"Carbon Credits" for Resource-Bounded Computations using Amortised Analysis

Steffen Jost[1], Hans-Wolfgang Loidl[2], Kevin Hammond[1],
Norman Scaife[3], and Martin Hofmann[2]

[1] St Andrews University, St Andrews, Scotland, UK
jost,kh@cs.st-andrews.ac.uk
[2] Ludwig-Maximilians University, Munich, Germany
hwloidl,mhofmann@tcs.ifi.lmu.de
[3] Université Blaise-Pascal, Clermont-Ferrand, France
Norman.Scaife@univ-bpclermont.fr

Abstract. Bounding resource usage is important for a number of areas, notably real-time embedded systems and safety-critical systems. In this paper, we present a fully automatic static type-based analysis for inferring upper bounds on resource usage for programs involving general algebraic datatypes and full recursion. Our method can easily be used to bound any countable resource, without needing to revisit proofs. We apply the analysis to the important metrics of worst-case execution time, stack- and heap-space usage. Our results from several realistic embedded control applications demonstrate good matches between our inferred bounds and measured worst-case costs for heap and stack usage. For time usage we infer good bounds for one application. Where we obtain less tight bounds, this is due to the use of software floating-point libraries.

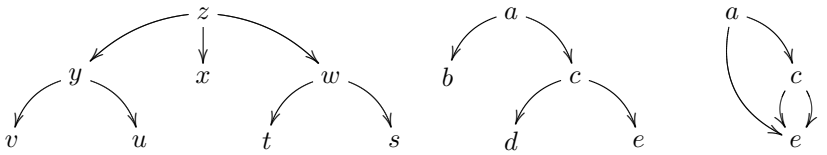
1 Introduction

Programs often produce undesirable "emissions", such as littering the memory with garbage. Our work is aimed at predicting limits on such emissions in advance of execution. "Emissions" here refer to any quantifiable resource that is used by the program. In this paper, we will focus on the key resources of worst-case execution time, heap allocations, and stack usage. Predicting emissions limits is clearly desirable in general, and can be vital in safety-critical, embedded systems.

Our method can be explained by analogy to an attempted countermeasure to global warming: some governments are attempting to reduce industrial pollution by issuing tradable carbon credits. The law then dictates that each CO_2 emission must be offset by expending an appropriate number of carbon credits. It follows that the total amount of emissions is *a priori* bounded by the number of carbon credits that have been previously issued by the authorities. Following this analogy, we will similarly issue credits for computer programs. The "emissions" of each program operation must then be *immediately* justified by spending a corresponding number of credits. The use of "carbon credits" for software analysis does, however, have several advantages over the political situation: i) we can

prove that each and every emission that occurs is legitimate and that it has been properly paid for by spending credits; ii) we have zero bureaucratic overhead, since we use an efficient *compile-time* analysis, there need be no modifications whatever to the original program, and we therefore do not change actual execution costs; and iii) we provide an *automatic static analysis* that, when successful, provides a *guaranteed* upper bound on the number of credits that must be issued initially to ensure that a program can run to completion, rather than using a heuristic to determine the requirements. The amount of credits a program is allowed to spend is specified as part of its type. This allows the absolute number of credits to vary in relation to the actual input, as shown below.

Example: Tree Processing. Consider a tree-processing function *mill*, whose argument has been determined by our analysis to have type $\mathsf{tree}(\mathsf{Node}\langle 7\rangle \mid \mathsf{Leaf}\langle 0.5\rangle)$. Given this type, we can determine that processing the first tree below requires at most $23 = \lfloor 23.5 \rfloor$ credits[†]: 7 credits per node and 0.5 credits for each leaf reference; and that processing either of the other trees requires at most $\lfloor 15.5 \rfloor$ credits, regardless of aliasing.



In fact, the type given by our analysis allows us to easily determine an upper bound on the cost of *mill* for any input tree. For example, for a tree of 27 nodes and 75 leaves, we can compute the credit quota from the type as $7 \cdot 27 + 0.5 \cdot 75 = 226.5$, without needing to consider the actual node or leaf values. The crucial point is that while we are analysing *mill*, our analysis only needs to keep track of this single number. Indeed, the entire dynamic state of the program at any time during its execution *could* be abstracted into such a number, representing the total unspent credits at that point in its execution. Because the number of credits must always be non-negative, this then establishes an upper bound on the total future execution costs (time or space, etc.) of the program. Note that since this includes the cost incurred by all subsequent function calls, recursive or otherwise, it follows that our analysis will also deal with outsourced emissions.

Novel contributions made by this paper: We present a fully automatic compile-time analysis for inferring upper bounds on generic program execution costs, in the form of a new resource-aware type system. The underlying principle used in our automatic analysis is a modified version of Tarjan's *amortised cost analysis* [17], as previously applied to heap allocation by Hofmann and Jost [11]. We prove that the annotated type of terms describes its maximal resource requirement with respect to a given operational semantics. Our analysis becomes automatic by providing type inference for this system and solving any constraints that are generated by using an external linear programming solver.

[†] Note while only whole credits may be spent, fractional credits can be accumulated.

Moreover, we extend previous work:

a) by dealing with arbitrary (recursive) algebraic datatypes;
b) by providing a unified generic approach that presents a soundness proof that holds for arbitrary cost metrics and for many different operational models;
c) by applying the approach to real-world examples, notably worst-case execution time on the Renesas M32C/85U processor.

Section 2 introduces a simple functional language that exhibits our analysis. We consider the soundness of our analysis in Section 5, discuss several example programs in Section 6 and cover related work in Section 7. Section 8 concludes.

2 The Schopenhauer Notation

We illustrate our approach using a simple, strict, purely functional programming language Schopenhauer (named after the German philosopher), which includes recursive datatypes and full recursion, and which is intended as a simple core language for richer notations, such as our own Hume language [9]. Schopenhauer programs comprise a set of one or more (possibly mutually recursive) function declarations. For simplicity, functions and datatypes are monomorphic (we are currently investigating the extension to polymorphic definitions).

$$
\begin{array}{lcll}
\mathit{prog} & ::= & \mathit{varid}_1\ \ \mathit{vars}_1\ \texttt{=}\ \mathit{expr}_1\ \texttt{;}\ \ \ldots\ \ \texttt{;}\ \mathit{varid}_n\ \ \mathit{vars}_n\ \texttt{=}\ \mathit{expr}_n & n \geq 1 \\
\mathit{vars} & ::= & \langle\ \mathit{varid}_1\ \texttt{,}\ \ \ldots\ \ \texttt{,}\ \ \mathit{varid}_n\ \rangle & n \geq 0 \\
\mathit{expr} & ::= & \mathit{const} \mid \mathit{varid} \mid \mathit{varid}\ \ \mathit{vars} \mid \mathit{conid}\ \ \mathit{vars} & \\
 & \mid & \texttt{case}\ \ \mathit{varid}\ \ \texttt{of}\ \ \mathit{conid}\ \ \mathit{vars}\ \ \texttt{->}\ \ \mathit{expr}_1\ \ \texttt{|}\ \ \mathit{expr}_2 & \\
 & \mid & \texttt{let}\ \ \mathit{varid}\ \ \texttt{=}\ \ \mathit{expr}_1\ \texttt{in}\ \mathit{expr}_2 & \\
 & \mid & \texttt{LET}\ \ \mathit{varid}\ \ \texttt{=}\ \ \mathit{expr}_1\ \texttt{IN}\ \mathit{expr}_2 &
\end{array}
$$

The Schopenhauer syntax is fairly conventional, except that: i) we distinguish variable and constructor identifiers; ii) pattern matches are not nested and only allow two branches; iii) we have two forms of let-expression; and iv) function calls are in let-normal form, i.e. arguments are always simple variables. The latter restriction is purely for convenience, since it simplifies the construction of our soundness proof in Section 5 by removing some tedious redundancies. There is no drawback to this since Schopenhauer features two kinds of let-expressions, `let` and `LET`, the former appearing in source programs, and the latter introduced as a result of internal translation. Both forms have identical semantics but they may have differing operational costs, depending on the desired operational model and on the translation into let-normal form. Since the ordinary let-expression usually incurs some overhead for managing the created reference, it cannot be used to transform expressions into let-normal form in a cost-preserving manner.

3 Schopenhauer Operational Semantics

Our operational semantics (Figure 1) is fairly standard, using a program signature Σ to map function identifiers to their defining bodies. The interesting

feature of our semantics is that it is instrumented by a (non-negative) resource counter, which *defines* the cost of each operation. This counter is intended to *measure* execution costs, with the execution being stuck if the counter becomes negative. We will prove later that our analysis determines an upper bound on the smallest starting value for this counter, and so prevents this from happening.

An environment, $\mathcal{V}$, is a mapping from variables to locations, denoted by ℓ. A heap, $\mathcal{H}$, is a partial map from locations to values w. $\mathcal{H}[\ell \mapsto w]$ denotes a heap that maps ℓ to value w and otherwise acts as $\mathcal{H}$. Values are simple tuples whose first component is a flag that indicates the kind of the value, e.g. $(\mathsf{bool}, \mathtt{tt})$ for the boolean constant *true*, $(\mathsf{int}, 42)$ for the integer 42, etc. The judgement

$$\mathcal{V}, \mathcal{H} \vdash^{n}_{n'} e \rightsquigarrow \ell, \mathcal{H}'$$

then means that under the initial environment $\mathcal{V}$ and heap $\mathcal{H}$, the expression e evaluates to location ℓ (all values are boxed) and post-heap $\mathcal{H}'$, provided at least n units of the selected resource are available before the computation. Furthermore, n' units are available after the computation. Hence, for example, $\mathcal{V}, \mathcal{H} \vdash^{3}_{1} e \rightsquigarrow \ell, \mathcal{H}'$ simply means that 3 resource units are sufficient for evaluating e, and that exactly one is unused after the computation. This one unit might, or might not, have been used temporarily. We will simply write $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \ell, \mathcal{H}'$ if there exists n, n' such that $\mathcal{V}, \mathcal{H} \vdash^{n}_{n'} e \rightsquigarrow \ell, \mathcal{H}'$.

Cost Parameters. The operational rules involve a number of constants which serve as parameters for an arbitrary cost model. For example, the constant `KmkInt` denotes the cost for an integer constant. If an integer occupies two heap units, and we are interested in heap usage, we set this constant to two; if each pointer occupies a single stack unit, and we are interested in stack usage, we set this value to one; and so on. Some cost parameters are parametrised to allow better precision to be obtained, e.g. for execution time, the cost of matching a constructor may vary according to the number of arguments it has.

It is important to note that our soundness proof does not rely on any specific values for these constants. Any suitable values may be used according to the required operational cost model. While it would be possible to expand the cost parameters to vectors, in order to deal with several simultaneous metrics, for example, this would require similar vector annotations in our type systems, requiring a high notational overhead, without making a new contribution.

4 Schopenhauer Type Rules

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

$$T ::= \mathsf{int} \mid X \mid \mu X.\{\ c_1{:}(q_1, \overrightarrow{T_1}) \mid \ldots \mid c_k{:}(q_k, \overrightarrow{T_k})\ \} \mid \overrightarrow{T} \xrightarrow[p']{p} T'$$

where X is a type variable, $c_i \in \mathsf{Constrs}$ are constructor labels; p, p', q_i are either non-negative rational constants or resource variables belonging to the infinite

$$\frac{n \in \mathbb{Z} \qquad \ell \notin \mathrm{dom}(\mathcal{H})}{\mathcal{V}, \mathcal{H} \;\vdash^{\frac{q' + \texttt{KmkInt}}{q'}}\; n \rightsquigarrow \ell, \mathcal{H}\big[\ell \mapsto (\mathsf{int}, \mathsf{n})\big]} \qquad \frac{\mathcal{V}(x) = \ell}{\mathcal{V}, \mathcal{H} \;\vdash^{\frac{q' + \texttt{KpushVar}}{q'}}\; x \rightsquigarrow \ell, \mathcal{H}}$$

$$\frac{\Sigma(\mathit{fid}) = \big(e_f; y_1, \ldots, y_k; C; \psi\big) \qquad [y_1 \mapsto \mathcal{V}(x_1), \ldots, y_k \mapsto \mathcal{V}(x_k)], \mathcal{H} \vdash^{\frac{q - \texttt{Kcall}(k)}{q' + \texttt{Kcall}'(k)}} e_f \rightsquigarrow \ell, \mathcal{H}'}{\mathcal{V}, \mathcal{H} \;\vdash^{\frac{q}{q'}}\; \mathit{fid}\ \langle x_1, \cdots, x_k \rangle \rightsquigarrow \ell, \mathcal{H}'}$$

$$\frac{c \in \mathsf{Constrs} \qquad \ell \notin \mathrm{dom}(\mathcal{H}) \qquad k \geq 0 \qquad w = \big(\mathsf{constr}_c, \mathcal{V}(x_1), \ldots, \mathcal{V}(x_k)\big)}{\mathcal{V}, \mathcal{H} \;\vdash^{\frac{q' + \texttt{KCons}(k)}{q'}}\; c\ \langle x_1, \ldots, x_k \rangle \rightsquigarrow \ell, \mathcal{H}[\ell \mapsto w]}$$

$$\frac{\mathcal{H}(\mathcal{k}) = \big(c, \mathcal{k}_1, \ldots, \mathcal{k}_k\big) \qquad \mathcal{V}[y_1 \mapsto \mathcal{k}_1, \ldots, y_k \mapsto \mathcal{k}_k], \mathcal{H} \;\vdash^{\frac{q - \texttt{KCaseT}(k)}{q' + \texttt{KCaseT}'(k)}}\; e_1 \rightsquigarrow \ell, \mathcal{H}'}{\mathcal{V}[x \mapsto \mathcal{k}], \mathcal{H} \;\vdash^{\frac{q}{q'}}\; \texttt{case}\ x\ \texttt{of}\ c\ \langle y_1, \ldots, y_k \rangle\ \texttt{->}\ e_1 \,|\, e_2 \rightsquigarrow \ell, \mathcal{H}'}$$

$$\frac{\mathcal{H}\big(\mathcal{V}(x)\big) \neq \big(c, \mathcal{k}_1, \ldots, \mathcal{k}_k\big) \qquad \mathcal{V}, \mathcal{H} \;\vdash^{\frac{q - \texttt{KCaseF}(k)}{q' + \texttt{KCaseF}'(k)}}\; e_2 \rightsquigarrow \ell, \mathcal{H}'}{\mathcal{V}, \mathcal{H} \;\vdash^{\frac{q}{q'}}\; \texttt{case}\ x\ \texttt{of}\ c\ \langle y_1, \ldots, y_k \rangle\ \texttt{->}\ e_1 \,|\, e_2 \rightsquigarrow \ell, \mathcal{H}'}$$

$$\frac{\mathcal{V}, \mathcal{H} \;\vdash^{\frac{q_1 - \texttt{KLet1}}{q_2}}\; e_1 \rightsquigarrow \ell_1, \mathcal{H}_1 \qquad \mathcal{V}[x \mapsto \ell_1], \mathcal{H}_1 \;\vdash^{\frac{q_2 - \texttt{KLet2}}{q' + \texttt{KLet3}}}\; e_2 \rightsquigarrow \ell_2, \mathcal{H}_2}{\mathcal{V}, \mathcal{H} \;\vdash^{\frac{q_1}{q'}}\; \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \rightsquigarrow \ell_2, \mathcal{H}_2}$$

Note that the rule for `LET ... IN` *is identical to that for* `let ... in` *above, except in replacing constants* `KLet1`*,* `KLet2` *and* `KLet3` *with* `KLET1`*,* `KLET2` *and* `KLET3`*, respectively.*

Fig. 1. Schopenhauer Operational Semantics

set of *resource variables* CV ranging over $\mathbb{Q}^+$; and we write $\overrightarrow{T}$ for $\langle\ T_1\ \ldots\ T_n\ \rangle$ where $n \geq 0$. For convenience, we extend all operators pointwise when used in conjunction with the vector notation i.e. $\overrightarrow{A} = \overrightarrow{B}$ stands for $\forall i\,.\,A_i = B_i$. Let ψ, ϕ, ξ range over sets of linear inequalities over resource variables. We write $\psi \Rightarrow \phi$ to denote that ψ entails ϕ, i.e. all valuations $v : \mathsf{CV} \to \mathbb{Q}^+$ which satisfy ψ also satisfy all constraints in ϕ. We write $v \Rightarrow \phi$ if the valuations satisfies all constraints. We extend valuations to types and type contexts in the obvious way. Valuations using non-negative real numbers are permissible, but rational annotations are of most interest since they allow the use of in-place update, as described in [11].

Algebraic datatypes are defined as usual, except that the type carries a resource variable for each constructor. The type rules for Schopenhauer then govern how credits are associated with runtime values of an annotated type. The number of credits associated with a runtime value w of type A is denoted by $\Phi^v_{\mathcal{H}}(w : A)$, formalised in Definition 2 in Section 5. Intuitively, it is the sum over all constructor nodes reachable from w, where the weight of each constructor in the sum is determined by the type A. As we have seen in the tree/mill example in the introduction, a single constructor node may contribute many times to this

sum, possibly each time with a different weight, determined by the type of the reference used to access it. While this definition is paramount to our soundness proof, any practical application only requires the computation of this number for the initial memory configuration, for which it can always be easily computed. It is easy to see, for example, that the number of credits associated with a list of integers having the type $\mu X.\{\texttt{Nil} : (z_0, \langle\rangle) | \texttt{Cons} : (z, \langle \mathsf{int}, X\rangle)\}$ is simply $z_0 + n \cdot z$, where n is the length of the list. We naturally extend this definition to environments and type contexts by summation over the domain of the context.

We can now formulate the type rules for Schopenhauer (which are standard apart from the references to cost and resource variables). Let Γ denote a typing context mapping identifiers to annotated Schopenhauer types. The Schopenhauer typing judgement $\Gamma \vdash^{q}_{q'} e : A \mid \phi$ then reads "for all valuations v that satisfy all constraints in ϕ, the expression e has Schopenhauer type $v(A)$ under context $v(\Gamma)$; moreover evaluating e under environment $\mathcal{V}$ and heap $\mathcal{H}$ requires at most $v(q) + \Phi^{v}_{\mathcal{H}}(\mathcal{V} : \Gamma)$ credits and leaves at least $v(q') + \Phi^{v}_{\mathcal{H}}(\mathcal{V} : \Gamma)$ credits available afterwards". The types thus bound resource usage and we will formalise the above statement as our main theorem (Theorem 1), which requires as a precondition that the context, environment and heap are all mutually consistent.

A Schopenhauer program is a mapping Σ, called the *signature* of the program, which maps function identifiers *fid* belonging to the set Var to a quadruple consisting of: i) a term defining the function's body; ii) an ordered list of argument variables; iii) a type; and iv) a set of constraints involving the annotations of the type. Since the signature Σ is fixed for each program to be analysed, for simplicity, we omit it from the premises of each type rule. A Schopenhauer program is *well-typed* if and only if for each identifier *fid*

$$\Sigma(\mathit{fid}) = (e_{\mathit{fid}}; y_1, \ldots, y_a; \langle A_1, \ldots, A_a\rangle \xrightarrow[p']{p} C; \psi) \qquad \Longrightarrow$$
$$y_1{:}A_1, \ldots, y_a{:}A_a \vdash^{p - \texttt{Kcall}(a)}_{p' + \texttt{Kcall}'(a)} e_{\mathit{fid}} : C \mid \psi$$

Basic Expressions. Primitive terms have fixed costs. Requiring all available credits to be spent simplifies proofs, without imposing any restrictions, since a substructural rule allows costs to be relaxed where required.

$$\frac{n \in \mathbb{Z}}{\varnothing \vdash^{\texttt{KmkInt}}_{0} n : \mathsf{int} \mid \emptyset} \text{(Int)} \qquad \frac{}{x{:}A \vdash^{\texttt{KpushVar}}_{0} x : A \mid \emptyset} \text{(Var)}$$

Function Call. The cost of function application is represented by the constants $\texttt{Kcall}(k)$ and $\texttt{Kcall}'(k)$, which specify, respectively, the absolute costs of setting up before the call and clearing up after the call. In addition, each argument may carry further credits, depending on its type, to pay for the function's execution. For simplicity, we have prohibited zero-arity function calls.

$$\frac{\Sigma(\mathit{fid}) = \left(e_{\mathit{fid}}; y_1, \ldots, y_k; \langle A_1, \ldots, A_k\rangle \xrightarrow[p']{p} C; \psi\right) \qquad k \geq 1}{y_1{:}A_1, \ldots, y_k{:}A_k \vdash^{p + \texttt{Kcall}(k)}_{p' - \texttt{Kcall}'(k)} \mathit{fid}\, \langle y_1, \cdots, y_k\rangle : C \mid \psi} \text{(App)}$$

Algebraic Datatypes. The CONSTR rule plays a crucial role in our annotated type system, since this is where available credits may be associated with a new data structure. Credits cannot be used while they are associated with data.

$$\frac{c \in \mathsf{Constrs} \qquad C = \mu X.\{\cdots \mid c : (p, \langle B_1, \ldots, B_k\rangle) \mid \cdots\} \qquad A_i = B_i\big[C \,/\, X\big] \text{ (for } i = 1, \ldots, k)}{x_1{:}A_1, \ldots, x_k{:}A_k \;\vdash^{p \,+\, \mathtt{KCons}(k)}_{0}\; c\ \langle x_1, \ldots, x_k\rangle : C \mid \emptyset} \quad (\textsc{Constr})$$

The dual to the above rule is the CASE rule; the only point where credits associated with data can be released again. This is because this is the only point where we know about the actual constructor that is referenced by a variable, i.e. where we know whether a variable of a list type refers to a non-empty list, etc.

$$\frac{\begin{array}{c} c \in \mathsf{Constrs} \qquad \Gamma, y_1{:}B_1[A/X], \ldots, y_k{:}B_k[A/X] \;\vdash^{q_t}_{q'_t}\; e_1 : C \mid \psi_t \\ A = \mu X.\{\cdots \mid c : (p, \langle B_1, \ldots, B_k\rangle) \mid \cdots\} \qquad \Gamma, x{:}A \;\vdash^{q_f}_{q'_f}\; e_2 : C \mid \psi_f \\ \psi = \left\{ \begin{array}{rr} p + q = q_t + \mathtt{KCaseT}(k) & q'_t = q' + \mathtt{KCaseT'}(k) \\ q = q_f + \mathtt{KCaseF}(k) & q'_f = q' + \mathtt{KCaseF'}(k) \end{array} \right\} \end{array}}{\Gamma, x{:}A \;\vdash^{q}_{q'}\; \mathtt{case}\ x\ \mathtt{of}\ c\ \langle y_1, \ldots y_k\rangle\ \mathtt{->}\ e_1 \,|\, e_2 : C \mid \psi \cup \psi_t \cup \psi_f} \quad (\textsc{Case})$$

Let-bindings. The two rules for let-expressions are the only ones that thread credits sequentially through the sub-rules. As in the operational semantics rules, the type rule for `LET ... IN` is identical to that below, except in replacing `KLet1`, `KLet2`, `KLet3` with `KLET1`, `KLET2`, `KLET3`, respectively. Note that we use a comma for the disjoint union of contexts throughout, hence duplicated uses of variables must be introduced through the SHARE rule, described in the next paragraph.

$$\frac{\begin{array}{c} \Gamma \;\vdash^{q_1}_{q'_1}\; e_1 : A_1 \mid \psi_1 \qquad \Delta, x{:}A_1 \;\vdash^{q_2}_{q'_2}\; e_2 : A_2 \mid \psi_2 \\ \psi_0 = \left\{ q_1 = q - \mathtt{KLet1} \quad q_2 = q'_1 - \mathtt{KLet2} \quad q' = q'_2 - \mathtt{KLet3} \right\} \end{array}}{\Gamma, \Delta \;\vdash^{q}_{q'}\; \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2\ :\ A_2 \mid \psi_0 \cup \psi_1 \cup \psi_2} \quad (\textsc{Let})$$

Substructural rules. We use explicit substructural type rules. Apart from simplifying proofs, the SHARE rule makes multiple uses of a variable explicit. Unlike in a strictly linear type system, variables can be used several times. However, the types of all occurrences must "add up" in such a way that the total credit associated with all occurrences is no larger than the credit initially associated with the variable. It is the job of the SHARE rule to track multiple occurrences, and it is the job of the $\curlyvee$-function to apportion credits.

$$\frac{\Gamma, x{:}B \;\vdash^{q}_{q'}\; e : C \mid \phi \quad \psi \vdash A <: B}{\Gamma, x{:}A \;\vdash^{q}_{q'}\; e : C \mid \phi \cup \psi} \quad (\textsc{Supertype}) \qquad \frac{\Gamma \;\vdash^{q}_{q'}\; e : D \mid \phi \quad \psi \vdash D <: C}{\Gamma \;\vdash^{q}_{q'}\; e : C \mid \phi \cup \psi} \quad (\textsc{Subtype})$$

$$\frac{\Gamma \vdash^{p}_{p'} e : A \mid \psi \qquad \phi \Rightarrow \psi \cup \{q \geq p, q - p \geq q' - p'\}}{\Gamma \vdash^{q}_{q'} e : A \mid \phi}\ (\textsc{Relax}) \qquad \frac{\Gamma \vdash^{q}_{q'} e : C \mid \psi}{\Gamma, x{:}A \vdash^{q}_{q'} e : C \mid \phi}\ (\textsc{Weak})$$

$$\frac{\Gamma, x{:}A_1, y{:}A_2 \vdash^{q}_{q'} e : C \mid \phi}{\Gamma, z{:}A \vdash^{q}_{q'} e[z/x, z/y] : C \mid \phi \cup \curlyvee(A \,|\, A_1, A_2)}\ (\textsc{Share})$$

The ternary function $\curlyvee(A \,|\, B, C)$ is only defined for structurally-identical type-triples which differ in at most the names of resource variables. It 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. The crucial property of this function is expressed in Lemma 4. For example,

$$A = \mu X.\{\texttt{Nil}{:}(a,\langle\rangle) \mid \texttt{Cons}{:}(d,\langle \mathsf{int}, X\rangle)\} \quad B = \mu X.\{\texttt{Nil}{:}(b,\langle\rangle) \mid \texttt{Cons}{:}(e,\langle \mathsf{int}, X\rangle)\}$$
$$C = \mu X.\{\texttt{Nil}{:}(c,\langle\rangle) \mid \texttt{Cons}{:}(f,\langle \mathsf{int}, X\rangle)\} \quad \curlyvee(A \,|\, B, C) = \{a = b + c, d = e + f\}$$

Subtyping. The type rules for subtyping depend on another inductively defined relation $\xi \vdash A <: B$ between two types A and B, relative to constraint set ξ. For any fixed constraint set ξ, the relation is both *reflexive* and *transitive*.

$$\frac{}{\xi \vdash A <: A} \qquad \frac{\text{for all } i \text{ holds } \xi \Rightarrow \{p_i \geq q_i\} \text{ and } \xi \vdash \overrightarrow{A_i} <: \overrightarrow{B_i}}{\xi \vdash \mu X.\{\cdots \mid c_i{:}(p_i, \overrightarrow{A_i}) \mid \cdots\} <: \mu X.\{\cdots \mid c_i{:}(q_i, \overrightarrow{B_i}) \mid \cdots\}}$$

$$\frac{\xi \Rightarrow \{p \leq q\,,\, p' \geq q'\} \qquad \xi \vdash \overrightarrow{B} <: \overrightarrow{A} \qquad \xi \vdash C <: D}{\xi \vdash \overrightarrow{A} \xrightarrow[p']{p} C <: \overrightarrow{B} \xrightarrow[q']{q} D}$$

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. The standard typing derivation is then traversed *once* to gather all the constraints. Since we found this easier to implement, substructural rules have been amalgamated with the other typing rules. Because all types have been annotated with fresh resource variables, subtyping is required throughout. Subtyping simply generates the necessary inequalities between corresponding resource variables, and will always succeed, since it is only permitted between types that differ at most in their resource annotations. Likewise, the Relax rule is applied at each step, using the minimal constraints shown in the rule. (However, inequalities are turned into equalities using explicit slack variables, in order to minimise wasted credits.) The Weak and Share rule are applied based on the free variables of the subterms.

In the final step, the constraints that have been gathered are fed to an LP-solver [2]. Any solution that is found is presented to the user in the form of an annotated type and a human-readable closed cost formula. In practice, we have found that these constraints can be easily solved by a standard LP-solver running on a typical laptop or desktop computer, partly because of their structure [11]. Since only a single pass over the program code is needed to construct these constraints, this leads to a highly efficient analysis.

5 Soundness of the Analysis

We now *sketch* the important steps for proving the main theorem. We first formalise the notion of a "well-formed" machine state, which simply says that for each variable, the type assigned by the typing context agrees with the actual value found in the heap location assigned to that variable by the environment. This is an essential invariant for our soundness proof.

Definition 1. *A memory configuration consisting of heap $\mathcal{H}$ and stack $\mathcal{V}$ is* well-formed *with respect to context Γ and valuation v, written $\mathcal{H} \vDash_v \mathcal{V} : \Gamma$, if and only if $\mathcal{H} \vDash_v \mathcal{V}(x) : \Gamma(x)$ can be derived for all variables $x \in \Gamma$.*

$$\frac{\mathcal{H}(\ell) = (\mathsf{int}, \mathsf{n}) \qquad \mathsf{n} \in \mathbb{Z}}{\mathcal{H} \vDash_v \ell : \mathsf{int}} \qquad \frac{\mathcal{H} \vDash_v \ell : A \qquad \exists \phi \,.\, v \Leftarrow \phi \wedge \phi \vdash A <: B}{\mathcal{H} \vDash_v \ell : B}$$

$$\frac{\mathcal{H}(\ell) = (\mathsf{constr}_c, \ell_1, \ldots, \ell_k) \qquad C = \mu X.\{\cdots \mid c : (q, \langle B_1, \ldots, B_k \rangle) \mid \cdots\} \qquad \forall i \in \{1, \ldots, k\} \,.\, \mathcal{H} \vDash_v \ell_i : B_i\big[C \,/\, X\big]}{\mathcal{H} \vDash_v \ell : C}$$

Lemma 1. *If $\mathcal{H} \vDash_v \mathcal{V} : \Gamma$ and $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \ell, \mathcal{H}'$ then also $\mathcal{H}' \vDash_v \mathcal{V} : \Gamma$.*

We remark that one might wish to prove a stronger statement to the effect that the result ℓ of the valuation is also well-formed given that the expression e was typeable. Unfortunately such a statement cannot be proven on its own and must necessarily be interwoven in Theorem 1.

We now formally define how credits are associated with runtime values, following our intuitive description from the previous section.

Definition 2. *If $\mathcal{H} \vDash_v \ell : A$ holds, then $\Phi^v_{\mathcal{H}}(\ell{:}A)$ denotes the number of credits associated with location ℓ for type A in heap $\mathcal{H}$ under valuation v. This value is always zero, except when A is a recursive datatype in which case it is recursively defined by*

$$\Phi^v_{\mathcal{H}}(\ell{:}A) = v(q) + \sum_i \Phi^v_{\mathcal{H}}(\ell_i{:}B_i[A/X])$$

when $A = \mu X.\{\cdots \mid c{:}(q, \langle B_1, \ldots, B_k \rangle) \mid \cdots\}$ and $\mathcal{H}(\ell) = (\mathsf{constr}_c, \ell_1, \ldots, \ell_k)$. We extend to contexts by $\Phi^v_{\mathcal{H}}(\mathcal{V} : \Gamma) = \sum_{x \in \mathrm{dom}(\Gamma)} \Phi^v_{\mathcal{H}}\big(\mathcal{V}(x){:}v\big(\Gamma(x)\big)\big)$

Subsumption cannot increase the number of associated credits.

Lemma 2. *If $\mathcal{H} \vDash_v \ell : A$ and $\phi \vdash A <: B$ holds and v is a valuation satisfying ϕ, then $\Phi^v_{\mathcal{H}}(\ell{:}A) \geq \Phi^v_{\mathcal{H}}(\ell{:}B)$*

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

Lemma 3. *If $\curlyvee(A \,|\, B, C) = \phi$ holds then also $\phi \vdash A <: B$ and $\phi \vdash A <: C$.*

The number of credits attached to any value of a certain type is always linearly shared between the two types introduced by sharing. In other words, the overall amount of available credits does not increase by using Share.

Lemma 4. *If the judgements* $\mathcal{H} \vDash_v \ell{:}A$ *and* $\curlyvee(A\,|B, C\,) = \phi$ *hold and* v *satisfies the constraint set* ϕ *then* $\Phi^v_{\mathcal{H}}(\ell{:}A) = \Phi^v_{\mathcal{H}}(\ell{:}B) + \Phi^v_{\mathcal{H}}(\ell{:}C)$*. Moreover, for* $A = B$ *and* $A = C$*, it follows that* $\Phi^v_{\mathcal{H}}(\ell{:}A) = 0$ *also holds.*

We can now formulate the main theorem (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 statements hold*

$$\Gamma \vdash^{q}_{q'} e{:}A \mid \phi \tag{5.1}$$

$$\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \ell, \mathcal{H}' \tag{5.2}$$

$$v : \mathsf{CV} \to \mathbb{Q}^+, \ \textit{satisfying } \phi \tag{5.3}$$

$$\mathcal{H} \vDash_v \mathcal{V}{:}v(\Gamma) \tag{5.4}$$

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

$$m \geq v(q) + \Phi^v_{\mathcal{H}}\big(\mathcal{V} : v(\Gamma)\big) + r \tag{5.5}$$

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

$$\mathcal{V}, \mathcal{H} \vdash^{m}_{m'} e \rightsquigarrow \ell, \mathcal{H}' \tag{5.6}$$

$$m' \geq v(q') + \Phi^v_{\mathcal{H}'}\big(\ell{:}v(A)\big) + r \tag{5.7}$$

$$\mathcal{H}' \vDash_v \ell{:}v(A) \tag{5.8}$$

The proof is by induction on the lengths of the derivations of (5.2) and (5.1) 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 the case of function application, which increases the length of the typing derivation. On the other hand, the length of the derivation for the term evaluation never increases, but may remain unchanged where the final step of the typing derivation was obtained by a substructural rule. In these cases, the length of the typing derivation does decrease, 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 "hide" excess credits when applying the induction hypothesis for subexpressions, which leaves those credits untouched. We show one case to provide some flavour of the overall proof:

$\rightsquigarrow$ Case Succeed**:** By the induction hypothesis, we obtain for all $m_0 \geq v(q_t) + \Phi^v_{\mathcal{H}}(\mathcal{K}_i{:}B_i[A/X]) + \Phi^v_{\mathcal{H}}(\mathcal{V}{:}\Gamma) + r$ a suitable $m_0' \geq v(q_t') + \Phi^v_{\mathcal{H}}(\ell{:}C) + r$ such that e_1 evaluates under the annotated operational semantics with m_0 and m_0'. Observe that we have $\Phi^v_{\mathcal{H}}(\mathcal{K}{:}A) = v(p) + \sum_i \Phi^v_{\mathcal{H}}(\mathcal{K}_i{:}B_i[A/X])$ and $v(p) + v(q) = v(q_t) + \texttt{KCaseT}(k)$ and $v(q_t') = v(q') + \texttt{KCaseT}'(k)$. Therefore $m = m_0 + \texttt{KCaseT}(k) \geq v(q) + v(p) + \Phi^v_{\mathcal{H}}(\mathcal{K}_i{:}B_i[A/X]) + \Phi^v_{\mathcal{H}}(\mathcal{V}{:}\Gamma) + r = v(q) + \Phi^v_{\mathcal{H}}(\mathcal{K}{:}A) + \Phi^v_{\mathcal{H}}(\mathcal{V}{:}\Gamma) + r$ and $m' = m_0' - \texttt{KCaseT}'(k) \geq val(q') + \Phi^v_{\mathcal{H}}(\ell{:}C) + r$ as required.

Constant	Stack	Heap	WCET[‡]	Constant	Stack	Heap	WCET[‡]
`KmkInt`	1	2	83	`KCaseF`(k)	0	0	205
`KpushVar`	1	0	39	`KCaseF′`(k)	0	0	$56 + \textsc{Ret}$
`Kcall`(k)	$4 + k$	0	142	`KLet1`	1	0	142
`Kcall′`(k)	$-(4 + k)$	0	$53 + \textsc{Ret}$	`KLet2`	0	0	0
`KCons`(k)	$1 - k$	$2 + k$	$107 + 54k$	`KLet3`	-1	0	$3 + \textsc{Ret}$
`KCaseT`(k)	$k - 1$	0	$301 + 80k$	`KLET1`	0	0	0
`KCaseT′`(k)	$-k$	0	$65 + \textsc{Ret}$	`KLET2`	0	0	0
				`KLET3`	0	0	0

Table 1. Table of Resource Constants for Stack, Heap and Time

	N = 1			N = 2			N = 3			N = 4			N = 5		
	Heap	Stack	Time	Heap	Stack	Time	Heap	Stack	Time	Heap	Stack	Time	Heap	Stack	Time
revApp															
Analysis	14	25	2440	24	26	3596	34	27	4752	44	28	5908	54	29	7064
Measured	14	24	1762	24	24	2745	34	24	3725	44	24	4707	54	24	5687
Ratio	1	1.04	1.39	1	1.08	1.31	1	1.13	1.27	1	1.17	1.26	1	1.21	1.24
flatten															
Analysis	17	24	3311	34	34	6189	51	44	9067	68	54	11945	85	64	14823
Measured	17	24	2484	34	33	4372	51	43	6260	68	43	8148	85	43	10036
Ratio	1	1.00	1.33	1	1.03	1.42	1	1.02	1.45	1	1.26	1.47	1	1.49	1.48

Table 2. Measurement and Analysis Results for Tree-Flattening

6 Example Cost Analysis Results

In this section, we compare the bounds inferred by our analysis with concrete measurements for one operational model. Heap and stack results were obtained by instrumenting the generated code. Time measurements were obtained from unmodified code on a 32MHz Renesas M32C/85U embedded micro-controller with 32kB RAM. The cost parameters used for this operational model are shown in Table 1. The time metrics were obtained by applying AbsInt GmbH's **aiT** tool [8] to the compiled code of individual abstract machine instructions.

Our first example is a simple tree-flattening function and its auxiliary function, reverse-append. The *heap space consumption* inferred by our analysis is encoded as the following annotated type

```
SCHOPENHAUER typing for HumeHeapBoxed:
0, (tree[Leaf<10>:int|Node:#,#]) -(2/0)-> list[C:int,#|N] ,0
```

which reads, "for a given tree with l leaves, the heap consumption is $10l + 2$." Table 2 compares analysis and measurement results. As test input, we use

[‡] Returns are performed through a fixed size table. On the Renesas M32C/85U this is compiled to a series of branches and the WCET therefore depends on the number of calling points in the program. We have $\textsc{Ret} = \max\big(116, 51 + 15 \cdot (\#\text{ReturnLabels})\big)$.

balanced trees with $N = 1 \ldots 5$ leaves. Heap prediction is an exact match for the measured results. Stack prediction follows a linear bound over-estimating actual costs, which are logarithmic in general, using a tail-recursive reverse function. This linear bound is due to the design of our analysis, which cannot infer re-use of stack space in all cases (Campbell has described an extension to our approach [4] that may improve this). The predicted time costs are between 33% and 48% higher than the measured worst-cases.

6.1 Control Application: Inverted Pendulum

Our next example is an inverted pendulum controller. This implements a simple, real-time control engineering problem. A pendulum is hinged upright at the end of a rotating arm. Both rotary joints are equipped with angular sensors, which are the inputs for the controller (arm angle θ and pendulum angle α). The controller should produce as its output the electric potential for the motor that rotates the arm in such a way that the pendulum remains in an upright position.

The Hume code comprises about 180 lines of code, which are translated into about 800 lines of Schopenhauer code for analysis. The results for heap and stack usage (upper part of Table 3) show exact matches in both cases. For time we have measured the best-case (36118), worst-case (47635) and average number of clock cycles (42222) required to process the controlling loop over 6000 iterations during an actual run, where the Renesas M32C/85U actually controlled the inverted pendulum. Compared to the worst-case execution time (WCET) bound given by our automated analysis (63678) we have a margin of 33.7% between the predicted WCET and the worst measured run. The hard real-time constraint on this application is that the pendulum controller can only be made stable with a loop time of less than about 10ms. The measured loop time is 1.488ms, while our predicted loop time would be 1.989ms, showing that our controller is guaranteed to be fast enough to successfully control the pendulum under all circumstances.

6.2 Control Application: Biquadratic Filter

Our final control application is a second-order recursive linear filter, a biquadratic filter, so named because the transfer function is a ratio of two quadratic polynomials. It is commonly used in audio work and can be used to implement low-pass, high-pass, band-pass and notch filters. Well-known algorithms exist for computing filter coefficients from the desired gain, centre frequency and sample rate [14].

The lower part of Table 3 compares analysis results against measured costs for the components of the biquadratic filter. For heap, we obtain exact bounds for all but one box. For stack, we find a close match of the bounds with the measured values (within 12% units). For time, however, the bounds are significantly worse than the measured values. This is mainly due to the heavy use of floating-point operations in this application, which are implemented in software on the Renesas M32C/85U. This means that the WCET bounds for the primitive operations in the analysis cost table are already very slack.

Box	Analysis			Measured			Ratio		
	Heap	Stack	Time	Heap	Stack	Time	Heap	Stack	Time
pendulum									
control	299	93	63678	299	93	47635	1.00	1.00	1.34
biquad									
biquad	33	32	10330	33	32	5848	1.00	1.00	1.77
compute_filter	73	62	26392	73	59	13176	1.00	1.05	2.00
compute_params	40	38	47307	40	34	16107	1.00	1.12	2.94
scale_in	14	15	3919	10	15	1844	1.40	1.00	2.13
scale_out	33	33	16044	33	33	5920	1.00	1.00	2.71

Table 3. Comparison of Results for Pendulum and Biquad Filter Applications

The critical path through the system comprises the `scale_in`, `biquad` and `scale_out` boxes. If we sum their bounds, we obtain a total of 30293 clock cycles, or 947μs. This gives us a sample rate of about 1.056kHz, obviously well short of audio sampling rates of about 48kHz. However, this is a concrete guarantee for the application, and it tells us at an early stage, *without any measurement*, that the hardware we are using is not fast enough for real-time audio processing. The heap and stack bounds confirm, however, that we can fit static and dynamic memory on the board. The components are executed sequentially, so the largest component governs the dynamic memory for the entire system: this is 73 *heap* + 62 *stack* cells for the `compute_filter` box, or a maximum of 540 bytes of memory, well within our design maximum of 32kB.

One distinctive feature of our analysis is that it attributes costs to individual data type constructors. Therefore, our bounds are not only size-dependent, as would be expected, but more generally data-dependent. For a worst-case execution time analysis of `compute_filter`, we produce the following explanation:

```
Worst-case Time-units required to compute box compute_filter once:
  359 + 9374*X1 + 16659*X2 + 16123*X3 + 14570*X4    where
        X1 = one if 1. wire is live, zero if the wire is void
        X2 = number of "BPF" nodes at 1. position
        X3 = number of "HPF" nodes at 1. position
        X4 = number of "LPF" nodes at 1. position
```

In particular, since `BPF`, `HPF` and `LPF` are elements of an enumeration type, selecting a band pass, high pass or low pass filter, respectively, we know that only one of the three costs attached to these constructors (16659, 16123 or 14570) will apply. Furthermore, in the case where a null filter is selected, by providing `NULLF` as input, none of these three costs applies and the time bound for this case is, therefore, 9733 clock cycles. Being data-dependent, this parametrised bound is more accurate than the worst-case bound specified in Table 3, where we take the worst-case over all constructors to derive a value of 26392 clock cycles.

7 Related Work

While there has been significant interest in the use of amortised analysis for resource usage, in contrast to the work presented in this paper, none of this work considers multiple resources, none of the work has studied worst-case execution time, and none of it covers arbitrary recursive data structures. In particular, a notable difference to Tarjan's seminal work [17] (in addition to the fact that we perform automatic inference) is that credits are associated on a *per-reference* basis and not on the basis of the pure data layout in the memory. Okasaki [15] resorted to the use of *lazy evaluation* to solve this problem. In contrast, our per-reference credits can be directly applied to strict evaluation.

Hofmann and Jost were the first to develop an *automatic* amortised analysis for heap consumption [11], exploiting a difference metric similar to that used by Crary and Weirich [7] (the latter, however, only *check* bounds, and do not *infer* them, as we do). Hofmann and Jost have extended their method to cover a comprehensive subset of Java, including imperative updates, inheritance and type casts [12]. Shkaravska et al. subsequently considered heap consumption inference for first-order polymorphic lists, and are currently studying extensions to non-linear bounds [16]. Finally, Campbell [4] has developed the ideas of depth-based and temporary credit uses to give better results for stack usage.

A related idea is that of *sized types* [13], which express bounds on data structure sizes, and are attached to types in the same way as our *weights*. The difference to our work is that sized types express bounds on the size of the underlying data structure, whereas our weights are factors of the corresponding sizes, which may remain unknown. The original work on sized types was limited to type checking, but subsequent work has developed inference mechanisms [5,18].

A number of authors have recently studied analyses for heap usage. Albert et al. [1] present a fully automatic, live heap-space analysis for an object-oriented bytecode language with a scoped-memory manager. Most notably it is not restricted to a certain complexity class, and produces a closed-form upper bound function over the size of the input. However, unlike our system, data-dependencies cannot be expressed. Braberman et al. [3] infer polynomial bounds on the live heap usage for a Java-like language with automatic memory management. However, unlike our system, they do not cover general recursive methods. Finally, Chin et al. [6] present a heap and a stack analysis for a low-level (assembler) language with explicit (de-)allocation, which is also restricted to linear bounds.

8 Conclusions and Further Work

By developing a new type-based analysis, we have been able to automatically infer linear bounds on real-time, heap and stack costs for strict functional programs with algebraic data-types. The use of *amortised costs* allows us to determine a *provable* upper bound on the overall resource cost of running a program, by attaching numerical annotations to constructors. Thus, our analysis is not just

size-dependent but also data-dependent. We have extended previous work on the inference of amortised costs [11] by considering arbitrary (recursive) data structures and by constructing a generic treatment of resource usage through our resource tables. In this way, we are able to separate the mechanics of our approach from the operational semantics that applies to the usage of a given resource. Previous work [10,11,12,18] has been restricted to the treatment of a single resource type, and usually also to list homomorphisms. For all programs studied here, we determine very tight bounds on both heap and stack usage. Our results show that the bounds we infer for worst-case execution times can be within 33.7% of the measured costs. However, in some degenerate cases they can be significantly higher (in some cases due to the use of *software* floating-point operations whose time behaviour can be difficult to analyse effectively).

We are currently experimenting with a number of further extensions. We have developed a working prototype implementation dealing with higher-order functions with flexible cost annotations and partial application (`http://www.embounded.org/software/cost/cost.cgi`). The corresponding (and extensive) theoretical proof is still, however, in preparation. This implementation also deals with many useful extended language constructs, such as optimised conditionals for a boolean base type, pattern-matches having multiple cases, multiple let-definitions, etc. Most of these extensions are theoretically straightforward, and in the interest of brevity, we have therefore excluded them from this paper.

We now intend to study how to improve our time results, to determine how to extend our work to non-linear bounds, and to determine whether sized types can be effectively combined with amortised analysis. We are also working to extend our study of worst-case execution time so that it covers other interesting embedded systems architectures, e.g. the Freescale MPC555 for automotive applications. Since this has a hardware floating-point unit, we anticipate that the issues we have experienced with software floating-point operations on the Renesas M32C/85U will no longer be a concern on this new architecture.

Acknowledgements

We thank Hugo Simões and our anonymous reviewers for their useful comments, and Christoph Herrmann for performing measurements on the Renesas architecture. This work is supported by EU Framework VI grants IST-510255 (EmBounded) and IST-15905 (Mobius); and by EPSRC grant EP/F030657/1 (Islay).

References

1. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Live Heap Space Analysis for Languages with Garbage Collection. In *Proc. ISMM 2009: Intl. Symp. on Memory Management*, pages 129–138, Dublin, Ireland, June 2009. ACM.
2. M. Berkelaar, K. Eikland, and P. Notebaert. lp_solve: Open source (mixed-integer) linear programming system. GNU LGPL (Lesser General Public Licence). `http://lpsolve.sourceforge.net/5.5`.

3. V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *Proc. ISMM 2008: Intl. Symp. on Memory Management*, pages 141–150, Tucson, USA, June 2008. ACM.
4. B. Campbell. Amortised Memory Analysis Using the Depth of Data Structures. In *Proc. ESOP 2009: 18th European Symposium on Programming*, LNCS 5502, pages 190–204, York, UK, Mar. 2009. Springer.
5. W.-N. Chin and S.-C. Khoo. Calculating Sized Types. *Higher-Order and Symbolic Computing*, 14(2,3):261–300, Sept. 2001.
6. W.-N. Chin, H. Nguyen, C. Popeea, and S. Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *Proc. ISMM 2008: Intl. Symp. on Memory Management*, pages 151–160, Tucson, USA, June 2008. ACM.
7. K. Crary and S. Weirich. Resource Bound Certification. In *Proc. POPL 2000: ACM Symp. on Principles of Prog. Langs.*, pages 184–198, Boston, USA, Jan. 2000. ACM.
8. C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache Behavior Prediction by Abstract Interpretation. *Science of Comp. Prog.*, 35(2):163–189, 1999.
9. K. Hammond and G. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. GPCE 2003: Intl. Conf. on Generative Prog. and Component Eng.*, LNCS 2830, pages 37–56, Erfurt, Germany, Sept. 2003. Springer.
10. M. Hofmann. A Type System for Bounded Space and Functional In-Place Update. *Nordic Journal of Computing*, 7(4):258–289, Winter 2000.
11. M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *Proc. POPL 2003: ACM Symp. on Principles of Prog. Langs.*, pages 185–197, New Orleans, USA, Jan. 2003. ACM.
12. M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *Proc. ESOP 2006: 15th European Symposium on Programming*, LNCS 3924, pages 22–37, Vienna, Austria, Mar. 2006. Springer.
13. R. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *Proc. POPL 1996: ACM Symp. on Principles of Prog. Langs.*, pages 410–423, St. Petersburg Beach, USA, Jan. 1996. ACM.
14. S. M. Kuo, B. H. Lee, and W. Tian. *Real-Time Digital Signal Processing: Implementations and Applications*. Wiley, 2nd edition, Apr. 2006.
15. C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
16. O. Shkaravska, R. van Kesteren, and M. van Eekelen. Polynomial Size Analysis of First-Order Functions. In *Proc. TLCA 2007: Intl. Conf. on Typed Lambda Calculi and Applications*, LNCS 4583, pages 351–365, Paris, France, June 2007. Springer.
17. R. E. Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, April 1985.
18. P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *Proc. IFL 2003: Intl. Workshop on Impl. of Functional Languages*, LNCS 3145, pages 86–101, Edinburgh, UK, Sept. 2003. Springer.