Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs

Hugo Simões
Pedro Vasconcelos
Mário Florido
LIACC, Universidade do Porto,
Porto, Portugal
{hrsimoes,pbv,amf}@dcc.fc.up.pt

Steffen Jost
Ludwig Maximillians Universität,
Munich, Germany
jost@tcs.ifi.lmu.de

Kevin Hammond
University of St Andrews,
St Andrews, UK
kh@cs.st-andrews.ac.uk

Abstract

This paper describes the first successful attempt, of which we are aware, to define an automatic, type-based static analysis of resource bounds for lazy functional programs. Our analysis uses the automatic amortisation approach developed by Hofmann and Jost, which was previously restricted to eager evaluation. In this paper, we extend this work to a lazy setting by capturing the costs of unevaluated expressions in type annotations and by amortising the payment of these costs using a notion of *lazy potential*. We present our analysis as a proof system for predicting heap allocations of a minimal functional language (including higher-order functions and recursive data types) and define a formal cost model based on Launchbury's natural semantics for lazy evaluation. We prove the soundness of our analysis with respect to the cost model. Our approach is illustrated by a number of representative and non-trivial examples that have been analysed using a prototype implementation of our analysis.

1. Introduction

Non-strict functional programming languages, such as Haskell [36], offer important benefits in terms of modularity and abstraction [23]. A key practical obstacle to their wider use, however, is that extra-functional properties, such as time- and space-behaviour, are often difficult to determine prior to actually running the program. Recent advances in static cost analyses, such as *sized-timed types* [43, 44] and *type-based amortisation* [18, 19] have enabled the *automatic* prediction of resource bounds for eager functional programs, including uses of higher-order functions [29]. This paper extends type-based amortisation to lazy evaluation, describing a static analysis for determining *a-priori* worst-case bounds on execution costs (specifically, dynamic memory allocations).

This paper makes the following novel contributions:

a) we present the first successful attempt, of which we are aware, to produce an automatic, type-based, static analysis of resource bounds for lazy evaluation;
b) we introduce a cost model for heap allocations for a minimal lazy functional language based on Launchbury's natural semantics for lazy evaluation [30], and use this as the basis for developing a resource analysis;
c) we have proved the soundness of our analysis with respect to the cost-instrumented semantics (due to space limitations, we present only a proof sketch); and
d) we provide results from a prototype implementation to show the applicability of our analysis to some non-trivial examples.*

Our amortised analysis derives costs with respect to a cost semantics for lazy evaluation that derives from Launchbury's natural operational semantics of graph reduction. It deals with both first-order and higher-order functions, but does not consider polymorphism. For simplicity, we restrict our attention to heap allocations†, but previous results have shown that the amortised analysis approach also extends to other countable resources, such as worst-case execution time [28]. In order to ensure a good separation of concerns, our analysis assumes the availability of Hindley-Milner type information. We extend Hofmann and Jost's type annotations for capturing *potential* costs [19] with information about the lazy evaluation context. The analysis produces a set of constraints over cost variables that we solve in our prototype implementation using an external LP-solver. We have thus demonstrated all the steps that are necessary to produce a fully-automatic analysis for determining bounds on resource usage for lazily-evaluated programs.

2. A Cost Model for Lazy Evaluation

Our cost model is built on Sestoft's revision [40] of Launchbury's natural semantics for lazy evaluation [30]. Launchbury's semantics forms one of the earliest and most widely-used operational accounts of lazy evaluation for the λ-calculus. De la Encina and Peña-Marí [13, 14] subsequently proved that the *Spineless Tagless G-Machine* [24] is sound and complete w.r.t. one of Sestoft's ab-

* The detailed soundness proof and a web version of the analysis is available at `http://www.dcc.fc.up.pt/~pbv/cgi/aalazy.cgi`

† Note that, because we do not consider deallocation, we model total allocation but not residency.

stract machines. We therefore have a high degree of confidence that the cost model for lazy evaluation developed here is not just theoretically sound, but also that it could, in principle, be extended to model real implementations of lazy evaluation.

2.1 Syntax

The syntax of *initial expressions* (the subject of our cost analysis) is the λ-calculus extended with local bindings, data constructors and pattern matching:

$$\begin{array}{rcl} e & ::= & x \;\mid\; \lambda x.\, e \;\mid\; e\, x \\ & \mid & \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \;\mid\; \mathsf{letcons}\ x = c(\vec{y})\ \mathsf{in}\ e \\ & \mid & \mathsf{match}\ e_0\ \mathsf{with}\ c(\vec{x})\ \texttt{->}\ e_1\ \mathsf{otherwise}\ e_2 \end{array}$$

As in Launchbury's semantics, we restrict the arguments of applications to be variables and we require that nested applications be translated into nested let-bindings.[‡] *let*-expressions bind variables to possibly recursive terms. In line with common practice in non-strict functional languages, we do not have a separate *letrec* form, as in ML. For simplicity, we consider only single-variable let-bindings: multiple let-bindings can be encoded, if needed, using pairs and projections. Note that constructor applications $c(\vec{x})$ will never occur in the initial expression. They are only ever introduced through evaluation of *letcons*-expressions. This is the main difference between our notation and those of Launchbury or Sestoft. The difference is motivated by the need to syntactically distinguish *allocating* a new constructor from simply *referencing* an existing one. De la Encina and Peña-Marí use a similar notation. Our operational semantics is defined over *augmented expressions*, $\widehat{e}$, that include these constructor applications:

$$\widehat{e} \quad ::= \quad e \;\mid\; c(\vec{\ell})$$

An evaluation result is then an (augmented) expression w, which is in *weak head normal form* (*whnf*), i.e. it is a λ-abstraction or constructor application.

$$w \quad ::= \quad \lambda x.e \;\mid\; c(\vec{\ell})$$

In the remainder of this paper we will use lowercase letters x, y for bound variables in initial expressions and $\ell, \not{k}$ for "fresh" variables (designated *locations*) that are introduced through evaluation of *let*- and *letcons*-expressions.

2.2 Cost-instrumented operational semantics

Figure 1 defines an instrumented big-step operational semantics for lazy evaluation that we will use as the basis for our analysis. Our semantics is given as a relation $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} \widehat{e} \Downarrow w, \mathcal{H}'$, where $\widehat{e}$ is an augmented expression; $\mathcal{H}$ is a *heap* mapping variables to augmented expressions (*thunks*, that may require evaluation to weak head normal form); $\mathcal{S}$ is a set of bound variables that are used to ensure the freshness condition in the LET$_\Downarrow$/LETCONS$_\Downarrow$ rules; and $\mathcal{L}$ is a set of variables used to record thunks that are under evaluation and to prevent cyclic evaluation (similar to the well-known "black-hole" technique used in [30]). The result of evaluation is an expression w in *whnf* and a final heap $\mathcal{H}'$. The parameters m, m' are non-negative integers representing the number of available heap locations before and after evaluation, respectively. The purpose of the analysis that will be developed in Section 3 is to obtain a static approximation for m that will safely allow execution to proceed. For readability, we may omit the resource information from judgements when they are not otherwise mentioned, writing simply $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash \widehat{e} \Downarrow w, \mathcal{H}'$ instead of $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} \widehat{e} \Downarrow w, \mathcal{H}'$.

[‡] This transformation does not increase worst-case costs because, in a call-by-need setting, function arguments must, in general, be heap-allocated in order to allow in-place update and sharing of normal forms.

$$\frac{w \text{ is in whnf}}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m} w \Downarrow w, \mathcal{H}} \qquad (\text{WHNF}_\Downarrow)$$

$$\frac{\ell \notin \mathcal{L} \qquad \mathcal{H}, \mathcal{S}, \mathcal{L} \cup \{\ell\} \vdash^{m}_{m'} \mathcal{H}(\ell) \Downarrow w, \mathcal{H}'}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} \ell \Downarrow w, \mathcal{H}'[\ell \mapsto w]} \qquad (\text{VAR}_\Downarrow)$$

$$\frac{\begin{array}{c}\ell \text{ is fresh} \qquad e_1' = e_1[\ell/x] \qquad e_2' = e_2[\ell/x] \\ \mathcal{H}[\ell \mapsto e_1'], \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e_2' \Downarrow w, \mathcal{H}'\end{array}}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m+1}_{m'} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow w, \mathcal{H}'} \qquad (\text{LET}_\Downarrow)$$

$$\frac{\begin{array}{c}\ell \text{ is fresh} \qquad y_i' = y_i[\ell/x] \qquad e' = e[\ell/x] \\ \mathcal{H}[\ell \mapsto c(\vec{y'})], \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e' \Downarrow w, \mathcal{H}'\end{array}}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m+1}_{m'} \mathsf{letcons}\ x = c(\vec{y})\ \mathsf{in}\ e \Downarrow w, \mathcal{H}'} \quad (\text{LETCONS}_\Downarrow)$$

$$\frac{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e \Downarrow \lambda x.\, e', \mathcal{H}' \qquad \mathcal{H}', \mathcal{S}, \mathcal{L} \vdash^{m'}_{m''} e'[\ell/x] \Downarrow w, \mathcal{H}''}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m''} e\, \ell \Downarrow w, \mathcal{H}''} \qquad (\text{APP}_\Downarrow)$$

$$\frac{\begin{array}{c}\mathcal{H}, \mathcal{S} \cup \{\vec{x}\} \cup \mathrm{BV}(e_1) \cup \mathrm{BV}(e_2), \mathcal{L} \vdash^{m}_{m'} e_0 \Downarrow c(\vec{\ell}), \mathcal{H}' \\ \mathcal{H}', \mathcal{S}, \mathcal{L} \vdash^{m'}_{m''} e_1[\vec{\ell}/\vec{x}] \Downarrow w, \mathcal{H}''\end{array}}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m''} \mathsf{match}\ e_0\ \mathsf{with}\ c(\vec{x})\texttt{->}e_1\ \mathsf{otherwise}\ e_2 \Downarrow w, \mathcal{H}''} \quad (\text{MATCH}_\Downarrow)$$

$$\frac{\begin{array}{c}\mathcal{H}, \mathcal{S} \cup \{\vec{x}\} \cup \mathrm{BV}(e_1) \cup \mathrm{BV}(e_2), \mathcal{L} \vdash^{m}_{m'} e_0 \Downarrow w', \mathcal{H}' \\ w' \neq c(\vec{\ell}) \qquad \mathcal{H}', \mathcal{S}, \mathcal{L} \vdash^{m'}_{m''} e_2 \Downarrow w, \mathcal{H}''\end{array}}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m''} \mathsf{match}\ e_0\ \mathsf{with}\ c(\vec{x})\texttt{->}e_1\ \mathsf{otherwise}\ e_2 \Downarrow w, \mathcal{H}''} \quad (\text{FAIL}_\Downarrow)$$

Figure 1. Cost-instrumented Operational Semantics

The only rules that bind variables to expressions in the heap are LET$_\Downarrow$ and LETCONS$_\Downarrow$. These are therefore the only places where new fresh locations are needed. These heap allocations may either allocate new constructors (*letcons*), or thunks or λ-abstractions (*let*). For simplicity, but without loss of generality, we choose to use a uniform cost model: evaluation will cost one (heap) unit for each fresh heap location that is needed during evaluation. Other cost models are also possible [28], modelling the usage of other countable resources such as execution time, or stack usage, for example. The WHNF$_\Downarrow$ rule for weak-head normal forms (λ-expressions and constructors) incurs no cost. Any costs must have been already accounted for by an initial *let*- or *letcons*-expression. The VAR$_\Downarrow$ and APP$_\Downarrow$ rules are identical to the equivalent ones in Launchbury's semantics. The VAR$_\Downarrow$ rule is restricted to locations that are not marked as being under evaluation (so enforcing "black-holing"). The MATCH$_\Downarrow$ and FAIL$_\Downarrow$ cases deal respectively with successful/unsuccessful pattern matches against a constructor. These rules record the bound variables in e_1 plus the new bound variables in $\vec{x}$ solely in order to ensure freshness in the LET$_\Downarrow$/LETCONS$_\Downarrow$ rules.

We now give the auxiliary definition[§] that formalises the notion of freshness of variables and a lemma regarding the preservation of locations that are marked as "black-holes".

Definition 2.1 (Freshness). A variable x is *fresh* in judgement $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash \widehat{e} \Downarrow w, \mathcal{H}'$ if x does not occur in either $\mathrm{dom}(\mathcal{H})$, $\mathcal{L}$ or $\mathcal{S}$ nor does it occur bound in either $\widehat{e}$ or $\mathrm{ran}(\mathcal{H})$.

[§] Due to de La Encina and Peña-Marí [13].

Lemma 2.2 (Invariant Black Holes)**.** *If $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash \widehat{e} \Downarrow w, \mathcal{H}'$ then for all $\ell \in \mathcal{L}$ we have $\mathcal{H}'(\ell) = \mathcal{H}(\ell)$. In other words, heap locations that are under evaluation are preserved during intermediate evaluations.*

Proof. By inspection of the operational semantics (Figure 1) we observe that $\text{VAR}_{\Downarrow}$ is the only rule that modifies an existing location ℓ and that this rule does not apply when $\ell \in \mathcal{L}$. □

2.3 Example: call-by-need versus call-by-value/call-by-name

Consider the expression below, which includes a divergent term:

$$\mathsf{let}\ z = z\ \mathsf{in}\ (\lambda x.\, \lambda y.\, y)\ z \tag{2.1}$$

Under a *call-by-value* semantics, this would fail to terminate, because z does not admit a normal form. In our *call-by-need* semantics, however, evaluation succeeds:

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{1}_{0} \mathsf{let}\ z = z\ \mathsf{in}\ (\lambda x.\, \lambda y.\, y)\ z \Downarrow \lambda y.\, y, \mathcal{H}[\ell_3 \mapsto \ell_3]$$

The final heap is augmented with a fresh location ℓ_3 whose content is a cyclic self-reference; because the argument z is discarded by the application, its evaluation is never attempted. We can see that the semantics is call-by-need rather than call-by-name by observing the sharing of normal forms. Consider,

$$\begin{aligned} &\mathsf{let}\ f = \mathsf{let}\ z = z\ \mathsf{in}\ (\lambda x.\, \lambda y.\, y)\ z \\ &\quad \mathsf{in}\ \mathsf{let}\ i = \lambda x.\, x\ \mathsf{in}\ \mathsf{let}\ v = f\ i\ \mathsf{in}\ f\ v \end{aligned} \tag{2.2}$$

where f is bound to the thunk (2.1) and applied twice to the identity function. Evaluation of $f\ v$ forces the thunk. After the thunk is evaluated, the location ℓ_0 that is associated with f is updated with the corresponding *whnf*, $\lambda y.\, y$. The second evaluation of f does not *not* re-evaluate the thunk (2.1). Starting from the empty configuration, we derive:

$$\begin{aligned} \emptyset, \emptyset, \emptyset \vdash^{4}_{0}\ &(2.2) \Downarrow \lambda x.\, x, \\ &[\ell_0 \mapsto \lambda y.\, y, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \lambda x.\, x, \ell_3 \mapsto \ell_3] \end{aligned}$$

Evaluating expression (2.2) thus costs four heap cells, that is, one cell for each let-expression. Under a *call-by-name* semantics, the cost would instead be 5, since the let-expression that is bound to f would then be evaluated twice, rather than once as here.

3. An Amortised Analysis for Lazy Evaluation

Our type-based cost analysis is based on the principle of *amortisation*, that is, averaging the costs of individual operations over a sequence of such operations. It is often possible to obtain better worst-case bounds by amortisation than by reasoning about the costs of single operations. For example, we may obtain a worst-case bound of $O(n)$ for a sequence of n operations even if some of the individual operations cost more than $O(1)$. Amortisation has been successfully used in manual complexity analysis of data structures in both imperative [42] and functional settings [35] and for automatic resource analysis of strict functional languages [18, 19, 27, 29]. It has never been previously used for automatic resource analysis of lazy evaluation. One method for deriving amortised bounds starts by defining a *potential function* from data structures to real numbers. The *amortised cost* of an operation is defined as $t + \phi' - \phi$, where t is the actual cost of the operation (e.g. time or memory) and ϕ, ϕ' are the potentials of the data before and after the operation. The key objective is to choose the potential function so that it simplifies the amortised costs, e.g. so that the change in potential offsets any variation in actual costs, and the amortised costs are therefore constant.

We assign potential to data structures in a type-directed way: recursive data types are annotated with positive coefficients that specify the contribution of each constructor to the potential of the data structure. For example, if we annotate the empty list constructor with q_{nil} and the non-empty list constructor with $q_{\mathtt{cons}}$, then the overall potential of a list of n elements (ignoring any potential for the list elements themselves) is $q_{\mathtt{nil}} + n \times q_{\mathtt{cons}}$, as expected. The principal advantage of this choice is that we can use efficient linear constraint solvers to automatically determine suitable type annotations. The main limitation is that we can only express potentials and costs that are linear functions of the number of constructors in a data structure. Recent work by Hoffmann et al. [18] shows that multivariate polynomial cost functions can also be efficiently inferred, however, and still only require linear constraint solving.

A crucial difference between classic amortised analysis [35, 42] and type-based amortised analysis is that the type system can keep track of data sharing through an explicit structural rule. This allows potential to be defined *per-reference* reflecting how often a data structure is accessed. The advantage is that we do not require ephemeral usage of data structures to ensure the soundness of amortisation. The disadvantage is that (fully evaluated) cyclic data can only be assigned either zero or infinite potential, and that the type system requires an extra structural rule.

It is important to note that, although we are defining a static analysis, the overall potential for any actual data structure can only be known dynamically, when the concrete data size is known. We never actually need to compute this potential, however, but rather concern ourselves with the *change* in potential along all possible computation paths.

3.1 Annotated types and contexts

The syntax of annotated types includes type variables, functions, thunks and (possibly recursive) data types over labelled sums of products, representing the types of each constructor.

$$\begin{aligned} A, B, C \quad ::= \quad & X \mid A \xrightarrow[q']{q} B \mid \mathsf{T}^{q}_{q'}(A) \\ & \mid \quad \mu X.\{c_1 : (q_1, \vec{B}_1) \mid \cdots \mid c_n : (q_n, \vec{B}_n)\} \end{aligned}$$

We use meta-variables A, B, C for types, X, Y for type variables and p, q for annotations (i.e. non-negative rational numbers, representing potential). Typing contexts are *multisets* of pairs $x{:}A$ of variables and annotated types; we use multisets to allow separate potential to be accounted for in multiple references. We use Γ, Δ etc for contexts and $\Gamma\restriction_x$ for the multiset of types associated with x in Γ, i.e. $\Gamma\restriction_x = \{A \mid x{:}A \in \Gamma\}$.

The annotations q, q' in the function type $A \xrightarrow[q']{q} B$ express the resources before and after evaluation (hence its cost); similarly, the annotations q, q' in a type $\mathsf{T}^{q}_{q'}(A)$ capture the cost of evaluating a thunk (this can be zero if the thunk is known to be in *whnf*). For simplicity, we exclude *resource parametricity* [29], since this is only important for functions that are re-used in different circumstances, and not for thunks that are evaluated at most once. It is thus orthogonal to this paper.

In a (possibly recursive) data type $\mu X.\{c_1 : (q_1, \vec{B}_1) \mid \ldots \mid c_n : (q_n, \vec{B}_n)\}$ each coefficient q_i represents the potential associated with one application of constructor c_i. We consider only recursive data types that are *non-interleaving* [32], i.e. we exclude μ-types whose bound variables overlap in scope (e.g. $\mu X.\big\{c_1 : \big(\ldots, \mu Y.\{c_2 : (\ldots, X)\}\big)\big\}$). This helps us prove a crucial lemma on cyclic structures in the key soundness proof (Theorem 1). Note that this restriction does *not* prohibit nested data types; e.g. the type of lists of lists of naturals is $\mu Y.\{\mathtt{nil} : (q'_n, ()), \mathtt{cons} : (q'_c, (\mathsf{LN}, Y))\}$, where $\mathsf{N} = \mu X.\{\mathtt{zero} : (q_z, ()), \mathtt{succ} : (q_s, X)\}$ is the type of naturals and $\mathsf{LN} = \mu Y.\{\mathtt{nil} : (q_n, ()), \mathtt{cons} : (q_c, (\mathsf{N}, Y))\}$ is the type of list of naturals. Note also that distinct lists can be assigned different constructor annotations in their types, thus improving the precision of the cost analysis.

$$\frac{}{\curlyvee(A \mid \emptyset)} \quad (\textsc{ShareEmpty})$$

$$\frac{}{\curlyvee(X \mid X, \ldots, X)} \quad (\textsc{ShareVar})$$

$$\frac{\begin{array}{c} B_i = \mu X.\{c_1 : (q_{i1}, \vec{B}_{i1}) \mid \cdots \mid c_m : (q_{im}, \vec{B}_{im})\} \\ \curlyvee\Big(\vec{A}_j \,\Big|\, \vec{B}_{1j}, \ldots, \vec{B}_{nj}\Big) \qquad p_j \geq \sum_{i=1}^{n} q_{ij} \qquad (1 \leq i \leq n,\ 1 \leq j \leq m) \end{array}}{\curlyvee\Big(\mu X.\{c_1 : (p_1, \vec{A}_1) \mid \cdots \mid c_m : (p_m, \vec{A}_m)\} \ \mid\ B_1, \ldots, B_n\Big)} \quad (\textsc{ShareDat})$$

$$\frac{\curlyvee(A_i \mid A) \qquad \curlyvee(B \mid B_i) \qquad q_i \geq q \qquad q_i - q \geq q'_i - q' \qquad (1 \leq i \leq n)}{\curlyvee\Big(A \xrightarrow[q']{q} B \,\Big|\, A_1 \xrightarrow[q'_1]{q_1} B_1, \ldots, A_n \xrightarrow[q'_n]{q_n} B_n\Big)} \quad (\textsc{ShareFun})$$

$$\frac{\curlyvee(A \mid A_1, \ldots, A_n) \qquad q_i \geq q \qquad q_i - q \geq q'_i - q' \qquad (1 \leq i \leq n)}{\curlyvee\Big(\mathsf{T}^{q}_{q'}(A) \,\Big|\, \mathsf{T}^{q_1}_{q'_1}(A_1), \ldots, \mathsf{T}^{q_n}_{q'_n}(A_n)\Big)} \quad (\textsc{ShareThunk})$$

$$\frac{\curlyvee(A_j \mid B_{1j}, \ldots, B_{nj}) \qquad m = |\vec{A}| = |\vec{B}_i| \qquad (1 \leq i \leq n,\ 1 \leq j \leq m)}{\curlyvee\Big(\vec{A} \,\Big|\, \vec{B}_1, \ldots, \vec{B}_n\Big)} \quad (\textsc{ShareVec})$$

$$\frac{}{\curlyvee(\Gamma \mid \emptyset)} \quad (\textsc{ShareEmptyCtx})$$

$$\frac{\curlyvee(A \mid B_1, \ldots, B_n) \qquad \curlyvee(\Gamma \mid \Delta)}{\curlyvee(x : A, \Gamma \mid x : B_1, \ldots, x : B_n, \Delta)} \quad (\textsc{ShareCtx})$$

Figure 2. Sharing Relation

3.2 Sharing and Subtyping

Figure 2 shows the syntactical rules for an auxiliary judgement $\curlyvee(A \mid B_1, \ldots, B_n)$ that is used to *share* a type A among a finite multiset of types $\{B_1, \ldots, B_n\}$. It is used to limit contraction in our type system. Datatype annotations for potential associated with A are linearly distributed by the $\curlyvee$ relation among $B_1, \ldots, B_n$, whereas cost annotations for functions and thunks are preserved. Sharing also allows the relaxing of annotations to subsume subtyping (i.e. potential annotations can decrease, cost annotations may increase). It is important to note that a *decrease* of cost annotations for thunks (possibly down to zero) can only be achieved through the PREPAY structural rule (Figure 4) and not through these sharing rules. "Pre-paying" allows us to correctly model the reduced costs of lazy evaluation by allowing costs to be accounted only once for a thunk. The SHAREEMPTY, SHAREVAR and SHAREVEC rules are trivial. The SHAREDAT rule allows potential from the data constructors that comprise A to be shared among the B_i. The SHAREFUN and SHARETHUNK rules allow any costs for functions and thunks, respectively, to be replicated. The SHARECTXEMPTY and SHARECTX rules extend the sharing relation for typing contexts in a pointwise manner: Γ shares to Δ *iff* for each type assignment $x{:}A$ in Γ there exists $x{:}B_1, \ldots, x{:}B_n$ in Δ and A shares to $B_1, \ldots, B_n$. The special case of sharing one type to a single other corresponds to a *subtyping relation*; we define the shorthand notation $A <: B$ to mean $\curlyvee(A \mid B)$. This relation expresses the relaxation of potentials and costs: informally, $A <: B$ implies that A, B have identical underlying types but B has *lower or equal potential* and *greater or equal cost* than that of A. As usual in structural subtyping, this relation is contravariant in the left argument of functions (SHAREFUN). A special case occurs when sharing a type or context to itself: because of non-negativity $\curlyvee(A \mid A, A)$ (respectively $\curlyvee(\Gamma \mid \Gamma, \Gamma)$), requires that the potential annotations in A (respectively Γ) be zero. We use this property to impose a constraint that types or contexts carry no potential. A variant of this is $\curlyvee(A \mid A, A')$, which implies that A' is a subtype of A that holds no potential.

3.3 Typing judgements

Our analysis is presented in Figures 3 and 4 as a proof system that derives judgments of the form $\Gamma \vdash^{p}_{p'} \widehat{e} : A$, where Γ is a typing context, $\widehat{e}$ is an augmented expression, A is an annotated type and p, p' are non-negative numbers approximating the resources available before and after the evaluation of $\widehat{e}$, respectively. For simplicity, we will omit these annotations whenever they are not explicitly mentioned. Because variables reference heap expressions, rules dealing with the introduction and elimination of variables also deal with the introduction and elimination of thunk types: VAR eliminates an assumption of a thunk type, i.e. of the form $x : \mathsf{T}^{q}_{q'}(A)$. Dually, LET and LETCONS introduce an assumption of a thunk type. Note that LETCONS is not simply identical to a LET rule that allows augmented expressions to be bound, since it accounts for the constructor potential q differently. In order to avoid duplicating potential where a λ-abstraction is applied more than once, ABS ensures that Γ does not carry potential, by forcing it to share with itself. APP ensures that the argument and function types match and includes the cost of the function in the final result. The CONS rule simply ensures consistency between the arguments and the result type. Since constructors cannot appear in source forms, the rule is used only when we need to assign types either to heap expressions or to evaluation results. The MATCH rule deals with pattern-matching over an expression of a (possibly recursive) data type. The rule requires that both branches admit an identical result type and that estimated resources after execution of either branch are equal; fulfilling such a condition may require relaxing type and/or cost information using the structural rules below. The matching branch uses extra resources corresponding to the potential annotation on the matched constructor. The structural rules of Figure 4 allow the analysis to be relaxed in various ways: WEAK allows the introduction of an extra hypothesis in the typing context; RELAX

$$\frac{}{x{:}\mathsf{T}^{p}_{p'}(A)\ \vdash^{p}_{p'}\ x : A} \qquad (\textsc{Var})$$

$$\frac{x \notin \mathrm{dom}(\Gamma,\Delta) \qquad \curlyvee(A \mid A, A') \qquad q \geq q' \qquad \Gamma, x{:}\mathsf{T}^{0}_{0}(A')\ \vdash^{q}_{q'}\ e_1 : A \qquad \Delta, x{:}\mathsf{T}^{q}_{q'}(A)\ \vdash^{p}_{p'}\ e_2 : C}{\Gamma, \Delta \vdash^{1+p}_{p'} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : C} \qquad (\textsc{Let})$$

$$\frac{\begin{array}{c} A = \mu X.\{\cdots \mid c : (q, \vec{B}) \mid \cdots\} \qquad x \notin \mathrm{dom}(\Gamma,\Delta) \qquad \curlyvee(A \mid A, A') \\ \Gamma, x{:}\mathsf{T}^{0}_{0}(A')\ \vdash^{0}_{0}\ c(\vec{y}) : A \qquad \Delta, x{:}\mathsf{T}^{0}_{0}(A)\ \vdash^{p}_{p'}\ e : C \end{array}}{\Gamma, \Delta \vdash^{1+q+p}_{p'} \mathsf{letcons}\ x = c(\vec{y})\ \mathsf{in}\ e : C} \qquad (\textsc{Letcons})$$

$$\frac{\Gamma, x{:}A \vdash^{q}_{q'} e : C \qquad x \notin \mathrm{dom}(\Gamma) \qquad \curlyvee(\Gamma \mid \Gamma, \Gamma)}{\Gamma \vdash^{0}_{0} \lambda x.e : A \xrightarrow[q']{q} C} \qquad (\textsc{Abs})$$

$$\frac{\Gamma \vdash^{p}_{p'} e : A \xrightarrow[q']{q} C}{\Gamma, y{:}A \vdash^{p+q}_{p'+q'} e\, y : C} \qquad (\textsc{App})$$

$$\frac{B = \mu X.\{\cdots \mid c : (q, \vec{A}) \mid \cdots\}}{y_1{:}A_1[B/X], \ldots, y_k{:}A_k[B/X] \vdash^{0}_{0} c(\vec{y}) : B} \qquad (\textsc{Cons})$$

$$\frac{\begin{array}{c} B = \mu X.\{\cdots \mid c : (q, \vec{A}) \mid \cdots\} \qquad |\vec{A}| = |\vec{x}| = k \qquad x_i \notin \mathrm{dom}(\Delta)\,,\ \text{for all } i \\ \Gamma \vdash^{p}_{p'} e_0 : B \qquad \Delta \vdash^{p'}_{p''} e_2 : C \qquad \Delta, x_1{:}A_1[B/X], \ldots, x_k{:}A_k[B/X] \vdash^{p'+q}_{p''} e_1 : C \end{array}}{\Gamma, \Delta \vdash^{p}_{p''} \mathsf{match}\ e_0\ \mathsf{with}\ c(\vec{x})\text{->}e_1\ \mathsf{otherwise}\ e_2 : C} \qquad (\textsc{Match})$$

Figure 3. Syntax Directed Type Rules

$$\frac{\Gamma \vdash^{p}_{p'} e : C}{\Gamma, x{:}A \vdash^{p}_{p'} e : C} \qquad (\textsc{Weak})$$

$$\frac{\Gamma \vdash^{p_0}_{p'_0} e : A \qquad p \geq p_0 \qquad p - p_0 \geq p' - p'_0}{\Gamma \vdash^{p}_{p'} e : A} \qquad (\textsc{Relax})$$

$$\frac{\Gamma, x{:}\mathsf{T}^{q_0}_{q'}(A)\ \vdash^{p}_{p'}\ e : C \qquad q_0 \geq q'}{\Gamma, x{:}\mathsf{T}^{q_0+q_1}_{q'}(A)\ \vdash^{p+q_1}_{p'}\ e : C} \qquad (\textsc{Prepay})$$

$$\frac{\Gamma, x{:}A_1, x{:}A_2 \vdash^{p}_{p'} e : C \qquad \curlyvee(A \mid A_1, A_2)}{\Gamma, x{:}A \vdash^{p}_{p'} e : C} \qquad (\textsc{Share})$$

$$\frac{\Gamma, x : B \vdash^{p}_{p'} e : C \qquad A <: B}{\Gamma, x : A \vdash^{p}_{p'} e : C} \qquad (\textsc{Supertype})$$

$$\frac{\Gamma \vdash^{p}_{p'} e : B \qquad B <: C}{\Gamma \vdash^{p}_{p'} e : C} \qquad (\textsc{Subtype})$$

Figure 4. Structural Type Rules

allows argument costs to be relaxed; PREPAY allows (part of) the cost of a thunk to be paid for, so reducing the cost of further uses; SUPERTYPE and SUBTYPE allow supertyping in a hypothesis and subtyping in the conclusion, respectively; finally, SHARE allows the use of sharing to split potential in a hypothesis.

Because our semantics does not deallocate resources, it can be expected that all the "lower" annotations in the type system can be set to zero, i.e. the p' in a type judgement, and the q' in function and thunk types (but *not* the m' in an evaluation judgement). However, fixing them to zero would increase the complexity of our soundness proof [26, Section 2.1] and we have therefore retained them.

3.4 Worked examples

We now present type derivations for the examples from Section 2.3 in order to illustrate how the type rules of Figures 3 and 4 model the costs of our operational semantics. Recall example (2.1) which demonstrates that unneeded redexes are not reduced (i.e., that the semantics is non-strict):

$$\mathsf{let}\ z = z\ \mathsf{in}\ (\lambda x.\, \lambda y.\, y)\, z$$

Evaluation of this term in our operational semantics succeeds and requires one heap cell (for allocating the thunk named by z):

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{1}_{0} \mathsf{let}\ z = z\ \mathsf{in}\ (\lambda x.\, \lambda y.\, y)\, z \Downarrow \lambda y.y, \mathcal{H}'$$

An analysis for this term is given in Figure 5 as an annotated type derivation with the following final judgement:

$$\emptyset \vdash^{1}_{0} \mathsf{let}\ z = z\ \mathsf{in}\ (\lambda x.\lambda y.y)\, z : \mathsf{T}^{q}_{q'}(B) \xrightarrow[q']{q} B$$

The annotations in the turnstile of this judgement give a cost estimate of one heap cell, matching the exact cost of the operational se-

$$\dfrac{\left\{\begin{array}{c} \dfrac{}{z{:}\mathsf{T}^0_0(A) \vdash^{0}_{0} z : A}\,\textsc{Var} \\[2ex] \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y{:}\mathsf{T}^q_{q'}(B) \vdash^{q}_{q'} y : B}\,\textsc{Var}}{\emptyset \vdash^{0}_{0} \lambda y.y : \mathsf{T}^q_{q'}(B) \xrightarrow[q']{q} B}\,\textsc{Abs}}{x{:}\mathsf{T}^p_{p'}(A) \vdash^{0}_{0} \lambda y.y : \mathsf{T}^q_{q'}(B) \xrightarrow[q']{q} B}\,\textsc{Weak}}{\emptyset \vdash^{0}_{0} \lambda x.\lambda y.y : \mathsf{T}^p_{p'}(A) \xrightarrow[0]{0} \mathsf{T}^q_{q'}(B) \xrightarrow[q']{q} B}\,\textsc{Abs}}{z{:}\mathsf{T}^p_{p'}(A) \vdash^{0}_{0} (\lambda x.\lambda y.y)\, z : \mathsf{T}^q_{q'}(B) \xrightarrow[q']{q} B}\,\textsc{App} \end{array}\right.}{\emptyset \vdash^{1}_{0} \mathsf{let}\ z = z\ \mathsf{in}\ (\lambda x.\lambda y.y)\, z : \mathsf{T}^q_{q'}(B) \xrightarrow[q']{q} B}\,\textsc{Let}$$

$$\text{where } p \geq p', q \geq q', \curlyvee(A \mid A, A) \tag{3.1}$$

Figure 5. Type derivation for non-strict evaluation example (2.1).

mantics. The result of the evaluation is the identity function, $\lambda y.y$. The type annotations q, q' represent the cost of the thunk for the argument. These parameters can be arbitrary, subject only to the side conditions $q \geq q'$. The type B is similarly arbitrary.

The second example (2.2) illustrates the sharing of normal forms, i.e. lazy evaluation:

$$\begin{array}{l} \mathsf{let}\ f = \mathsf{let}\ z = z\ \mathsf{in}\ (\lambda x.\, \lambda y.\, y)\, z \\ \quad \mathsf{in}\ \mathsf{let}\ i = \lambda x.\, x\ \mathsf{in}\ \mathsf{let}\ v = f\ i\ \mathsf{in}\ f\ v \end{array}$$

Evaluating $f\ v$ forces the thunk f; following evaluation, the location associated with f is updated with the *whnf*. Subsequent evaluation of f re-uses this result. Evaluation of the overall expression therefore costs 4 cells:

$$\begin{array}{l} \emptyset, \emptyset, \emptyset \vdash^{4}_{0} (2.2) \Downarrow \lambda x.\, x, \\ \qquad [\ell_0 \mapsto \lambda y.\, y, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \lambda x.\, x, \ell_3 \mapsto \ell_3] \end{array}$$

The type derivation in Figure 6 shows the analysis for this example, with the final type judgement replicating the exact operational cost of 4 heap cells. Note that we use the structural rule Prepay to pay the cost of the thunk that is bound to f precisely once. We also employ Share to allow the function f to be used twice. The duplication is justified because the type of f carries no potential (i.e. it shares to itself).

4. Experimental results

We have constructed a prototype implementation of an inference algorithm for the type system of Figures 3 and 4.[¶] The inference algorithm is fully automatic (it does not require type annotations from the programmer) and may either produce an admissible annotated typing or fail (meaning that cost bounds could not be found). Our analysis is therefore a *whole program* analysis. Inference is conducted in three stages:

a) We first perform Damas-Milner type inference to obtain an unannotated Hindley-Milner version of the type derivation using the syntax-directed rules in Figure 3. The unannotated types form a free algebra and can be determined using standard first-order unification.

b) We then decorate the Hindley-Milner types with fresh annotation variables for the types of thunks, arrows and data constructors and perform a traversal of the type derivation gathering linear constraints among annotations according to the sharing and subtyping conditions.

c) Finally, we feed the linear constraints to a standard linear programming solver[||] with the objective of minimizing the overall expression cost. Any solution gives rise to a valid annotated typing derivation, and hence to a concrete formula bounding evaluation costs in terms of the program's input data sizes.

The implementation allows some trivial syntactic extensions to the term language, namely, multiple constructor branches in match-expressions and omission of the default alternative. Also, as in ML or Haskell, we require that data constructors are associated with a single data type. This ensures that the use of the Cons rule is syntax-directed.

It remains to explain how to decide when to use the structural rules from Figure 4. We use Share to split the context Γ into two Γ_1, Γ_2 when typing sub-expressions (e.g. when typing e_1 and e_2 in $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$); note that this does not lose precision unnecessarily, since the unused types can be assigned zero potential. We consequently delegate the task of finding the best assignment (i.e. one yielding the least cost) to the LP solver. We use Weak depending on the remaining free variables in the sub-expressions. We allow Prepay to be used for the body e_2 of any let-expression $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$. Once again, this does not lose precision because the rule can be used to pay any part of the cost (possibly zero); hence, we allow the LP solver to decide how to use it for each individual thunk in order to achieve an overall optimal solution. Finally, we allow the use of Relax at every node of the derivation and Subtype at the application rule (to enforce compatibility between the function and its argument) and at the Match rule (to obtain a compatible result type). This may generate more constraints and variables for intermediate types than necessary; the resulting increase in size has negligible cost for current LP-solvers (in fact, all our examples were solved by a typical desktop computer in less than one second). Hoffman and Jost have shown that the LP problems that are generated for the eager amortised analysis exhibit regularities that allow lower complexity than general LP solving [19]. We conjecture that this should also be true for our analysis.

4.1 List reversal

Our first recursive example is the classical list reversal using an accumulating parameter:

```
let rev_acc = \xs ys -> match xs with
      Nil () -> ys
    | Cons(x,xs') -> letcons ys' = Cons(x,ys)
                     in rev_acc xs' ys'
```

The analysis fails to find an annotated typing for the above fragment. This is because the recursion is over the first argument of a Curried function and the Abs rule only allows potential in the last argument (since it requires the context to share to itself in order to avoid duplicating potential). Two solutions are possible: either rewrite the function to use a pair of lists instead of using Currying or simply flip the argument order. We choose the latter:

```
let rev_acc' = \ys xs -> match xs with
      Nil () -> ys
    | Cons(x,xs') -> letcons ys' = Cons(x,ys)
                     in rev_acc' ys' xs'
```

The analysis can now yield an informative type. If we abbreviate the type of lists of A as:

$$\begin{array}{l} \mathsf{L}(q_c, q_n, A) \overset{\text{def}}{=} \\ \qquad \mu X.\{\, \mathtt{Cons} : (q_c,\ (\mathsf{T}^0_0(A), \mathsf{T}^0_0(X)))\ \mathtt{|}\ \mathtt{Nil} : (q_n,\ ())\, \} \end{array}$$

[¶] Available at `http://www.dcc.fc.up.pt/~pbv/cgi/aalazy.cgi`.

[||] We use the GLPK library: `http://www.gnu.org/software/glpk`.

$$\dfrac{(\text{Figure 5})}{f{:}\mathsf{T}^1_0(\mathsf{T}^0_0(B)\xrightarrow[0]{0}B)\vdash^{1}_{0}\mathsf{let}\ z=z\ \mathsf{in}\ (\lambda x.\lambda y.\,y)\ z : \mathsf{T}^0_0(B)\xrightarrow[0]{0}B}\ \textsc{Weak}$$

$$\dfrac{\dfrac{\dfrac{}{x{:}\mathsf{T}^0_0(B)\vdash^{0}_{0} x : B}\ \textsc{Var}}{\emptyset\vdash^{0}_{0}\lambda x.\,x : B}\ \textsc{Abs}}{i : \mathsf{T}^0_0(B)\vdash^{0}_{0}\lambda x.\,x : B}\ \textsc{Weak}$$

$$\dfrac{\dfrac{\dfrac{}{f{:}\mathsf{T}^0_0(\mathsf{T}^0_0(B)\xrightarrow[0]{0}B)\vdash^{0}_{0} f : \mathsf{T}^0_0(B)\xrightarrow[0]{0}B}\ \textsc{Var}}{f{:}\mathsf{T}^0_0(\mathsf{T}^0_0(B)\xrightarrow[0]{0}B),\ i{:}\mathsf{T}^0_0(B)\vdash^{0}_{0} f\,i : B}\ \textsc{App}\qquad \dfrac{\dfrac{}{f{:}\mathsf{T}^0_0(\mathsf{T}^0_0(B)\xrightarrow[0]{0}B)\vdash^{0}_{0} f : \mathsf{T}^0_0(B)\xrightarrow[0]{0}B}\ \textsc{Var}}{f{:}\mathsf{T}^0_0(\mathsf{T}^0_0(B)\xrightarrow[0]{0}B),\ v{:}\mathsf{T}^0_0(B)\vdash^{0}_{0} f\,v : B}\ \textsc{App}}{\dfrac{f{:}\mathsf{T}^0_0(\mathsf{T}^0_0(B)\xrightarrow[0]{0}B),\ f{:}\mathsf{T}^0_0(\mathsf{T}^0_0(B)\xrightarrow[0]{0}B),\ i{:}\mathsf{T}^0_0(B)\vdash^{1}_{0}\mathsf{let}\ v=f\,i\ \mathsf{in}\ f\,v : B}{f{:}\mathsf{T}^0_0(\mathsf{T}^0_0(B)\xrightarrow[0]{0}B),\ i{:}\mathsf{T}^0_0(B)\vdash^{1}_{0}\mathsf{let}\ v=f\,i\ \mathsf{in}\ f\,v : B}\ \textsc{Share}}\ \textsc{Let}$$

$$\dfrac{\dfrac{\cdots}{f{:}\mathsf{T}^0_0(\mathsf{T}^0_0(B)\xrightarrow[0]{0}B)\vdash^{2}_{0}\mathsf{let}\ i=\lambda x.\,x\ \mathsf{in}\ \ldots : B}\ \textsc{Let}}{f{:}\mathsf{T}^1_0(\mathsf{T}^0_0(B)\xrightarrow[0]{0}B)\vdash^{3}_{0}\mathsf{let}\ i=\lambda x.\,x\ \mathsf{in}\ \ldots : B}\ \textsc{Prepay}$$

$$\dfrac{\cdots}{\emptyset\vdash^{4}_{0}\mathsf{let}\ f=(\mathsf{let}\ z=z\ \mathsf{in}\ (\lambda x.\lambda y.\,y)\ z)\ \mathsf{in}\ \mathsf{let}\ i=\lambda x.\,x\ \mathsf{in}\ \mathsf{let}\ v=f\,i\ \mathsf{in}\ f\,v : B\,,\qquad \text{where } B=\mathsf{T}^0_0(C)\xrightarrow[0]{0}C}\ \textsc{Let}$$

Figure 6. Type derivation for lazy-evaluation example (2.2).

then we obtain:

$$\texttt{rev\_acc'} : \mathsf{T}^0_0(\mathsf{L}(0,0,A))\xrightarrow[0]{0}\mathsf{T}^0_0(\mathsf{L}(1,0,A))\xrightarrow[0]{0}\mathsf{L}(0,0,A)$$

This annotated type assigns a potential of 1 heap cell to each `Cons` in the recursion argument xs. The first argument ys and the result both have no potential. Thus, the analysis gives a bound of n heap cells for reversing a list of length n, which is, in fact, the exact cost.

4.2 Functional queues

We now consider Okasaki's purely functional queues, implemented as pairs of lists [35]. This data structure allows $O(1)$ amortised access time to both ends of the queue, and is commonly used as an example for deriving amortised bounds. The translation into our language is shown in Figure 7. It consists of three functions: `mkqueue` normalizes a pair of front and back lists by reversing the back list when the front list is empty, so ensuring that the front is empty *iff* the queue as a whole is empty; the `enqueue` function adds an element to the back of the queue; and the `dequeue` function returns a new queue without the front element. We omit the auxiliary definition of `reverse` which uses `rev_acc'` from Section 4.1. Assuming normalized queues, the `enqueue` function has constant worst-case cost. The `dequeue` function may involve reversing a variable-size list, so its worst-case is $O(n)$; however, the amortised cost for both operations is $O(1)$. The types inferred by our analysis are shown in Figure 8. They express amortised bounds that correspond exactly to Okasaki's analysis, which assigns 1 unit of potential for each element in the back list of the queue. More precisely:

- `mkqueue` consumes a fixed cost of 3 heap cells plus 1 cell for each node in the back list; furthermore, the result queue preserves 1 unit of potential for each node in the new back list;
- `enqueue` and `dequeue` have fixed amortised costs (5 & 3 units, respectively), preserving 1 unit of potential in the back list.

4.3 Infinite structures

Our next example concerns the use of lazy evaluation to define infinite lists (i.e. *streams*). Consider two definitions of a function that generates a stream of identical values:

```
let repeat = \x -> letcons ys = Cons(x,ys)
                   in ys
```

```
let mkqueue = \f r -> match f with
        Nil() -> let f' = reverse r
                 in letcons r' = Nil()
                 in letcons q = Pair(f',r')
                 in q
        otherwise letcons q = Pair(f,r) in q

let enqueue = \x q -> match q with
        Pair(f,r) -> letcons r' = Cons(x,r)
                     in mkqueue f r'

let dequeue = \q -> match q with
        Pair(f,b) -> match f with
             Cons(x,f') -> mkqueue f' b
```

Figure 7. Okasaki's purely functional queues.

```
let repeat' = \x -> let xs = repeat' x
                    in letcons ys = Cons(x,xs)
                    in ys
```

The two definitions yield exactly the same infinite list of values. However, the first one is more efficient: `repeat` will generate a cyclic structure occupying a single heap node, while `repeat'` will allocate many (identical) nodes as the result stream is traversed. We can observe these non-functional properties in the types that our analysis infers for the two definitions:

$$\texttt{repeat} : \mathsf{T}^0_0(A)\xrightarrow[0]{1}\mu X.\{\texttt{Cons} : (0,\ (\mathsf{T}^0_0(A),\mathsf{T}^0_0(X)))\ \mid\ \ldots\}$$
$$\texttt{repeat'} : \mathsf{T}^0_0(A)\xrightarrow[0]{2}\mu X.\{\texttt{Cons} : (0,\ (\mathsf{T}^0_0(A),\mathsf{T}^2_0(X)))\ \mid\ \ldots\}$$

First note that, because the results of both functions are infinite structures, they must have zero potential, hence the zero annotation on `Cons`. The type for `repeat` shows that it costs 1 heap cell to generate the first node and that subsequent nodes have no further cost (because the thunk annotations are zero). The type for `repeat'`, however, shows that evaluating each tail thunk of the result list costs 2 cells (plus 2 cells for the first node).

$$\begin{aligned}
\texttt{mkqueue} &: \mathsf{T}^0_0(\mathsf{L}(0,0,A)) \xrightarrow[0]{0} \mathsf{T}^0_0(\mathsf{L}(1,0,A)) \xrightarrow[0]{3} \mathsf{T}^0_0(\mathsf{L}(0,0,A)) \times \mathsf{T}^0_0(\mathsf{L}(1,0,A))\\
\texttt{enqueue} &: \mathsf{T}^0_0(A) \xrightarrow[0]{0} \mathsf{T}^0_0(\mathsf{T}^0_0(\mathsf{L}(0,0,A)) \times \mathsf{T}^0_0(\mathsf{L}(1,0,A))) \xrightarrow[0]{5} \mathsf{T}^0_0(\mathsf{L}(0,0,A)) \times \mathsf{T}^0_0(\mathsf{L}(1,0,A))\\
\texttt{dequeue} &: \mathsf{T}^0_0(\mathsf{T}^0_0(\mathsf{L}(0,0,A)) \times \mathsf{T}^0_0(\mathsf{L}(1,0,A))) \xrightarrow[0]{3} \mathsf{T}^0_0(\mathsf{L}(0,0,A)) \times \mathsf{T}^0_0(\mathsf{L}(1,0,A))
\end{aligned}$$

Figure 8. Analysis of the functional queues example.

$$\texttt{map} : \mathsf{T}^0_0(\mathsf{T}^0_0(A) \xrightarrow[0]{0} B) \xrightarrow[0]{0} \mathsf{T}^0_0(\mu X.\{\texttt{Cons} : (3, (\mathsf{T}^0_0(A), \mathsf{T}^0_0(X)))\,|\,\texttt{Nil} : (1, ())\}) \xrightarrow[0]{0} \mu X.\{\texttt{Cons} : (0, (\mathsf{T}^0_0(B), \mathsf{T}^0_0(X)))\,|\,\ldots\} \quad (4.1)$$
$$\texttt{map} : \mathsf{T}^0_0(\mathsf{T}^0_0(A) \xrightarrow[0]{1} B) \xrightarrow[0]{0} \mathsf{T}^0_0(\mu X.\{\texttt{Cons} : (3, (\mathsf{T}^0_0(A), \mathsf{T}^0_0(X)))\,|\,\texttt{Nil} : (1, ())\}) \xrightarrow[0]{0} \mu X.\{\texttt{Cons} : (0, (\mathsf{T}^1_0(B), \mathsf{T}^0_0(X)))\,|\,\ldots\} \quad (4.2)$$
$$\texttt{map} : \mathsf{T}^0_0(\mathsf{T}^0_0(A) \xrightarrow[0]{0} B) \xrightarrow[0]{0} \mathsf{T}^0_0(\mu X.\{\texttt{Cons} : (0, (\mathsf{T}^0_0(A), \mathsf{T}^0_0(X)))\,|\,\texttt{Nil} : (0, ())\}) \xrightarrow[0]{3} \mu X.\{\texttt{Cons} : (0, (\mathsf{T}^0_0(B), \mathsf{T}^3_0(X)))\,|\,\ldots\} \quad (4.3)$$
$$\texttt{map} : \mathsf{T}^0_0(\mathsf{T}^0_0(A) \xrightarrow[0]{1} B) \xrightarrow[0]{0} \mathsf{T}^0_0(\mu X.\{\texttt{Cons} : (0, (\mathsf{T}^0_0(A), \mathsf{T}^0_0(X)))\,|\,\texttt{Nil} : (0, ())\}) \xrightarrow[0]{3} \mu X.\{\texttt{Cons} : (0, (\mathsf{T}^1_0(B), \mathsf{T}^3_0(X)))\,|\,\ldots\} \quad (4.4)$$

Figure 9. Analyses of `map` for finite (4.1) (4.2) and infinite lists (4.3) (4.4).

4.4 Higher order functions over lists

Consider now the higher-order function `map` that applies a function to every element in a list:

```
let map = \f xs -> match xs with
          Nil () -> letcons nil=Nil() in nil
        | Cons(x,xs') -> let y = f x
                         in let ys' = map f xs'
                         in letcons ys = Cons(y,ys')
                         in ys
```

Figure 9 shows four distinct typings inferred depending on use: (4.1) and (4.2) were inferred for mapping over a *finite* list (which can carry potential) while (4.3) and (4.4) were inferred for mapping over an *infinite* one (which must have zero potential). Thus, the first two typings (4.1) and (4.2) offset costs with potential from the argument list (three heap cells for each `Cons` and one for each `Nil`) while (4.3) and (4.4) defer costs to the tail thunk of the result lists. Note also that (4.1) and (4.3) allow a zero-cost argument function while (4.2) and (4.4) allow a unit-cost argument function; the effect of this change is reflected on the thunk costs for the head of the result lists. Finally, we remark that the analysis chooses these typings automatically according to use.**

5. Soundness

This section establishes the soundness of our analysis with respect to the operational semantics of Section 2. We begin by stating some auxiliary proof lemmas and preliminary definitions, notably formalizing the notion of *potential* from Section 3. We then define the principal invariants of our system, namely, *type consistency* and *type compatibility* relations between a heap configuration of the operational semantics and global types, contexts and balance. We conclude with the soundness result proper (Theorem 1).

5.1 Auxiliary Lemmas

We now present some auxiliary proof lemmas for our type system. The first lemma allows us to replace variables in type derivations. Note that because of the lazy evaluation semantics (and unlike the usual substitution lemma for the λ-calculus), we substitute only variables but not arbitrary expressions.

Lemma 5.1 (Substitution)**.** *If* $\Gamma, x{:}A \overset{p}{\underset{p'}{\vdash}} \widehat{e} : C$ *and* $y \notin \mathrm{dom}(\Gamma) \cup \mathrm{FV}(\widehat{e})$ *then also* $\Gamma, y{:}A \overset{p}{\underset{p'}{\vdash}} \widehat{e}[y/x] : C$.

Proof. By induction on the height of derivation of $\Gamma, x{:}A \overset{p}{\underset{p'}{\vdash}} \widehat{e} : C$, simply replacing any occurrences of x for y. □

The next two lemmas establish inversion properties for constructors and λ-abstractions.

Lemma 5.2 (CONS inversion)**.** *If* $\Gamma \vdash c(\vec{y}) : B$ *then* $B = \mu X.\{\ldots \mid c : (q, \vec{A}) \mid \ldots\}$ *and* $\curlyvee(\Gamma \mid \vec{y}{:}\vec{A}[B/X])$.

Lemma 5.3 (ABS inversion)**.** *If* $\Gamma \vdash \lambda x.e : A \xrightarrow[q']{q} C$ *then there exists* Γ' *such that* $\curlyvee(\Gamma \mid \Gamma')$, $\curlyvee(\Gamma' \mid \Gamma', \Gamma')$, $x \notin \mathrm{dom}(\Gamma')$ *and* $\Gamma', x{:}A \overset{q}{\underset{q'}{\vdash}} e : C$.

Proof Sketch for both lemmas. A typing with conclusion $\Gamma \vdash c(\vec{y}) : B$ must result from axiom CONS followed by (possibly zero) uses of structural rules. Similarly, a typing $\Gamma \vdash \lambda x.e : A \xrightarrow[q']{q} C$ must result from an application of the rule ABS followed by uses of structural rules. The proof follows by induction on the structural rules, considering each rule separately. □

The final auxiliary lemma allows splitting contexts used for typing expressions in *whnf* according to a split of the result type.

Lemma 5.4 (Context Splitting)**.** *If* $\Gamma \overset{0}{\underset{0}{\vdash}} w : A$*, where* w *is an expression in whnf and* $\curlyvee(A \mid A_1, A_2)$*; then there exists* Γ_1, Γ_2 *such that* $\curlyvee(\Gamma \mid \Gamma_1, \Gamma_2)$, $\Gamma_1 \overset{0}{\underset{0}{\vdash}} w : A_1$ *and* $\Gamma_2 \overset{0}{\underset{0}{\vdash}} w : A_2$.

Proof Sketch. The proof follows from an application of Lemma 5.2 (if w is a constructor) or Lemma 5.3 (if w is an abstraction) together with the definition of sharing. □

5.2 Global Types, Contexts and Balance

We now define some auxiliary mappings that will be necessary for formulating the soundness of our type system. The mapping $\mathcal{M}$ from locations to types, written $\{\ell_1 \mapsto A_1, \ldots, \ell_n \mapsto A_n\}$, records the *global type* of a location, which accounts for all potential in all references to that location. The mapping $\mathcal{C}$ from locations to typing contexts, written $\{\ell_1 \mapsto \Gamma_1, \ldots, \ell_n \mapsto \Gamma_n\}$, associates each location with its *global context* that justifies its global type. We extend the projection operation from (local) contexts to global contexts in the natural way:

$$\mathcal{C}\!\upharpoonright_\ell = \{\ell_1 \mapsto \Gamma_1, \ldots, \ell_n \mapsto \Gamma_n\}\!\upharpoonright_\ell \overset{\text{def}}{=} (\Gamma_1, \ldots, \Gamma_n)\!\upharpoonright_\ell$$

We also extend subtyping to global types in the natural way, namely $\mathcal{M} <: \mathcal{M}'$ if and only if $\mathrm{dom}(\mathcal{M}) \subseteq \mathrm{dom}(\mathcal{M}')$ and for all $\ell \in \mathrm{dom}(\mathcal{M})$ we have $\mathcal{M}(\ell) <: \mathcal{M}(\ell')$. This relation will be

** Note, however, that using the same definition with both finite and infinite structures would generate infeasible constraints due to the absence of *"resource parametricity"* (introduced in [29]).

used to assert that the potential assigned to global types is always non-increasing during execution. Furthermore, we introduce an auxiliary *balance* (or *lazy potential*) mapping $\mathcal{B}$ from locations to non-negative rational numbers. This keeps track of the partial costs of thunks that have been paid in advance by applications of the PREPAY rule. Note that these auxiliary mappings are needed only in the soundness proof of the analysis for bookkeeping purposes, but are *not* part of the operational semantics — in particular, they do not incur runtime costs.

5.3 Potential

We define the potential of an augmented expression with respect to a heap and an annotated type. The potential of expressions that are not *whnf*s (i.e. thunks) and λ-abstractions is always zero. For data constructors, the potential is obtained by summing the type annotation with the (recursive) potential contributed by each of the arguments. Note that for cyclic data structures, the potential is only defined if all the type annotations of all nodes encountered along a cycle are zero (the overall potential must therefore also be zero).

Definition 5.5 (Potential)**.** The potential assigned to an augmented expression $\widehat{e}$ of type A under heap $\mathcal{H}$, written $\phi_{\mathcal{H}}(\widehat{e}{:}A)$, is defined in (5.1) within Figure 10.

Equation (5.2) extends the definition to typing contexts in the natural way. Equation (5.3) defines potential for global contexts, but considers only thunks that are not under evaluation. Finally, (5.4) defines a convenient shorthand notation for a similar summation over the balance. The next two lemmas formalize the intuition that sharing splits the potential of a type and that a supertype of a type A has potential that is no greater than A.

Lemma 5.6 (Potential Splitting)**.** *If* $\curlyvee(A \,|\, A_1, \ldots, A_n)$ *then for all* $\widehat{e}$ *such that the potentials are defined, we have* $\phi_{\mathcal{H}}(\widehat{e}{:}A) \geq \sum_i \phi_{\mathcal{H}}(\widehat{e}{:}A_i)$.

This lemma has an important special case when A occurs as one of the types on the right hand side: if $\curlyvee(A \,|\, A, B_1, \ldots, B_n)$ then $\phi_{\mathcal{H}}(\widehat{e}{:}B_i) = 0$ for all i.

Proof Sketch. First note that the results follow immediately if $\widehat{e}$ is not in *whnf* or is a λ-abstraction (because potentials are zero in those cases). The potential is also zero if $\widehat{e}$ is a constructor that is part of a cycle (since otherwise it would be undefined). The remaining case is for a constructor with no cycles, i.e. a directed acyclic graph (DAG). The proof is then by induction on the length of the longest path. □

Lemma 5.7 (Potential Subtype)**.** *If* $A <: B$ *then for all* $\widehat{e}$ *such that the potentials are defined, we have* $\phi_{\mathcal{H}}(\widehat{e}{:}A) \geq \phi_{\mathcal{H}}(\widehat{e}{:}B)$.

Proof. By the definition of subtyping, this is a direct corollary of Lemma 5.6 for the case when $n = 1$. □

5.4 Consistency and Compatibility

We now define the principal invariants for proving the soundness of our analysis, namely, *consistency* and *compatibility* relations between a heap configuration and the global types, contexts and balance. We proceed by first defining type consistency of a single location and then extend it to a whole heap.

Definition 5.8 (Type consistency of locations)**.** We say that location ℓ admits type $\mathsf{T}^q_{q'}(A)$ under context Γ, balance $\mathcal{B}$, heap configuration $(\mathcal{H}, \mathcal{L})$, and write $\Gamma, \mathcal{B}; \mathcal{H}, \mathcal{L} \vdash_{\text{LOC}} \ell : \mathsf{T}^q_{q'}(A)$, if $q \geq q'$ and one of the following cases holds:

(LOC1) $\mathcal{H}(\ell)$ is in *whnf* and $\Gamma \mathrel{\vdash^{0}_{0}} \mathcal{H}(\ell) : A$

(LOC2) $\mathcal{H}(\ell)$ not in *whnf* and $\ell \notin \mathcal{L}$ and $\Gamma \mathrel{\vdash^{q\,+\,\mathcal{B}(\ell)}_{q'}} \mathcal{H}(\ell) : A$

(LOC3) $\mathcal{H}(\ell)$ not in *whnf* and $\ell \in \mathcal{L}$ and $\Gamma = \emptyset$

The three cases in the above definition are mutually exclusive: LOC1 applies when the expression in the heap is already in *whnf*; otherwise LOC2 and LOC3 apply, depending on whether the thunk is or is not under evaluation. For LOC2, the balance $\mathcal{B}(\ell)$ associated with location ℓ is added to the available resources for typing the thunk $\mathcal{H}(\ell)$, effectively reducing its cost by the prepaid amount. Once evaluation has begun (LOC3), or once it has completed (LOC1), the balance is considered spent. However, we never lower or reset the balance, since it is simply ignored in such cases.

Definition 5.9 (Type consistency of heaps)**.** We say that a heap state $(\mathcal{H}, \mathcal{L})$ is consistent with global contexts, global types and balance, and write $\mathcal{C}, \mathcal{B} \vdash_{\text{MEM}} (\mathcal{H}, \mathcal{L}) : \mathcal{M}$, if and only if for all $\ell \in \text{dom}(\mathcal{H})$: $\mathcal{C}(\ell), \mathcal{B}; \mathcal{H}, \mathcal{L} \vdash_{\text{LOC}} \ell : \mathcal{M}(\ell)$ holds.

Definition 5.10 (Global compatibility)**.** We say that a global type $\mathcal{M}$ is *compatible* with context Γ and a global context $\mathcal{C}$, written $\curlyvee(\mathcal{M} \,|\, \Gamma, \mathcal{C})$, if and only if $\curlyvee(\mathcal{M}(\ell) \,|\, \Gamma{\restriction}_\ell, \mathcal{C}{\restriction}_\ell)$ for all $\ell \in \text{dom}(\mathcal{M})$.

Definition 5.9 requires the type consistency of each specific location. Definition 5.10 requires that the global type of each location accounts for the joint potential of all references to it in either the local or global contexts.

5.5 Soundness of the proof system

We can now state the soundness of our analysis as an augmented type preservation result.

Theorem 1 (Soundness)**.** *Let* $t \in \mathbb{Q}^+$ *be fixed, but arbitrary. If the following statements hold*

$$\Gamma \mathrel{\vdash^{p}_{p'}} e : A \tag{1.A}$$

$$\mathcal{C}, \mathcal{B} \vdash_{\text{MEM}} (\mathcal{H}, \mathcal{L}) : \mathcal{M} \tag{1.B}$$

$$\curlyvee(\mathcal{M} \,|\, (\Gamma, \Theta), \mathcal{C}) \tag{1.C}$$

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash e \Downarrow w, \mathcal{H}' \tag{1.D}$$

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

$$m \geq t + p + \phi_{\mathcal{H}}(\Gamma) + \phi_{\mathcal{H}}(\Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{C}) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{B}) \tag{1.E}$$

there exist m', Γ', $\mathcal{C}'$, $\mathcal{B}'$ *and* $\mathcal{M}'$ *such that*

$$\mathcal{M} <: \mathcal{M}' \tag{1.F}$$

$$\Gamma' \mathrel{\vdash^{0}_{0}} w : A \tag{1.G}$$

$$\mathcal{C}', \mathcal{B}' \vdash_{\text{MEM}} (\mathcal{H}', \mathcal{L}) : \mathcal{M}' \tag{1.H}$$

$$\curlyvee(\mathcal{M}' \,|\, (\Gamma', \Theta), \mathcal{C}') \tag{1.I}$$

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \mathrel{\vdash^{m}_{m'}} e \Downarrow w, \mathcal{H}' \tag{1.J}$$

$$m' \geq t + p' + \phi_{\mathcal{H}'}(w{:}A) + \phi_{\mathcal{H}'}(\Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}'}(\mathcal{C}') + \Phi^{\mathcal{L}}_{\mathcal{H}'}(\mathcal{B}') \tag{1.K}$$

Informally, the soundness theorem reads as follows: if an expression e admits a type A (1.A), the heap can be consistently typed (1.B) (1.C) and the evaluation is successful (1.D), then the result *whnf* also admits type A (1.G). Furthermore, the resulting heap can can also be typed (1.H) (1.I) and the static bounds that are obtained from the typing of e give safe resource estimates for evaluation (1.E) (1.J) (1.K). The arbitrary value t is used to carry over excess potential which is not used for the immediate evaluation but will be needed in subsequent ones (i.e. for the argument of an application). Similarly, the context Θ is used to preserve types for variables that are not in the current scope but that are necessary for subsequent evaluations (i.e. the alternatives of the match). Because of space limitations, we present here only a proof sketch; a detailed proof is available at `http://www.dcc.fc.up.pt/~pbv/AALazyExtended.pdf`.

$$\phi_{\mathcal{H}}(\widehat{e}{:}A) \overset{\text{def}}{=} \begin{cases} p + \sum_i \phi_{\mathcal{H}}(\mathcal{H}(\ell_i){:}B_i[A/X]) & \text{if } A = \mu X.\{\cdots \mid c{:}(p, \vec{B}) \mid \cdots\} \text{ and } \widehat{e} = c(\vec{\ell}) \\ \phi_{\mathcal{H}}(\widehat{e}{:}B) & \text{if } A = \mathsf{T}^{q}_{q'}(B) \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

$$\phi_{\mathcal{H}}(\Gamma) \overset{\text{def}}{=} \sum \left\{ \phi_{\mathcal{H}}(\mathcal{H}(x){:}A) \;\middle|\; x{:}A \in \Gamma \right\} \tag{5.2}$$

$$\Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{C}) \overset{\text{def}}{=} \sum \left\{ \phi_{\mathcal{H}}(\mathcal{C}(\ell)) \;\middle|\; \ell \in \text{dom}(\mathcal{H}) \text{ and } \ell \notin \mathcal{L} \text{ and } \mathcal{H}(\ell) \text{ is not a whnf} \right\} \tag{5.3}$$

$$\Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{B}) \overset{\text{def}}{=} \sum \left\{ \quad \mathcal{B}(\ell) \quad \middle|\; \ell \in \text{dom}(\mathcal{H}) \text{ and } \ell \notin \mathcal{L} \text{ and } \mathcal{H}(\ell) \text{ is not a whnf} \right\} \tag{5.4}$$

Figure 10. Potential

Proof Sketch. The proof is by induction on the lengths of the derivations of (1.D) and (1.A) ordered lexicographically, with the derivation of the evaluation taking priority over the typing derivation. We proceed by case analysis of the typing rule used in premise (1.A), considering just some representative cases.

Case VAR*:* The typing premise $\ell{:}\mathsf{T}^{p}_{p'}(A) \;\vdash^{p}_{p'}\; \ell : A$ is an axiom. By inversion of the evaluation premise, we obtain $\mathcal{H}, \mathcal{S}, \mathcal{L} \cup \{\ell\} \vdash \mathcal{H}(\ell) \Downarrow w, \mathcal{H}'$. In order to apply induction to the evaluation of the thunk $\mathcal{H}(\ell)$, we take the typing context from the hypothesis of type consistency for the location ℓ. We apply induction to a typing with the global type $\mathcal{M}(\ell)$ rather than the local type $\mathsf{T}^{p}_{p'}(A)$ in the local context. This gives us a stronger conclusion with a context that we can then split using Lemma 5.6 to justify type consistency for the heap update and the local context answer for the answer. Finally, we require an auxiliary result to ensure that if the update introduces a cycle, the locations on the cycle can be assigned a type with zero potential (a lemma contained in the full proof).

Case LET*:* The typing premise is $\Gamma, \Delta \;\vdash^{1+p}_{p'}\; \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : C$ and evaluation premise gives $\mathcal{H}_0, \mathcal{S}, \mathcal{L} \vdash e_2[\ell/x] \Downarrow w, \mathcal{H}'$ where $\mathcal{H}_0 = \mathcal{H}[\ell \mapsto e_1[\ell/x]]$ is the heap extended with a new location ℓ and thunk. To apply induction to the evaluation of $e_2[\ell/x]$ we re-establish the consistency to the new location ℓ; this is done using Γ from the typing hypothesis together with an idempotent type for self-references to ℓ. Applying induction then yields all required conclusions.

Case MATCH*:* The typing premise is:

$$\Gamma, \Delta \;\vdash^{p}_{p''}\; \mathsf{match}\ e_0\ \mathsf{with}\ c(\vec{x})\text{->}e_1\ \mathsf{otherwise}\ e_2 : C$$

By inversion of the type rule, we get a typing $\Gamma \;\vdash^{p}_{p'}\; e_0 : B$ for e_0, where $B = \mu X.\{\cdots \mid c : (q, \vec{A}) \mid \cdots\}$ is some data type with a constructor c. We apply induction to the evaluation of e_0 and then do a case analysis on the evaluation rule used (i.e. MATCH$_\Downarrow$ or FAIL$_\Downarrow$). We then apply induction to either $e_1[\vec{\ell}/\vec{x}]$ or e_2 and obtain the proof obligation. To establish the premise (1.E) on m for the MATCH$_\Downarrow$ case, we use definition of potential: $\phi_{\mathcal{H}}(c(\vec{\ell}){:}B) = q + \sum_i \phi_{\mathcal{H}}(\ell_i{:}A_i[B/X])$ — i.e. the potential of the constructor is the sum of the type annotation q plus the potential of its context. □

6. Related Work

As described above, we build heavily on Launchbury's natural semantics for lazy evaluation [30], as subsequently adapted by Sestoft, and exploit ideas that were developed by de la Encina and Peña-Marí [14, 15]. There is a significant body of other work on the semantics of call-by-need evaluation. Pre-dating Launchbury's work, Josephs [25] gave a *denotational* semantics of lazy evaluation, using a continuation-based semantics to model sharing, and including an explicit store. However, this approach doesn't fit well with standard proof techniques. Maraist et al. [31] subsequently defined both natural and reduction semantics for the call-by-need lambda calculus, so enabling equational reasoning, and a similar approach was independently described by Ariola and Felleisen [4].

Bakewell and Runciman [6] have previously defined an operational semantics for Core Haskell that gives time and space execution costs in terms of Sestoft's semantics for his Mark 1 abstract machine. The work has subsequently been extended to give a model that can be used to determine space leaks by comparing the space usage for two evaluators using a bisimulation approach [5]. Gustavsson and Sands [17] have similarly defined a space-improvement relation that guarantees that some optimisation can never lead to asymptotically worse space behaviour for call-by-need programs and Moran and Sands [33] have defined an improvement relation for call-by-need programs that can be used to determine whether one terminating program improves another in all possible contexts.

Finally, like de la Encina and Peña-Marí, Mountjoy [34] derived an operational semantics for the Spineless Tagless G-Machine from the natural semantics of Launchbury and Sestoft, including polyapplicative λ-expressions. The main differences between these approaches are that de la Encina and Peña-Marí correct some mistakes in Mountjoy's presentation, that they provide correctness proofs, that their semantics correctly deals with partial applications in the Spineless Tagless G-Machine, that they deal with partial applications as normal forms, and that they consider two distinct implementation variants, based on push/enter versus apply/eval. Our own work differs from this body of earlier work in that we not only provide an operational semantics to *model* lazy evaluation, but also provide a corresponding cost semantics from which we derive a static analysis to automatically determine upper bounds on the memory requirements of lazily evaluated programs.

Resource analysis based on profiling and manual code inspection has long formed the state-of-the-art and still is current practice in many cases. Indeed, for non-strict functional languages, such as Haskell, ad-hoc techniques, manual analysis or symbolic profiling are the only currently viable approaches: as we have seen, the dynamic *demand-driven* nature of lazy functional programming creates particular problems for resource analysis, whether manual or automatic. There has therefore been very little work on static resource analysis for lazy functional programs, and, to our knowledge, no *previous automatic analysis has ever been produced.* The most significant previous work in the area is that by Sands [37, 38], whose PhD thesis proposed a cost calculus for reasoning about sufficient and necessary execution time for lazily evaluated higher-order programs, using an approach based on *evaluation contexts* [39, 45] to capture information about evaluation degree and appropriate *projections* [47] to project this information to the required approach. Wadler [45] had earlier proposed a similar approach to that taken by Sands, but using *strictness analysis* combined with appropriate projections, rather than the *neededness analysis* that Sands uses. A primary disadvantage of such approaches lies in the complexity of the domain structure and associated projections that must be used when analysing even simple data structures such as lists. In contrast, our approach easily extends to arbitrarily complex data structures. A secondary disadvantage is that, unlike the self-contained analysis we have described, projection-based approaches rely on the existence of a complex and powerful external *neededness analysis* to determine evaluation contexts for expressions. These are serious practical disadvantages: in fact, to date, we are not aware of any fully automatic static analysis that has been produced using these techniques.

A number of authors have proposed analysis approaches based on transforming lazy programs to eager ones (e.g. Bjerner and Holmström [7], Fradet and Métayer [16]). The resulting programs may then be analysed using (simpler) techniques for eagerly evaluated programs, such as the automatic amortised analysis we have previously developed [19, 28, 29]. Unlike our work, these approaches are generally restricted to first-order programs, and suffer from the problems that they are, in general, not cost-preserving, that they lead to potentially exponential code explosion, and that, because they alter the program, they are not suitable for use with standard compilers for lazy functional languages.

Several authors have proposed *symbolic profiling* approaches, where programs are annotated with additional cost parameters. For example, Wadler [46] uses *monads* to capture execution costs through a tick-counting function; Albert et al. [1] adds additional cost parameters to each function, using logic variables to capture sharing information and so avoid cost duplication; and Hope [22] describes how to derive an instrumented function for determining time and space usage, including a simple deallocation model, for a strict functional language and outlines how this could be extended to lazy evaluation. Danielsson [12] takes this work a stage further, describing a library that can be used to annotate (lazy) functions with the time that is needed to compute their result. An annotated monad is then used to combine these time complexity annotations. This can be used to verify (but *not infer*) the time complexity of (lazy) functional data structures and algorithms against Launchbury's semantics, using a dependent type approach. Provided the cost model is sufficiently accurate, symbolic profiling approaches can give "exact" costs for specific program inputs. They are also easy to implement. However, unlike the work described here, the cost information is input-dependent, cannot give a guaranteed worst-case except in trivial cases, and transforms the program in a way that may not be cost-preserving for all metrics. Unlike our analysis, such approaches therefore cannot produce upper bounds on resource usage for all possible program inputs.

The amortised analysis approach has been previously studied by a number of authors, but has never previously been used to automatically determine the costs of lazy evaluation. Tarjan [42] first described amortised analysis, but as a manual technique. Okasaki [35] subsequently described how Tarjan's approach could be applied to (lazy) data structures, but again as a manual technique. While there has subsequently been significant interest in the use of amortised analysis for automatic resource usage analysis, using an advanced per-reference potential, none of this newer work, however, considers lazy evaluation. Hofmann and Jost [19] were the first to develop an *automatic* amortised analysis for heap consumption, exploiting a difference metric similar to that used by Crary and Weirich [11] (the latter, however, only *check* bounds, and therefore do not perform an automatic static analysis of the kind we require); Hofmann et al. have extended their method to cover a comprehensive subset of Java, including imperative updates, inheritance and type casts [20, 21]; Shkaravska et al. [41] subsequently considered heap consumption inference for first-order polymorphic lists; and Campbell [9] has developed the ideas of depth-based and temporary credit uses to give better results for stack usage. Hoffmann et al. [18] achieved another breakthrough by extending the technique to infer multivariate polynomial cost functions, still only requiring efficient LP solving. Finally, several authors have recently studied analyses for heap usage in eager languages, without considering lazy evaluation. For example, Albert et al. [2] present a fully automatic, live heap-space analysis for an object-oriented bytecode language with a scoped-memory manager, and have subsequently extended this to consider garbage collection [3], but, unlike our system, data-dependencies cannot be expressed. Braberman et al. [8] infer polynomial bounds on the live heap usage for a Java-like language with automatic memory management, but do not cover general recursive methods. Finally, Chin et al. [10] present a linearly-bounded heap and stack analysis for a low-level (assembler) language with explicit (de)-allocation, but do not cover lazy evaluation or high-level functional programming constructs.

7. Conclusions and Further Work

This paper has introduced a new automatic type-based analysis for accurately determining bounds on the execution costs of lazy (higher-order) functional programs. The analysis uses the new idea of *lazy potential* as part of an amortised analysis technique that is capable of directly analysing lazy programs without requiring defunctionalisation or other non-cost-preserving program transformations. Our analysis deals with (potentially infinite) recursive data structures, nested data structures, and cyclic data structures. It is defined for arbitrary data types (including e.g. trees). We have proved the soundness of this analysis against an operational semantics derived from Launchbury's natural semantics of graph reduction, and analysed some non-trivial examples of lazy evaluation using a prototype implementation of the analysis.

A number of extensions to this work would repay further investigation. Firstly, to reduce complexity, our system is restricted to monomorphic definitions. It should be straightforward, albeit laborious, to adapt our previous work on polymorphism [29] to also cover the lazy setting, including "resource parametricity", which allows function applications to have different costs depending on context. Secondly, we have only considered linear cost functions. Although it would increase complexity, Hoffmann et al. [18]'s approach to polynomial cost functions, which infers asymptotically tight bounds for many practical examples, should also be applicable here. Thirdly, while we have previously constructed [28, 29] analyses that are capable of dealing with arbitrary countable resources for strict languages, for simplicity, in this paper we have restricted our attention to heap allocations. Analysing time and stack usage should follow a similar structure to that presented here, but requires a richer operational semantics than that given by Launchbury. Finally, it would be interesting to extend this work to a full production abstract machine such as the Spineless Tagless G-Machine [24]. This would allow us to confirm our results against real functional programs written in non-strict languages such as Haskell.

Acknowledgements

This work is supported by EU grants SCIEnce (RII3-CT-026133), ADVANCE (IST-248828) and ParaPhrase (IST-288570), and EPSRC grant HPC-GAP (EP/G 055181). Hugo Simões would like to thank "Fundação para a Ciência e Tecnologia, Portugal" for Ph.D. grant SFRH/BD/17096/2004.

References

[1] E. Albert, J. Silva, and G. Vidal. Time Equations for Lazy Functional (Logic) Languages. In *Proc. AGP-2003: 2003 Joint Conf. on Declarative Prog., Reggio Calabria, Italy, Sept. 3-5, 2003*, pages 13–24, 2003.

[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. ISBN 978-1-60558-347-1.

[3] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Parametric Inference of Memory Requirements for Garbage Collected Languages. In *Proc. 2010 International Symposium on Memory Management*, ISMM '10, pages 121–130, New York, NY, USA, 2010. ACM.

[4] Z. M. Ariola and M. Felleisen. The Call-by-Need Lambda Calculus. *J. Funct. Program.*, 7:265–301, May 1997.

[5] A. Bakewell and C. Runciman. A Model for Comparing the Space Usage of Lazy Evaluators. In *Proc. PPDP 2000: Intl. Conf. on Principles and Practice of Declarative Prog., Quebec, Canada*, pages 151–162, 2000.

[6] A. Bakewell and C. Runciman. A Space Semantics for Core Haskell. *Electr. Notes Theor. Comput. Sci.*, 41(1), 2000.

[7] B. Bjerner and S. Holmström. A Compositional Approach to Time Analysis of First Order Lazy Functional Programs. In *Proc. FPCA '89: Conf. on Functional Prog. Langs. and Comp. Arch.*, pages 157–165, 1989.

[8] V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *Proc. ISMM 2008: Intl. Symp. on Memory Management*, pages 141–150, New York, NY, USA, June 2008.

[9] B. Campbell. Amortised Memory Analysis Using the Depth of Data Structures. In G. Castagna, editor, *Proc. ESOP 2009: 18th European Symposium on Programming, York, UK*, pages 190–204. Springer LNCS 5502, 2009.

[10] 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. ISBN 978-1-60558-134-7.

[11] K. Crary and S. Weirich. Resource Bound Certification. In *Proc. POPL 2000: ACM Symp. on Principles of Prog. Langs.*, pages 184–198, Jan. 2000.

[12] N. A. Danielsson. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Proc. POPL 2008: Symp. on Principles of Prog. Langs., San Francisco, USA, January 7-12, 2008*, pages 133–144. ACM, 2008.

[13] A. de la Encina and R. Peña-Marí. Proving the Correctness of the STG Machine. In *Proc. IFL '01: Impl. of Functional Langs., Stockholm, Sweden, Sept. 24-26, 2001*, pages 88–104. Springer LNCS 2312, 2002.

[14] A. de la Encina and R. Peña-Marí. Formally Deriving an STG Machine. In *Proc. 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*, pages 102–112. ACM, 2003.

[15] A. de la Encina and R. Peña-Marí. From Natural Semantics to C: a Formal Derivation of two STG Machines. *J. Funct. Program.*, 19(1): 47–94, 2009.

[16] P. Fradet and D. L. Métayer. Compilation of functional languages by program transformation. *ACM Transactions on Programming Languages and Systems*, 13(1):21–51, January 1991.

[17] J. Gustavsson and D. Sands. A Foundation for Space-Safe Transformations of Call-by-Need Programs. *Electronic Notes on Theoretical Computer Science*, 26, 1999.

[18] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. In *38th Symp. on Principles of Prog. Langs. (POPL'11)*, pages 357–370, 2011.

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

[20] M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis (for an Object-Oriented Language). In *Proc. ESOP '06: European Symposium on Prog.*, pages 22–37, Mar. 2006.

[21] M. Hofmann and D. Rodriguez. Efficient type-checking for amortised heap-space analysis. In *Proc. CSL '09: 18th EACSL Annual Conf. on Computer Science Logic*, pages 317–331, 2009.

[22] C. Hope. *A Functional Semantics for Space and Time*. PhD thesis, 2008. University of Nottingham.

[23] R. Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, 1989.

[24] S. L. P. Jones. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine. *J. Funct. Program.*, 2 (2):127–202, 1992.

[25] M. B. Josephs. The semantics of lazy functional languages. *Theor. Comput. Sci.*, 68(1):105–111, 1989.

[26] S. Jost. Static prediction of dynamic space usage of linear functional programs, 2002. Diploma thesis. Darmstadt University of Technology.

[27] S. Jost. *Automated Amortised Analysis*. PhD thesis, LMU Munich, Sept. 2010.

[28] S. Jost, H.-W. Loidl, K. Hammond, N. Scaife, and M. Hofmann. "Carbon Credits" for Resource-Bounded Computations Using Amortised Analysis. In *Proc. FM 2009: Intl. Conf. on Formal Methods*, pages 354–369. Springer LNCS 5850, 2009.

[29] S. Jost, H.-W. Loidl, K. Hammond, and M. Hofmann. Static determination of quantitative resource usage for higher-order programs. In *Proc. POPL 2010: ACM Symp. on Principles of Prog. Langs., Madrid, Spain*, pages 223–236, Jan. 2010.

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

[31] J. Maraist, M. Odersky, and P. Wadler. The Call-by-Need Lambda Calculus. *J. Funct. Program.*, 8:275–317, May 1998.

[32] R. Matthes. *Extensions of System F by Iteration and Primitive Recursion on Monotone Induction Types*. PhD thesis, LMU Munich, 1998.

[33] A. Moran and D. Sands. Improvement in a Lazy Context: An Operational Theory for Call-by-Need. In *POPL*, pages 43–56, 1999.

[34] J. Mountjoy. The Spineless Tagless G-machine, naturally. In *Proc. ICFP '98: Intl. Conf. on Functional Prog.*, pages 163–173, 1998.

[35] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[36] S. Peyton Jones (ed.), L. Augustsson, B. Boutel, F. Burton, J. Fasel, A. Gordon, K. Hammond, R. Hughes, P. Hudak, T. Johnsson, M. Jones, J. Peterson, A. Reid, and P. Wadler. Report on the Non-Strict Functional Language, Haskell (Haskell98). Technical report, Yale University, 1999.

[37] D. Sands. Complexity Analysis for a Lazy Higher-Order Language. In *Proc. ESOP '90: European Symposium on Programming, Copenhagen, Denmark*, Springer LNCS 432, pages 361–376, 1990.

[38] D. Sands. *Calculi for Time Analysis of Functional Programs*. PhD thesis, Imperial College, University of London, September 1990.

[39] D. Sands. Computing with Contexts: A Simple Approach. In *Proc. HOOTS II: Higher-Order Operational Techniques in Semantics*, Electr. Notes in Theoretical Comp. Sci. 1998.

[40] P. Sestoft. Deriving a Lazy Abstract Machine. *J. Functional Programming*, 7(3):231–264, 1997.

[41] O. Shkaravska, R. van Kesteren, and M. van Eekelen. Polynomial Size Analysis of First-Order Functions. In *Proc. TLCA 2007: Typed Lambda Calculi and Applications (TLCA 2007)*, pages 351–365, Paris, France, June 26–28, June 2007. Springer LNCS 4583.

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

[43] P. B. Vasconcelos. *Space Cost Analysis Using Sized Types*. PhD thesis, University of St Andrews, 2008.

[44] P. B. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *Proc. IFL '03: Impl. of Functional Languages*, pages 86–101, Edinburgh, UK, 2004. Springer LNCS 3145.

[45] P. Wadler. Strictness Analysis aids Time Analysis. In *Proc. POPL '88: ACM Symp. on Princ. of Prog. Langs.*, pages 119–132, 1988.

[46] P. Wadler. The Essence of Functional Programming. In *Proc. POPL '92: ACM Symp. on Principles of Prog. Langs.*, pages 1–14, Jan. 1992.

[47] P. Wadler and J. Hughes. Projections for Strictness Analysis. In *Proc. FPCA'87: Intl. Conf. on Functional Prog. Langs. and Comp. Arch.*, Springer LNCS 274, pages 385–407, Sept. 1987.