Static Determination of Quantitative Resource Usage for Higher-Order Programs

Steffen Jost  Kevin Hammond
University of St Andrews, St Andrews, UK
{jost,kh}@cs.st-andrews.ac.uk

Hans-Wolfgang Loidl*  Martin Hofmann
Ludwig-Maximilians University, Munich, Germany
{hwloidl,mhofmann}@tcs.ifi.lmu.de

Abstract
We describe a new automatic static analysis for determining upper-bound functions on the use of quantitative resources for strict, higher-order, polymorphic, recursive programs dealing with possibly-aliased data. Our analysis is a variant of Tarjan’s manual amortised cost analysis technique. We use a type-based approach, exploiting linearity to allow inference, and place a new emphasis on the number of references to a data object. The bounds we infer depend on the sizes of the various inputs to a program. They thus expose the impact of specific inputs on the overall cost behaviour.

The key novel aspect of our work is that it deals directly with polymorphic higher-order functions without requiring source-level transformations that could alter resource usage. We thus obtain safe and accurate compile-time bounds. Our work is generic in that it deals with a variety of quantitative resources. We illustrate our approach with reference to dynamic memory allocations/deallocations, stack usage, and worst-case execution time, using metrics taken from a real implementation on a simple micro-controller platform that is used in safety-critical automotive applications.

Categories and Subject Descriptors F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Program analysis

General Terms Languages, Reliability, Performance, Theory.

Keywords Functional Programming, Resource Analysis, Types.

1. Introduction
Automatically obtaining good quality information about resource usage (e.g. space/time behaviour) is important to a number of areas including real-time embedded systems, parallel systems, and safety-critical systems. While there has been significant work on automatic analyses for first-order programs, to date there has been correspondingly little work on analyses for higher-order programs. Developing such analyses is important both to enable the deployment of functional programming languages, and to assist the increasing number of conventional programming approaches that rely on higher-order information (e.g. aspect orientation).

* Current affiliation: Heriot-Watt University, Edinburgh. Part of this work was done while being employed by the University of St Andrews.

This paper introduces a new automatic static analysis for determining upper-bound functions on the resource usage of strict, higher-order, polymorphic, recursive functional programs. “Resource” may here refer to any quantifiable resource. In particular, we discuss and analyse worst-case execution time, stack-space usage, and heap-memory consumption. The bounds that we obtain are simple linear expressions that depend on the input sizes. They thus expose the impact of the size of each input on overall execution cost. These bounds can be inferred both easily and efficiently.

This is the first automatic amortised analysis that can determine costs for higher-order functions directly rather than relying on program transformations such as defunctionalisation [33] to transform higher-order programs into first-order ones. Such transformations are not acceptable for several reasons. Firstly, they usually change time and space properties. This is unacceptable in any context where the preservation of costs is important, such as the increasingly important class of resource-aware applications. Moreover, they may also change which programs can be costed (e.g. by making linear programs non-linear, etc.), and they can destroy the programmers’ intuitions about cost. Unlike transformation methods such as defunctionalisation, our approach is fully compositional. This is important, since compositionality enhances modularity. Our technique can produce usage-dependent upper-bound functions on costs for closed-source libraries of (possibly higher-order) functions. In order to analyse a program that uses such a library, it is only necessary to know the previously inferred annotated type for any function that is exported, and not its definition.

Our automatic analysis is a variant of the amortised cost analysis that was first described by Tarjan [37]. Amortised cost analysis is a manual technique, which works as follows: using ingenuity, one devises a mapping from all possible machine states to a non-negative rational number, henceforth referred to as the potential of that state. This map must be constructed in such a way that the actual cost of each machine operation is amortised by the difference in potentials before and after the execution of the operation. For example, for heap space an operation that allocates \( n \) memory units must always lead to states whose potential is then decreased by \( n \). It follows that the cost of each operation, including entire loops or complete recursive calls, becomes zero, and the overall execution cost is then equal to the potential of the initial state.

There are two main problems to be overcome. Firstly, devising a useful mapping from each machine state to the number representing its potential is a difficult task. Secondly, Okasaki notes that [31]:

“As we have seen, amortized data structures are often tremendously effective in practice. […] traditional methods of amortization break in presence of persistence.”

Our type-based variant solves both of these issues: i) we can automatically determine the abstraction through efficient linear programming; and ii) we can deal with the persistent data structures that are commonly found in a functional setting by assigning po-
our method is currently limited to linear cost formulas (a restriction evaluation strategy as Okasaki does [31]. The price we pay is that using a standard external linear programming solver. Our main associated fully-automatic type inference, which has been implemented morphic programs. We prove that the type system is sound with tion costs for various resource metrics on strict, higher-order, poly-


The use of let-normal form means that the threading of resources is limited to let-expressions. This simplification avoids the need to replicate large parts of the soundness proof for let-expressions in the proofs for the other cases shown in Section 5. However, a transformation to let-normal form could, obviously, alter execution costs. We avoid this by adding a second LET-construct that is used only for transformed expressions. By assigning a different direction to this construct (generally local), we make the transformation to let-normal form entirely cost-neutral. The LET-construct also allows us to construct an accurate cost metric for stack space usage despite the fact that we have chosen to use a big-step semantics. We explain the rationale for this choice in Section 6.1.1.

Since non-monotone cost metrics are interesting to deal with, Schopenhauer includes a primitive for deallocation, which we combine with pattern matching (case). We do not deal with the safety of deallocations, since this is an orthogonal and complex problem that deserves its own treatment (see, for example, Walker and Morriset’s alias types [40], or the bunched implication logic of Ish- tiq and O’Hearn [23]). We encapsulate this problem by adopting essentially a storeless semantics [34, 24]. While we do deal with explicit memory addresses, these should be considered as symbolic handles, as used, for example, in early versions of the JVM. A deallocated memory address is then simply overwritten with the special tag Bad. This prevents its reuse and so guarantees that evaluation halts when dereferencing any stale pointer. As a consequence, we can prove that the required resource bounds are maintained.

3. Schopenhauer Operational Semantics

We now state how Schopenhauer programs are executed, and define the cost for a specific execution sequence, thereby fixing a (resource-aware) operational semantics. The Schopenhauer type rules in Section 4 govern how potential is associated with the runtime values of a particular type. The operational semantics is independent of the type rules. Evaluation may, however, get stuck for untypable programs.

An environment $\mathcal{V}$ is a partial map from variables $x$ to locations $\ell$. Our semantics is therefore based on a boxed heap model. By varying the cost parameters explained below, we can, however, also capture evaluation costs for an unboxed heap model. A heap $\mathcal{H}$ is a partial map from locations to labelled values $w$. $\mathcal{H}[\ell \mapsto w]$ denotes a heap that maps $\ell$ to value $w$ and otherwise acts as $\mathcal{H}$. All values are labelled for simplicity, e.g. (bool, $t$), (int, 7), (constr, $\ell_1, \ldots, \ell_n$), ($x:e, V$). The Ind$(\ell)$ is a special value modelling an indirection. To follow such indirections we define next$(\mathcal{H}, \ell) = \ell$ if $\mathcal{H}(\ell) = \text{Ind}(\ell)$ and next$(\mathcal{H}, \ell) = t$ otherwise. These indirections are needed to model recursive definitions, which we explain with an example at the end of this section. As discussed above, deallocated locations are overwritten with the tag Bad to prevent stale pointers.

Our operational semantics is fairly standard, except that it is instrumented by a resource counter, which defines the cost of each operation. The cost counter is used to measure execution costs. If
This counter becomes negative, then program execution becomes stuck. We are interested in finding the smallest number for each input that safely allows execution. The purpose of the analysis in Section 4 is to provide an upper bound on this number for large classes of inputs, without evaluating the program in any way.

The judgement $\forall, \mathcal{H} \vdash m \downarrow e \leadsto \ell, \mathcal{H}'$ means that under the initial environment $\forall$ and heap $\mathcal{H}$, the expression $e$ evaluates to location $\ell$, containing the result value, and post-heap $\mathcal{H}'$, provided that there are at least $m \in \mathbb{N}$ units of the selected resource available before the computation. Furthermore, $m \in \mathbb{N}$ units will be available after the computation. We write $\forall, \mathcal{H} \vdash e \leadsto \ell, \mathcal{H}'$ to denote that $e$ evaluates to $\ell$ using an unknown, but finite, amount of resources.

For example, $\forall, \mathcal{H} \vdash 3 \downarrow e \leadsto \ell, \mathcal{H}'$ means that three resource units are sufficient to allow $e$ to be evaluated, and that exactly one resource unit is unused after the computation. This unused resource unit might or might not have been used temporarily. Note that this tracks both the overall net resource costs as well as the minimum number of free resources that are necessary for the computation to be started. These two numbers may be different if there is some temporary resource usage, as with stack space usage.

Lemma 3.1. For all $k \geq 0$, if $\forall, \mathcal{H} \vdash m \downarrow e \leadsto \ell, \mathcal{H}'$ holds, then both $m' \geq 0$ and $\forall, \mathcal{H} \vdash m' + k \downarrow e \leadsto \ell, \mathcal{H}'$ hold.

The operational semantics rules for Schopenhauer are shown in Figure 2. Two rules are omitted because they are almost identical to other rules: OP CASE! FAIL is similar to OP CASE FAIL; and

OP LET (which covers LET $x = e_1$), is identical to OP LET if the cost metric parameters Klet1, Klet2 and Klet3 are replaced by KLET1, KLET2 and KLET3, respectively. The rules exploit a number of constant cost parameters. This allows us to deal with several different cost metrics without changing the operational model. Since our analysis uses the same constants regardless of the metric, our soundness proof is completely independent of the cost metric and so does not require to be performed anew for each new cost metric. These parameters must be chosen carefully so that the costs of the operational semantics match reality. For example, the constant KmkInt denotes the cost of constructing an integer constant. So, if we are interested in heap allocation and an integer occupies two heap units, as in our boxed heap model, then we set this constant to two. In an unboxed heap model, however, it is set to zero, since the integer is created directly in the stack. Likewise for stack usage, KmkInt is either the size of a pointer (in the boxed model) or the actual size of an integer (in the unboxed model); and for worst-case execution time (WCET) we set it to the greatest number of clock cycles needed to create an integer constant. For example, the commercial aIT WCET analyser (http://www.abaint.com) determines this to be 83 cycles on the Renesas M32/85U microcontroller.

Recursive let-bindings use directions. An indirection never points to another indirection, since indirections are only introduced in rule OP REC to locations which have been followed. This property is formalised in Definition 5.1, which serves as the invariant for our soundness theorem. This also allows a constant cost bound (Knext) when dereferencing an indirection.
4. Schopenhauer Type Rules

We use $\alpha, \beta, \gamma$ to denote type variables. Let $C$ be an infinite set of resource variables ranging over $Q^+$, usually denoted by $q, p, r, s$, being disjoint from the identifier sets for variables and constructors $\mathsf{Var}, \mathsf{Const}$. Sets of type and resource variables are referred to using the vector notation, e.g. $\vec{\alpha}, \vec{\beta}$. All other decorations stand for different entities. We use $\psi, \phi, \xi$ to range over sets of linear inequalities over non-negative rational constants and resource variables, plus special terms involving type variables that are mapped to linear inequalities when the type variables are substituted with closed type terms. A valuation $v$ is a two-fold mapping, that maps resource variables to $Q^+$ and type variables to closed types. We write $v \Rightarrow \phi$ if $v$ satisfies all constraints in $\phi$, and $v \Rightarrow \phi$ to denote that $v$ entails $\phi$, i.e. that all valuations that satisfy $v$ also satisfy all constraints in $\phi$.

The annotated types of Schopenhauer are then given by the following grammar:

$$T ::= \begin{array}{l}
\mathsf{int} \mid \mathsf{bool} \mid \alpha \\
\forall v \in \mathsf{V} \forall q \in \mathsf{Q} : T q \mid \mathsf{FV}(\alpha, \beta) \mid \mathsf{Nil};(v) \mid y^x \mathsf{int} \\
\forall v \in \mathsf{V} \forall q \in \mathsf{Q} : T q \mid \mathsf{FV}(\alpha, \beta)
\end{array}$$

where $c_i \in \mathsf{Const}$ are constructor labels and $\vec{T}$ stands for $T_1 \ldots T_n$ where $n \geq 0$. Algebraic datatypes are defined as usual, except that each constructor also carries a resource variable in addition to the usual type information.

The types contain two different universal-quantifiers: one for resource variables, and one for type variables. For example, the type of a function counting the length of a list could be:

$$\forall v \in \mathsf{V} \forall q \in \mathsf{Q} : \mathsf{Nil};(v) \mid y^x \mathsf{int} \text{ with } \phi = \{x \geq 156 + y, u \geq 940\}.$$ 

So the type tells us that this length function can be applied to lists of any type ($\forall v:\emptyset$). Furthermore, it admits several resource behaviours, since $\forall v : (x, y, u, v) \in \phi$, tells us that we can rename $x, y, u, v$ to independent resource variables. Of course, the constraints $\phi$ must be substituted accordingly. The admissible valuation $x = 156, y = 0, u = 940, v = 0$ would then indicate that evaluating the function requires at most 156 resource units (in this case clock cycles), plus at most 940 resource units per $\mathsf{Nil}$ constructor in the input. In other words, if $n$ is the length of the input list, the execution cost is bounded by $940n + 156$. However, the connection between the actual cost of running a program and its annotated type, such as the one above, is only guaranteed by Theorem 1.

Continuing with the annotated type example, we also see that the above function can be called with more resources available, since the valuation $x = 256, y = 100, u = 999$ is also admissible, leading to the bound $999n + 256$. Of these resources, at least

$$\text{100} \text{ can be recovered after the call (the value of } y).$$

So list types having extra potential may be accepted, but their additional potential would be lost. This is safe, since it increases the upper bound on resource usage, but of course we will usually avoid such a loss.

The free resource variables of the type and constraint sets are denoted by $\mathsf{FV}(\alpha)$. We also define a mapping $|\cdot|$ from annotated types to standard unannotated types, which simply erases all annotations. For $\forall \alpha \in \mathsf{V} : T q \Rightarrow p + r \Rightarrow T'$, we require that $\alpha \subseteq \mathsf{FV}(T') \cup \{p, r\} \cup \mathsf{FV}(T')$ holds, but that $\mathsf{FV}(\psi)$ is a subset of $\alpha$. Any intermediate variables which would then only occur in $\psi$ can be eliminated by projecting the polytope described by $\phi$ on the relevant dimensions. This ensures that subtyping remains decidable.

The type rules for Schopenhauer govern how potential is associated with each particular runtime value through its type. We denote the part of the potential associated with a runtime value $w$ of type $A$ by $\Phi_{\mathsf{Sch}}(w : A)$ (see Definition 5.4). Intuitively, this is defined as the sum of the weights of all constructors that are reachable from $w$, where the weight of each constructor in the sum is determined by the type $A$. A single constructor at a certain location may contribute to this sum several times, if there is more than one reference to it. It is natural to extend this definition to environments and contexts by summation, i.e. $\Phi_{\mathsf{Sch}}(V : \Gamma) = \sum V(x : \Gamma(x))$ for all variables $x$ in $\Gamma$. Since the potential depends on the state, (static) type rules do not have access to this number, but only govern the relative changes. Note that we never actually need to compute the potential (apart from the initial state), so the potential mainly serves as an invariant in our soundness proof.

We now formulate the type rules for Schopenhauer. These differ from standard Hindley-Milner typing judgments only in that they also refer to cost and resource variables. Note that we are not concerned with type inference itself (a generally solved problem), but only with the inference of our new type annotations. Let $\Gamma$ denote a typing context mapping identifiers to annotated Schopenhauer types. The Schopenhauer typing judgement

$$\Gamma \vdash \phi : e : A \mid \psi$$

then reads “for all valuations $\nu$ that satisfy all constraints in $\phi$, the expression $e$ has Schopenhauer type $\nu(A)$ under context $\nu(\Gamma)$; furthermore, evaluating $e$ under environment $V$ and heap $H$ requires a potential of at most $\nu(q)$ and $\Phi_{\mathsf{Sch}}(V : \Gamma)$ and leaves a potential of at least $\nu(q) + \Phi_{\mathsf{Sch}}(\nu(A) : \Gamma)$ available afterwards, where $\ell$ is the result value and $\mathfrak{c} \ell$ is the post-heap”. In Section 5, we will formalise this statement as Theorem 1 (our main theorem), which requires as a pre-condition that the context, environment and heap all agree.

We use a compressed notation that makes the following two formulations equivalent for $\psi = \{q = q_1 + q_2, q_2 = q'_1 + c_2\}$:

$$\Gamma | q_2 / q_1 q_2 | e_2 : A_2 \mid \phi \quad \Gamma | q_2 / q_1 q_1 + c_2 | e_2 : A_2 \mid \phi$$

The constraints $\psi$ that were explicitly introduced in the left-hand form have thus become implicit in the compressed notation on the right. We believe that, with a little practice, the compressed notation is actually easier to read. It is also closer to our implementation, which avoids the introduction of unnecessary intermediate variables. Note that we do not simplify constraints after they are generated, since the LP-solver is much faster if we do not do so.

### Basic Expressions.

```
| n ∈ Z | x : A | $\mathsf{AplusVar}$ | x : A | $\emptyset$ | (VAR)
| $\emptyset$ | $\mathsf{Knull}$ | 0 | n : int | $\emptyset$ | (INT)
| e ∈ $\{\text{true, false}\}$ | $\mathsf{Kandbool}$ | 0 | e : A | $\emptyset$ | (BOOL)
```

©ACM, 2010. This is the authors version of the work. It is posted here by permission of ACM for your personal use. Preprint. Not for redistribution.
Since primitive terms such as integers (INT) or variables (VAR) always have fixed evaluation costs, a fixed initial potential and a returned potential of zero suffices. The restriction to empty contexts and the use of explicit weakening, rule WEAK below, just serves to simplify our soundness proof by removing redundancies. For our prototype implementation, we have merged the WEAK and RELAX rules into all terminal rules.

Structural rules. We use explicit structural rules for weakening and sharing (contraction), while exchange is built-in. It is necessary to track pointers that are discarded (WEAK) or duplicated (SHARE), since such operations may affect resource consumption. An additional structural rule (RELAX) allows potential to be discarded both before or after a term, as well as allowing a constant amount of potential to bypass a term.

In our system, unlike in a strictly linear type system, variables can be used several times. However, the sum of all the potential bestowed by each type of all the existing references must not exceed the potential that was originally attached to the type associated with the entity when it was created. It is the job of the SHARE rule to track multiple occurrences of a variable; and it is the job of the \( \gamma \)-function to apportion potential appropriately.

The application of these rules is straightforward. For example, where there are multiple uses of a variable, sharing is used only at the latest point; WEAK is applied before each terminal rule; and RELAX is built-in throughout the rules with an additional slack variable that is punished in the objective function, so discouraging the LP-solver from using relaxations.

\[
\Gamma \vdash_{\mathcal{P}} e : A \mid \phi \quad \text{(RELAX)}
\]

\[
\Gamma \vdash_{\mathcal{Q}} e : A \mid \phi \cup \{q \geq p, q - p \geq q' - p'\}
\]

\[
\Gamma \vdash_{\mathcal{P}} e : C \mid \psi \quad \phi \Rightarrow \psi \quad \text{(WEAK)}
\]

\[
\Gamma, x : A \vdash_{\mathcal{Q}} e : C \mid \phi \quad \text{(SHARE)}
\]

The ternary function \( \chi(A | B, C) \) returns a set of constraints that enforce the property that each resource variable in \( A \) is equal to the sum of its counterparts in \( B \) and \( C \). This function is only defined for structurally identical types \( A, B, C \), i.e., types that differ at most in the names of their resource variables. For example, we have

\[
A = \mu X. (\{\text{nil}(a)\} | \text{Cons}(\text{d}, \{\text{int}, X\}))
\]

\[
B = \mu X. (\{\text{nil}(b)\} | \text{Cons}(\text{f}, \{\text{int}, X\}))
\]

\[
\chi(A | B, C) = \{a = b + \alpha, \quad d = e + f\}
\]

For type variables we simply record \( \chi(\alpha | \beta_1, \beta_2) \) within the constraints, and replace it by the according constraints upon specialisation. The crucial property of sharing is expressed in Lemma 5.7.

Function Abstraction & Application.

\[
\text{dom}(\Gamma) = \text{FV}(e) \setminus x \quad B = A \stackrel{\phi}{\rightarrow} C \quad \phi \cup \psi \Rightarrow \xi
\]

\[
\Gamma, x : A \vdash_{\mathcal{Q}} e : C \mid \xi \quad \phi \Rightarrow \bigcup_{D \in \text{ran}(\Gamma)} \chi(D | D, D)
\]

\[
\Gamma \vdash_{\text{Kapp}(\Gamma)} \mu x.e : \forall \forall \psi \in B \mid \phi
\]

Since the potential stored in the function closure becomes available for each function application, in order to allow the unlimited repeated application of functions, we must restrict the potential stored in a function closure to zero. This is achieved by abusing the sharing operator \( \gamma \). Here, \( \gamma(D | D, D) \) just generates the constraint \( x = x + x \) for each resource variable in \( D \), forcing them all to zero. All the potential required during the execution of the function body must therefore be provided by its arguments, except for a constant amount.

This, relatively minor, restriction only affects functions that recurse over a captured free variable, but not over one of their inputs. We have not yet encountered an interesting program example where this restriction would be an issue. In order to deal with such functions, potential could be allowed within the closure, provided that a static bound on the number of calls to such functions could be determined. We plan to experiment with use-n-times functions in future work, if this restriction turns out to be a real issue. Alternatively, knowing the sizes of potential-bearing entities captured in a closure would allow us to recharge their potential at each call. Combining our work with a "sized-type" analysis (e.g. [39]) might thus also avoid this limitation.

Each function body is only analysed once, associating a set of constraints with the function. At each application, these constraints are copied from the type. All resource variables that only occur in the function’s type and constraints, but nowhere else, are given fresh names for each application. Thus, although each function is only analysed once, the LP-solver may still choose a different solution for each individual application of the function.

\[
\sigma : \overrightarrow{\Gamma} \Rightarrow CV \text{ a substitution to fresh resource variables}
\]

\[
\sigma(B) = A \stackrel{q}{\rightarrow} C
\]

\[
x : A, y : \forall \forall \rho \in \psi, B \vdash_{\text{app}} \Gamma \quad \rho \quad \text{APP}
\]

Note that the LET-construct can be used to specialise the function before application. This is required anyway, if we follow the convention suggested in Section 6.1.1 that normal sub-expressions should always be unique variables, and that these are introduced by a LET-construct immediately before their single use.

Algebraic Datatypes and Conditionals.

\[
\Gamma \vdash_{\text{KifT}} e_1 : A \mid \phi \quad \Gamma \vdash_{\text{KifF}} e_2 : A \mid \psi
\]

\[
\Gamma, x : \text{bool} \vdash_{q'} \text{if } x \text{ then } e_1 \text{ else } e_2 : A \mid \phi \cup \psi \quad \text{(CONDITIONAL)}
\]

\[
e \in \text{Constrs} \quad C = \mu X. (\cdots | c : (p, \{B_1, \ldots, B_k\})) \cdots
\]

\[
A_i = B_i | C \mid X \quad \left(\text{for } i = 1, \ldots, k\right)
\]

\[
x_1 : A_1, \ldots, x_k : A_k \vdash_{p + \text{Kalloc}(c)} 0 \quad c \mid x_1, \ldots, x_k : C \mid \emptyset \quad \text{(CONSTR)}
\]

The Constr rule plays a crucial role in our annotated type system, since this is where available potential may be associated with a new data structure. Potential cannot be used while it is associated with data; it can only be used once it has been released using the CASE rule that forms the dual to the Constr rule. A successful match releases the potential associated with the corresponding constructor.

\[
e \in \text{Constrs} \quad A = \mu X. (\cdots | c : (p, \{B_1, \ldots, B_k\})) \cdots
\]

\[
\Delta = y_1 : B_1 | A \mid X, \ldots, y_k : B_k | A \mid X
\]

\[
\Gamma, \Delta \vdash_{q + \text{Kdealloc}(c) \text{ - KcaseF}(c)} e_1 : C \mid \phi
\]

\[
\Gamma, x : A \vdash_{q + \text{KcaseT}(c)} \text{case } x \text{ of } c \mid y_1, \ldots, y_k \Rightarrow e_1 | e_2 : C \mid \phi \cup \psi \quad \text{(CASE!)}
\]

The CASE rule for the read-only case pattern-match is identical to CASE!, except that it doesn’t include the cost parameter Kdealloc(c), the (possibly negative) cost of deallocating constructor constrC.
The type rule for the alternative form of let-expression

\[ (\text{LET}) \]

is almost identical, except it substitutes the cost constants on the constants

Let-bindings.

\[
\frac{q_1}{x_1} : A_1 \mid \psi \quad \Delta, x: A_1 \frac{q_2}{x_2} : A_2 \mid \phi}{\Gamma \frac{q_3}{x_3} \frac{q_4}{x_4} \Delta \vdash \text{let } x = e \text{ in } e_2 : A_2 \mid \psi \cup \phi}
\]

The recursive let rec construct allows circular data to be constructed. In contrast to non-circular aliased data, which may be created carrying per-reference potential, as usual, circular data is ill-suited for bounding recursion since its type-based potential must be either infinite or zero. The REC rule therefore enforces zero potential by abusing the sharing operator \( \gamma \) in the same manner as the ABS rule. As previously noted, function types are always assigned zero potential, and so are not affected, since the potential that is required to execute the body of a function must come from the arguments to the function.

Polymorphism.

\[
\frac{\alpha \notin \text{dom}(\Gamma)}{\Gamma \frac{\alpha}{\psi} e : \forall \alpha.\phi. C \mid \psi}
\]

(\text{GENERALISE})

Rec-bindings.

\[
\Delta = x_1: A_1, \ldots, x_n: A_n
\]

\[
\forall i \in \{1, \ldots, n\}, \quad \Gamma, \Delta, x_i \frac{q_{i_1}}{x_{i_1}} + \frac{q_{i_2}}{x_{i_2}} e_i : A_i \mid \psi_i
\]

\[
\phi \Rightarrow \xi \cup \bigcup_{i=1}^{n} \psi_i \cup \bigcup_{B \in \text{trans}(\Delta)} Y(B | B, B)
\]

\[
\phi \Rightarrow \{q \geq q_1 + \text{Krec1} + n \cdot \text{Knext}\}
\]

\[\Gamma_1, \ldots, \Gamma_{n+1} \frac{q}{\psi} \text{let rec} \{x_1 = e_1; \ldots; x_n = e_n\} \text{ in } e : C \mid \phi\]

(REC)

Subtyping.

The type rules for subtyping depend on another inductively defined relation \( \xi, \phi \Rightarrow A <: B \) between two types \( A \) and \( B \), defined below, which is relative to a constraint set \( \xi \), which is in general a constraint set. The general rule is:

\[
\frac{\Gamma, x : B \frac{\alpha}{\psi} e : C \mid \phi \quad \psi \Rightarrow \xi[\overline{B} / \overline{\alpha}]}{\Gamma \frac{\alpha}{\psi} e : C[\overline{B} / \overline{\alpha}] \mid \phi}
\]

(SUBTYPE)

For any fixed constraint set \( \xi \), the following relation is both reflexive and transitive, but not necessarily anti-symmetric.

For all \( i \) holds \( \xi = \{p_i \geq q_i\} \) and \( \xi \Rightarrow \overline{A_i} <: \overline{B_i} \)

\[
\xi \Rightarrow \mu X.\{\cdots c_i : (p_i, A_i) \cdots\} <: \mu X.\{\cdots c_i : (q_i, B_i) \cdots\}
\]

\[\sigma : \tilde{\alpha} \rightarrow \tilde{\beta} \text{ a substitution} \quad \xi \cup \sigma \Rightarrow \sigma(\psi) \quad \xi \vdash \sigma(A) : B \]

The inference itself follows straightforwardly from these type rules. First, a standard typing derivation is constructed, and each type occurrence is annotated with fresh resource variables. We insert the structural rules as outlined above and then traverse the type derivation precisely once to gather all the constraints. Because all types have been annotated with fresh resource variables, subtyping is required throughout, but this will always succeed and it will generate the necessary inequalities. We illustrate this process in more detail with a simple example in Section 6.1.2.

In the final step, the constraints that have been gathered are solved by a standard LP-solver [4]. In practice, we have found that the sparse LPs that are generated can be easily solved, partly because they have a simple structure [19]. Furthermore, the number of constraints that are generated is linear in the size of the analysed program without resource parametricity; and at most quadratic with resource parametricity. Since only a single pass over the program code is needed to construct these constraints, this leads to a highly efficient analysis. An online demo of our analysis is available at http://www.embounded.org/software/cost/cost.cgi.

5. Soundness of the Analysis

We now sketch the most important steps for formulating our main soundness theorem. We first formalise the notion of a ‘well-formed’ environment, written \( \mathcal{H} = : \mathcal{V} \Gamma \), which simply states that for each variable, the type assigned by the typing context agrees with the actual value found in the heap location that is assigned to that variable by the environment. This is an essential invariant for our soundness proof.

Definition 5.1. A memory configuration consisting of heap \( \mathcal{H} \) and stack \( \mathcal{V} \) is well-formed with respect to context \( \mathcal{I} \) and valuation \( v \), written \( \mathcal{H} \models \mathcal{V} \Gamma \), if and only if for all variables \( x \in \Gamma \) the statement \( \mathcal{H} \models \mathcal{V}(x) \Gamma(x) \) can be derived by the rules in Figure 3.

Lemma 5.2. If \( \mathcal{H} \models \mathcal{V} \Gamma \) then for all \( \ell \) also \( \mathcal{H}[\ell \mapsto \text{Bad}] \models \mathcal{V} \Gamma \)

It is an obvious requirement that evaluation must maintain a well-formed memory configuration.
Lemma 5.3. If $\mathcal{H} \vdash \ell, V, \Gamma$ and $V, \mathcal{H} \vdash e \leadsto \ell, \mathcal{H}'$ then $\mathcal{H}' \vdash V, \Gamma$.

We remark that one might wish to prove an extended statement that the result $\ell$ of the valuation is also well-formed if the expression $e$ was typable. Unfortunately such a statement cannot be proven on its own and must be interwoven into Theorem 1.

Definition 5.4 (Potential). If $\mathcal{H} \vdash: e: A$ holds, then the potential of location $\ell$ for type $A$ in heap $\mathcal{H}$ under valuation $\nu$, written $\Phi_{\mathcal{H}}^\nu(\ell; A)$, is recursively defined for recursive datatypes by

$$
\Phi_{\mathcal{H}}^\nu(\ell; A) = \nu(q) + \sum_{i} \Phi_{\mathcal{H}}^\nu(\ell_i; B_i[A/X])
$$

when both $A = \mu X. \{ \cdots \}$ and also $\mathcal{H}(\text{next}(\mathcal{H}, \ell)) \in \mathcal{H}$ and $\nu(\ell)$ holds, and zero in all other cases. We extend this definition to contexts by

$$
\Phi_{\mathcal{H}}^\nu(V : \nu(\Gamma)) = \sum_{x \in \text{dom}(\Gamma)} \Phi_{\mathcal{H}}^\nu(V(x) : \nu(\Gamma(x))
$$

Subtyping must respect the well-formed environments and the amount of potential associated with any value of that type.

Lemma 5.5. If $\mathcal{H} \vdash: \ell: A \land \nu \vdash e: B$ holds and $\nu$ is a valuation satisfying $\phi$, then $\mathcal{H} \vdash: \ell: B$ and $\Phi_{\mathcal{H}}^\nu(\ell; A) \geq \Phi_{\mathcal{H}}^\nu(\ell; B)$.

If a reference is duplicated, then the type of each duplicate must be a subtype of the original type.

Lemma 5.6. If $\mathcal{H}(A[B, C]) = \phi$ holds then also $\phi \vdash A < B$.

The potential attached to any value of a certain type is always shared linearly among types introduced by sharing. In other words, the SHARE rule does not increase the total available potential.

Lemma 5.7. If $\mathcal{H} \vdash: \ell: A$ and $\mathcal{H}(A[B, C]) = \phi$ holds then $\Phi_{\mathcal{H}}(\ell; A) = \Phi_{\mathcal{H}}(\ell; B) + \Phi_{\mathcal{H}}(\ell; C)$. Moreover, for $A = B$ and $A = C$, it follows that $\Phi_{\mathcal{H}}(\ell; A) = 0$ also holds.

Lemma 5.8 (Inversion). If $\Gamma \vdash^\ell \lambda x.e : B \mid \phi$ holds, then there exists $\Delta, \xi, \forall s \in \psi : \Phi_{\mathcal{H}}^\xi(A \forall s \rightarrow C) < B$. From all of the following hold

$$
\begin{align*}
\phi & \vdash (\forall s \in \psi : A \forall s \rightarrow C) < B : \Delta, x : A \vdash^\ell e : C : \xi \\
\Delta & \subseteq \Gamma : \text{dom}(\Delta) = \text{FV}(e) \times x \\
\phi & \cup \psi \trianglerighteq \xi \\
\phi & \Rightarrow \bigcup_{D : \text{dom}(\Delta)} \mathcal{H}(D[D, D]) \mid \phi \Rightarrow p \geq p' + \text{KmkFun}(\{\Delta\})
\end{align*}
$$

We can now formulate the main theorem, as described intuitively in Section 4.

Theorem 1 (Soundness). Fix a well-typed Schopenhauer program. Let $r \in \mathbb{Q}^+$ be fixed, but arbitrary. If the following holds

$$
\begin{align*}
\Gamma & \vdash^\ell q : e : A : \phi \\
V, \mathcal{H} & \vdash e \leadsto \ell, \mathcal{H}' \\
\mathcal{H} & \vdash V, \psi(\Gamma) \\
v & : \text{a valuation satisfying } \nu(\phi)
\end{align*}
$$

then for all $m \in \mathbb{N}$ such that

$$
m \geq \nu(q) + \Phi_{\mathcal{H}}(V : \psi(\Gamma)) + r
$$

there exists $m' \in \mathbb{N}$ satisfying

$$
\begin{align*}
V, \mathcal{H} & \vdash^\ell m : e \leadsto \ell, \mathcal{H}' \\
m' & \geq \nu(q') + \Phi_{\mathcal{H}}(\ell; A) + r
\end{align*}
$$

The proof is by induction on the lengths of the derivations of (1.B) and (1.A) ordered lexicographically, with the derivation of the evaluation taking priority over the typing derivation. This is required since an induction on the length of the typing derivation alone would fail for function applications, since in this case we extend the length of the typing derivation by the typing judgment for the body, using the invariant for well-formed environments. On the other hand, the length of the derivation for the term evaluation never increases, but may remain unchanged where the last step of the typing derivation was obtained by a structural rule. In these cases, the length of the typing derivation decreases, allowing an induction over lexicographically-ordered lengths of both derivations.

The proof is complex but unsurprising for most rules. The arbitrary value $r$ is required to carry over excessive potential, which may be required in the second sub-expression of a let-expression, but left untouched by the first sub-expression. We sketch some important cases:

(ABS) In the case that the last step of the derivation for (1.A) was derived by rule ABS, we also know that the last step for (1.B) must have been performed according to rule OP ABS. We have $\mathcal{H}'(\ell) = (\lambda x.e, V^*)$. Fix $r \in \mathbb{Q}^+$ and choose any $m \in \mathbb{N}$ such that $m \geq \nu(q) + \Phi_{\mathcal{H}}(V : \psi(\Gamma)) + r$. By the definition of ABS and the observation that $\Gamma$ has no potential by Lemma 5.7, we have $m \geq \text{KmkFun}(|V^*|) + r$. Furthermore $m \geq \text{KmkFun}(|V^*|) + r$ since $|\Gamma| = |V^*|$. By $V^* = \bigcup_{V \in \text{Var}} \mathcal{H}(e)$ from OP ABS and dom($\Gamma$) = FV($V^*) \times x$ from ABS. By OP ABS and Lemma 3.1 we thus obtain $m' = m - \text{KmkFun}(|V^*|) + r$ which satisfies $m' \geq r$ as required, since the potential of $\ell$ is zero by Definition 5.4. This leaves us to prove (1.III), which follows in this case directly from WFPFun, since all existentially quantified requirements are among our premises, except for $\mathcal{H}'[\ell \mapsto \text{Bad}] \vdash V : \Gamma$ which follows by Lemmas 5.2 and 5.3 from (1.C).

(APP) From OP ABS we have $\mathcal{H}(\forall y.e) = (\lambda x.e, V^*)$ and from (1.III) through WFPFun we obtain the existence of a typing judgment for the function body. By the inversion Lemma 5.8, we obtain all the required properties to derive $\Gamma \vdash^\ell \lambda x.e : V, \psi(e) : B \mid \phi$ through the application of the ABS type rule. This allows us now to apply the induction hypothesis, together with the premise of (1.B) for the body of the function. The application of the induction hypothesis is justified despite the increased type derivation, since the evaluation was shortened by one step. Again, note that the potential of $\Gamma$ is zero. This follows from Lemmas 5.5 and 5.7. Lemma 5.5 is also important for deriving the necessary inequalities between $m$ and $m'$ and their counterparts from the induction hypothesis. Conclusion (1.III) follows from the induction hypothesis and the first part of Lemma 5.7.

(RELAX) Let $r \in \mathbb{Q}^+$ be fixed but arbitrary. We observe that $m \geq \nu(p) + \Phi_{\mathcal{H}}^\nu(V : \psi(\Gamma)) + r + v(q) + \Phi_{\mathcal{H}}(V : \Gamma) + r'$ if we choose $r' = r + v(p) - v(q)$ for applying the induction hypothesis. We can do this since $\nu(p) - v(q) \geq 0$ holds by the constraints of the RELAX rule. We thus obtain $m'$ with

$$
m' \geq \nu(q') + \Phi_{\mathcal{H}}^\nu(V : \Gamma) + r'
$$

which follows by $\nu(p) - v(q) \geq v(p') - v(q')$ from the other constraint added in RELAX.

©ACM, 2010. This is the authors version of the work. It is posted here by permission of ACM for your personal use. Preprint. Not for redistribution.
### 6. Example Cost Analysis Results

In this section, we compare the bounds inferred by our analysis against concrete measurements. Our measurement results were obtained from an instrumented version of the underlying abstract machine that counts resources used during the execution.

For readability, the programs in this section use a more compact functional notation than Schopenhauer, expression-level Hume [14], without a restriction to let-normal form. This Haskell-style notation uses multiple rules with pattern matching instead of top-level, asymmetric case expressions. The basic type of integers is parametrised with its bit-size precision. We use the familiar notation of [] for Nil and _:_ for Cons in the pre-defined list type: data [a] = Nil | Cons a [a]. This notation is automatically translated to the Schopenhauer code that is actually analysed.

The examples chosen in this section focus on the main language features that are of interest in this paper: higher-order functions, polymorphism and destructive pattern matching. The examples are deliberately kept small to demonstrate the applicability of our approach to these language features, without being side-tracked by previously-solved problems. For example, the variants of the sum-of-squares function demonstrate how our analysis faithfully reflects the increased performance that is achieved when turning a composition of higher order functions into direct recursion. The final evaluator example is interesting because it modifies the argument function as it is passed through the recursive calls.

#### 6.1 List-sum

Our first example computes the sum of a list of integers (Figure 4). In order to demonstrate the use of our analysis on higher-order functions, we define the sum function as an instance of the standard (left-) fold function. A bound on the heap usage for the sum function is given by the following enriched type, where # represents the \( \mu \)-type, i.e., list, with the constructors Cons and Nil.

```haskell
type num = int 16;
add :: num \rightarrow num \rightarrow num;
add x y = x * y;
fold :: (num \rightarrow num \rightarrow num) \rightarrow num \rightarrow [num] \rightarrow num;
fold f n [] = n;
fold f n (x:xs) = fold f (f x n) xs;
sum :: [num] \rightarrow num;
sum xs = fold add 0 xs;
```

**Figure 4.** Source code of list-sum

The argument type includes annotations for each constructor, separated by \( \lambda \). This shows that at most two units of heap are needed for every Cons constructor in the input list (shown by the annotation Cons<2>). In addition to this input-dependent part, the sum function needs at most 6 heap units, shown by the first annotation to the function type \( \lambda \). As shown by the second annotation (the zero) and the absence of annotations in the result type, the analysis could not find a guarantee that any portion of the requested heap memory is unused after execution. In total, given an input list of length \( n \), the heap consumption of this function is therefore bounded by \( 2n + 6 \).

This bound can be seen to be exact by direct inspection of the source code in Figure 4. In the sum function, a constant of 2 is needed to allocate the initial integer value of 0. Another constant of 4 is needed to create a closure for the add function (a closure includes a tag, a function pointer, plus counts of expected and supplied arguments). In the fold function, a new integer value is created in each iteration through the application of \( f \times n \). This requires two heap cells per iteration. This value is therefore attached to the Cons constructor of the input.

The bound on the stack consumption for sum, shown below, is a constant for this tail-recursive program. The absence of annotations to the Cons constructor indicates that the bound is independent of the size of the input list.

#### Table 1. Measurement and analysis results for list- and tree-processing functions

<table>
<thead>
<tr>
<th>Function</th>
<th>N = 1</th>
<th>N = 2</th>
<th>N = 3</th>
<th>N = 4</th>
<th>N = 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Heap</td>
<td>Stack</td>
<td>Time</td>
<td>Heap</td>
<td>Stack</td>
</tr>
<tr>
<td>sum (see Fig 4)</td>
<td>16</td>
<td>39</td>
<td>3603</td>
<td>24</td>
<td>39</td>
</tr>
<tr>
<td>Analysis</td>
<td>1</td>
<td>0</td>
<td>0.58</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Measured</td>
<td>1</td>
<td>0</td>
<td>0.58</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>flatten (see Fig 6)</td>
<td>53</td>
<td>96</td>
<td>16962</td>
<td>55</td>
<td>66</td>
</tr>
<tr>
<td>Analysis</td>
<td>1</td>
<td>0</td>
<td>0.58</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Measured</td>
<td>1</td>
<td>0</td>
<td>0.58</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>repmin</td>
<td>17</td>
<td>42</td>
<td>5020</td>
<td>35</td>
<td>69</td>
</tr>
<tr>
<td>Analysis</td>
<td>1</td>
<td>0</td>
<td>0.58</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Measured</td>
<td>1</td>
<td>0</td>
<td>0.58</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**SCHOPENHAUER typing for HeapBoxed:**

\[
\text{list}([\text{Cons}<2]\times \text{int},\# \text{Nil}) \rightarrow (6/0) \rightarrow \text{int}
\]

The argument type includes annotations for each constructor, separated by \( \lambda \). This shows that at most two units of heap are needed for every Cons constructor in the input list (shown by the annotation Cons<2>). In addition to this input-dependent part, the sum function needs at most 6 heap units, shown by the first annotation to the function type \( \lambda \). As shown by the second annotation (the zero) and the absence of annotations in the result type, the analysis could not find a guarantee that any portion of the requested heap memory is unused after execution. In total, given an input list of length \( n \), the heap consumption of this function is therefore bounded by \( 2n + 6 \).

This bound can be seen to be exact by direct inspection of the source code in Figure 4. In the sum function, a constant of 2 is needed to allocate the initial integer value of 0. Another constant of 4 is needed to create a closure for the add function (a closure includes a tag, a function pointer, plus counts of expected and supplied arguments). In the fold function, a new integer value is created in each iteration through the application of \( f \times n \). This requires two heap cells per iteration. This value is therefore attached to the Cons constructor of the input.

The bound on the stack consumption for sum, shown below, is a constant for this tail-recursive program. The absence of annotations to the Cons constructor indicates that the bound is independent of the size of the input list.

**SCHOPENHAUER typing for StackBoxed:**

\[
\text{list}([\text{Cons}:\text{int},\# \text{Nil}] \rightarrow (27/17) \rightarrow \text{int}
\]

Finally, we can infer an upper bound on the time consumption of this function using our worst-case execution costs in clock cycles for the Renesas M32C/85U processor. As expected, time consumption is linear in the length of the input list \( n \), namely \( 1714n + 909 \).

**SCHOPENHAUER typing for TimeM32:**

\[
\text{list}([\text{Cons}:\lambda \text{int},\# \text{Nil}:\lambda \text{int} \rightarrow \text{int}) \rightarrow (684/0) \rightarrow \text{int}
\]

The first block in Table 1 compares the analysis bounds above with our measured results, applying sum to the initial segments of the input list \( [1,2,3,4,5] \) of lengths \( N = 1, 2, 3, 4, 5 \). Since we analyse and measure the entire code, including the costs for generating the test input, the absolute values given in the table are slightly higher than the values calculated from the function types above. The ratio of inferred to measured costs is used to assess the quality of our bounds against actual behaviour. We can see that the predicted heap consumption is exact in all cases. For stack usage, the measured costs for this tail-recursive program are constant (34). The inferred...
Recall from the introduction that in order to remove annoying
boundaries are also constant but not exact in this case. Finally, our time
predictions are a close match to actual execution times, yielding
an estimate that is between 18% and 26% higher than the actual
cost. In general, we expect less accurate bounds for time, because
the entries in the cost table are already worst-case bounds for the
primitive operations of the abstract machine.

6.1.1 Let-normal form

Recall from the introduction that in order to remove annoying
redundancies from the proof of Theorem 1, we require programs to be
in let-normal form. Programs can automatically be transformed into
let-normal form without altering their (cost) behaviour using a
second LET-construct that simply has zero costs assigned to it.
This is achieved by adopting the policy that each sub-expression
must be a unique variable and that this variable is introduced by
the LET-construct immediately before its (single) use. For example, the 
fold function from Figure 4 would be transformed into the
let-normal form of Figure 5.

Under this policy, the rule for function calls can expect that all
arguments are available on the stack. The cost for pushing variables
on the top of the stack or creating constants was already modelled
by the ordinary VAR, INT and BOOL rules. It follows that only the
cost of popping the arguments from the stack, after returning from
the call, must be included in rule AP. An additional benefit is that
the order in which the arguments are placed on the stack is also made
explicit in the code by the order of the LET-bindings. Although
our prototype implementation always adheres to it, we have
framed from strictly enforcing this policy in Schopenhauer be-
cause it is not intrinsic to our analysis method, and it is conceivable
that other cost models might not require such a strict convention.

6.1.2 Manual amortised cost analysis demonstration

We now illustrate how the type rules are applied and perform a
manual analysis for a simplified version of one branch of the fold.

The first step is to enrich the type for the fold function with fresh
variables, representing the as yet unknown annotations.

\[
\begin{align*}
\text{fold } \mathbf{f} \mathbf{n} \mathbf{l} & = \text{LET } \mathbf{l} \mathbf{1} = 1 \text{ IN } \\
\text{case } \mathbf{l} \text{ of } [\mathbf{]} & \rightarrow \text{LET } \mathbf{n} \mathbf{l} = \mathbf{n} \text{ IN } \mathbf{n} \mathbf{l}; \\
| (x : \mathbf{x}) & \rightarrow \text{LET } \mathbf{x} \mathbf{1} = x \mathbf{x} \text{ IN } \\
\mathbf{n} & \rightarrow \mathbf{n} \text{ IN } \\
\mathbf{l} & \rightarrow \mathbf{x} \text{ IN } \\
\mathbf{x} & \rightarrow \mathbf{x} \text{ IN } \\
\mathbf{f} & \rightarrow \mathbf{x} \text{ IN } \\
\mathbf{f} & \rightarrow \mathbf{x} \text{ IN } \\
\mathbf{n} & \rightarrow \mathbf{n} \mathbf{l} \mathbf{1} \mathbf{x} \mathbf{s} ;
\end{align*}
\]

It is possible to spot the recursive nature of fold in these
constraints, since \(x\) occurs both on the left and right hand side, i.e.
the cost must be paid in full by \(p\). This is justified, since each recursive
step introduces a new \(\text{Cons}\)-constructor, bearing a potential of \(p\).
Both \(x\) and \(y\) can have arbitrary values, which is sound since we
ignore the terminating case branch in this example. For the full
fold function, we need to similarly examine the branch dealing with
\([\mathbf{]}\). This produces the constraint \(x \geq y + C_3\), restricting \(x\) and
\(y\), as expected.
data tree = Leaf num | Node tree tree;

dfsAcc ::(num -> [num] -> [num]) -> tree -> [num] -> [num];
dfsAcc g (Leaf x) acc = g x acc;
dfsAcc g (Node t1 t2) acc = let acc' = dfsAcc g t1 acc in dfsAcc g t2 acc';

cons :: num -> [num] -> [num];
cons x xs = x:xs;

revApp :: [num] -> [num] -> [num];
revApp [] acc = acc;
revApp (y:ys) acc = revApp ys (y:acc);

flatten :: tree -> [num];
flatten t = revApp (dfsAcc cons t []) [];

Figure 6. Source code of tree-flattening (flatten)

6.2 Tree operations

The next two examples operate over trees. The first is a tree flattening function, using a higher-order depth-first-traversal of a tree structure that is parametrised by the operation that is applied at the leaves of the tree. The source code is given in Figure 6.

Again, the bounds for heap, stack, and time consumption are linear in the number of leaves (l) and nodes (n) in the input structure: the heap consumption is $l + 8$, the stack consumption is $10l + 16n + 14$, and the time consumption is $2850l + 998n + 821$.

The second block in Table 1 compares analysis and measurement results for the tree-flattening example. Again the bounds for heap are exact. For stack, the analysis delivers a linear bound, whereas the measured costs are logarithmic in general. Here, we could usefully apply a further extension of the amortised cost based analysis. Campbell [6] has developed methods for associating potential in relation to the depth of data structures. This is more suitable for stack-space usage. It also allows temporary “borrowing” of potential. The time bounds give very good predictions, with an over-estimate of at most 13% for the range of inputs shown here.

The second operation on trees is the $\text{repmin}$ function which replaces all leaves in a tree with the element with the minimal value. This function is implemented in two phases, both using higher-order functions: the first phase computes the minimal element using a tree-fold operation; the second phase fills in this minimal element at most $13\%$ of the measured costs.

The second block in Figure 1 compares analysis and measurement results for the $\text{repmin}$ function. Again the bounds for heap are exact. For stack, we observe a linear bound with a more pronounced difference between the measured and analysed costs. This is due to the two tree traversals. The time bounds, however, show a good match against the measured costs, with an over-estimate of at most 22%.

6.3 Sum-of-squares

In this section, we study 3 variants of the classic sum-of-squares example (Figure 7). This function takes an integer $n$ as input and calculates the sum of all squares ranging from 1 to $n$. The first variant is a first-order program using direct recursion, which does not

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum-of-squares (variant 1: direct recursion)</td>
<td>Analysis</td>
<td>22</td>
<td>114</td>
</tr>
<tr>
<td></td>
<td>Measured</td>
<td>22</td>
<td>108</td>
</tr>
<tr>
<td></td>
<td>Ratio</td>
<td>1.00</td>
<td>1.06</td>
</tr>
<tr>
<td>sum-of-squares (variant 2: with map and fold)</td>
<td>Analysis</td>
<td>56</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td>Measured</td>
<td>56</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td>Ratio</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>sum-of-squares (variant 3: also unfold)</td>
<td>Analysis</td>
<td>71</td>
<td>272</td>
</tr>
<tr>
<td></td>
<td>Measured</td>
<td>71</td>
<td>272</td>
</tr>
<tr>
<td></td>
<td>Ratio</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 2. Results for three variants of the sum-of-squares function

As discussed before, our inference engine is largely independent of the actual resource being inferred. We can therefore easily adapt our analysis to other resources simply by replacing the basic cost table that is used to model the program execution costs. We exploit

©ACM, 2010. This is the authors version of the work. It is posted here by permission of ACM for your personal use. Preprint. Not for redistribution.
this capability here to infer bounds on the total number of function calls in a program expression. This metric is of particular interest for higher-order programs (this is discussed in more detail in [30]).

The results for this resource are given in the second column of Table 2. The first variant exhibits the lowest number of function calls, since all three phases of the computation are covered by one recursive function. Thus, we have one function call to the square function and one recursive call for each integer value. Additionally, we have one call to the top level function:

SCHOPENHAUER typing for CallCount: (int<2>) -(2/1)-> int

The second variant separates the phases of generating a list, computing the squares and summing them. The generation phase, implemented using direct recursion, needs one call per iteration. The other two phases each need two calls per iteration: one for the higher-order function, and one for the function being applied. In total we have $5n + 6$ calls, as encoded by the following type:

SCHOPENHAUER typing for CallCount: (int<5>) -(6/0)-> int

The third variant again has three phases. Now all three phases use higher-order functions, with the enumeration being implemented through a call to unfold. The number of calls therefore increases to $7n + 1$. This is encoded by the following type:

SCHOPENHAUER typing for CallCount: (int<7>) -(1/0)-> int

### 6.4 Polymorphic functions

As an example of a simple polymorphic function we examine the resource consumption of the `twice` and `quad` functions:

```haskell
type afct = a -> a;
twice f x = f (f x);
quad f x = let f' = twice f in twice f' x;
```

We obtain the following polymorphic type as heap bound for `quad`:

SCHOPENHAUER typing for HeapBoxed: 
(a -(0/0)-> a) -(0/0)-> a -(5/0)-> a

The resource consumption for `quad` is expressed by the annotation on the top level function type: five heap cells are required to build a closure for `twice f`, which contains one fixed argument. The zeros for the function argument are provisional, the LP-solver has simply chosen a possible solution. When applied to a concrete function, the merged constraints will need to be solved once again. Applying the successor function `succ`, which has a fixed cost of four heap units, then yields the correct typing of:

SCHOPENHAUER typing for HeapBoxed: int -(21/0)-> int

Using a call count metric for the number of calls to the function `succ` in the expression `quad succ 1`, we obtain the following bound. This accurately indicates that `succ` is called precisely four times.

SCHOPENHAUER typing for CallCount: 4, int,0

### 6.5 Destructive pattern matching

A primary motivation for our analysis is to prove bounded resource consumption on resource constrained hardware, such as embedded systems. It is, therefore, important that our analysis can cover techniques that are frequently employed to produce programs with a small resource footprint. We address this issue here, and in our subsequent examples, by i) testing our analysis on programs with destructive pattern matching, and ii) by using a more space-efficient, unboxed representation of the heap. Due to the flexible design of our inference engine, both aspects can be modelled without modifying the engine itself: only the cost tables need to be changed. Our first example to test these features is in-place list reversal:

```haskell
reverse xs = revApp [] xs;
```

### 6.6 An evaluator for expressions

Our final example is an evaluator function for a small subset of the Schopenhauer language itself, using only integer types and without function calls. Even this loop-free version of the language is interesting, since it uses a function to model the environment, and the evaluation of a let-expression modifies this function. The code for the evaluation function is shown in Figure 9).

The analysis of the eval function produces the following heap bound for an unboxed heap model:

SCHOPENHAUER typing for HeapUnboxed:
(int -(0/0)-> int)-(0/0)-> exp[Const:int
|VarOp:int |
|IfOp:int,#,# |
|LetExpOp:int,#,# |
|UnOp:(int -(0/ANY)-> int<ANY>,# |
|BinOp:(int -> int -(0/ANY)-> int<ANY>,#,#)-(0/0)->int

Most notably the analysis distinguishes between different constructors when examining an expression. For constants or variables,
type num = int 16; type val = num;
type var = int 16; type env = var -> val;

data exp = Const val | VarOp var | IfOp var exp exp | LetOp var exp exp |
  | UnOp (val->val) exp | BindOp (val->val->val) exp exp;

_\text{true} = 1; _\text{false} = 0;

\text{eval :: env \rightarrow exp \rightarrow val;}
\text{eval rho (Const n) = n;}
\text{eval rho (VarOp v) = rho v;}
\text{eval rho (IfOp v e1 e2) = if (rho v)==_false then eval rho e2 else eval rho e1;}
\text{eval rho (LetOp v e1 e2) = let x = eval rho e1 ; rho' v' = if v==v' then x else rho v' in eval rho' e2;}
\text{eval rho (UnOp f m) = f (eval rho e1);}
\text{eval rho (BindOp f m n) = f (eval rho e1) (eval rho e2);}

**Figure 9.** Source code of the evaluator example

no heap costs are incurred, since the result value is returned on the stack. For an if-expression, the total costs comprise those for the sub-expressions, represented as \# in the type. No further costs are added for the variable lookup. For a let-expression, a modified environment is defined. This amounts to the construction of a closure with two fixed variables in the heap (9 heap cells in total). Finally, the primitive unary and binary operators do not use any heap cells, since the result value will be produced directly on the stack.

### 6.6.1 Resource parametric recursion

Interestingly, the eval function cannot be analysed under the boxed heap cost model — analysing this function would require polymorphic recursion [15, 26], which we do not support. The second recursive call in the case dealing with LetOp requires a different type, since the annotated type of the first argument has changed. Function rho' is more expensive to execute than rho, because adding the equality operation allocates a boolean value in the boxed heap cost model, and this cannot be amortised against the inputs of rho'. In future, we intend to investigate whether the considerable increase in complexity brought about by polymorphic recursion might be warranted by the possible gain in expressivity.

### 7. Related Work

Our discussion of related work focuses on analyses for strict, higher-order programs. A discussion of analyses for first-order programs is given in another paper [25].

#### 7.1 Amortised Analysis

The focus of most previous work on automatic amortised cost analyses has been on determining the costs of first-order rather than higher-order programs. For example, Hofmann’s linearly-typed functional programming language LFPL [17] uses linear types to determine resource usage in terms of the number of constructors used by a program. First-order LFPL definitions can be computed in bounded space, even in the presence of general recursion. Adding higher-order functions to LFPL raises the expressive power in terms of complexity theory from linear space (LFPL) to exponential time [18]. Hofmann and Jost subsequently described an automatic inference mechanism for heap-space consumption in a functional, first-order language [19], using an amortised cost model. This work uses a deallocation mechanism similar to that we have used here, but is built on a difference metric similar to that of Crary and Weirich [9]. The latter, however, only checks bounds, and does not infer them, as we do.

Taha et al.’s GeHB [36] staged notation automatically generates first-order, heap-bounded LFPL programs from higher-order specifications, but likewise requires the use of non cost-preserving transformations. We are not aware of any other work targeting automatic amortised analysis for higher-order definitions. However, Campbell [6] has studied how the Hofmann/Jost approach can be applied to stack analysis for first-order programs, using “give-back” annotations to return potential. This improves the quality of the analysis results that can be obtained for stack-like metrics. While, in order to keep the presentation clear, we have not done so here, there is no technical reason why “give-back” potential cannot also be applied to the higher-order analysis that we have described. Recent work has aimed to overcome the linearity restriction when analysing first-order programs. For example, Shkaravska et al. aim to extend the amortised cost approach to non-linear bounds using resource functions in the constraints, rather than simple variables [35].

#### 7.2 Sized Types

*Sized types* [22] express bounds on data structure sizes. They are attached to types in the same way as the weights we have used here. The difference is that sized types express bounds on the size of the underlying data structure, whereas our weights are factors of a linear resource bound. Hughes, Pareto and Sabry [22] originally described a *type checking* algorithm for a simple higher-order, non-strict functional language to determine progress in a reactive system. This work was subsequently developed to describe space usage in Embedded ML [21], a strict functional language using regions to control memory usage. Abel [1] extended higher-order sized types to allow higher-kinded types with embedded function spaces. He used this system to formalise termination checking but did not tackle resource consumption in general. A combination of sized types and regions is also being developed by Peña and Segura [32], building on information provided by ancillary analyses on termination and safe destruction. The focus of this work is on determining safety properties rather than resource usage, however. Chin and Khoo [7] introduced a type inference algorithm that is capable of computing size information from high-level program source. Chin et al. [8] presented a heap and a stack analysis for a low-level (assembler) language with explicit (de-)allocation. By inferring path-sensitive information and using symbolic evaluation they are able to infer exact stack bounds for all but one example program.

Vasconcelos and Hammond have independently developed automatic inferences that are capable of deriving cost equations for abstract time- and heap-consumption from unannotated program source expressions based on the inference of sized types for recursive, polymorphic, and higher-order programs [39]. Vasconcelos’ PhD thesis [38] extended these previous approaches by using abstract interpretation techniques to automatically infer linear approximations of the sizes of recursive data types and the stack and heap costs of recursive functions. By including user-defined sizes, it is possible to infer sizes for algorithms on non-linear data structures, such as binary trees.

Finally, Danielsson [10] has recently introduced a library of functions that he claims makes the analysis of a number of purely functional data structures and algorithms almost fully formal. He does this by using a dependent type system to encode information about execution time, and then by combining individual costs into an overall cost using an annotated monad.
7.3 Abstract Interpretations
While having the attraction of being very general, one major disadvantage of abstract interpretations is that analysis results usually depend on the existence of concrete data values. Where they can be applied, impressive results can, however, be obtained even for large commercial applications. For example, AbsInt’s aIT tool [11], and Cousot et al.’s ASTREE system [5] have both been deployed in the design of the software of Airbus Industrie’s Airbus A380. Typically, such tools are limited to non-recursive programs, and may require significant programmer effort to use effectively. We are aware of very little work that considers user-defined higher-order programs, though Le Métayer’s work [28] can handle predefined higher-order functions with known costs, and Benzinger’s work on worst-case complexity analysis for NuPrl [3] similarly supports higher-order functions if the complexity information is provided explicitly. Huelsergen, Larus and Aiken [20] have defined an abstract interpretation of a higher-order, strict language for determining computation costs that depend on the size of data structures. This static analysis is combined with run-time size information to deliver dynamic granularity estimates.

Gulwani, Mehra and Chilimbi’s SPEED system [13] uses a symbolic evaluation approach to calculate non-linear complexity bounds for C/C++ procedures using an abstract interpretation-based invariant generation tool. Precise loop bounds are calculated for 50% of the production loops that have been studied. Unlike our work, they target only first-order programs. Also unlike our work, they consider only time bounds. They do, however, consider non-linear bounds and disjunctive combination of cost information.

The COSTA system [2] performs a fully automatic resource analysis for an object-oriented bytecode language. It produces a closed-form upper bound function over the size of the input. Unlike our system, however, data-dependencies cannot be expressed.

Finally, Gómez and Liu [12] have constructed an abstract interpretation for determining time bounds on higher-order programs. This executes an abstract version of the program that calculates cost parameters, but which otherwise mirrors the normal program execution strategy. Unlike our type-based analysis, the cost of this analysis therefore depends directly on the complexity (or actual values) of the input data and the number of iterations that are performed, does not give a general cost metric for all possible inputs, and will fail to terminate when applied to non-terminating programs.

8. Conclusions and Further Work
By developing a new type-based, resource-generic analysis, we have been able to automatically infer linear bounds on real-time, heap usage, stack usage and number of function calls for strict, higher-order functional programs. The use of amortised costs allows us to determine upper bound cost functions on the overall resource cost of running a program, which take the sizes of program arguments as their inputs. We have extended previous work on amortised-cost-based inference [19, 25] by considering higher-order and polymorphic programs, and by constructing a generic treatment of resource usage through resource tables that can be specialised to different cost metrics and execution models. In this way we achieve a clean separation of the mechanics of inference from the concrete cost metrics that we use. We have demonstrated the flexibility of the resource table approach by building an analysis to determine the number of function calls in a higher-order program. Another key advantage of this separation is that our basic soundness proof applies regardless of the cost metric that we use.

Our results for a range of higher-order programs demonstrate the high quality of the bounds that we can infer. For heap space, we can generally achieve an exact prediction. For worst-case execution time, the bounds we achieve are within 30% of the measured costs. For stack, we generally achieve good results, but occasionally obtain bounds that are linear where the measured costs are constant. This is not inherent to our analysis. For example, Campbell has studied how to improve stack bounds for amortised analysis [6].

Crucial to the usability of our inference is its high degree of efficiency, its full automation and the absence of mandatory programmer annotations. Being built on a high-performance linear program solver our inference is very efficient: for the examples that we have used in this paper, the sizes of the constraint sets vary between 64 and 350 constraints, with the analysis runtime never exceeding 1 second, including constraint solving. However, the restriction to a linear constraint system does impose limits on the range of programs whose costs can be analysed. Precisely classifying the programs that can be analysed is an interesting theoretical question for all forms of cost analysis. While it would be possible to construct a restrictive classification on source-level programs, this would either exclude many programs that are, in fact, analysable, or include many programs that were not analysable. This does not, therefore, seem to be a constructive activity. The most precise classification is that our analysis will succeed exactly where the cost equations have a linear bound. While the inclusion of tail-call optimisations and other cost-simplifying optimisations can actually extend the range of programs that can be costed, the restriction to linearity remains both a theoretical and practical limitation.

8.1 Further Work
Incorporating Sized Types. As we have seen, sized-type systems provide information about data structure sizes. Although they can be used to provide cost information when combined with a suitable constraint inference algorithm [39], they are complementary to the amortised cost approach described here, in that our weights for data structures are multiples of input data structure sizes. Sized type systems should allow these sizes to be inferred statically for a number of common data structures.

Non-Linear Constraints. An extension of the amortised cost based approach to polynomial bounds for a first-order language is ongoing work [16]. We have also begun to investigate whether combining our approach with a sized-type analysis might also allow the inference of super-linear bounds, while still using efficient LP-solver technology (multiple times).

Non-Strictness. Our work is restricted to strict programming languages. An extension of our work to non-strict programming languages, such as Haskell, requires the solution of two technical problems: firstly, we must identify when computations are needed; and, secondly, we must have a formal operational semantics of non-strict evaluation that will allow us to identify resource usage in the way we have done here. We are in the process of producing a cost model and analysis based on Launchbury’s semantics for graph reduction [27], which incorporates notions of evaluation- and sharing-contexts to determine where potentials may be used.

Acknowledgements
We would like to thank our colleagues Pedro Vasconcelos and Hugo Simões (both Universidade do Porto, Portugal) for their fruitful discussion of the rules and proofs, and the anonymous reviewers for their helpful suggestions. This work has been supported by EU Framework VI grants IST-510255 (EmBounded), IST-15905 (Mobius), and RI3-CT-2005-026133 (SCIEnce); and by EPSRC grant EP/F030657/1 (Islay).

---

1 There is, however, significant recent work on determining loop bounds for iterative programs as part of a worst-case execution time analysis, e.g. [29].
References


