Chapter 1

AHA: *A*mortized *H*eap Space Usage *A*nalysis – *Project Paper* –

Marko van Eekelen, Olha Shkaravska, Ron van Kesteren,
Bart Jacobs, Erik Poll, and Sjaak Smetsers[1]

Abstract: This paper introduces **AHA**, an NWO-funded[2] 344K Euro project involving research into an amortized analysis of heap-space usage by functional and imperative programs. Amortized analysis is a promising technique that can improve on simply summing worst case bounds. The project seeks to combine this technique with type theory in order to obtain non-linear bounds on heap-space usage for functional languages and to adapt the results for the lazy functional case and for imperative languages.

1.1 INTRODUCTION

Estimating heap consumption is an active research area as it becomes more and more an issue in many applications. This project seems to be part of an upcoming trend since a growing number of projects are addressing this as a research topic (see section 1.6 on related work). Examples of possible application areas

[1] All authors are members of the Security of Systems Department, Institute for Computing and Information Sciences, Radboud University Nijmegen, Toernooiveld 1, Nijmegen, 6525 ED, The Netherlands; Project leader contact: `marko@cs.ru.nl`.

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

include programming for small devices, e.g. smart cards, mobile phones, embedded systems and distributed computing, e.g GRID. It is important to give as accurate bounds for heap consumption as possible to avoid unnecessarily expensive and even unpractical estimates for small devices and high integrity real-time applications.

A promising technique to obtain accurate bounds of resource consumption and gain is amortized analysis. An amortized estimate of a resource does not target a single operation but a sequence of operations. One assigns to an operation some amortized cost that may be higher or lower than its actual cost. For the sequence considered it is important that its overall amortized cost covers its overall actual cost. An amortized cost of the sequence lies between its actual cost and the simple multiplication of the worst-case of one operation by the length of the sequence. An amortized cost of the sequence is in many cases easier to compute than its actual cost and it is obviously better than the worst-case estimate.

Combining amortization with type theory allows the inference of linear heap consumption bounds for functional programs with explicit memory deallocation [10]. The **AHA** project aims to adapt this method for *non-linear* bounds within (lazy) functional programs and transfer the results to the object-oriented programming. In this way the project both enhances fundamental theory and practical impact.

1.1.1 Relevance

Accurate estimates of heap space consumption are directly relevant for robustness, execution time and safety of programs. For instance, memory exhaustion may cause abrupt termination of an application or invoke garbage collection. In the latter case, heap management 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 optimizations 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 for programs downloaded from external sources. It is one of the most important properties that 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.

1.1.2 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:

– How can the existing type-based linear heap consumption analysis of functional programs [10] be improved such that a wider class of resource usage bounds can be guaranteed? The question is how complex the type-checking and inference procedures may be. In particular, which arithmetic and constraint solvers will be needed for which classes of function definitions?

– 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 adapt 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 of Java programs. The plan is to start with a functional subset of Java that encompasses classes admitting algebraic data type operations, like constructors and get-field methods (corresponding to nondestructive pattern matching) and generalize from there.

1.1.3 Outline of the paper

Amortization for resource-aware program analysis is explained in section 1.2. In section 1.3 we give an overview of the existing amortization-related type system which is used to infer linear heap-consumption bounds for first-order functional programs. The research questions from section 1.1.2, which concern generalizations of the type system, are to be answered according to the project plan from section 1.4. The motivation and more detailed generalization of the type system for non-linear heap bounds for strict languages and related results on size inference are presented in section 1.5. We finish the paper with the overview of related projects devoted to quantitative resource analysis and define the place of **AHA** amongst this variety in section 1.6.

1.2 INTRODUCTION TO AMORTIZATION

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 of programs. “Payments” in a program are done by its operations or by the data structures that participate in the computation, see [15]. 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.

1.2.1 Amortization of resources in program 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 number 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
```

To construct a stack one needs a function `push`:

```
push :: Int → Stack Int → Stack Int
push x s = x:s
```

If the actual costs of each function call (such as `multipop` and `push`) is 1 time unit, then the actual cost of the program `multipop k` S is $min(s,k)+1$ time units, where s is the size of the stack S.

Assigning amortized costs for `multipop` and `push` one may think in the following way. Each operation `push` has actual cost 1, but it "takes care" of 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 corresponding part of a call of `multipop`. Thus, the complete cost of `multipop k S` is paid while constructing the input S using `push`. *After construction of the stack S*, the amortized cost for `multipop` is just 1 for the call of `multipop k []`. Hence, the amortized cost of the construction of S followed by `multipop` is $2s+1$, which is an upper bound for the actual cost being $s+min(s,k)+1$.

The correctness of an amortized analysis for a sequence of n operations is defined by $\Sigma_{i=1}^{j} a_i \geq \Sigma_{i=1}^{j} t_i$, for all $j \leq n$, a_i is the amortized cost of the ith 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.

1.2.2 Views to Amortization

A general understanding of amortization [17] is based on a graph representation 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 replacing *if-then-else* by nondeterministic choice.

In the *physicist's view* of amortization one assigns to any state s a real number $\Phi(s)$ called the *potential* of the state s. We consider only non-negative potentials. Negative potentials can never be introduced since the typing rules insist that the potential is kept non-negative (see section 1.5.3). The first intuition behind the potential function is that it reflects the number 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 view. The opposite transformation is more complicated. The banker's approach is more general than the physicist's one, because one considers particular paths instead of their initial and end points. However, it has been shown [17] 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), and $a'(s_1, s_2) \leq a(s_1, s_2)$ for any edge $e(s_1, s_2)$. Thus, without loss of generality one can consider *conservative* amortized analysis only.

1.2.3 Amortization for Heap Consumption Gives Size of Live Data

Now we interpret amortization for heap consumption analysis. A potential of a structure is a number of free heap units associated with this structure. An initial potential is the potential of an input structure before the program runs. Any data structure, which exists during the computation of a function, may be constructed either from heap units taken from the initially allocated units (defined by the initial potential function) or taken from reused heap cells (for a language with destructive pattern matching).

If heap management is performed via maintaining a free list, then the heap layouts before and after the computations are presented by the scheme in Figure 1.1. One can view maintaining a free list as an ideal garbage collector: once a location is destructed 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 the size of input data define an upper bound on the size of the live data at any moment of computation. In

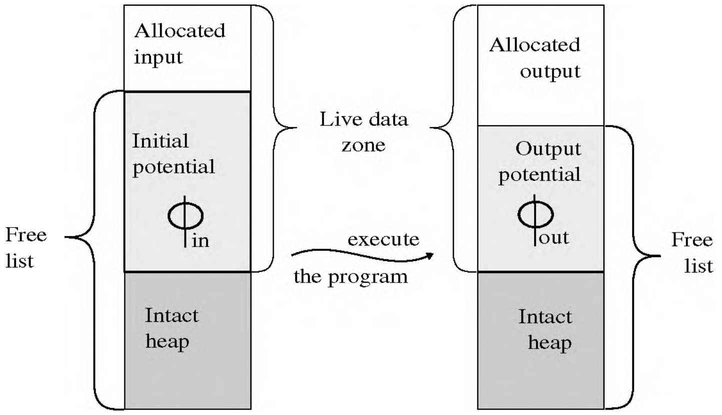

FIGURE 1.1. Heap layouts before and after the computations

general, we have the following dependency:

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

1.3 STATE OF THE ART: A TYPE SYSTEM FOR LINEAR BOUNDS

One can implement a heap-aware amortized analysis via an annotated type system. In this section we consider an annotated type system introduced by Hofmann and Jost [10] for linear bounds on heap consumption. Given a first-order function definition this system allows us to infer an upper bound (if it exists and is linear) on the number of freshly-allocated heap units.

The operations that affect 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 function that creates a fresh copy of a list of integers

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

has the annotated signature $[\mathtt{Int}]_1\ {}_0\!\rightarrow_0\ [\mathtt{Int}]_0$ (we adapted the notation of [10]). It reflects the fact that for each element of an input list 1 extra heap unit (credit) must be supplied to fix the space for its copy. Furthermore, it indicates via a 0-annotated arrow that it is not necessary to add extra heap cells for evaluating the function. Also, no cells at all will be released: nor a number of cells depending on the size of the result (since a 0 is assigned to the result), nor independent of that (since a 0 is assigned to the function arrow).

In general, the heap consumption by a function f with the credit-annotated type $[\mathtt{Int}]_k\ {}_{k_0}\!\rightarrow_{k_0'}\ [\mathtt{Int}]_{k'}$ 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' the sizes of the input and output lists, respectively. The potential of a list $[\texttt{Int}]_k$ of length n is $k \cdot n$. In fact, the type system above infers two (linear) potentials of a given function: the potential of an input and the potential of an output. The potential of the input may be discharged during evaluation of the program expression and the potential of the output may be used in further computations. Non-zero arrow annotations typically appear due to destructive pattern-matching.

It is possible to extend this approach for non-linear bounds. One of the aims of the **AHA** project is to study such extensions.

1.4 AHA PROJECT PLAN

To answer the three research questions posed in section 1.1.2, 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 an imperative object-oriented setting.

In view of the breadth of the proposed research, which looks both at functional and imperative languages, the project will use the funding for two positions, a three year post-doc and a four year PhD student. Cooperating with the post-doc the PhD student will not only study the more fundamental issues but the PhD student will also be responsible for creating prototypes demonstrating the effectiveness of the developed analyses. Master students will be actively encouraged to participate in creating these demonstrators.

Ultimately, we want to implement the type systems for heap space usage to obtain prototypes that can check whether a given (functional or imperative) program, augmented with resource-aware type annotations, meets a given bound on heap space usage. Ideally, we will be able to infer such bounds in many cases.

1.4.1 Amortized analysis with non-linear bounds

There are many interesting examples that require non-linear heap space, for instance matrix multiplication and the Cartesian product. Also, e.g. 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. The sports competition has $n \cdot n - n$ matches, where n is the number of teams. So, the program will require at least $n \cdot n - n$ heap space.

Arguments and results of functions are represented as (intermediate) structures in the heap. Sizes of results depend generally on the sizes of the arguments. For example, the number of matches (the size of the result) in an implemented sports competition[3] depends on the number of teams (the size of the argument).

[3] Using amortization in section 1.5 it is shown that for a specific sports competition

So, deriving such size dependencies is an important first step before computing amortized bounds that take temporary structures into account.

Methodology. To begin with, we tackle the derivation of size relations separately from heap-space usage to keep both systems as simple as possible. The results from the derivation of the size relations are input for the amortized analysis. The amortized analysis will be an extension of the existing linear analysis in [10].

1.4.2 Amortized Heap Analysis of a Lazy Language

Applying a strict-semantic-based type system to a lazy evaluation strategy may lead to significant mis-evaluations of heap consumption. Indeed, one may count heap cells for a structure that is not actually allocated or allocated in a "zipped" form, or one counts a heap consumed/released by a function that is not called.

Consider, for instance, a lazy list of integers `lazy_list n` containing integers from n to 1. An element of this list is a record $(n,\ r)$ which consist of the integer field for n, and a reference field with r, where r is the address a of the closure computing tail, if $n > 0$, and $r = \mathsf{nil}$ otherwise. The closure is the function `tail` $t = \lambda$ `i.` **if** `i>1` **then** `(i-1, a)` **else** `(0, nil)`. So, the size of this structure is *constant: the size of integer + the size of a pointer + the size of the closure*, whereas the size of the corresponding strict list is *(the size of integer + the size of a pointer)*$\times n$. The lazy list is unfolded once it is needed (and may be memoized after that).

One of the ways to provide a transition from a strict semantics to a lazy one is to augment a strict language with an explicit suspension constructor `S` and `force` operator, as it is done in [15]: `datatype` α `susp` $=$ `S` α. Then for the example above one has:

```
val l=S(lazy_list 1000) (*the constant heap is allocated*)
...
val x=force l (*proportional to 1000 cells are allocated*)
```

One may consider typing rules for explicit suspensions and forces, like

$$\frac{\Gamma \vdash_{\Sigma} e : \tau}{\Gamma \vdash_{\Sigma} \mathit{force}(Se) : \tau}\ \textsc{Force}$$

Amortized time analysis for call-by-need[4] languages is considered in [15]. Instead of credits it uses *debts* to cover costs of 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

programme implementation that it actually requires $n \cdot (3 \cdot n + 3)$ heap space.

[4] Following [15] 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.

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 analyzed 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 functional language with polymorphism and algebraic data types. To enable analysis of heap usage, in Camelot one can syntactically make the distinction between destructive and non-destructive pattern matching, 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 change the strict semantics into a mixed lazy/strict semantics and require that suspensions and forces are explicit in our input language. This corresponds to assuming that compiler optimizations and program transformations have been performed before the analysis starts. We will investigate the consequences for the operational semantics and for 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 1.5.3 and in the SHARE-rule in [10].

1.4.3 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 formalization of the heap. This will first require accurate 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 of yet without any clear semantics. We want to provide a rigorous semantics for these properties about heap space usage and then develop an associated program logic for proving such properties.

Methodology. We will start to adjust the analysis of Section 1.5.3 by applying it to classes that admit a functional *algebraic data-type (ADT) interface*. These classes posses "basic" methods that have counterparts in functional programming. Constructors correspond to functional constructors and get-field methods ("observers") correspond to non-destructive pattern matching. Heap-aware program-logic rules are to be defined for these basic methods and the language constructs such as `if`-branching, sequencing and `while`-repetition (a-la "recursive function call"). Then, a field assignment, for example, may be presented as a composition of the destructive match and a constructor.

Next, research will be done to alleviate the restrictions. For that purpose, we will investigate the possibility of introducing amortized variants of existing specific analyzes (such as the non-recursive [6] and the symbolic [7] which treats aliasing). One of the main problems for heap space analysis is 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.

1.5 FIRST STEPS: NON-LINEAR BOUNDS AND SIZED TYPES

In this section we show why a more general treatment of credits (generalizing from constants to functions) is required for non-linear heap consumption analysis. We give examples, one of which illustrates the advantage of combination amortization with types and the other one is about non-linear heap consumption. Further, we present an experimental type system that combines sizes and amortization. Sizes are needed to determine generalized credits and may be considered independently. Finally, we give a summary of the first results of the project dealing with strictly sized types.

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

It is convenient to measure the potentials of data structures in terms of their sizes. For instance, the potential of a list of length n 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 function of type $([\mathtt{String32}]^n, [\mathtt{String32}]^m) \to [(\mathtt{Int}, \mathtt{Int})]^{n \cdot m}$ that creates an initial table from input lists of length n and m of strings of fixed length (say 32). We use superscripts for sizes and subscripts for credits. It is natural to assume that for the input of type $([\mathtt{String32}]^n, [\mathtt{String32}]^m)$ the potential $\Phi(n, m)$ depends on n and m.

The system assigns a credit to each constructor of a data structure. For instance, in [10] each constructor of a list of type $[\alpha]_k$ 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 function the annotated type of its input may be $([\mathtt{String32}]^n_k, [\mathtt{String32}]^m_0)$, where $k(position, n, m) = m$.

In the linear heap-consumption analysis of [10] these dependencies are not taken into account. This makes the analysis very simple, because it reduces to solving a linear programming task. 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

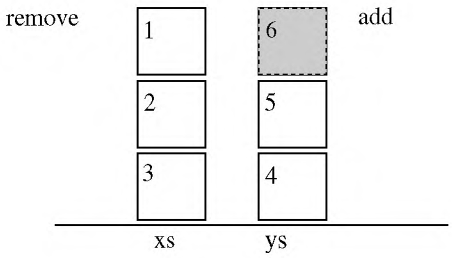

FIGURE 1.2. Adding to the queue.

checking and inference. We will study classes of function definitions for which type checking and inference of non-linear bounds are decidable.

1.5.2 Examples: going on with amortization-and-types

The linear heap-consumption analysis shows that amortization and types can be combined naturally. In this section we consider 2 examples. One example illustrates the advantages of this combination. The other one motivates the study of annotated types for non-linear heap consumption.

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 relatively 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 $xs + +(reverse\ ys)$. The head of the list xs is the first element of the queue, and the head of ys is the last element 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 1.2 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.)

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 cells at all.

The annotated type for `remove` looks as follows:

$(\texttt{[Int]}^{n}_{0},\ \texttt{[Int]}^{m}_{1})\ {}_{0}{\rightarrow}_{0}\ (\texttt{Int},\ (\texttt{[Int]}^{p_1}_{0},\ \texttt{[Int]}^{p_2}_{1}))$

where p_1 and p_2 are defined piece-wise: if $n = 0$ then $p_1 = m - 1$, $p_2 = 0$; otherwise $p_1 = n - 1$, $p_2 = m$. As for the credits: if $n = 0$ then the potential of the second argument $1 \cdot m$ is spent by `reverse` and the potential of the second list on the r.h.s. is $0 = 1 \cdot p_2$. If $n > 0$ then the second argument and its potential $1 \cdot m$ are intact, and the potential of the second list on the r.h.s. is $m = 1 \cdot p_2$.

So, amortization keeps track of the resources that are left after computation and that may be used afterwards. The effect of combining amortization with types may be seen at the composition of `remove` with copy3 that returns a fresh copy of the third argument. The type of copy3 is $(\texttt{Int},\ \texttt{[Int]}^{n}_{0},\ \texttt{[Int]}^{m}_{1})\ {}_{0}{\rightarrow}_{0}\ \texttt{[Int]}^{m}_{0}$.

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 requires detailed abstract program analysis of the *entire composition* and leads to a piecewise 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 $(\texttt{Int},\ \texttt{[Int]}^{n}_{0},\ \texttt{[Int]}^{m}_{1})\ {}_{0}{\rightarrow}_{0}\ \texttt{[Int]}^{m}_{0}$ of copy3(remove(−)) is easily obtained by composition. It means that the composition consumes $1 \cdot m$ heap units. Type derivation for `remove` is done once and forever and the type is applicable for any other composition yielding a modular amortized analysis.

Example of the use of nonlinear bounds

We illustrate the kind of types we plan to derive by the following small example.

Consider the function definition 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 $(-1, -1)$. This function is used for creating the initial table for a tournament, like a round in a soccer championship.

The initial table is used as follows. During a round, each team plays two games – at home and as a guest. Let, for instance, "PSV Eindhoven" be number 1 in the list and play in Eindhoven with "AZ Alkmaar" being number 3 withe result $2 - 1$. Then one places $(2, 1)$ in the position $(1, 3)$ in the table. This may be done in non-destructive or destructive (in-place update) way. At the end of the round the table, except the diagonal, is filled with the results.

We need an auxiliary initializing function `init_row`. Note, that a node of a list of pairs of integers allocates 3 heap units: one per each integer and one for the reference to the next element. The main "working" function is `init_table`. Finally, the function `init_round` creates the initial tournament table.

$$\texttt{init\_row ::}\ [\texttt{String32}]^{n}_{3}\ {}_{0}{\rightarrow}_{0}\ [(\texttt{Int},\texttt{Int})]^{n}_{0}$$

```
init_row []     = []
init_row (h:t) = (-1, -1) : init_row t
```

$$\texttt{init\_table ::}\ ([(\texttt{Int, Int})]^{n}_{0},\ [\texttt{String32}]^{m}_{3n})\ {}_{0}{\rightarrow}_{0}\ [[(\texttt{Int, Int})]^{n}_{0}]^{m}_{0}$$

```
init_table  row []     = []
init_table  row (h:t) = copy row : init_table t
```

$$\texttt{init\_round ::}\ [\texttt{String32}]^{n}_{3n+3}\ {}_{0}{\rightarrow}_{0}\ [[(\texttt{Int, Int})]^{n}_{0}]^{n}_{0}$$

```
init_round teams = init_table (init_row teams) teams
```

The size dependency (superscripts in types) indicates that the result is of size $n \cdot n$. One might have expected size $n \cdot (n-1)$ but the program does not remove the superfluous diagonal. Taking into account credits (subscripts in types) for the intermediate structure produced by `init_row` we derived that te heap consumption of `init_round` is: $n \cdot (3n+3)$.

1.5.3 Experimental Type System

We start with a type system for a first-order call-by-value functional language over integers and polymorphic lists. First we consider only *shapely* function definitions, that is definitions for which the size of the output (polynomially) depends on the sizes of input lists. Below, we sketch the basic ideas of such a type system.

Language and Types

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

$$\begin{array}{lcll}
\textit{Basic} & b & ::= & c \mid \mathsf{nil} \mid \mathsf{cons}(x,y) \mid f(x_1,\ldots,x_n) \\
\textit{Expr} & e & ::= & \mathsf{letfun}\ f(x_1,\ldots,x_n) = e_1\ \mathsf{in}\ e_2 \\
 & & & \mid b \mid \mathsf{let}\ x = b\ \mathsf{in}\ e \mid \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\
 & & & \mid \mathsf{match}\ x\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow e_1 \mid \mathsf{cons}(x_{\texttt{hd}},x_{\texttt{tl}}) \Rightarrow e_2
\end{array}$$

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

Size expressions that annotate types are polynomials representing lengths of finite lists and arithmetic operations over these lengths (at a later stage this may be extended to piecewise-defined polynomial functions):

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

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 $k: \mathbb{N} \to \mathcal{R}^+$ is that $k(i)$ is the credit, that is, a number of 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 has credit $k(n)$. Fractional credits may be used to achieve more flexibility in defining distribution of extra heap cells across an overall data structure.

As we have noticed in 1.5.1, 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 $\mathbb{N} \times \ldots \times \mathbb{N} \to (\mathbb{N} \to \mathcal{R}^+)$. However here, for the sake of simplicity, we consider typing rules with the simplest credits of type $k: \mathbb{N} \to \mathcal{R}^+$, and k denotes a parametric credit function.

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

$$\mathit{Types} \quad \tau \quad ::= \quad \mathtt{Int} \mid \alpha \mid [\tau]_k^p \qquad\qquad \alpha \in \mathit{TypeVar}$$

where α is a type variable. For now, lists represent matrix-like structures and must have size expressions at every position in the "nested-list" type.

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

$$\begin{array}{lll} \mathit{FTypes} \quad \tau^f & ::= & \tau_1^\circ \times \ldots \times \tau_n^\circ \to \tau_{n+1} \\ & & \text{such that } \mathit{FVS}(\tau_{n+1}) \subseteq \mathit{FVS}(\tau_1^\circ) \cup \cdots \cup \mathit{FVS}(\tau_n^\circ) \end{array}$$

where $\mathit{FVS}(\tau)$ denotes free size variables of a type τ and K, K' are non-negative rational constants. Here, we abstracted from a few technical details concerning the equivalence of empty lists, like $[[\alpha]^p]^0 \equiv [[\alpha]^q]^0$. Full definitions are in [18].

Typing rules

Consider a few typing rules that generalize the type system of Hofmann and Jost [10] using credit functions in stead of credit constants.

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* (i.e. equations with integer coefficients with variables varying over natural numbers) used to keep track of the size information. The signature Σ contains the type assumptions for the functions 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.

The type system allows non-negative potentials only. The credit functions k and the constants K and K' are always non-negative. Due to the side conditions (like $K \geq K' + 1 + k(p' + 1)$ below) the typing rules guarantee that potentials fully cover the cost of computation. For the rule below the side condition guarantees that there will be enough heap cells to evaluate CONS.

$$\frac{\begin{array}{l} K \geq K' + 1 + k(p'+1) \\ D \vdash p = p' + 1 \end{array}}{D;\ \Gamma,\ hd\!:\tau,\ tl\!:[\tau]_k^{p'};\ K \ \vdash_\Sigma \mathsf{cons}(hd, tl) : [\tau]_k^p;\ K'}\ \textsc{Cons}$$

The non-destructive pattern-matching rule takes into account that a 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:

$$\frac{\begin{array}{c} p = 0,\ D;\ \Gamma,\ x\!:[\tau']_k^p;\ K \ \vdash_\Sigma e_{\mathsf{nil}} : \tau;\ K' \\ D;\ \Gamma, hd\!:\tau',\ x\!:[\tau']_0^p,\ tl\!:[\tau']_k^{p-1};\ K + k(p) \ \vdash_\Sigma e_{\mathsf{cons}} : \tau;\ K' \end{array}}{D;\ \Gamma,\ x\!:[\tau']_k^p;\ K \ \vdash_\Sigma \mathsf{match}\ x\ \mathsf{with} \begin{array}[t]{l} \mid \mathsf{nil} \Rightarrow e_{\mathsf{nil}} \\ \mid \mathsf{cons}(hd, tl) \Rightarrow e_{\mathsf{cons}} \end{array} : \tau;\ K'}\ \textsc{Match}$$

The function application rule for a function `f` may be viewed as a generalization of the CONS-rule with `f` instead of cons and the function's arguments instead of *hd*, *tl*. Note that the precondition requires the information $\Sigma(\mathtt{f})$ 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 may have 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

$$\frac{\begin{array}{l} D \vdash K \geq \Sigma_{i=1}^p k'(i) \\ D;\ \Gamma, x\!:[\tau]_k^p;\ K \ \vdash_\Sigma e : \tau';\ K' \end{array}}{D;\ \Gamma, x\!:[\tau]_{k+k'}^p;\ K - \Sigma_{i=1}^p k'(i) \ \vdash_\Sigma e : \tau';\ K'}\ \textsc{ShuffleIn}$$

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.

1.5.4 First Results: 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.

As a first result we have designed a pure size-aware type system, which is obtained from the one presented in section 1.5.3 by erasing credit functions and resource constants [18]. This type system treats *non-monotonic* polynomial size dependencies. We have shown that, in general, type-checking for this system is undecidable. Indeed, consider the matching rule. Its nil-branch 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* (i.e. existence of an algorithm which given any Diophantine equation decides if it has roots or not). Hilbert's tenth problem is shown to be undecidable [14]. Thus, type checking is undecidable in general.

We have identified a syntactical restriction such that the equations to solve are 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. Technically, it amounts to solving systems of polynomial equations that may be non-linear. So, to infer types we propose an altogether different approach [18]. 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 function on test-data a hypothesis for the size equations can be determined. If size dependency has indeed the type assumed, checking the hypothesis in the type system gives a positive result. By repeating the process for increasing degrees, the sized type for a function definition will eventually be found, if the function is typable. In case it does not exist, or the function does not terminate, the procedure does not terminate. Thus, the sized-type inference problem is *semi-decidable* for terminating functions. Complete shape-checking and inference procedures, even for the expressions subject to the syntactic condition, cannot exist. So, the type system is incomplete, non-termination due to absence of a type means either that the function is not shapely, or shapely, but not typable. However, a large non-trivial class of shapely functions is typable in the system.

A further development of this system would, amongst others, include an adaptation to upper and lower bounds and support for other data structures.

1.6 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*, [16]. 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]. 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. Firstly, it is aimed to extend the high-level type system of MRG to type-systems for non-linear heap consumption bounds. Secondly, 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. Finally, 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 analyzes 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 abrupt lack-of-memory termination of an expensive user's request in a GRID application.

Together, these projects seem to constitute an upcoming resource consumption trend in functional programming research.

1.7 ACKNOWLEDGEMENTS

We would like to thank Martin Hofmann for the foundational results that inspired this project and the anonymous reviewers for their helpful comments and suggestions.

1.8 CONCLUSION

The **AHA** project aims to contribute to the analysis of the resource consumption by improving the state of the art in inferring upper bounds for 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. Aspinalll 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, 2007. To appear.

[5] J. van den 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. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5):31–58, June 2006.

[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 Cliff Steinet. *Introduction to algorithms.* MIT press, 2001.

[9] K. Hammond, R. Dyckhoff, Ch. 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 Marko van Eekelen, editor, *Trends in Functional Programming*, volume 6, pages 195–210. Intellect.

[10] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM Symposium on Principles of Programming Languages*, volume 38-1, pages 185–197. ACM Press, 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, pages 134–153. Springer, 2004.

[12] M. Konechny. Typing with conditions and guarantees for functional in-place update. In *TYPES 2002 Workshop, Nijmegen*, volume 2646 of *LNCS*, pages 182 – 199. Springer, 2003.

[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] Yu. Matiyasevich and J. P. Jones. Proof of recursive unsolvability of Hilbert's tenth problem. *American Mathematical Monthly*, 98(10):689–709, October 1991.

[15] Ch. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[16] 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 Marko van Eekelen, editor, *Trends in Functional Programming*, volume 6, pages 211–226. Intellect.

[17] B. Schoenmakers. *Data Structures and Amortized Complexity in a Functional Setting*. PhD thesis, Eindhoven University of Technology, September 1992.

[18] O. Shkaravska, R. van Kesteren, and M. van Eekelen. Polynomial size analysis for first-order functions. In S. Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications (TLCA'2007), Paris, France*, volume 4583 of *LNCS*, pages 351 – 366. Springer, 2007.