

AHA: Amortized Heap Space Usage Analysis

– Project Paper –

Marko van Eekelen, Olha Shkaravska, Ron van Kesteren,
Bart Jacobs, Erik Poll, and Sjaak Smetsers

Project Contact: `marko@cs.ru.nl`

Institute for Computing and Information Systems,
Radboud University Nijmegen,
Toernooiveld 1, 6525 ED Nijmegen, Netherlands

Abstract

This paper introduces **AHA**, an NWO-funded¹ 344K Euro research project involving research into an amortized analysis of heap-space usage by functional and imperative programs. Amortized analysis is a promising technique that can significantly improve on simply summing worst case bounds. The project seeks to apply this technique to obtain non-linear bounds on heap-space usage for lazy functional languages and to adapt the results for imperative languages.

1 INTRODUCTION

Estimating heap consumption is an active research area as it becomes more and more an issue in many applications. Examples include programming for small devices, e.g. smart cards, mobile phones, embedded systems and distributed computing, e.g. GRID computing. The standard technique for estimating heap consumption gives unrealistically high bounds in many cases. As a consequence, in practice amounts of heap are used that are unnecessarily expensive and for small devices highly unpractical. A more accurate analysis is wanted for these cases in particular and for high integrity real-time applications in general.

A promising technique to obtain accurate bounds of resource consumption and gain is amortized analysis. The amortized analysis of a resource considers not the worst case of a single operation but the worst case of a sequence of operations. The overall amortized cost of a sequence is calculated by taking into account both the higher costs of one operation and the lower costs of another weighing them according to their distribution. In many cases amortized analysis can give much more accurate resource consumption estimates than the standard worst case analysis.

Combining amortization with type theory allows to check linear heap consumption bounds for functional programs with explicit memory deallocation. The **AHA** project aims to adapt this method to deal with *non-linear* bounds within (lazy)

¹This project is sponsored by the Netherlands Organization for Scientific Research (NWO) under grantnr. 612.063.511.

functional programs as well as to transfer the results of the functional programming community to the imperative object-oriented programming world by applying the amortized method to derive accurate bounds for heap usage of Java programs. In this way the project both enhances fundamental theory and practical impact.

1.1 Relevance

Because memory exhaustion will invoke garbage collection, heap usage can indirectly slow down execution and hence influence time complexity. A better heap space analysis will therefore enable a more accurate estimation of time consumption. This is relevant for time-critical applications. Analyzing resource usage is also interesting for optimizations in compilers for functional languages, in particular of memory allocation and garbage collection techniques. A more accurate estimation of heap usage enables allocation of larger memory chunks beforehand instead of allocating memory cells separately when needed, leading to a better cache performance.

Resource usage is an important aspect of any safety or security policy, as exhausting available resources typically causes system failure. Indeed, it is one of the most important properties one wants to specify and verify for Java programs meant to be executed on (embedded) Java-enabled devices with limited amounts of memory, such as smart-cards implementing the Java Card platform, and MIDP mobile phones implementing the Java 2 Micro Edition (J2ME) platform.

The Java Programming Language (JML) already provides some rudimentary possibilities for specifying resource usage of Java programs. However, there is only syntax for specifying this, without any clear semantics, and there are no tools for actually monitoring – let alone, proving – that such constraints are met.

1.2 Research questions

The **AHA** project investigates the possibilities for analyzing heap usage for both functional and imperative object-oriented languages, more specifically, Clean and Java. It aims to answer the following research questions:

- It is clear, that the heap analysis for functional languages can be improved so that a wider class of resource usage bounds than just linear bounds can be guaranteed. The question is how complex the type-checking and inference procedures may be. In particular, which arithmetics and constraint solvers will be needed and for which classes of programs?
- Can heap space analysis be done for lazy functional languages?

Heap space analysis for lazy functional languages is clearly more complicated than for strict languages, because the heap space is also used for unevaluated expressions (closures). The amount of memory that is used at a certain moment depends on the evaluation order of expressions, which in its turn is influenced by the strictness analyzer in the code generating compiler.

- How successfully can one adopt the approach for object-oriented imperative languages? The aim here is to be able to prove – or, better still, derive – properties about the heap space consumption for Java programs specified in an extension of JML (Java Modeling Language).

2 INTRODUCTION TO AMORTIZATION

2.1 Amortization of resources in program analysis

The term “amortization” came to computer science from the financial world. There it denotes a process of ending a debt by regular payments into a special fund. In computer science amortization is used to estimate time and heap consumption by programs. “Payments” in a program are done by its operations or the data structures that participate in the computation, see [14]. These payments must cover the overall resource usage. Methods of distribution of such “payments” across operations or data structures form the subject of amortized analysis.

To begin with, consider amortized time costing. Given a sequence of operations, one often wants to know not the costs of the individual operations, but the cost of the entire sequence. One assigns to an operation an *amortized cost*, which can be greater or less than its actual cost. All one is interested in, is that the sum of the amortized costs is large enough to cover the overall time usage. Thus, one redistributes the run time of the entire sequence over the operations. The simplest way to arrange such redistribution is to assign to each operation the average cost $T(n)/n$, where $T(n)$ is the overall run time, and n is the amount of operations. A *rich* operation is an operation for which its amortized cost, say, $T(n)/n$, exceeds its actual cost. Rich operations pay for “poor” ones.

Consider the Haskell-style version of the function `multipop` from [8] that, given a stack S and a counter k , pops an element from the top of the stack till the stack is empty or the counter is zero:

```

multipop :: Int → Stack Int → Stack Int
multipop k []      = []
multipop 0 (x:xs) = x:xs
multipop k (x:xs) = multipop (k-1) xs

```

If the actual costs of push and pop are 1 each, then the actual cost of the entire program is $\min(s, k)$, where s is the size of the stack S . Assigning amortized costs one may think in the following way. Each operation push has the actual cost 1, but it “takes care” about the future of the element it pushes on the stack. This element may be popped out. So push obtains the amortized cost 2 to pay for itself and for the possible call of pop. Thus, all possible calls of pop are payed while constructing the input S for `multipop`. *After construction of the stack S* , the amortized cost for pop is 0, and the amortized cost of `multipop`, which is the sum of the amortized costs of pops, is zero as well. The amortized cost of the construction of S followed by `multipop` is $2s$, whereas its actual cost is $s + \min(s, k)$.

The correctness of an amortized analysis for a sequence of n operations is defined by $\sum_{i=1}^j a_i \geq \sum_{i=1}^j t_i$, where $j \leq n$, a_i is the amortized cost of the i th operation, and t_i is its actual cost. In this way one ensures that, at any moment of the computation, the overall amortized cost covers the overall actual cost.

2.2 Views to Amortization

A general understanding of amortization [16] is based on a graph presentation of programs. A program is viewed as a directed graph with *states* (i.e. data structures) as nodes and *edges* (i.e. basic operators or constructs) as transitions between them. A possible *computation* is a path in the graph.

Branching in the graph appears due to non-determinism or due to the fact that states may be abstract. In other words, states may represent not concrete operational states like memory layouts, but their “projections”. When concrete information is lost, the *if-then-else* construct is presented in the graph by branching.

In a *physicist’s view* of amortization one assigns to any state s a real number $\Phi(s)$ called the *potential* of the state s . For the time being we consider only non-negative potentials. The first intuition behind the potential function is that it reflects the amount of resources (heap units, time ticks) that may be discharged during a computation, starting from the state s . In the physicist’s approach the amortized cost of an *any* path between some s and s' is the difference $\Phi(s') - \Phi(s)$.

To introduce a *banker’s view* we first note the following. Each edge $e(s_1, s_2)$ has its actual cost $t(s_1, s_2)$ defined by the corresponding basic command or the construct. Let it have an amortized cost $a(s_1, s_2)$. The difference $a(s_1, s_2) - t(s_1, s_2)$ for the edge $e(s_1, s_2)$ is called a *surplus*. If the difference $a(s_1, s_2) - t(s_1, s_2)$ is positive, it is called a *credit*, it may be used to cover the actual costs of further computations. The actual/amortized cost of a path π , between some s and s' , is the sum of actual/amortized costs of edges. In principle, the costs of two paths π and π' between the same vertices may differ. If for any two states s and s' it holds that $a(s, s') = t(s, s') + \Phi(s') - \Phi(s)$, then the analysis is called *conservative*.

It is clear that for any physicist’s view one can find a corresponding banker’s approach. The opposite transformation is more complicated. The banker’s approach is more general than the physicist’s one, because one consider particular paths, but not only their initial and end points. However, it has been shown [16] that for any banker’s amortization distribution a there is a “better” conservative distribution a' and a potential function Φ for it, such that:

- $a'(s, s') = t(s, s') + \Phi(s') - \Phi(s)$ (a conservative analysis),
- the new analysis has the same set of pluspoints as the old one. A *pluspoint* is a vertex for which the surpluses of all paths from the initial state to it are non-negative,
- $a'(s_1, s_2) \leq a(s_1, s_2)$ for any edge $e(s_1, s_2)$.

Thus, without loss of generality we restrict our attention to a *conservative* amortized analysis, where amortized costs depend only on the end and initial vertices, but not on a concrete path.

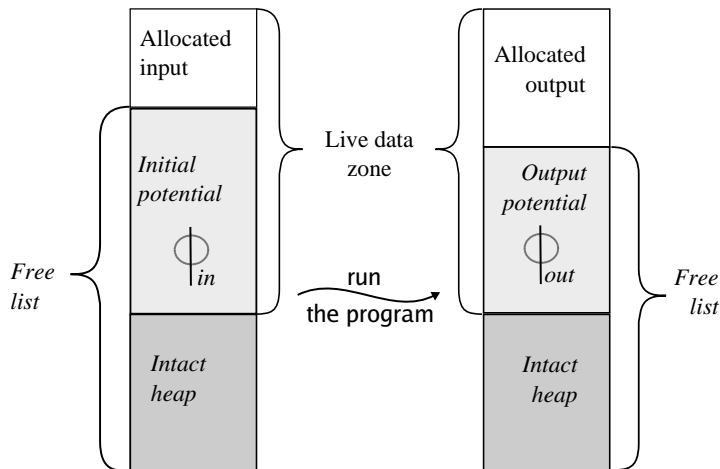


FIGURE 1. Heap layouts before and after the computations

2.3 Amortization for Heap Consumption Gives Size of Live Data

Any data structure that exists during the computation of a function may be constructed either from heap units from the initially allocated units defined by the initial potential function or reused heap cells of initial data (for the language with destructive pattern matching).

If a heap management is performed via maintaining a free list, then the heap layouts before and after the computations are presented by the schema in Figure 1. One can see maintaining a free list as an ideal garbage collector: once a location is not used it is put on the top of the free list. A fresh cell is taken from the top of the free list. Thus, a potential function and a size of input data define an upper bound on the size of the live data at any moment of computation. In general, the physicist's approach gives the following dependency:

$$size(input) + \Phi_{in} = size(data_current) + \Phi_{current} = size(output) + \Phi_{out}$$

Below we discuss how amortization may be incorporated into a type system.

3 STATE OF THE ART: A TYPE SYSTEM FOR LINEAR BOUNDS ON THE SIZE OF LIVE DATA

One can implement a heap-aware amortized analysis via an annotated type system. In this section we consider an annotated type system that corresponds to a banker's view. It was introduced by Hofmann and Jost [10] for linear bounds on heap consumption. Given a first-order program this system allows to infer an upper bound (if it exists and is linear) on the amount of freshly-allocated heap units. More precisely, the size of live data during a run of the program will never exceed the size

of the input plus the inferred linear function. The operations that effect heap consumption are constructors and pattern matching. The coefficients of linear bounds appear in the form of numerical annotations (constants) for types. For instance, a program that creates a fresh copy of a list of integers

```
copy :: [Int] → [Int]
copy []      = []
copy (x:xs) = x : copy xs
```

has the annotated signature $L(\text{Int}, 1), 0 \rightarrow L(\text{Int}, 0), 0$. It means that each element of an input list must be supplied with 1 extra heap unit to fix the space for its copy. The connection with the banker's view is obvious: the annotations, attached to constructors, play a role of credits. The potential of a list $L(\text{Int}, 1)$ of length n is $k \cdot n$. The heap consumption by a program pf with the signature $L(\text{Int}, k), k_0 \rightarrow L(\text{Int}, k'), k'_0$ does not exceed $k \cdot n + k_0$ heap units, and at the end of the computation at least $k' \cdot n' + k'_0$ heap units are available, with n and n' be the sizes of the input and output lists, respectively.

In fact, the type system above inferred two (linear) potential functions of a given program: the potential of an input and the potential of an output. The potential of the input may be discharged during the run of the program, and the potential of the output may be used in further computations.

It is possible to extend this approach for non-linear bounds. The aim of the presented project is to study such extensions.

4 PROJECT PLAN

To answer the three research questions posed in Section 1.3, the project is partitioned into an initial step followed by two parallel research lines. The initial step serves as a pre-requisite for the two lines and will establish the foundations of amortized analysis with non linear bounds for strict languages. After that, a fundamental theoretical research line will extended this analysis to a lazy language. A parallel practical line will transfer the theoretical results to a more practical imperative object-oriented setting.

Ultimately, we want to implement the type systems for heap space usage to obtain an implementation that can check whether a given program, possibly with some type-annotations, meets a given bound on heap space usage or an implementation that can actually compute such a bound.

4.1 Amortized analysis with non-linear bounds

There are many interesting examples that require non-linear heap space, for instance matrix multiplication and Cartesian products. Also the generation of a sports competition programme, in which every team plays a home and an away match against every other team, needs a non-linear amount of heap space.

In all of these examples, the size of the output is also non-linear when expressed in terms of the input: the sports competition has $n \cdot n - n$ matches, where n is the number of teams. Therefore, it is clear that the size relations are important when computing amortized bounds.

Methodology. On the theoretical level (without implementing it for an actual programming language) we will tackle the derivation of size relations separately from heap-space usage to keep both systems as simple as possible. The results from the system that derives the size relations serves as input for the amortized analysis. The amortized analysis will be an extension of the existing linear analysis [10].

4.2 Amortized Heap Analysis of a Lazy Language

An amortized time analysis for call-by-need² languages is considered in [14]. Instead of credits it uses *debts* to cover costs of *closures* (suspensions). A closure is allowed to be forced only after its debt is “payed off” by the operations preceding the operation which forces the closure.

Choice of Programming Language. To consider heap usage analysis for lazy functional programming languages, we will begin with a strict version of core-Clean. We have chosen Clean since Clean’s uniqueness typing [3] makes Clean more suited as a starting point than e.g. Haskell, since with uniqueness typing reuse of nodes can be analysed in a sophisticated manner. For this strict Core-Clean language we will define an alternative operational semantics which will take heap usage into account, and then formulate a type system in which annotations in types express costs.

Methodology. Camelot [13] is an ML-like strict first order functional language with polymorphism and algebraic data types. To enable analysis of heap usage Camelot makes a syntactic distinction between destructive and non-destructive pattern matchings, where destructive pattern matching allows a node of heap space to be reclaimed; it is expected to be relatively easy to transfer such a distinction to a language that has uniqueness typing, as this can enforce the safe use of destructive pattern matching. Therefore, we expect that the results achieved for Camelot will be quickly transferred to the strict version of core-Clean. Then, we will make incremental changes, by changing the strict semantics into a mixed lazy/strict semantics and then investigate the effect on the operational semantics and the type system. This is not a big step in the dark since the heap-aware inference system from [10] already has some flavor of the call-by-need semantics. *Shared* usage of variables by several expressions is treated, for instance, in the MATCH-rule given below in Section 5.3 and in the SHARE-rule in [10].

²Following [14] we associate *call-by-value* with strict languages, *call-by-name* with lazy languages without memoization, and *call-by-need* with lazy languages with memoization.

4.3 Adaptation to Object-Orientation

Choice of Programming Language. As the object-oriented programming languages to be studied we have chosen Java. We will use the Java semantics developed in the LOOP project [11], which includes an explicit formalisation of the heap. This will first require accurately accounting of heap usage in the type-theoretic memory model underlying the LOOP tool [5].

The Java Modeling Language JML, a specification language tailored to Java, already provides a syntax for specifying heap usage, but this part of JML is as yet without any clear semantics. We want to provide a rigorous semantics for these properties about heap space usage and then develop an associated programming logic for proving such properties.

Methodology. We will start to adjust the analysis of Section 5.3 by applying it to classes that admit a functional *algebraic data-type (ADT) interface*. These classes can be considered as defining a number of operations on an algebraic data type. One extracts basic imperative routines which contain explicit allocation/deallocation, and correspond to (co)algebraic operations, and have functional counterparts, like data constructors and pattern matching(s). A field assignment, for example, may be presented as a composition of the destructive match and a constructor. Heap-aware typing judgments must be defined for these macro-operations and the language constructs like `if`-branching, sequencing and `while`-repetition.

Next, research will be done to alleviate the restrictions that are set upon the classes in order to make the analysis applicable. For that purpose, we will investigate the possibility of introducing amortized variants of existing specific analyses (such as the non-recursive [6] and the symbolic [7] which treats aliasing).

One of the main problems for heap space analysis will be aliasing. Aliasing-aware type systems and logics presented in [1, 12] may be considered separately from the resource-aware typing system and are to be combined with it at the very last stage of the design of the proof system.

5 FIRST STEPS

5.1 Towards non-linear upper bounds on the size of live data

It is convenient to measure the potentials of data structures in terms of the sizes. For instance, for a list of length n its potential may be a function of n , that is $\Phi(n)$. In general one assigns a potential to an *overall data structure*. In other words, a potential is assigned to the abstract state that is the collection of the sizes of the structures existing in a given concrete state. Now, consider a program that creates the initial table from a list of n rows and a list of m columns. For its input of type $L_n(\text{string}) \times L_m(\text{string})$ the potential $\Phi(n, m)$ should be proportional to $n \cdot m$.

The banker's view is reduced to assigning a credit to each constructor of a data structure. For instance, in [10] each constructor of a list of type $L(\alpha, k)$ has a constant credit k , and thus the potential of the list is $k \cdot n$, where n is its length.

In general the credit of a node may be a function. It may depend on the position of the node in the list, and/or on the size of the list, as well as on the size of “neighboring” data structure, etc. For instance, in the table-creating program the annotated type of its input may be $L_n(\text{Int}, k) \times L_m(\text{Int}, 0)$, where $k(\text{position}, n, m) = m$.

In the linear heap-consumption analysis [10] these dependencies were not taken into account. This makes the analysis very simple, because it reduces to solving linear inequalities. It covers a large class of functional programs with linear heap consumption, where coefficients of linear functions are credits of constructors.

Introducing dependencies will significantly increase the complexity of type checking and inference. We study classes of programs for which type checking and inference of non-linear bounds are decidable.

5.2 Examples: going on with amortization-and-types

The linear heap-consumption analysis shows that amortization and types suit each other. In this section we consider examples one of which illustrates the advantages of their combination and the other one motivates study of annotated types for non-linear heap consumption.

5.2.1 Type systems bring modularity to amortized analysis

In the following example the naive worst-case analysis significantly overestimates the real heap consumption and the precise analysis is rather complicated. We show that with the help of types annotated with credits, one obtains a very good upper bound for a “reduced price”: types make the analysis modular and, thus, simpler and more suitable for automated checking or inference.

Consider queues (“first-in-first-out” lists) presented as pairs of lists in the usual way. A queue q is represented by a pair (xs, ys) , such that q is $xs ++ (\text{reverse } ys)$. The head of the list xs is the head of the queue, and the head of ys is the tail of the queue. For instance, the queue $[1, 2, 3, 4, 5]$ may be presented as $([1, 2, 3], [5, 4])$. One adds elements to the queue by pushing them on the head of ys , see Figure 2 below. After adding 6 the resulting queue is presented by $([1, 2, 3], [6, 5, 4])$. The function “remove from the queue”, will pop 1 from xs . Consider the code for `remove`, where `reverse` creates a *fresh copy* of the reversed list:

```
remove :: ([Int], [Int]) -> (Int, ([Int], [Int]))
remove ([], []) = error
remove ([], ys) = remove (reverse ys, [])
remove (x:xs, ys) = (x, (xs, ys))
```

We assume that input pairs and output triples are *not boxed*, that is, two input pointer values are taken from the operand stack and in the case of normal termination three values will be pushed on the operand stack. (This helps to avoid technical overhead with heap consumption for pairs and triples creation.)

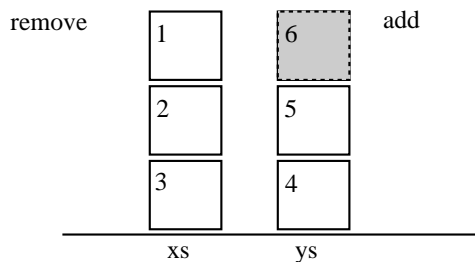


FIGURE 2. Adding to the queue.

Let n denote the length of `remove`'s first argument and m denote the length of the second argument. If $n = 0$, then `remove` consumes m heap cells, otherwise `remove` does not consume at all.

Annotate the function type in the spirit of Physicist's point of view:

$$L_n(\text{Int}) \times L_m(\text{Int}), \Phi \longrightarrow \text{Int} \times L_{p_1}(\text{Int}) \times L_{p_2}(\text{Int}), \Phi'$$

where $p_1 = m - 1$ and $p_2 = 0$ if $n = 0$, and $p_1 = n - 1$ and $p_2 = m$ if $n > 0$. Φ denotes the potential of input data before the computations, and Φ' denotes the potential of the data after the computation. Clearly, $\Phi' = \Phi - m$ if $n = 0$, and $\Phi' = \Phi$ if $n > 0$. *The drawback of this presentation is that Φ' is defined piecewise.*

The typing corresponding to the banker's view is more elegant, since it allows to escape piecewise definitions for amortization:

$$L_n(\text{Int}, 0) \times L_m(\text{Int}, 1), 0 \longrightarrow \text{Int} \times L_{p_1}(\text{Int}, 0) \times L_{p_2}(\text{Int}, 1), 0$$

Indeed, if $n = 0$ then the potential of the second argument $1 \cdot m$ is spent by `reverse`, $p_2 = 0$ and the potential of the second list on the r.h.s. is $0 = 0 \cdot p_2$. If $n > 0$ then the potential of the second argument $1 \cdot m$ is not spent, and the second argument is intact, $p_2 = m$ and the potential of the second list on the r.h.s. is $1 \cdot p_2$. So, amortization keeps track on the resources that are left after computation and may be used afterwards. The effect of combination of amortization with types may be seen on composition of `remove` and `copy3`: $\text{Int} \times L_n(\text{Int}, 0) \times L_m(\text{Int}, 1), 0 \longrightarrow L_m(\text{Int}, 0), 0$ that returns a fresh copy of the third argument.

The naive worst-case analysis consists in summation of two worst-case heap consumption estimations: for `remove` it is m , and for `copy3` it is m . So, the naive worst-case for `copy3(remove(xs, ys))` is $2 \cdot m$.

The precise worst-case analysis means detailed abstract program analysis of the *entire composition* and leads to piece-wise definition of the consumption function, which is later simplified to a linear function $p(n, m) = m$:

n	m	remove consumes	p_2	copy3 consumes	copy3(remove(-)) consumes
0	m	m	0	$p_2 = 0$	$m + 0 = m$
> 0	m	0	m	$p_2 = m$	$0 + m = m$

The type of `copy3(remove(-))` is easily obtained by composition:

$$\text{Int} \times \text{L}_n(\text{Int}, 0) \times \text{L}_m(\text{Int}, 1), 0 \longrightarrow \text{L}_m(\text{Int}, 0), 0.$$

It means that the composition consumes $1 \cdot m$ heap units. Some program analysis has been done here as well but only on the level of `remove`, to obtain its type. This is done once and forever and the type is applicable for any other composition.

5.2.2 Nonlinear bounds

In this subsection we consider an example to illustrate which extensions we should be able to cover.

Consider the program that given two lists of strings, of length n and m respectively creates the initial $n \times m$ table of pairs of integer numbers, filled with $(0, 0)$. This program is used for creating the initial table for a tournament, like a round in a soccer championship.

We start with a comment on how the initial table is used. During a round, each team plays two games – at home and as a guest. Let, for instance, “Ajax” which is number 1 in the list (in the alphabetical order), plays in Amsterdam with “Feyenoord”, number 3. The result is $2 - 1$. One places $(2, 1)$ in the position $(1, 3)$ in the table. Let “Feyenoord” play in Rotterdam with “Ajax”, $1 - 1$. One places $(1, 1)$ in the position $(3, 1)$. At the end of the round the table, except the diagonal, is filled with the results.

We need the auxiliary function:

```
init_row :: [String] → [(Int, Int)]
init_row [] = []
init_row (h:t) = (0, 0) : init_row t
```

Its annotated type is $\text{L}_n(\text{String}, 2), 0 \rightarrow \text{L}_n(\text{Int} \times \text{Int}, 0), 0$. Note, that here pairs of integers are allocated in the heap, and we assume that a pair allocate one heap unit, as well as a cons-cell.

The main “working” function is:

```
init_table: [(Int, Int)] → [String] → [[(Int, Int)]]
init_table row [] = []
init_table row (h:t) = copy row : init_table t
```

with type $\text{L}_n(\text{Int} \times \text{Int}, 2m) \times \text{L}_m(\text{String}, 0), 0 \rightarrow \text{L}_m(\text{L}_n(\text{Int} \times \text{Int}, 0), 0), 0$.

The function that creates the initial tournament table

```
init_tour :: [String] → [[(Int, Int)]]
init_tour teams = init_table (init_row teams) teams
```

has the annotated type $\text{L}_n(\text{String}, 2n + 2), 0 \rightarrow \text{L}_n(\text{L}_n(\text{Int} \times \text{Int}, 0), 0), 0$.

5.3 Experimental Type System

We start with a type system for a first-order call-by-value functional language over integers and polymorphic lists. So far, we consider *shapely* programs, that is programs for which the size of the output (polynomially) depends on the sizes of input lists.

5.3.1 Language and Types

The *abstract* syntax of the language is defined by the following grammar, where t ranges over integer constants, x and y denote zero-order program variables, and f denotes a function name:

$$\begin{aligned} \text{Basic } b & ::= c \mid \text{nil} \mid \text{cons}(x, y) \mid f(x_1, \dots, x_n) \\ \text{Expr } e & ::= \text{letfun } f(x_1, \dots, x_n) = e_1 \text{ in } e_2 \\ & \quad \mid b \mid \text{let } x = b \text{ in } e \mid \text{if } x \text{ then } e_1 \text{ else } e_2 \\ & \quad \mid \text{match } x \text{ with } \mid \text{nil} \Rightarrow e_1 \mid \text{cons}(x_{\text{hd}}, x_{\text{tl}}) \Rightarrow e_2 \end{aligned}$$

We have been studying a type and effect system in which types are annotated with size expressions (lowercase indices) and *credit functions*.

Size expressions that annotate types (see below) are polynomials representing lengths of finite lists and arithmetic operations over these lengths:

$$\text{SizeExpr } p ::= \mathbb{N} \mid n \mid p + p \mid p - p \mid p * p$$

where n , possibly decorated, denotes a size variable, which ranges over integer numbers. Semantics for lists with negative sizes is not defined: these lists are ill-formed.

In the simplest case, the intuition behind a credit function $k : \mathbb{N} \rightarrow \mathcal{R}^+$ is that $k(i)$ is the credit, that is, an amount of the free heap units, assigned to the i -th cons-cell of a given list. Note that we count cons-cells from nil, that is the head of a list of length n . Fractional credits are used to achieve more flexibility in defining distribution of extra heap cells across an overall data structure.

As we have noticed in 5.1 credits may depend not only on the position of a cons-cell, but also on other parameters, like the length of the outer list or the sizes of “neighboring” lists. In general a credit function is of type $\mathbb{N} \times \dots \times \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathcal{R}^+)$. The symbol k denotes a parametric or non-parametric credit function.

Zero-order types are assigned to program values, which are integers and annotated finite lists:

$$\text{Types } \tau ::= \text{Int} \mid \alpha \mid L_p(\tau, k) \quad \alpha \in \text{TypeVar}$$

where α , possibly decorated, is a type variable. For now, lists must have size expressions at every position in the type. Hence, they represent matrix-like structures.

First-order types are assigned to shapely functions over values of a zero-order type. Let τ° denote a zero-order type where all the size annotations are size variables. First-order types are defined by:

$$\tau^f ::= \tau_1^\circ \times \dots \times \tau_l^\circ, K \rightarrow \tau_{l+1}, K'$$

where the free size variables of the annotations of τ_{l+1} are the size variables of the input type, and K, K' are non-negative rational constants.

5.3.2 Typing rules

Consider a type system that generalises the type system of Hofmann and Jost [10].

A typing judgment is a relation of the form $D; \Gamma; K \vdash_{\Sigma} e : \tau; K'$, where D is a set of Diophantine equations and disequations, which is used to keep track of the size information. The signature Σ contains the type assumptions for the functions that are going to be checked.

In the typing rules, $D \vdash p = p'$ means that $p = p'$ is derivable from D in first-order logic. $D \vdash \tau = \tau'$ is a shorthand that means that τ and τ' have the same underlying type and equality of their credit and size annotations is derivable. Consider some of the typing rules that define a typing judgment relation formally.

$$\frac{K \geq K' + 1 + k(p' + 1) \quad D \vdash p = p' + 1}{D; \Gamma, hd : \tau, k, tl : L_{p'}(\tau, k); K \vdash_{\Sigma} \text{cons}(hd, tl) : L_p(\tau, k); K'} \text{CONS}$$

The non-destructive pattern-matching rule takes into account that the list and its tail are shared and, therefore, they share the potential. In the simplified version below all, but the head-cell's, potential is transferred to the tail. The head-cell's credit is "opened" for usage:

$$\frac{p = 0, D; \Gamma, x : L_p(\tau', k); K \vdash_{\Sigma} e_{\text{nil}} : \tau; K' \quad D; \Gamma, hd : \tau', x : L_p(\tau', 0), tl : L_{p-1}(\tau', k); K + k(p) \vdash_{\Sigma} e_{\text{cons}} : \tau; K'}{D; \Gamma, x : L_p(\tau', k); K \vdash_{\Sigma} \text{match } x \text{ with } \begin{array}{l} \text{nil} \Rightarrow e_{\text{nil}} \\ \text{cons}(hd, tl) \Rightarrow e_{\text{cons}} \end{array} : \tau; K'} \text{MATCH}$$

Function application rule for a function f may be viewed as the generalisation of the CONS-rule with f instead of cons , the function's arguments instead of hd, tl . Note that in the precondition there must be the information $\Sigma(f)$ about the type of the function. In this way one achieves the finiteness of the derivation tree if the function is recursive. The information may be not complete, that is the type has unknown parameters in annotations. Type inference for the annotated types consists in finding these parameters.

To deal with inter-structural exchange of resources, one needs rules like

$$\frac{D \vdash K \geq \sum_{i=1}^p k'(i) \quad D; \Gamma, x : L_p(\tau, k); K \vdash_{\Sigma} e : \tau'; K'}{D; \Gamma, x : L_p(\tau, k + k'); K - \sum_{i=1}^p k'(i) \vdash_{\Sigma} e : \tau'; K'} \text{SHUFFLEIN}$$

This rule is non-syntax driven and increases complexity of type-checking. We plan to establish conditions that define how such inference rules must be applied.

5.4 Ongoing Research: Sized Types

Whilst exploring possible research directions, it became clear that an important aspect of any advanced amortized analysis is static derivation of the sizes of data structures. More specifically, the relation between the sizes of the argument and the size of the result of a function has to be known. The size of a data structure, for now, is the number of nodes it consists of.

We have studied the pure size-aware type system, which is obtained from the presented one by erasing credit functions and resource constants K . We have shown, that in general type-checking for this system is undecidable. Indeed, consider the matching rule. In its nil-branch it contains the Diophantine equation that reflects the fact that the list is empty. At the end of type checking one may need to determine if a branch is going to be entered or not. To check this the Diophantine equations have to be solved. So, type-checking is reducible to Hilbert's tenth problem and, thus, undecidable in general [17].

However, we have formulated the syntactical restriction that makes the solving of the equations trivially decidable: let-expressions are not allowed to contain pattern matching as a sub-expression.

It is not known whether type inference is decidable for the size-aware system. It amounts to solving systems of polynomial equations that may be non-linear [17].

So, to infer types we propose an altogether different approach [17]. The idea is simple. First, note that the size dependencies are exact and polynomial. From interpolation theory it is known that any polynomial of finite degree is determined by a finite number of data points. Hence, if a degree of the polynomial is assumed and enough pairs of input-output sizes are measured by running the program on test-data a hypothesis for the size equations can be determined. If size dependency has indeed the degree assumed, checking the hypothesis in the type system gives a positive result. By repeating the process for increasing degrees, any polynomial dependency will eventually be found, if it exists. In case it does not exist, or the program does not terminate, the procedure does not terminate. So, we named this *weak type inference*.

A further development of this system would, amongst others, include an adaptation to upper and lower bounds and support for higher-order functions.

6 RELATED WORK

The presented combination of amortization and types generalizes the approach from [10] which forms the foundational basis of the EU funded project *Mobile Resource Guarantees*, [15]. The project *has developed the infrastructure needed to endow mobile code with independently verifiable certificates describing its resource behavior (space, time)*. Its functional language *Camelot* is an implementation of the underlying language from [10]. Numerical annotations are not the part of its typing, they are computed later, on top of a standard type-inference procedure. A Camelot program is compiled into *Grail*, which a structured version

of the Java Byte Code. The high-level type system is mirrored in a specialized heap-aware Hoare logic for the byte-code.

The **AHA** project can be considered as one of the successors of MRG:

- it is aimed to extend the high-level type system of MRG to type-systems for non-linear heap consumption bounds,
- applications of the methodology to object-oriented programming will involve MRG experience with the byte-code: one considers imperative object-oriented structures that have counterparts in functional programming,
- soundness of the type systems, type-checking and inferences procedures, object-oriented extensions will be implemented in an environment similar to the program-logic environment designed for MRG.

MRG has a few other successors. First, one should mention a large consortium *Mobius* [4], which, as well as MRG, runs under EU framework *Global Computing*. Its aim is to design a byte-code verification tool that allows to employ a large variety of formal methods. The byte-code properties of interest include information flows and resource consumption.

The aims of the *EmBounded* project [9] are *to identify, to quantify and to certify resource-bounded code in Hume, a domain-specific high-level programming language for real-time embedded systems*. The project develops static analyses for time and space consumption, involving size and effect type systems. The foundational results have realistic applications for embedded systems.

The *ReQueSt* project [2], funded by UK government's agency EPSRC, aims to prevent the situations when, for instance, an expensive user's request fails due to the lack of memory.

7 CONCLUSION

The **AHA** project aims to improve the state of the art in inferring upper bounds on heap-space usage. Improvements lie in the complexity of the bounds and the applicability to widely used languages. Ultimately, we want to implement both a type checking and a type inference system for heap space usage bounds of lazy functional and imperative programs.

REFERENCES

- [1] D. Aspinall and M. Hofmann. Another type system for in-place update. In *ESOP'2002*, volume 2305 of *LNCS*, pages 36 – 52, 2002.
- [2] R. Atkey and K. MacKenzie. Request: Resource quantification for e-science technologies. In *International Workshop on Proof-Carrying Code*, 2006.
- [3] E. Barendsen and S. Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6:579–612, 1996.

- [4] G. Barthe, L. Beringer, P. Crégut, B. Grégoire, M. Hofmann, P. Müller, E. Poll, G. Puebla, I. Stark, and E. Vétillard. Mobius: Mobility, ubiquity, security. objectives and progress report. In *TGC 2006: Proceedings of the second symposium on Trustworthy Global Computing*, LNCS. Springer-Verlag, 2006. To appear.
- [5] J. v. d. Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. In D. Bert and C. Choppy, editors, *Recent Trends in Algebraic Development Techniques (WADT'99)*, volume 1827 of LNCS. Springer, 2000.
- [6] V. Braberman, D. Garbervetsky, and S. Yovine. Synthesizing parametric specifications of dynamic memory utilization in object-oriented programs. In *FTJJP 2005: Formal Techniques for Java-like Programs. Glasgow, Scotland*.
- [7] W.-N. Chin, H. H. Nguen, S. Qin, and M. Rinard. Predictable memory usage for object-oriented programs. Technical report, National University of Singapore, Massachusetts Institute of Technology, 2004.
- [8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press.
- [9] K. Hammond, R. Dyckhoff, C. Ferdinand, R. Heckmann, M. Hofmann, S. Jost, H.-W. Loidl, G. Michaelson, R. Pointon, N. Scaife, J. Sérot, and A. Wallace. Project start paper: The embounded project. In M. van Eekelen, editor, *Trends in Functional Programming*, volume 6. Intellect.
- [10] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. *SIGPLAN Not.*, 38(1):185–197, 2003.
- [11] B. Jacobs and E. Poll. Java program verification at Nijmegen: Developments and perspective. In *International Symposium on Software Security (ISSS'2003), Tokyo, Japan*, LNCS. Springer, 2004.
- [12] M. Konechny. Typing with conditions and guarantees for functional in-place update. In *TYPES 2002 Workshop, Nijmegen*, volume 2646 of LNCS.
- [13] H.-W. Loidl and K. MacKenzie. *A Gentle Introduction to Camelot*, September 2004. <http://groups.inf.ed.ac.uk/mrg/camelot/Gentle-Camelot/>.
- [14] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
- [15] D. Sanella, M. Hofmann, D. Aspinall, S. Gilmore, I. Stark, L. Beringer, H.-W. Loidl, K. MacKenzie, A. Momigliano, and O. Shkaravska. Project evaluation paper: Mobile resource guarantees. In M. van Eekelen, editor, *Trends in Functional Programming*, volume 6. Intellect.
- [16] B. Schoenmakers. *Data Structures and Amortized Complexity in a Functional Setting*. PhD thesis, Eindhoven University of Technology, September 1992.
- [17] O. Shkaravska, R. van Kesteren, and M. van Eekelen. Polynomial size analysis of first-order functions. Technical Report ICIS-R07004, Radboud University Nijmegen, 2007. www.cs.ru.nl/icis.