' = max(\sigma_{nil},\sigma_{cons}), max(\phi_{nil},\phi_{cons})}{\mathscr{C};\sigma;\phi \vdash match\ e\ with \mid Nil \to e_{nil} \mid Cons\ x\ xs \to e_{cons} : \sigma';\phi'}\ \text{(CSP-Match)}$$

$$\frac{\mathscr{C};\sigma;\phi \vdash e : \sigma_e;\phi_e \qquad \phi' = add(\phi_e, l, size(e))}{\mathscr{C};\sigma;\phi \vdash ref_l\ e : \sigma';\phi'}\ \text{(CSP-Ref)}$$

When a reference is created, we need the corresponding amount of memory. The interesting thing about references appears when assignments are made. We have to update the amount of memory reachable by the corresponding reference. This is what the function *update* performs on ϕ. This update depends on the nature of the assignment. If cumulative then we add a few memory words to the already available amount. If non-cumulative, then we introduce a max operator to keep the maximum amount of reachable memory.

$$\frac{\mathscr{C};\sigma;\phi \vdash e_f : \sigma_r;\phi_r \qquad \mathscr{C};\sigma_r;\phi_r \vdash e_v : \sigma_v;\phi_v \qquad \phi' = update(\phi_v, label(e_r), size(e_v))}{\mathscr{C};\sigma;\phi \vdash e_r := e_v : \sigma';\phi'} \text{ (CSP-Assign)}$$

To allocate a new region, you need to have at least the amount of memory necessary to store a region handler.

$$\frac{\sigma' = update(\sigma, \mathscr{C}_{rgn}, \rho)}{\mathscr{C};\sigma;\phi \vdash newrgn\ () : \sigma';\phi} \text{ (CSP-New)}$$

$$\frac{record(\sigma) \qquad \sigma' = free(\sigma,\rho) \qquad \phi' = clean(\phi,\rho)}{\mathscr{C};\sigma;\phi \vdash freergn\ \rho : \sigma';\phi'} \text{ (CSP-Free)}$$

The type system checks that the region corresponding to ρ can be reclaimed in a safe way. Hence, we can ignore the size of this region to compute an upper bound of live memory. As we are freeing memory, there is a local maximum. We need to save the sum of region sizes to track the maximum amount of live memory. At the end of the analysis, we take the maximum between each local maximum. We do not need to consider the amount of memory available at the end of the execution because the typing discipline checks that no region remains unclaimed.

7 Example

The following example shows how the analysis is performed. The main function is *rev_append* which concatenates two lists by reversing the first one to be tail recursive. This function can be written in at least two different styles: purely functional and imperative.

This program builds two regions, *r* and *rr*, and allocates two lists, *xs* and *ys*, in *r* and *rr* respectively. Then, it concatenates *xs* and *ys* thanks to *rev_append* and reclaims the region *rr*.

```
let r = newrgn () in
let ys = [12; 15; 18] @ r in
let rr = newrgn () in
let xs = [3; 6; 9] @ rr in
let zs = rev_append xs ys r in
  freergn rr
```

The left version employs a purely functional style and the right version an imperative one with the side-effect on the reference *rs* along the computation. The effect system captures this difference and allows different analyses to be performed and combined.

functional `rev_append`	imperative `rev_append`

`let rec rev_append xs ys r =`
` match xs with`
` | Nil -> ys`
` | Cons (h, t) ->`
`  let ys' = Cons(h,ys) @ r in`
`  rev_append t ys' r`
	`let rev_append xs ys r =`
	` let rs = ref ys in`
	` let rec loop xs =`
	`  match xs with`
	`  | Nil -> ()`
	`  | Cons (h, t) ->`
	`  rs += Cons(h,!rs) @ r;`
	`  loop t`
	` in loop xs; !rs`

As you can see here, each region contains the structure of the list. The type of the *rev_append* function gives us information about its memory allocation behavior.

$$rev\_append : (\alpha\ list, r_a) \to (\alpha\ list, r_b) \to hnd\ r_b \to (\alpha\ list, r_b)$$

The first list can be allocated in a region r_a, the second in a region r_b but in the end the returned list will be allocated in region r_b. This information is useful to track the different lifespans of the regions involved in the computations.

Pure functions are analyzed with the automatic amortized analysis. It extracts a set of constraints of the function and tries to minimize it. In *rev_append*, the interesting part is the application of the data constructor *Cons*. If before this application, the amount of memory available is n, then the constraint $n \leq \mathscr{C}_{Cons} + n'$, where n' is the amount of memory available after, needs to be satisfied. $\mathscr{C}_{Cons}$ represents the amount of memory necessary to allocate an element of a list. In this case, the extracted cost is proportional to the size of the first list. Here, *rev_append* behaves like the function *append*. This allows us to infer size relations. Pure functions do not act on the program state, hence there is no information related to side-effects to propagate.

The imperative version of *rev_append* is analyzed thanks to invariants on iteration spaces. Here, the side-effect is local to the function. Hence, the amount of allocated memory is the only information propagated. Here, the invariant is $length\ !rs = size\ xs + size\ ys$ where $size\ ys$ is a constant. It is linear and could be obtained in an automatic way. In other cases, we would rely on programmer annotations. From this, we can deduce the amount of allocated memory. In this case, it is also proportional to the length of the first list.

In this example, both analyses return similar results. Then, we can instantiate symbolic expressions to get the amount of memory necessary to execute the program in a safe way. Here, we can see that the region *rr* is freed at the end. If the program would be larger, this region would have been considered as non-existent to analyze the rest of the memory allocated.

8 Conclusion

Providing an upper bound at compile-time on the amount of live memory at runtime would allow the introduction of high-level languages in the embedded system communities. This paper proposes to conceive a language and an analysis to move towards this end.

We present a language *à la ML* mixing pure and imperative features with an explicit region mechanism. Memory management is performed by the programmer through a set of primitives and checked at compile time. This mechanism provides information about the heap topology and lifespans of allocated values.

The analysis relies mainly on an effect system and a region mechanism. The effect system allows us to combine several analyses depending on the programming style employed by the programmer. Regions offer lifespans of allocated values. This prevents overpessimistic bounds because we can consider reclaimed regions at compile time. Automatic amortized analysis is used on pure functions and invariants on iteration spaces are employed on imperative functions.

Correctness of this analysis relies on the correctness of the type system which has been proved through progress and preservation lemmas which haven't been presented in this paper but proofs are similar to [4]. To validate this approach in practice, a prototype is currently being developed.

References

[1] Albert, E., P. Arenas, S. Genaim and G. Puebla, *Closed-Form Upper Bounds in Static Cost Analysis*, J. Autom. Reasoning **46** (2011), pp. 161–203.

[2] Braberman, V. A., D. Garbervetsky, S. Hym and S. Yovine, *Summary-based inference of quantitative bounds of live heap objects*, Sci. Comput. Program. **92** (2014), pp. 56–84.

[3] Cousot, P. and R. Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in: *Conference Record of the Fourth Annual Symposium on Principles of Programming Languages*, POPL 77 (1977), pp. 238–252.

[4] Fluet, M., G. Morrisett and A. J. Ahmed, *Linear Regions Are All You Need*, in: *Proceedings of the 15th European Symposium on Programming*, ESOP 06, 2006, pp. 7–21.

[5] Hoffmann, J., A. Dash and S.-C. Weng, *Towards automatic resource bound analysis for ocaml* (2015). URL http://www.cs.cmu.edu/~janh/papers/HoffmannW15.pdf

[6] Hofmann, M. and S. Jost, *Static Prediction of Heap Space Usage for First-order Functional Programs*, in: *Proceedings of the 30th Symposium on Principles of Programming Languages*, POPL '03 (2003), pp. 185–197.

[7] Hofmann, M. and S. Jost, *Type-Based Amortised Heap-Space Analysis*, in: *Proceedings of the 15th European Symposium on Programming*, ESOP 06, 2006, pp. 22–37.

[8] Hofmann, M. and D. Rodriguez, *Automatic Type Inference for Amortised Heap-Space Analysis*, in: *Proceedings of the 22sn European Symposium on Programming*, ESOP 13, 2013, pp. 593–613.

[9] Hughes, J., L. Pareto and A. Sabry, *Proving the Correctness of Reactive Systems Using Sized Types*, in: *Proceedings of the 23rd Symposium on Principles of Programming Languages, POPL 96* (1996), pp. 410–423.

[10] Montenegro, M., R. Peña and C. Segura, "Foundational and Practical Aspects of Resource Analysis: First International Workshop, FOPARA 2009, Eindhoven, The Netherlands, November 6, 2009, Revised Selected Papers," Springer Berlin Heidelberg, Berlin, Heidelberg, 2010 pp. 34–50.

[11] Tarjan, R. E., *Amortized computational complexity*, SIAM Journal on Algebraic and Discrete Methods **6** (1985), pp. 306–318.

[12] Tofte, M. and J.-P. Talpin, *Implementation of the Typed Call-by-value λ–calculus using a Stack of Regions*, in: *Proceedings of the 21st Symposium on Principles of Programming Languages*, POPL 94, 1994, pp. 188–201.

[13] Vasconcelos, P., "Space Cost Analysis Using Sized Types," Ph.D. thesis, University of St Andrews (2008).

[14] Wegbreit, B., *Mechanical Program Analysis*, Commun. ACM **18** (1975), pp. 528–539.