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*).

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-

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

POPL'10, January 17–23, 2010, Madrid, Spain.

$$
\begin{array}{lll}
\mathit{vars} & ::= & \langle\ \mathit{varid}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{varid}_n\ \rangle \qquad n \geq 0 \\
\mathit{expr} & ::= & \mathit{const} \mid \mathit{varid} \mid \mathit{varid}\ \ \mathit{vars} \mid \mathit{conid}\ \ \mathit{vars} \\
& \mid & \lambda\, \mathit{varid}\, .\, \mathit{expr} \\
& \mid & \texttt{if}\ \mathit{varid}\ \texttt{then}\ \mathit{expr}_1\ \texttt{else}\ \mathit{expr}_2 \\
& \mid & \texttt{case}\ \ \mathit{varid}\ \texttt{of}\ \mathit{conid}\ \mathit{vars}\ \texttt{->}\ \mathit{expr}_1 \mid \mathit{expr}_2 \\
& \mid & \texttt{case!}\ \mathit{varid}\ \texttt{of}\ \mathit{conid}\ \mathit{vars}\ \texttt{->}\ \mathit{expr}_1 \mid \mathit{expr}_2 \\
& \mid & \texttt{let}\ \mathit{varid}\ \texttt{=}\ \mathit{expr}_1\ \texttt{in}\ \mathit{expr}_2 \\
& \mid & \texttt{LET}\ \mathit{varid}\ \texttt{=}\ \mathit{expr}_1\ \texttt{IN}\ \mathit{expr}_2 \\
& \mid & \texttt{let rec}\ \left\{ \begin{array}{l} \mathit{varid}_1\ \ \texttt{=}\ \mathit{expr}_1\texttt{;} \\ \qquad\cdots \\ \mathit{varid}_n\ \ \texttt{=}\ \mathit{expr}_n \end{array} \right\}\ \texttt{in}\ \mathit{expr} \qquad n \geq 1
\end{array}
$$

Figure 1. Schopenhauer Syntax

tential on a *per-reference* basis, rather than resorting to a lazy-evaluation strategy as Okasaki does [31]. The price we pay is that our method is currently limited to linear cost formulas (a restriction which is not inherent to amortised cost analysis). However, we believe that an efficient automatic analysis that can be run repeatedly at the press of a button is a major advantage over a cumbersome and error-prone manual analysis requiring some human ingenuity.

It is important to realise that the implementation of our method is indeed quite simple, being based on a standard type system, augmented by a small set of linear constraints that are collected as each type rule is applied. We do not need to count references: it is sufficient to examine the points in the rules where new aliases are introduced. Furthermore, the automatically-inferred potential mappings always allow the initial potential to be determined simultaneously for large classes of inputs. These mappings can thus be transformed into simple closed cost formulas.

Contributions: We present a type system for a compile-time analysis that infers input-dependent upper bounds on program execution costs for various resource metrics on strict, higher-order, polymorphic programs. We prove that the type system is sound with respect to a given operational semantics. We also present the associated fully-automatic type inference, which has been implemented using a standard external linear programming solver. Our main novel contributions are to extend previous work [19, 25]:

a) by analysing the resource usage of *higher-order* functions, which may be both polymorphic and mutually recursive, in a cost-preserving way;

b) by dealing with polymorphism, also in a cost-preserving way;

c) by considering the *resource parametricity* of (polymorphic) higher-order functions, so allowing a function to have a different cost behaviour for its different uses, without re-analysing the function.

Other notable advances over our earlier work [19] are:

a) the handling of arbitrary (recursive) algebraic datatypes, possibly containing functions;

b) the use of a storeless semantics instead of the (awkward) "benign sharing" condition from [19];

c) a unified, generic approach that presents a single soundness proof for several resource metrics and for several different operational models, including dynamic memory, stack allocations and worst-case execution time (specifically for the Renesas M32C/85U embedded system microcontroller).

These are discussed in more depth in a companion paper [25].

2. The Schopenhauer Notation

We illustrate our approach using the simple Schopenhauer language, which acts as a compiler intermediate language. The syntax of Schopenhauer (Figure 1) is mostly conventional, except that: i) we distinguish between identifiers for variables and those for data constructors; ii) all expressions are in *let-normal form*, i.e. most sub-expressions are variables; iii) we have two let-constructs that have identical meaning, but differing costs (see the following paragraph); iv) pattern matches are not nested and allow only two branches; v) pattern matching comes in two variants – read-only and destructive. None of these peculiarities are actually required, but they have been chosen to simplify the presentation of our work. For example, our implementation readily deals with nested pattern matches with an arbitrary number of branches. Note that the recursive *let-rec* form allows not only the construction of recursive functions, but also that of aliased circular data.

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 cost to this construct (generally zero), we can 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 Morrisett's *alias types* [40], or the *bunched implication logic* of Ishtiaq 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 $\mathbf{Bad}$. 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 ℓ. 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 ℓ to value w and otherwise acts as $\mathcal{H}$. All values are labelled for simplicity, e.g. $(\mathsf{bool}, \mathtt{tt})$, $(\mathsf{int}, 7)$, $(\mathsf{constr}_c, \ell_1, \ldots, \ell_n)$, $(\lambda x.e\,,\ \mathcal{V}^\star)$. Here $\mathbf{Ind}(\widehat{\ell})$ is a special value modelling an indirection. To follow such indirections we define $\mathrm{next}(\mathcal{H}, \ell) = \widehat{\ell}$ if $\mathcal{H}(\ell) = \mathbf{Ind}(\widehat{\ell})$ and $\mathrm{next}(\mathcal{H}, \ell) = \ell$ 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 $\mathbf{Bad}$ 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

$$\frac{n \in \mathbb{Z} \qquad \ell \notin \mathrm{dom}(\mathcal{H})}{\mathcal{V}, \mathcal{H} \overset{m' + \texttt{KmkInt}}{\underset{m'}{\vdash}} n \rightsquigarrow \ell, \mathcal{H}[\ell \mapsto (\mathsf{int}, n)]} \text{ (OP CONST INT)}$$

$$\frac{w = (\mathsf{bool}, \mathbf{tt}/\mathbf{ff}) \qquad \ell \notin \mathrm{dom}(\mathcal{H})}{\mathcal{V}, \mathcal{H} \overset{m' + \texttt{KmkBool}}{\underset{m'}{\vdash}} \texttt{true/false} \rightsquigarrow \ell, \mathcal{H}[\ell \mapsto w]} \text{ (OP CONST BOOL)}$$

$$\frac{\mathrm{next}(\mathcal{H}, \mathcal{V}(x)) = \ell}{\mathcal{V}, \mathcal{H} \overset{m' + \texttt{KpushVar} + \texttt{Knext}}{\underset{m'}{\vdash}} x \rightsquigarrow \ell, \mathcal{H}} \text{ (OP VAR)}$$

$$\frac{\mathcal{V}^\star = \mathcal{V}|_{\mathrm{FV}(e) \setminus x} \qquad w = (\lambda x.e\,,\ \mathcal{V}^\star) \qquad \ell \notin \mathrm{dom}(\mathcal{H})}{\mathcal{V}, \mathcal{H} \overset{m' + \texttt{KmkFun}(|\mathcal{V}^\star|)}{\underset{m'}{\vdash}} \lambda x.e \rightsquigarrow \ell, \mathcal{H}[\ell \mapsto w]} \text{ (OP ABS)}$$

$$\frac{\mathrm{next}(\mathcal{H}, \mathcal{V}(x_0)) = \widehat{\ell} \qquad \mathcal{H}(\widehat{\ell}) = (\lambda x.e\,,\ \mathcal{V}^\star) \qquad \mathcal{V}^\star\,[x \mapsto \mathcal{V}(x_0)]\,, \mathcal{H} \overset{m - \texttt{Kapp}}{\underset{m' + \texttt{Kapp}'}{\vdash}} e \rightsquigarrow \ell, \mathcal{H}'}{\mathcal{V}, \mathcal{H} \overset{m + \texttt{Knext}}{\underset{m'}{\vdash}} y\ x_0 \rightsquigarrow \ell, \mathcal{H}'} \text{ (OP APP)}$$

$$\frac{\mathrm{next}(\mathcal{H}, \mathcal{V}(x)) = \widehat{\ell} \qquad \mathcal{H}(\widehat{\ell}) = (\mathsf{bool}, \mathbf{tt}) \qquad \mathcal{V}, \mathcal{H} \overset{m - \texttt{KifT}}{\underset{m' + \texttt{KifT}'}{\vdash}} e_t \rightsquigarrow \ell', \mathcal{H}'}{\mathcal{V}, \mathcal{H} \overset{m + \texttt{Knext}}{\underset{m'}{\vdash}} \texttt{if}\ x\ \texttt{then}\ e_t\ \texttt{else}\ e_f \rightsquigarrow \ell', \mathcal{H}'} \text{ (OP CONDITIONAL TRUE)}$$

$$\frac{\mathrm{next}(\mathcal{H}, \mathcal{V}(x)) = \widehat{\ell} \qquad \mathcal{H}(\widehat{\ell}) = (\mathsf{bool}, \mathbf{ff}) \qquad \mathcal{V}, \mathcal{H} \overset{m - \texttt{KifF}}{\underset{m' + \texttt{KifF}'}{\vdash}} e_f \rightsquigarrow \ell', \mathcal{H}'}{\mathcal{V}, \mathcal{H} \overset{m + \texttt{Knext}}{\underset{m'}{\vdash}} \texttt{if}\ x\ \texttt{then}\ e_t\ \texttt{else}\ e_f \rightsquigarrow \ell', \mathcal{H}'} \text{ (OP CONDITIONAL FALSE)}$$

$$\frac{k \geq 0 \qquad c \in \mathsf{Constrs} \qquad \ell \notin \mathrm{dom}(\mathcal{H}_k) \qquad w = \left(\mathsf{constr}_c, \mathcal{V}(x_1), \ldots, \mathcal{V}(x_k)\right)}{\mathcal{V}, \mathcal{H} \overset{m' + \texttt{Kalloc}(c)}{\underset{m'}{\vdash}} c\,\langle x_1, \ldots, x_k \rangle \rightsquigarrow \ell, \mathcal{H}[\ell \mapsto w]} \text{ (OP CONSTRUCTOR)}$$

$$\frac{\mathrm{next}(\mathcal{H}, \mathcal{V}(x)) = \widehat{\ell} \qquad \mathcal{H}(\widehat{\ell}) \neq \left(\mathsf{constr}_c, \ell_1, \ldots, \ell_k\right) \qquad \mathcal{V}, \mathcal{H} \overset{m - \texttt{KcaseF}(c)}{\underset{m' + \texttt{KcaseF}'(c)}{\vdash}} e_2 \rightsquigarrow \ell, \mathcal{H}'}{\mathcal{V}, \mathcal{H} \overset{m + \texttt{Knext}}{\underset{m'}{\vdash}} \texttt{case}\ x\ \texttt{of}\ c\,\langle y_1, \ldots, y_k \rangle\ \texttt{->}\ e_1 \,|\, e_2 \rightsquigarrow \ell, \mathcal{H}'} \text{ (OP CASE FAIL)}$$

$$\frac{\mathrm{next}(\mathcal{H}, \mathcal{V}(x)) = \widehat{\ell} \qquad \mathcal{H}(\widehat{\ell}) = \left(\mathsf{constr}_c, \ell_1, \ldots, \ell_k\right) \qquad \mathcal{V}[y_1 \mapsto \ell_1, \ldots, y_k \mapsto \ell_k], \mathcal{H} \overset{m - \texttt{KcaseT}(c)}{\underset{m' + \texttt{KcaseT}'(c)}{\vdash}} e_1 \rightsquigarrow \ell, \mathcal{H}'}{\mathcal{V}, \mathcal{H} \overset{m + \texttt{Knext}}{\underset{m'}{\vdash}} \texttt{case}\ x\ \texttt{of}\ c\,\langle y_1, \ldots, y_k \rangle\ \texttt{->}\ e_1 \,|\, e_2 \rightsquigarrow \ell, \mathcal{H}'} \text{ (OP CASE SUCCEED)}$$

$$\frac{\mathrm{next}(\mathcal{H}, \mathcal{V}(x)) = \widehat{\ell} \qquad \mathcal{H}(\widehat{\ell}) = \left(\mathsf{constr}_c, \ell_1, \ldots, \ell_k\right) \qquad \mathcal{V}^\star = \mathcal{V}[y_1 \mapsto \ell_1, \ldots, y_k \mapsto \ell_k] \qquad \mathcal{V}^\star, \mathcal{H}[\ell \mapsto \mathbf{Bad}] \overset{m - \texttt{KcaseT}(c) + \texttt{Kcealloc}(c)}{\underset{m' + \texttt{KcaseT}'(c)}{\vdash}} e_1 \rightsquigarrow \ell, \mathcal{H}'}{\mathcal{V}, \mathcal{H} \overset{m + \texttt{Knext}}{\underset{m'}{\vdash}} \texttt{case!}\ x\ \texttt{of}\ c\,\langle y_1, \ldots, y_k \rangle\ \texttt{->}\ e_1 \,|\, e_2 \rightsquigarrow \ell, \mathcal{H}'} \text{ (OP CASE! SUCCEED)}$$

$$\frac{\mathcal{V}, \mathcal{H} \overset{m_1 - \texttt{Klet1}}{\underset{m_2}{\vdash}} e_1 \rightsquigarrow \ell_1, \mathcal{H}_1 \qquad \mathcal{V}_1 = \mathcal{V}[x \mapsto \ell_1] \qquad \mathcal{V}_1, \mathcal{H}_1 \overset{m_2 - \texttt{Klet2}}{\underset{m' + \texttt{Klet3}}{\vdash}} e_2 \rightsquigarrow \ell_2, \mathcal{H}_2}{\mathcal{V}, \mathcal{H} \overset{m_1}{\underset{m'}{\vdash}} \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \rightsquigarrow \ell_2, \mathcal{H}_2} \text{ (OP LET)}$$

$$\frac{\begin{array}{c} m = m_1 + \texttt{Krec1} + n\texttt{Knext} \qquad \mathcal{V}^\star = \mathcal{V}[x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n] \\ \mathcal{H}_0 = \mathcal{H}[\ell_1 \mapsto \mathbf{Bad}, \ldots, \ell_n \mapsto \mathbf{Bad}] \\ \mathcal{H}' = \mathcal{H}_n[\ell_1 \mapsto \mathbf{Ind}(\widehat{\ell_1}), \ldots, \ell_n \mapsto \mathbf{Ind}(\widehat{\ell_n})] \\ \forall i \in \{1, \ldots, n\}\,.\ \ell_i \notin \mathrm{dom}(\mathcal{H}) \ \wedge\ \widehat{\ell_i} = \mathrm{next}(\mathcal{H}_n, \ell_i') \\ \forall i \in \{1, \ldots, n\}\,.\ \mathcal{V}^\star, \mathcal{H}_{i-1} \overset{m_i - \texttt{Krec2}}{\underset{m_{i+1}}{\vdash}} e_i \rightsquigarrow \ell_i', \mathcal{H}_i \\ \mathcal{V}^\star, \mathcal{H}' \overset{m_{n+1} - \texttt{Krec3}}{\underset{m' + \texttt{Krec4}}{\vdash}} e \rightsquigarrow \ell, \mathcal{H} \end{array}}{\mathcal{V}, \mathcal{H} \overset{m}{\underset{m'}{\vdash}} \texttt{let rec}\ \{x_1\ \texttt{=}\ e_1\texttt{;}\ \ldots\texttt{;}x_n\ \texttt{=}\ e_n\}\ \texttt{in}\ e \rightsquigarrow \ell, \mathcal{H}'} \text{ (OP REC)}$$

Figure 2. Schopenhauer Operational Semantics

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 $\mathcal{V}, \mathcal{H} \overset{m}{\underset{m'}{\vdash}} e \rightsquigarrow \ell, \mathcal{H}'$ means that under the initial environment $\mathcal{V}$ and heap $\mathcal{H}$, the expression e evaluates to location ℓ, 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 $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \ell, \mathcal{H}'$ to denote that e evaluates to ℓ using an unknown, but finite, amount of resources.

For example, $\mathcal{V}, \mathcal{H} \overset{3}{\underset{1}{\vdash}} e \rightsquigarrow \ell, \mathcal{H}'$ 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 $\mathcal{V}, \mathcal{H} \overset{m}{\underset{m'}{\vdash}} e \rightsquigarrow \ell, \mathcal{H}'$ holds, then both $m' \geq 0$ and $\mathcal{V}, \mathcal{H} \overset{m+k}{\underset{m'+k}{\vdash}} e \rightsquigarrow \ell, \mathcal{H}'$ 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 `IN` e_2), 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.absint.com`) determines this to be 83 cycles on the Renesas M32/85U microcontroller.

Recursive let-bindings use *indirections*. 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.

Let us see, for example, how $\texttt{let rec}\ \{x\ \texttt{=}\ \mathsf{cons}\langle x\rangle\}\ \texttt{in}\ x$ is evaluated in the empty heap and store (where $\mathsf{cons} \in \mathsf{Constrs}$ is assumed). We omit resource annotations. We put $\mathcal{H}_0 = [\ell_1 \mapsto \mathbf{Bad}]$ and $\mathcal{V}^\star = [x \mapsto \ell_1]$ and have

$$\mathcal{V}^\star, \mathcal{H}_0 \vdash \mathsf{cons}\langle x\rangle \rightsquigarrow \ell_1', [\ell_1 \mapsto \mathbf{Bad}, \ell_1' \mapsto (\mathsf{cons}, \ell_1)]$$

Now we have $\mathrm{next}(\mathcal{H}_1, \ell_1') = \ell_1'$. Defining the new heap $\mathcal{H}' = [\ell_1 \mapsto \mathbf{Ind}(\ell_1'), \ell_1' \mapsto (\mathsf{cons}, \ell_1)]$ we get

$$\emptyset, \emptyset \vdash \texttt{let rec}\ \{x\ \texttt{=}\ \mathsf{cons}\langle x\rangle\}\ \texttt{in}\ x \rightsquigarrow \ell_1, \mathcal{H}'$$

yielding the expected cyclic data structure with indirection.

4. Schopenhauer Type Rules

We use α, β, γ to denote type variables. Let CV be an infinite set of *resource variables* ranging over $\mathbb{Q}^+$, usually denoted by q, p, r, s, being disjoint from the identifier sets for variables and constructors $\mathsf{Var}, \mathsf{Constrs}$. Sets of type and resource variables are referred to using the vector notation, e.g. $\vec{\alpha}, \vec{q}$. All other decorations stand for different entities. We use ψ, ϕ, ξ 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 $\mathbb{Q}^+$ and type variables to closed types. We write $v \Rightarrow \phi$ if v satisfies all constraints in ϕ, and $\psi \Rightarrow \phi$ to denote that ψ entails ϕ, i.e. that all valuations that satisfy ψ also satisfy all constraints in ϕ.

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

$$\begin{array}{rcl} T ::= & \mathsf{int} \mid \mathsf{bool} \mid \alpha \\ & \mid & \mu\alpha.\{\ c_1{:}(q_1, \overrightarrow{T_1}) \mid \ldots \mid c_k{:}(q_k, \overrightarrow{T_k})\ \} \\ & \mid & \forall \vec{r} \in \psi.\overrightarrow{T} \xrightarrow[p']{p} T' \\ & \mid & \forall \vec{\alpha}{:}\psi.T \end{array}$$

where $c_i \in \mathsf{Constrs}$ are constructor labels and $\overrightarrow{T}$ stands for $\langle\ T_1\ \ldots\ T_n\ \rangle$ 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:

$$\forall\alpha{:}\emptyset.\forall\{x, y, u, v\} \in \phi.\mu\beta.\{\mathtt{Cons}{:}(u, \langle\alpha, \beta\rangle)\} \mid \mathtt{Nil}{:}(v)\} \xrightarrow[y]{x} \mathsf{int}$$

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 ($\forall\alpha{:}\emptyset.$). Furthermore, it admits several resource behaviours, since $\forall\{x, y, u, v\} \in \phi.$ tells us that we can rename x, y, u, v to independent resource variables. Of course, the constraints ϕ 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 `Cons` 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 100 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 $\mathrm{FV}_\diamond(\cdot)$. We also define a mapping $|\cdot|$ from annotated types to standard unannotated types, which simply erases all annotations. For $\forall\alpha \in \psi.\overrightarrow{T} \xrightarrow[p']{p} T'$, we require that $\alpha \subseteq \mathrm{FV}_\diamond(\overrightarrow{T}) \cup \{p, p'\} \cup \mathrm{FV}_\diamond(T')$ holds, but not that $\mathrm{FV}_\diamond(\psi)$ is a subset of α. Any intermediate variables which would then only occur in ψ can be eliminated by projecting the *polytope* described by ψ to 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 $\Phi^v_{\mathcal{H}}(w : A)$ (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. $\Phi^v_{\mathcal{H}}(\mathcal{V} : \Gamma)$ is the sum of $\Phi^v_{\mathcal{H}}(\mathcal{V}(x) : \Gamma(x))$ over all variables x in Γ. 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 judgements 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 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 $v(\Gamma)$; furthermore, evaluating e under environment $\mathcal{V}$ and heap $\mathcal{H}$ requires a potential of at most $v(q) + \Phi^v_{\mathcal{H}}(\mathcal{V} : \Gamma)$ and leaves a potential of at least $v(q') + \Phi^v_{\mathcal{H}'}(\ell : A)$ available afterwards, where ℓ is the result value and $\mathcal{H}'$ the post-heap". In Section 5, we will formalise this statement as Theorem 1 (our main theorem), which requires as a precondition that the context, environment and heap all agree.

We use a compressed notation that makes the following two formulations equivalent for $\psi = \{q_1 = q_2 + c_1,\ q2' = q_1' + c_2\}$:

$$\frac{\Gamma \vdash^{q_2}_{q_2'} e_2 : A_2 \mid \phi}{\Gamma \vdash^{q_1}_{q_1'} e_1 : A_1 \mid \phi \cup \psi} \qquad \frac{\Gamma \vdash^{q_2}_{q_1' + c_2} e_2 : A_2 \mid \phi}{\Gamma \vdash^{q_2 + c_1}_{q_1'} e_1 : A_1 \mid \phi}$$

The constraints ψ 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.

$$\frac{}{x{:}A \vdash^{\mathtt{KpushVar}}_{0} x : A \mid \emptyset} \ (\textsc{Var})$$

$$\frac{n \in \mathbb{Z}}{\varnothing \vdash^{\mathtt{KmkInt}}_{0} n : \mathsf{int} \mid \emptyset} \ (\textsc{Int}) \qquad \frac{e \in \{\mathtt{true}, \mathtt{false}\}}{\varnothing \vdash^{\mathtt{KmkBool}}_{0} e : A \mid \emptyset} \ (\textsc{Bool})$$

* Note that all operators are extended pointwise when used in conjunction with the vector notation and are only defined if both vectors have the same length, i.e. $\overrightarrow{A} = \overrightarrow{B}$ stands for $\forall i\,.\,A_i = B_i$.

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 $\curlyvee$-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.

$$\frac{\Gamma \vdash^{p}_{p'} e : A \mid \phi}{\Gamma \vdash^{q}_{q'} e : A \mid \phi \cup \{q \geq p, q - p \geq q' - p'\}} \text{(RELAX)}$$

$$\frac{\Gamma \vdash^{q}_{q'} e : C \mid \psi \qquad \phi \Rightarrow \psi}{\Gamma, x{:}A \vdash^{q}_{q'} e : C \mid \phi} \text{(WEAK)}$$

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

The ternary function $\curlyvee(A\,|B, C)$ 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

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

For type variables we simply record $\curlyvee(\alpha\,|\beta_1, \beta_2)$ 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.

$$\frac{\begin{array}{c}\mathrm{dom}(\Gamma) = \mathrm{FV}(e) \setminus x \qquad B = A \xrightarrow{\frac{q}{q'}} C \qquad \phi \cup \psi \Rightarrow \xi \\ \Gamma, x{:}A \vdash^{q}_{q'} e : C \mid \xi \qquad \phi \Rightarrow \bigcup_{D \in \mathrm{ran}(\Gamma)} \curlyvee(D\,|D, D) \\ \vec{r} \notin \mathrm{FV}_\diamond(\Gamma) \cup \mathrm{FV}_\diamond(\phi)\end{array}}{\Gamma \vdash^{\texttt{KmkFun}(|\Gamma|)}_{0} \lambda x.e : \forall \vec{r} \in \psi.B \mid \phi} \text{(ABS)}$$

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 $\curlyvee$. Here, $\curlyvee(D\,|D, D)$ 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.

$$\frac{\begin{array}{c}\sigma : \vec{r} \to \mathsf{CV} \text{ a substitution to fresh resource variables} \\ \sigma(B) = A \xrightarrow{\frac{q}{q'}} C\end{array}}{x{:}A,\ y{:}\forall \vec{r} \in \psi.B \vdash^{q + \texttt{Kapp} + \texttt{Knext}}_{q' - \texttt{Kapp}'} y\, x : C \mid \sigma(\psi)} \text{(APP)}$$

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.

$$\frac{\Gamma \vdash^{q - \texttt{KifT}}_{q' + \texttt{KifT}'} e_t : A \mid \phi \qquad \Gamma \vdash^{q - \texttt{KifF}}_{q' + \texttt{KifF}'} e_f : A \mid \psi}{\Gamma, x{:}\mathsf{bool} \vdash^{q + \texttt{Knext}}_{q'} \texttt{if}\ x\ \texttt{then}\ e_t\ \texttt{else}\ e_f : A \mid \phi \cup \psi} \text{(CONDITIONAL)}$$

$$\frac{\begin{array}{c}c \in \mathsf{Constrs} \qquad C = \mu X.\{\cdots \,|\, c : (p, \langle B_1, \ldots, B_k\rangle) \,|\, \cdots\} \\ A_i = B_i\big[C \,/\, X\big] \text{ (for } i = 1, \ldots, k)\end{array}}{x_1{:}A_1, \ldots, x_k{:}A_k \vdash^{p + \texttt{Kalloc}(c)}_{0} c\ \langle x_1, \ldots, x_k\rangle : C \mid \emptyset} \text{(CONSTR)}$$

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.

$$\frac{\begin{array}{c}c \in \mathsf{Constrs} \qquad A = \mu X.\{\cdots \,|\, c : (p, \langle B_1, \ldots, B_k\rangle) \,|\, \cdots\} \\ \Delta = y_1{:}B_1[A/X], \ldots, y_k{:}B_k[A/X] \\ \Gamma, \Delta \vdash^{q + p + \texttt{Kdealloc}(c) - \texttt{KcaseT}(c)}_{q' + \texttt{KcaseT}'(c)} e_1 : C \mid \phi \\ \Gamma, x{:}A \vdash^{q - \texttt{KcaseF}(c)}_{q' + \texttt{KcaseF}'(c)} e_2 : C \mid \psi\end{array}}{\Gamma, x{:}A \vdash^{q + \texttt{Keof}}_{q'} \texttt{case!}\ x\ \texttt{of}\ c\ \langle y_1, \ldots y_k\rangle\ \texttt{->}\ e_1 \,|\, e_2 : C \mid \phi \cup \psi} \text{(CASE!)}$$

The CASE rule for the read-only `case` pattern-match is identical to CASE!, except that it doesn't include the cost parameter $\texttt{Kdealloc}(c)$, the (possibly negative) cost of deallocating constructor constr_C.

Let-bindings.

$$\frac{\Gamma \vdash^{q_1}_{q_2 + \texttt{Klet2}} e_1 : A_1 \mid \psi \qquad \Delta, x{:}A_1 \vdash^{q_2}_{q_3} e_2 : A_2 \mid \phi}{\Gamma, \Delta \vdash^{q_1 + \texttt{Klet1}}_{q_3 - \texttt{Klet3}} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \ :\ A_2 \mid \psi \cup \phi} \text{(LET)}$$

The type rule for the alternative form of let-expression `LET ... IN` (LET), is almost identical, except it substitutes the cost constants `KLET1`, `KLET2`, `KLET3` for `Klet1`, `Klet2`, `Klet3`, respectively.

Rec-bindings.

$$\frac{\begin{array}{c}\Delta = x_1{:}A_1, \ldots, x_n{:}A_n \\ \forall i \in \{1, \ldots, n\}.\ \Gamma_i, \Delta \vdash^{q_i - \texttt{Krec2}}_{q_{i+1}} e_i : A_i \mid \psi_i \\ \Gamma_{n+1}, \Delta \vdash^{q_{n+1} - \texttt{Krec3}}_{q' + \texttt{Krec4}} e : C \mid \xi \\ \phi \Rightarrow \xi \cup \bigcup_{i=1,\ldots,n} \psi_i \cup \bigcup_{B \in \mathrm{ran}(\Delta)} \curlyvee(B \,|B, B\,) \\ \phi \Rightarrow \{q \geq q_1 + \texttt{Krec1} + n \cdot \texttt{Knext}\}\end{array}}{\Gamma_1, \ldots, \Gamma_{n+1} \vdash^{q}_{q'} \texttt{let rec}\ \{x_1 = e_1; \ldots; x_n = e_n\}\ \texttt{in}\ e : C \mid \phi} \text{(REC)}$$

Our 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 $\curlyvee$ 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.

$$\frac{\vec{\alpha} \notin \mathrm{dom}(\Gamma) \qquad \vec{\alpha} \notin \psi \qquad \Gamma \vdash^{q}_{q'} e : C \mid \psi \cup \phi}{\Gamma \vdash^{q}_{q'} e : \forall \vec{\alpha}{:}\phi.C \mid \psi} \text{(GENERALISE)}$$

$$\frac{\Gamma \vdash^{q}_{q'} e : \forall \vec{\alpha}{:}\xi.C \mid \psi \qquad \phi \Rightarrow \psi \cup \xi\big[\vec{B} \,/\, \vec{\alpha}\big]}{\Gamma \vdash^{q}_{q'} e : C\big[\vec{B} \,/\, \vec{\alpha}\big] \mid \phi} \text{(SPECIALISE)}$$

We use the standard Hindley-Milner rules for polymorphism: GENERALISE is used to generalise a type; and SPECIALISE allows a polymorphic type to be specialised to any valid type, as defined by the other type rules.

Subtyping. The type rules for subtyping depend on another inductively defined relation $\xi \vdash A <: B$ between two types A and B, defined below, which is relative to a constraint set ξ.

$$\frac{\Gamma, x{:}B \vdash^{q}_{q'} e : C \mid \phi \quad \psi \vdash A <: B}{\Gamma, x{:}A \vdash^{q}_{q'} e : C \mid \phi \cup \psi} \text{(SUPERTYPE)}$$

$$\frac{\Gamma \vdash^{q}_{q'} e : D \mid \phi \quad \psi \vdash D <: C}{\Gamma \vdash^{q}_{q'} e : C \mid \phi \cup \psi} \text{(SUBTYPE)}$$

For any fixed constraint set ξ, the following relation is both *reflexive* and *transitive*, but not necessarily *anti-symmetric*.

$$\frac{}{\xi \vdash A <: A}$$

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

$$\frac{\begin{array}{c}\sigma : \vec{s} \to \mathsf{CV} \text{ a substitution} \\ \xi \cup \phi \Rightarrow \sigma(\psi) \qquad \xi \cup \phi \Rightarrow \{\sigma(p) \leq q,\ \sigma(p') \geq q'\} \\ \xi \cup \phi \vdash D <: \sigma(C) \qquad \xi \cup \phi \vdash \sigma(A) <: B\end{array}}{\xi \vdash \forall \vec{s} \in \psi.C \xrightarrow{p}_{p'} A <: \forall \vec{r} \in \phi.D \xrightarrow{q}_{q'} B}$$

$$\frac{\mathcal{H}(\ell) = (\mathsf{int}, \mathsf{n}) \quad \mathsf{n} \in \mathbb{Z}}{\mathcal{H} \vDash_v \ell{:}\mathsf{int}} \text{(WFINT)} \qquad \frac{\mathcal{H}(\ell) = (\mathsf{bool}, \mathbf{tt}/\mathbf{ff})}{\mathcal{H} \vDash_v \ell{:}\mathsf{bool}} \text{(WFBOOL)}$$

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

$$\frac{\mathcal{H} \vDash_v \ell{:}A \qquad \exists \phi\,.\, v \Rightarrow \phi \wedge \phi \vdash A <: B}{\mathcal{H} \vDash_v \ell{:}B} \text{(WFSUBTYPE)}$$

$$\frac{\begin{array}{c}\mathcal{H}(\ell) = (\lambda x.e\ ,\ \mathcal{V}) \qquad \text{There exists } \Gamma, p, p', \phi \text{ such that:} \\ \mathcal{H}[\ell \mapsto \mathbf{Bad}] \vDash_v \mathcal{V}{:}\Gamma \qquad v \vDash \phi \qquad \Gamma \vdash^{p}_{p'} \lambda x.e : F \mid \phi\end{array}}{\mathcal{H} \vDash_v \ell{:}F} \text{(WFFUN)}$$

$$\frac{\mathcal{H}(\ell) = \mathbf{Bad}}{\mathcal{H} \vDash_v \ell{:}A} \text{(WFBAD)} \qquad \frac{\begin{array}{c}\mathcal{H}(\ell) = \mathbf{Ind}(\widehat{\ell}) \\ \mathcal{H}(\widehat{\ell}) \neq \mathbf{Ind}(\_) \\ \mathcal{H} \vDash_v \widehat{\ell}{:}A\end{array}}{\mathcal{H} \vDash_v \ell{:}A} \text{(WFINDIRECT)}$$

Figure 3. Derivation rules for "well-formed" environments.

$$\frac{\sigma : \vec{\alpha} \to \vec{\beta} \text{ a substitution} \qquad \xi \cup \phi \Rightarrow \sigma(\psi) \qquad \xi \vdash \sigma(A) <: B}{\xi \vdash \forall \vec{\alpha}{:}\psi.A \ <:\ \forall \vec{\beta}{:}\phi.B}$$

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 $\mathcal{H} \vDash_v \mathcal{V}{:}\Gamma$, 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 Γ and valuation v, written $\mathcal{H} \vDash_v \mathcal{V} : \Gamma$, if and only if for all variables $x \in \Gamma$ the statement $\mathcal{H} \vDash_v \mathcal{V}(x){:}\Gamma(x)$ can be derived by the rules in Figure 3.

Lemma 5.2. *If* $\mathcal{H} \vDash_v \mathcal{V}{:}\Gamma$ *then for all* ℓ *also* $\mathcal{H}[\ell \mapsto \mathbf{Bad}] \vDash_v \mathcal{V}{:}\Gamma$

It is an obvious requirement that evaluation must maintain a well-formed memory configuration.

Lemma 5.3. *If $\mathcal{H} \vDash_v \mathcal{V}:\Gamma$ and $\mathcal{V}, \mathcal{H} \vdash e \leadsto \ell, \mathcal{H}'$ then $\mathcal{H}' \vDash_v \mathcal{V}:\Gamma$.*

We remark that one might wish to prove an extended statement that the result ℓ 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 $\mathcal{H} \vDash_v \ell:A$ holds, then the *potential* of location ℓ for type A in heap $\mathcal{H}$ under valuation v, written $\Phi^v_{\mathcal{H}}(\ell{:}A)$, is recursively defined for recursive datatypes by

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

when both $A = \mu X.\{\cdots \mid c{:}(q, \langle B_1, \ldots, B_k\rangle) \mid \cdots\}$ and also $\mathcal{H}(\mathrm{next}(\mathcal{H}, \ell)) = (\mathsf{constr}_c, \ell_1, \ldots, \ell_k)$ holds, and zero in all other cases. We extend this definition to contexts by

$$\Phi^v_{\mathcal{H}}\Big(\mathcal{V} : v(\Gamma)\Big) = \sum_{x \in \mathrm{dom}(\Gamma)} \Phi^v_{\mathcal{H}}\Big(\mathcal{V}(x){:}v\big(\Gamma(x)\big)\Big)$$

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

Lemma 5.5. *If $\mathcal{H} \vDash_v \ell : A$ and $\phi \vdash A <: B$ holds and v is a valuation satisfying ϕ, then $\mathcal{H} \vDash_v \ell{:}B$ and $\Phi^v_{\mathcal{H}}(\ell{:}A) \geq \Phi^v_{\mathcal{H}}(\ell{:}B)$*

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

Lemma 5.6. *If $\curlyvee(A\,|B, C) = \phi$ 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 $\mathcal{H} \vDash_v \ell : A$ and $\curlyvee(A\,|B, C) = \phi$ holds and v satisfies 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.*

Lemma 5.8 (Inversion)**.** *If $\Gamma \overset{p}{\underset{p'}{\vdash}} \lambda x.e : B \mid \phi$ holds, then there exists $\Delta, \xi, \forall \vec{s}{\in}\psi.A \overset{q}{\underset{q'}{\longrightarrow}} C$ such that all of the following hold*

$$\phi \vdash \big(\forall \vec{a}{\in}\psi.A \overset{q}{\underset{q'}{\longrightarrow}} C\big) <: B \qquad \Delta, x{:}A \overset{q}{\underset{q'}{\vdash}} e : C \mid \xi$$

$$\Delta \subseteq \Gamma \qquad \mathrm{dom}(\Delta) = \mathrm{FV}(e) \setminus x \qquad \vec{s} \notin \mathrm{FV}_\diamond(\Delta) \cup \mathrm{FV}_\diamond(\phi)$$

$$\phi \cup \psi \Rightarrow \xi \qquad \phi \Rightarrow \bigcup_{D \in \mathrm{ran}\,\Delta} \curlyvee(D\,|D, D) \qquad \phi \Rightarrow p \geq p' + \mathtt{KmkFun}(|\Delta|)$$

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 statements hold*

$$\Gamma \overset{q}{\underset{q'}{\vdash}} e{:}A \mid \phi \tag{1.A}$$

$$\mathcal{V}, \mathcal{H} \vdash e \leadsto \ell, \mathcal{H}' \tag{1.B}$$

$$\mathcal{H} \vDash_v \mathcal{V}{:}v(\Gamma) \tag{1.C}$$

$$v \text{ : a valuation satisfying } v(\phi) \tag{1.D}$$

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

$$m \geq v(q) + \Phi^v_{\mathcal{H}}\Big(\mathcal{V} : v(\Gamma)\Big) + r \tag{1.E}$$

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

$$\mathcal{V}, \mathcal{H} \overset{m}{\underset{m'}{\vdash}} e \leadsto \ell, \mathcal{H}' \tag{1.I}$$

$$m' \geq v(q') + \Phi^v_{\mathcal{H}'}\Big(\ell{:}v(A)\Big) + r \tag{1.II}$$

$$\mathcal{H}' \vDash_v \ell{:}v(A) \tag{1.III}$$

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 excess 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 $\mathcal{H}'(\ell) = (\lambda x.e\,,\, \mathcal{V}^\star)$. Fix $r \in \mathbb{Q}^+$ and choose any $m \in \mathbb{N}$ such that $m \geq v(q) + \Phi^v_{\mathcal{H}}(\mathcal{V}{:}\Gamma) + r$. By the definition of ABS and the observation that Γ has no potential by Lemma 5.7, we have $m \geq \mathtt{KmkFun}(|\Gamma|) + r$. Furthermore $m \geq \mathtt{KmkFun}(|\mathcal{V}^\star|) + r$ since $|\Gamma| = |\mathcal{V}^\star|$ by $\mathcal{V}^\star = \mathcal{V}|_{\mathrm{FV}(e)\setminus x}$ from OP ABS and $\mathrm{dom}(\Gamma) = \mathrm{FV}(e) \setminus x$ from ABS. By OP ABS and Lemma 3.1 we thus obtain $m' = m - \mathtt{KmkFun}(|\mathcal{V}^\star|) + r$ which satisfies $m' \geq r$ as required, since the potential of ℓ is zero by Definition 5.4. This leaves us to prove (1.III), which follows in this case directly from WFFUN, since all existentially quantified requirements are among our premises, except for $\mathcal{H}'[\ell \mapsto \mathbf{Bad}] \vDash_v \mathcal{V} : \Gamma$ which follows by Lemmas 5.2 and 5.3 from (1.C).

(APP**)** From OP APP we have $\mathcal{H}\big(\mathcal{V}(y)\big) = (\lambda x.e\,,\, \mathcal{V}^\star)$ and from (1.III) through WFFUN we obtain the existence of a typing judgement for the function body. By the inversion Lemma 5.8, we obtain all the required properties to derive $\Gamma \overset{\mathtt{KmkFun}(|\Gamma|)}{\underset{0}{\vdash}} \lambda x.e : \forall \vec{r}{\in}\psi.B \mid \phi$ 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 Γ 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 $m \geq v(p) + \Phi^v_{\mathcal{H}}(\mathcal{V}{:}\Gamma) + r = v(q) + \Phi^v_{\mathcal{H}}(\mathcal{V}{:}\Gamma) + r'$ if we choose $r' = r + v(p) - v(q)$ for applying the induction hypothesis. We can do this since $v(p) - v(q) \geq 0$ holds by the constraints of the RELAX rule. We thus obtain m' with

$$\begin{aligned} m' &\geq v(q') + \Phi^v_{\mathcal{H}}(\mathcal{V}{:}\Gamma) + r' \\ &= v(q') + \Phi^v_{\mathcal{H}}(\mathcal{V}{:}\Gamma) + r + v(p) - v(q) \\ &\geq v(q') + \Phi^v_{\mathcal{H}}(\mathcal{V}{:}\Gamma) + r + v(p') - v(q') \\ &= v(p') + \Phi^v_{\mathcal{H}}(\mathcal{V}{:}\Gamma) + r \end{aligned}$$

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

	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
sum (see Fig 4)															
Analysis	16	39	3603	24	39	5615	32	39	7627	40	39	9639	48	39	11651
Measured	16	34	3066	24	34	4606	32	34	6146	40	34	7686	48	34	9226
Ratio	1.00	1.15	1.18	1.00	1.15	1.22	1.00	1.15	1.24	1.00	1.15	1.25	1.00	1.15	1.26
flatten (see Fig 6)															
Analysis	21	34	4485	38	60	8732	55	66	12979	72	82	17226	89	98	21473
Measured	21	34	4275	38	50	7970	55	50	11665	72	66	15360	89	66	19055
Ratio	1.00	1.00	1.05	1.00	1.20	1.10	1.00	1.32	1.11	1.00	1.24	1.12	1.00	1.49	1.13
repmin															
Analysis	17	42	5020	35	69	10991	53	96	16962	71	123	22933	89	150	28904
Measured	17	42	4633	35	52	9395	53	61	14157	71	62	18919	89	71	23681
Ratio	1.00	1.00	1.08	1.00	1.33	1.17	1.00	1.57	1.20	1.00	1.98	1.21	1.00	2.11	1.22

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

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 μ-type, i.e. `list`, with the constructors `Cons` and `Nil`.

```
type num = int 16;

add :: num -> num -> num;
add x y = x + y;

fold :: (num -> num -> num) -> num -> [num] -> num;
fold f n []     = n;
fold f n (x:xs) = fold f (f x n) xs;

sum :: [num] -> num;
sum xs = fold add 0 xs;
```

Figure 4. Source code of list-sum

```
SCHOPENHAUER typing for HeapBoxed:
 list[Cons<2>:int,#|Nil] -(6/0)-> int
```

The argument type includes annotations for each constructor, separated by `|`. 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 (`-(6/0)->`). 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 x 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:
 list[Cons:int,#|Nil] -(27/17)-> 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:
 list[Cons<1714>:int,#|Nil<225>] -(684/0)-> 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 $N = 1 \ldots 5$. 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

```
fold f n l = LET l1 = l IN
             case l1 of []     -> LET n1  = n       IN n1;
                      | (x:xs) -> LET xs1 = xs      IN
                                  LET n1  = n       IN
                                  LET x1  = x       IN
                                  LET z1  = f x1 n1 IN
                                  LET f1  = f       IN
                                  fold f1 z1 xs1;
```

Figure 5. Intermediate code form of function `fold`.

bounds 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. Another advantage of the `LET`-construct is that we can keep our big-step semantics for measuring worst-case execution time and stack space usage, for which small-step semantics are usually required. 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 APP. 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 refrained from strictly enforcing this policy in Schopenhauer because 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.

```
case l of (x:xs) -> LET z1 = f x n IN fold f z1 xs;
```

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

$$\big(\mathsf{int}\ \tfrac{a}{b}\!\!\rightarrow\ \mathsf{int}\ \tfrac{v}{w}\!\!\rightarrow\ \mathsf{int}\big)\ \tfrac{c}{d}\!\!\rightarrow\ \mathsf{int}\ \tfrac{e}{f}\!\!\rightarrow\ \mathsf{list}(\mathsf{int},p)\ \tfrac{x}{y}\!\!\rightarrow\ \mathsf{int}$$

where $\mathsf{list}(\mathsf{int},p)$ is a convenient shorthand for the simplified list type $\mu\alpha.\{\texttt{Cons}{:}(p,\langle\mathsf{int},\alpha\rangle)\,|\,\texttt{Nil}{:}(0)\}$. For the sake of simplicity, we immediately set the variables a, b, c, d, e and f to the zero value, since they are non-essential in this example.

We then follow the standard type derivation tree for the code, gathering constraints as we go. We must also reconstruct the implicit inequalities hidden by our compressed type rule notation. However, this is very simple, if we adopt the view that the value on top of the turnstile represents the "currently *available* resources" before executing the term and the one below the "guaranteed *remaining* resources" after. In that sense, we start with x resources available, since we are in the body of the fully applied function.

The outermost term constructor is a case distinction, so CASE applies. On top of the turnstile in the conclusion we have $q_1 + \texttt{Keof}$. Hence we gather the implicit inequality $x \geq q_1 + \texttt{Keof}$. We follow the branch of the type derivation for the successful match of the `Cons`-constructor, which according to the CASE rule now has $q_1 + p - \texttt{KcaseT}$ resources available.

Next, the LET rule applies. Matching the available resources yields the second inequality $q_1 + p - \texttt{KcaseT} \geq q_2 + \texttt{KLET1}$, and according to the first premise we have q_2 resources available for the call `f x n`.

According to the APP type rule, a call to function `f` requires us to pay $\texttt{Kapp} + \texttt{Knext}$. Hence we have the inequality $q_2 \geq v + \texttt{Kapp} + \texttt{Knext}$. In addition, we must apply subtyping to match the annotations of the argument types and the functions. However, no constraints are generated here, since both are unannotated numeric types. Furthermore, any constraints that are directly attached to function `f` are also added now. The inference renames all bound variables in these constraints, probably including v and w, but again, for simplicity, we ignore this here. The renaming allows different possible resource usages for each function application, as described in Section 6.6.1.

Since APP is a terminal rule, we are left with $w - \texttt{Kapp}'$ resources. Note that we always apply WEAK before any terminal rule, to allow excess resources to be carried over. Again, we ignore this in this example.

We have now returned to the second premise of rule LET and can obtain the constraint $w - \texttt{Kapp}' \geq q_3 + \texttt{KLET2}$, leaving us with q_3 resources for the body of the LET-expression, a recursive call. The application of WEAK is crucial in this case, so we obtain $q_3 \geq q_4$ and $q_3 - q_4 \geq q_6 - q_5$, where q_4 and q_6 are fresh, and q_5 represents the remaining resources after applying APP again. This in turn yields the constraints $q_4 \geq x + \texttt{Kapp} + \texttt{Knext}$ and $y - \texttt{Kapp}' \geq q_5$. We are therefore left with q_6 after the weakening.

Matching q_6 against the remaining resources guaranteed by LET then yields $q_6 - \texttt{KLET3} \geq q_7$. Finally, using CASE we obtain $q_7 - \texttt{KcaseT}' \geq y$ in a similar way.

Let ψ denote the set containing these constraints. If we instantiate the resource parameters according to the desired cost model and specify that all variables must be non-negative, ψ could now be solved by an LP-solver, yielding an annotated type for the function. However, if this function definition is part of a bigger program, we do not solve the constraints at this point, but rather use them to improve the type of the function to

$$\forall\vec{r}\in\psi.\big(\mathsf{int}\ \tfrac{0}{0}\!\!\rightarrow\ \mathsf{int}\ \tfrac{v}{w}\!\!\rightarrow\ \mathsf{int}\big)\ \tfrac{0}{0}\!\!\rightarrow\ \mathsf{int}\ \tfrac{0}{0}\!\!\rightarrow\ \mathsf{list}(\mathsf{int},p)\ \tfrac{x}{y}\!\!\rightarrow\ \mathsf{int}$$

where $\vec{r} = \{p, q_1, q_2, q_3, q_4, q_5, q_6, q_7, v, w, x, y\}$, so that the function may be used with differing resource behaviours.

Simplifying ψ by eliminating the intermediate variables (q_i) and summing the constant cost parameters to give some suitable constant C_i can help make the constraint set more comprehensible for human readers. These constraints in fact reduce to:

$$x + p \geq v + C_1 \qquad\qquad w \geq x + C_2$$

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 `Cons`-constructor, bearing a potential of p. Both x and y can have arbitrary values, which is sound since we ignored the terminating case branch in this example. For the full `fold` function, we need to similarly examine the branch dealing with `[]`, 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 $8l + 8$, the stack consumption is $10l + 16n + 14$, and the time consumption is $2850l + 938n + 821$.

```
SCHOPENHAUER typing for HeapBoxed:
 (tree[Leaf<8>:int|Node:#,#]) -(8/0)->
                                  list[Cons:int,#|Nil]
SCHOPENHAUER typing for StackBoxed:
 (tree[Leaf<10>:int|Node<16>:#,#]) -(14/23)->
                                  list[Cons:int,#|Nil]
SCHOPENHAUER typing for TimeM32:
 (tree[Leaf<2850>:int|Node<938>:#,#]) -(821/0)->
                                  list[Cons:int,#|Nil]
```

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 `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 using a tree-map function.

The third block in Figure 1 compares analysis and measurement results for the `repmin` example. Again the bounds for heap are exact. For stack, we observe a linear bound but 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 construct any intermediate list structures. The second variant uses the higher-order functions `map` and `fold` to compute the squares over all list elements and to sum the result, respectively. The third version additionally uses the higher-order function `unfold` to generate the initial list of numbers from 1 to n.

	N = 10			
	Calls	Heap	Stack	Time
sum-of-squares (variant 1: direct recursion)				
Analysis	22	114	30	18091
Measured	22	108	30	16874
Ratio	1.00	1.06	1.00	1.07
sum-of-squares (variant 2: with map and fold)				
Analysis	56	200	114	53612
Measured	56	200	112	42252
Ratio	1.00	1.00	1.02	1.27
sum-of-squares (variant 3: also unfold)				
Analysis	71	272	181	77437
Measured	71	272	174	59560
Ratio	1.00	1.00	1.04	1.30

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

```
-- common code
sq n = n*n;   add m n = m+n;
-- map, (left-) fold over lists are standard
-- enumFromTo m n generates [m..n] (tail-recursive)
-------------------------------------------------------
-- variant 1: direct recursion
sum_sqs' n m s = if (m>n)
                 then s
                 else sum_sqs' (n-1) m (s+(sq n));
sum_sqs n = sum_sqs' n 1 0;
-------------------------------------------------------
-- variant 2: uses h-o fcts fold and map
sum xs = fold add 0 xs;
sum_sqs n = sum (map sq (enumFromTo 1 n));
-------------------------------------------------------
-- variant 3: uses h-o fcts unfold, fold and map
data maybenum = Nothing | Just num;

unfoldr :: (num -> maybenum) -> num -> [num];
unfoldr f z = case f z of  Nothing -> []
                         | (Just z') -> z':(unfoldr f z');
countdown :: num -> maybenum;
countdown m = if (m<1) then Nothing else Just (m-1);

enum :: num -> [num];    -- this generates [n,n-1..1]
enum n = if (n<1) then [] else n:(unfoldr countdown n);

sum_sqs :: num -> num;
sum_sqs n = sum (map sq (enum n));
```

Figure 7. Source code of sum-of-squares (3 variants)

Table 2 summarises analysis and measurement results for all three variants. As expected, the higher-order versions of the code exhibit significantly higher resource consumption, notably for the second and third variants which generate two intermediate lists. These additional costs are accurately predicted by our analysis. In particular, the heap costs are exactly predicted and the stack costs are almost exact. The time results are within 30% of the measured costs. We consider this to be a very good worst-case estimate for higher-order code.

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

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:

```
type afct = a -> a;
twice :: afct -> afct;
twice f x = f (f x);

quad :: afct -> afct;
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:

```
revApp acc zs = case! zs of  [] -> acc
                           | (x:xs) -> revApp (x:acc) xs;
reverse xs = revApp [] xs;
```

```
type pred = num -> num -> bool;

insert :: pred -> num -> [num] -> [num];
insert cmp x zs = case! zs of
                    [] -> x : []
                  | (y:ys) -> if cmp x y then x : y : ys
                                         else y:insert cmp x ys;
sort :: pred -> [num] -> [num];
sort cmp zs = case! zs of
                [] -> []
              | (x:xs) -> insert cmp x (sort cmp xs);
leq :: pred;
leq x y = x<=y;

isort :: pred -> [num] -> [num];
isort xs = sort leq xs;
```

Figure 8. Source code of in-place insertion sort

The heap bound below shows that only constant heap space is needed for the `reverse` function. A constant heap space of 2 is needed for the initial `Nil` constructor passed to `revApp`. Due to the destructive nature of the pattern matches in `revApp`, the list cells of the input list can be re-used. Similarly, the `Nil` constructor of the input-list can be re-used for the result. Thus, the second 2 in the function type indicates that two resources are given back after completion of the `revApp` function. In total, the heap usage after execution is the same as before.

```
SCHOPENHAUER typing for HeapUnboxed:
list[Nil|Cons:int,#] -(2/2)-> list[Nil|Cons:int,#]
```

The stack and time consumption, however, are both linear:

```
SCHOPENHAUER typing for StackBoxed:
list[Nil<1>|Cons<3>:int,#] -(11/11)->
                                    list[Nil|Cons<1>:int,#]
SCHOPENHAUER typing for TimeM32:
list[Nil<225>|Cons<858>:int,#] -(481/0)->
                                       list[Nil|Cons:int,#]
```

A more interesting example is in-place insertion sort, parametrised over the comparison function (Figure 8). The `insert` function uses destructive pattern-matching to re-use the current cell of the input list when constructing the result list. The destructive pattern match in `sort` ensures that the call to `insert` has one list cell to start with. Using an unboxed heap model, which does not allocate heap space for the comparison operations, we can show that this function does not require any additional heap space:

```
SCHOPENHAUER typing for HeapUnboxed:
 (list[Cons:int,#|Nil]) -(0/0)-> list[Cons:int,#|Nil]
```

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,#,#
  |LetOp<9>: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
         | BinOp (val->val->val) exp exp;

_true = 1; _false = 0;

eval :: env -> exp -> val;
eval rho (Const n)       = n;
eval rho (VarOp v)       = rho v;
eval rho (IfOp v e1 e2)  = if (rho v)==_false
                             then eval rho e2
                             else eval rho e1;
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;
eval rho (UnOp  f m)   = f (eval rho e1);
eval rho (BinOp 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. Huelsbergen, Larus and Aiken [20] have defined an abstract interpretation of a higher-order, strict language for determining computation costs that dependent 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.

[†] 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].

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 RII3-CT-2005-026133 (SCIEnce); and by EPSRC grant EP/F030657/1 (Islay).

References

[1] A. Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006.

[2] 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.

[3] R. Benzinger. Automated Complexity Analysis of Nuprl Extracted Programs. *Journal of Functional Programming*, 11(1):3–31, 2001.

[4] M. Berkelaar, K. Eikland, and P. Notebaert. lp_solve: Open Source (Mixed-Integer) Linear Programming System. Published under GNU LGPL (Lesser General Public Licence). `http://lpsolve.sourceforge.net/5.5`.

[5] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *Proc. PLDI'03: Conf. on Programming Language Design and Implementation*, pages 196–207, San Diego, USA, June 2003. ACM.

[6] B. Campbell. Amortised Memory Analysis Using the Depth of Data Structures. In *Proc. ESOP 2009: European Symposium on Programming*, LNCS 5502, pages 190–204, York, UK, 2009. Springer.

[7] W.-N. Chin and S.-C. Khoo. Calculating Sized Types. *Higher-Order and Symbolic Computing*, 14(2,3):261–300, 2001.

[8] W.-N. Chin, H. Nguyen, C. Popeea, and S. Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *Proc. ISMM'08: Intl. Symp. on Memory Management*, pages 151–160, Tucson, USA, June 2008. ACM.

[9] K. Crary and S. Weirich. Resource Bound Certification. In *Proc. POPL'00: Symp. on Princ. of Prog. Langs*, pages 184–198, Boston, USA, 2000. ACM.

[10] N. Danielsson. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Proc. POPL'08: Symp. on Princ. of Prog. Langs*, pages 133–144, San Francisco, USA, 2008.

[11] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Proc EMSOFT'01: Intl Workshop on Embedded Software*, LNCS 2211, pages 469–485, Tahoe City, USA, Oct. 2001. Springer.

[12] G. Gomez and Y. Liu. Automatic Time-Bound Analysis for a Higher-Order Language. In *Proc. LCTES'98: Conf. on Languages, Compilers and Tools for Embedded Systems*, LNCS 1474, pages 31–40, Montreal, Canada, June 1998. Springer.

[13] S. Gulwani, K. Mehra, and T. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proc. POPL'09: Symp. on Princ. of Prog. Langs*, pages 127–139, Savannah, USA, Jan. 2009. ACM.

[14] 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.

[15] F. Henglein. Type Inference with Polymorphic Recursion. *ACM TOPLAS: Trans. Prog. Langs. Syst*s, 15(2):253–289, April 1993.

[16] J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polynomial Potential. In preparation.

[17] M. Hofmann. A Type System for Bounded Space and Functional In-Place Update. *Nordic Journal of Computing*, 7(4):258–289, 2000.

[18] M. Hofmann. The Strength of non Size-Increasing Computation. In *Proc. POPL'02: Symp. on Princ. of Prog. Langs*, pages 260–269, Portland, USA, Jan. 2002. ACM.

[19] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *Proc. POPL '03: Symp. on Princ. of Prog. Langs*, pages 185–197, New Orleans, USA, Jan. 2003. ACM.

[20] L. Huelsbergen, J. Larus, and A. Aiken. Using the Run-Time Sizes of Data Structures to Guide Parallel Thread Creation. In *Proc. LFP'94: Symp. on Lisp and Functional Programming*, pages 79–90, Orlando, USA, June 1994. ACM.

[21] R. Hughes and L. Pareto. Recursion and Dynamic Data Structures in Bounded Space: Towards Embedded ML Programming. In *Proc. ICFP '99: Intl. Conf. on Functional Programming*, pages 70–81, Paris, France, Sept. 1999. ACM.

[22] R. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *Proc. POPL'96: Symp. on Princ. of Prog. Langs*, pages 410–423, St. Petersburg Beach, USA, Jan. 1996. ACM.

[23] S. Ishtiaq and P. O'Hearn. BI as an Assertion Language for Mutable Data Structures. In *Proc. POPL'01: Symp. on Princ. of Prog. Langs.*, pages 14–26. ACM, Jan. 2001.

[24] H. Jonkers. Abstract Storage Structures. In *Algorithmic Languages*, pages 321–343. IFIP, North Holland, 1981.

[25] S. Jost, H.-W. Loidl, K. Hammond, N. Scaife, and M. Hofmann. "Carbon Credits" for Resource-Bounded Computations using Amortised Analysis. In *Proc. FM '09: Intl. Symp. on Formal Methods*, LNCS 5850, Eindhoven, the Netherlands, Nov. 2009. Springer.

[26] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type Reconstruction in the Presence of Polymorphic Recursion. *ACM TOPLAS: Trans. Prog. Langs. Syst*s, 15(2):290–311, April 1993.

[27] J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. POPL'93: Symp. on Princ. of Prog. Langs.*, pages 144–154, Charleston, USA, Jan. 1993. ACM.

[28] D. Le Métayer. ACE: An Automatic Complexity Evaluator. *ACM TOPLAS: Trans. on Prog. Langs. and Syst*s, 10(2), April 1988.

[29] B. Lisper. Fully Automatic, Parametric Worst-Case Execution Time Analysis. In *Proc. WCET '03: Intl. Workshop on Worst-Case Execution Time Analysis*, pages 99–102, 2003.

[30] H.-W. Loidl and S. Jost. Improvements to a Resource Analysis for Hume. In *Proc. FOPARA '09: Intl. Workshop on Foundational and Practical Aspects of Resource Analysis*, Nov. 2009. Submitted.

[31] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998. ISBN 0521663504.

[32] R. Pena, C. Segura, and M. Montenegro. A Sharing Analysis for Safe. In *Proc TFP'06: Symp. on Trends in Functional Programming*, pages 109–128, Nottingham, UK, Apr. 2006. Intellect.

[33] J. C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proc of the 25th ACM National Conference*, pages 717–740. ACM, 1972.

[34] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A Semantics for Procedure Local Heaps and its Abstractions. In *Proc. POPL'05: Symp. on Princ. of Prog. Langs*, pages 296–309. ACM, Jan. 2005.

[35] 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.

[36] W. Taha, S. Ellner, and H. Xi. Generating Heap-Bounded Programs in a Functional Setting. In *Proc. EMSOFT '03: Intl. Conf. on Embedded Software*, LNCS 2855, pages 340–355. Springer, 2003.

[37] R. E. Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, April 1985.

[38] P. Vasconcelos. *Cost Inference and Analysis for Recursive Functional Programs*. PhD thesis, University of St Andrews, Feb. 2008.

[39] 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.

[40] D. Walker and G. Morrisett. Alias Types for Recursive Data Structures. In *Proc. TIL'00: Types in Compilation*, LNCS 2071, pages 177–206. Springer, 2000.