Type-based Allocation Analysis for Co-Recursion in Lazy Functional Languages

Pedro Vasconcelos[1], Steffen Jost[2], Mário Florido[1], and Kevin Hammond[3]

[1] LIACC, Universidade do Porto, Porto, Portugal; {pbv,amf}@dcc.fc.up.pt
[2] Ludwig Maximillians Universität, Munich, Germany; jost@tcs.ifi.lmu.de
[3] University of St Andrews, St Andrews, UK; kevin@kevinhammond.net

Abstract. This paper presents a novel type-and-effect analysis for predicting upper-bounds on memory allocation costs for co-recursive definitions in a simple lazily-evaluated functional language. We show the soundness of this system against an instrumented variant of Launchbury's semantics for lazy evaluation which serves as a formal cost model. Our soundness proof requires an intermediate semantics employing indirections. Our proof of correspondence between these semantics that we provide is thus a crucial part of this work.
The analysis has been implemented as an automatic inference system. We demonstrate its effectiveness using several example programs that previously could not be automatically analysed.

1 Introduction

Co-recursion can be treated as a construction principle for infinite data structures: whereas recursion progressively deconstructs (finite) data structures, co-recursion progressively constructs (possibly infinite) data structures through synthesis from some base case [?]. In lazy functional programming, co-recursion allows concise and elegant definitions by separating data generation from control. For example, an infinite sequence of Fibonacci numbers, `fibs`, can be defined in Haskell by zipping a list with its own tail [?]:

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

There are two co-recursive base cases (`0` and `1`). The `zipWith` operation then builds the remainder of `fibs` constructively using both these base cases. Thanks to lazy evaluation, the above definition is efficient: each successive Fibonacci number is produced in constant cost. Furthermore, the flow of demand will ensure that each number is evaluated once only when it is needed. However, reasoning about execution costs requires a detailed understanding of the *operational* properties of lazy evaluation, particularly how intermediate results are shared. Moreover, apparently innocuous changes may have a significant impact on execution costs. As a simple example, consider two definitions of the function `cycle` that produces an infinite list by repeated concatenation:

```
cycle  xs = xs' where xs' = xs++xs'
cycle' xs = xs ++ cycle' xs
```

Although the two definitions are denotationally equivalent, the evaluation of `cycle'` will allocate space that is proportional to the number of elements that are demanded from the result, whereas `cycle` will generate a circular list and thus only use constant space. Difficulties in reasoning about space usage are often mentioned as a hindrance to the practical use of lazily evaluated languages, such as Haskell, especially in domains where predictability is a primary concern.

This paper presents a new static analysis for obtaining *a-priori* bounds on the dynamic costs of *co-recursive definitions* for a foundational subset of Haskell. The analysis is formulated as a proof system for inferring annotated types that express upper bounds on the costs of program fragments. For concreteness, we chose to bound the number of heap allocations performed by a standard operational semantics for lazy evaluation. Note that the semantics and our analysis do not model *deallocation* — hence our cost model does not account for residency of an implementation using e.g. garbage collection. Nonetheless, measuring allocations has the foundational benefit of being directly derivable from a standard semantics. Furthermore, the number of allocations has been shown to be a good predictor in practice for the effects of optimizations in real implementations of lazy languages [**?**].

The work presented here complements our previous analysis for lazy functional programs [**?**]. We have previously shown that amortisation allows cost bounds to be determined for recursive definitions over finite data, but also that it does not contribute to the analysis of co-recursion over infinite data. For clarity of presentation, we thefore omit amortisation here. Any automated analysis that is aimed at practical use could obviously combine both methods for improved precision. We do not foresee any problems in doing this (in fact, our implementation includes amortisation), but the technical complexity of the presentation and proofs is likely to increase substantially.

2 Language and cost semantics

We consider the λ-calculus extended with local bindings, data constructors and pattern matching:

$$e \;::=\; x \mid \lambda x.\, e \mid e\ x \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid c(\overline{x}) \mid \mathsf{match}\ e_0\ \mathsf{with}\ \{c_i(\overline{x_i})\texttt{->}e_i\}_{i=1}^{n}$$

Our semantics is built on Sestoft's revision [**?**] of Launchbury's natural semantics for lazy evaluation [**?**], which is one of the earliest and most widely-used operational accounts of lazy evaluation for the λ-calculus. As in Launchbury's semantics, we restrict arguments of applications to simple variables; nested applications must translated into nested let-bindings.[4] *Let*-expressions bind variables to possibly (co)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,

[4] 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.

$$\frac{}{\mathcal{H},\mathcal{S},\mathcal{L} \vdash^{m}_{m} w \Downarrow w, \mathcal{H}} \tag{$\textsc{Whnf}_\Downarrow$}$$

$$\frac{\ell \notin \mathcal{L} \qquad \mathcal{H}[\ell \mapsto e],\mathcal{S},\mathcal{L} \cup \{\ell\} \vdash^{m}_{m'} e \Downarrow w, \mathcal{H}'[\ell \mapsto e]}{\mathcal{H}[\ell \mapsto e],\mathcal{S},\mathcal{L} \vdash^{m}_{m'} \ell \Downarrow w, \mathcal{H}'[\ell \mapsto w]} \tag{$\textsc{Var}_\Downarrow$}$$

$$\frac{\ell \text{ is fresh} \qquad \mathcal{H}[\ell \mapsto e_1[\ell/x]],\mathcal{S},\mathcal{L} \vdash^{m}_{m'} e_2[\ell/x] \Downarrow w, \mathcal{H}'}{\mathcal{H},\mathcal{S},\mathcal{L} \vdash^{m+1}_{m'} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow w, \mathcal{H}'} \tag{$\textsc{Let}_\Downarrow$}$$

$$\frac{\mathcal{H},\mathcal{S},\mathcal{L} \vdash^{m}_{m'} e \Downarrow \lambda x.\, e', \mathcal{H}' \qquad \mathcal{H}',\mathcal{S},\mathcal{L} \vdash^{m'}_{m''} e'[\ell/x] \Downarrow w, \mathcal{H}''}{\mathcal{H},\mathcal{S},\mathcal{L} \vdash^{m}_{m''} e\ \ell \Downarrow w, \mathcal{H}''} \tag{$\textsc{App}_\Downarrow$}$$

$$\frac{\mathcal{H},\mathcal{S} \cup \left(\bigcup_{i=1}^{n} \{\overline{x}_i\} \cup \mathrm{BV}(e_i)\right),\mathcal{L} \vdash^{m}_{m'} e_0 \Downarrow c_k(\overline{\ell}), \mathcal{H}' \qquad \mathcal{H}',\mathcal{S},\mathcal{L} \vdash^{m'}_{m''} e_k[\overline{\ell}/\overline{x}_k] \Downarrow w, \mathcal{H}''}{\mathcal{H},\mathcal{S},\mathcal{L} \vdash^{m}_{m''} \mathsf{match}\ e_0\ \mathsf{with}\ \{c_i(\overline{x}_i)\texttt{->}e_i\}_{i=1}^{n} \Downarrow w, \mathcal{H}''} \tag{$\textsc{Match}_\Downarrow$}$$

Fig. 1. Instrumented version of Launchbury's operational semantics

we consider only single-variable let-bindings: multiple let-bindings can be encoded, if needed, using pairs and projections. Unlike [?], we do not require a distinguished let-construct for introducing constructors here.

Figure 1 defines an instrumented version of Launchbury's semantics, using a simple cost counting mechanism, against which we prove the soundness of our cost analysis. Our semantics is given as a relation $\mathcal{H},\mathcal{S},\mathcal{L} \vdash^{m}_{m'} e \Downarrow w, \mathcal{H}'$, where e is an expression; the *heap* $\mathcal{H}$ is a finite mapping from variables to possibly-unevaluated expressions (*thunks*):

$$\mathcal{H} ::= \emptyset \;\mid\; \mathcal{H}[x \mapsto e]$$

Some notation conventions: we will write $\mathrm{dom}(\mathcal{H})$ for the set of variables occurring in the left-hand side of all mappings in $\mathcal{H}$. We also assume that variables are assigned at most once, i.e. the notation $\mathcal{H}[x \mapsto e]$ requires $x \notin \mathrm{dom}(\mathcal{H})$, and we will use heaps as partial functions, i.e. use $\mathcal{H}(x)$ for the (possibly-undefined) expression associated with x in $\mathcal{H}$. The set $\mathcal{S}$ contains bound variables that are used to ensure the freshness condition in the $\textsc{Let}_\Downarrow$ rule; and $\mathcal{L}$ is a set of variables used to record thunks that are under evaluation, thereby preventing cyclic evaluation (similar to the well-known "black-hole" technique used in [?]). The result of evaluation is an expression w in weak head normal form *(whnf)* and a final heap $\mathcal{H}'$. Note that we use lowercase letters $x, y, \ldots$ for bound variables in the original expression and $\ell, \ell', \ldots$ for "fresh" variables (designated *locations*) introduced by the evaluation of let-expressions. The parameters m, m' are non-negative integers representing the number of available heap locations before and after evaluation, respectively. The choice of instrumenting the semantics with before-and-after resources, as opposed to net costs, is conceptually simpler because it does rely on an a-priori assumption of cost additivity. Furthermore, it also makes it easier to adapt to cost models that allow e.g. for deallocation in the future.

The purpose of the analysis that will be developed in Section 3 is to obtain static bounds on m and m' that will allow the execution to proceed. For readability, we may omit the resource information from judgements when they are not otherwise mentioned, writing simply $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash e \Downarrow w, \mathcal{H}'$ instead of $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e \Downarrow w, \mathcal{H}'$.

Under a lazy evaluation model, expressions are evaluated *only* when they are *demanded* (that is when their value is needed in order to progress evaluation). In our operational semantics, this happens: i) when we need the value of a variable in $\textsc{Var}_{\Downarrow}$ (which is looked up from the environment); ii) when we need the value of a function (a λ-expression) in $\textsc{App}_{\Downarrow}$; or iii) when we need the value of the constructor argument in a `match`-expression. $\textsc{Let}_{\Downarrow}$ is the only rule that augments the heap with a new expression bound to a "fresh" location. Accordingly, it is the only rule that requires a positive heap cost in the annotation above the turnstile; all other rules simply "thread" costs from sub-expressions to the outermost one. For simplicity, but without loss of generality, we choose to use a uniform cost model where each freshly allocated location is counted as a single cost unit. More complex cost model, e.g. for determining the usage of other resources such as execution time, or stack usage (as in [?]), could be easily substituted, if required.

The $\textsc{Whnf}_{\Downarrow}$ rule deals with weak-head normal forms (λ-expressions and constructors) that require no further evaluation, and hence it incurs no cost.

The $\textsc{Var}_{\Downarrow}$ and $\textsc{App}_{\Downarrow}$ rules are identical to their equivalents in Launchbury's semantics, except for passing on m,m',etc. Note that the $\textsc{Var}_{\Downarrow}$ rule is restricted to locations that are not marked as being under evaluation, $\ell \notin \mathcal{L}$ (so enforcing "black-holing" that explicitly excludes some non-terminating evaluations).

The $\textsc{Match}_{\Downarrow}$ rule deals with pattern matching against a constructor. The variables bound in the matching pattern are replaced in the corresponding branch expression e_k by the locations within the heap (also just variables, but we use the meta-variable ℓ to range over variables within the domain of the heap), which is then evaluated. Regardless of the actual branch taken, all possibly bound variables are added to $\mathcal{S}$; this is done solely to ensure the freshness condition in subsequent applications of the $\textsc{Let}_{\Downarrow}$ rule.

For the sake of completeness, we state the auxiliary definition that formalises the notion of variable freshness. This is due to de La Encina and Peña-Marí [?].

Definition 1 (Freshness). *A variable x is* fresh *in judgement $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash e \Downarrow w, \mathcal{H}'$ if x does not occur in either dom($\mathcal{H}$), $\mathcal{L}$ or $\mathcal{S}$ nor does it occur bound in either e or the range of the heap $\mathcal{H}$.*

3 Type and Effect Analysis

The syntax of annotated types is as follows:

$$A \quad ::= \quad X \mid A_1 \xrightarrow{q} A_2 \mid \mathsf{T}^p(A) \mid \mu X.\{c_i : \overline{A}_i\}_{i=1}^{n}$$

We use meta-variables A, B, C for types, X, Y for type variables and p, q for *cost annotations* (i.e. non-negative rational numbers). Function types $A_1 \xrightarrow{q} A_2$

```
let one  = 1 in
in letcons ones = Cons(one,ones)
in let map = \f xs -> match xs with   Nil () -> letcons r = Nil() in r
                      | Cons (x,xs') -> let y = f x
                                        in let ys = map f xs'
                                        in letcons r = Cons(y,ys) in r
in let f  = (\x -> let two=2 in two * x)
in map f ones
```

Fig. 2. Map over an infinite cyclic list

are annotated with a cost q of evaluating the function; thunk types $\mathsf{T}^p(A)$ are annotated with a cost p of evaluating the thunk to *whnf*.

Both recursive and non-recursive algebraic data types are encoded as μ-types $\mu X.\{c_i : \overline{A_i}\}$ where the c_i are constructors, $\overline{A_i}$ is a sequence of argument types and and X is a recursively-bound type variable. For example, the type of lists with elements of type A can be encoded as $\mu X.\{\mathsf{Nil} : () \mid \mathsf{Cons} : (A, X)\}$. Note that we do not distinguish co-recursive data types syntactically, hence finite and infinite lists have the same type.

3.1 Worked Example

Before introducing the formal type rules we start by providing some intuition for the type annotations through a simple example. This is also available through the web version of our analysis at `http://kashmir.dcc.fc.up.pt/cgi/lazy.cgi`.

The example in Fig. 2 multiplies by two every integer in an infinite list, using the canonical `map` function. To this end we augment expressions with primitive integer constants and associated arithmetic operations and a primitive type for integers. Our prototype analysis infers the following annotated types:

$$\begin{aligned} f &: \mathsf{T}^0(\mathsf{Int}) \xrightarrow{1} \mathsf{Int} \\ \mathit{ones} &: \mu X.\{\mathsf{Nil} : () \mid \mathsf{Cons} : (\mathsf{T}^0(\mathsf{Int}), \mathsf{T}^0(X))\} \\ \mathit{map}\ \ f\ \ \mathit{ones} &: \mu X.\{\mathsf{Nil} : () \mid \mathsf{Cons} : (\mathsf{T}^1(\mathsf{Int}), \mathsf{T}^3(X))\} \end{aligned}$$

From these types we observe the following:

1. the type for function f has a cost of 1 on the arrow; this is because each evaluation of f allocates one integer (`let two=2 in`...);
2. the type of *ones* has 0 costs assigned to the head and tail thunks; this implies that forcing elements from this infinite list does not incur more allocations (because *ones* is a cyclic list);
3. the type of the result expression *map f ones* has costs of 1 and 3 units assigned to the head and tail thunks, respectively; this means that evaluating each tail of of the result list costs 3 allocations (for the *map*) while evaluating each head costs 1 allocation (for the argument function f);

4. from the type of *map f ones* we can also read a closed formula of $3n + 1$ for the cost of evaluating the n-th element of the result list (n times the evaluation of the tail thunks plus one evaluation of a head thunk).

Note, however, that if we transform the program by floating the *let*-binding for *two* outwards (i.e. `let two=2 in let f = \x -> two * x`) we infer annotated types that reflect this optimization.

$$f : \mathsf{T}^0(\mathsf{Int}) \xrightarrow{0} \mathsf{Int}$$
$$map \;\; f \;\; ones : \mu X.\{\mathsf{Nil} : () \mid \mathsf{Cons} : (\mathsf{T}^0(\mathsf{Int}), \mathsf{T}^3(X))\}$$

After let-floating, f does not incur any allocation (a single allocation is done for all evaluations). Hence, the type for the result list has now only positive costs for the each successive tail. Note that the cost of *3 per successive list node* is accurate because under lazy evaluation, applying *map* to a cyclic list produces an infinite acyclic list.

3.2 Formal Description of Type Rules

Our analysis is presented in Figures 3 and 4 as a proof system that derives annotated typing judgments for expressions. The rules use two auxiliary relations on types (subtyping and lowering thunk costs) defined in Figures 5 and 6.

An annotated type judgment has the form $\Gamma \;\frac{\;p\;}{p'}\!\!\vdash e : A$ where Γ is a context assigning types to variables,[5] e is an expression, A is an annotated type and p, p' are non-negative numbers approximating the available resources before and after evaluation of e, respectively; these annotations are used for "threading" resources through sub-expressions in rules Let and Match. As with the operational semantics, we omit the annotations on the turnstile whenever they are not further referenced.

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 $x : \mathsf{T}^q(A)$. Dually, Let introduce an assumption of a thunk type. Note how the cost of evaluating a thunk is deferred from Let to Var. Similarly, the cost of evaluating the body of a λ-abstraction is deferred to application. Rules Abs and App are otherwise standard.

The type rules Cons and Match for constructors and pattern matching are straightforward.[6] The Cons rule just ensures consistency between the arguments to a constructor and its result type. In correspondence with our operational semantics, there is no extra cost for constructors, since allocation is accounted for in rule Let. The Match rule deals with pattern-matching over an expression of a (possibly recursive) data type. The rule requires that all branches admit an

[5] We use the standard notation $x : A$ to denote the singleton context mapping variable x to type A, and a comma between two contexts denotes disjoint union.

[6] Note that these rules are simpler than in our earlier work [?], since data constructors do not carry *potential* as required for the amortisation technique.

$$\frac{}{\Gamma,\ x{:}\mathsf{T}^{q}(A) \vdash^{q}_{0} x : A} \qquad (\textsc{Var})$$

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

$$\frac{\Gamma, x{:}A \vdash^{q}_{0} e : C}{\Gamma \vdash^{0}_{0} \lambda x.e : A \xrightarrow{q} C} \qquad (\textsc{Abs})$$

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

$$\frac{B = \mu X.\{\cdots \mid c : \overline{A} \mid \cdots\}}{\Gamma,\ \overline{y}{:}\overline{A}[B/X] \vdash^{0}_{0} c(\overline{y}) : B} \qquad (\textsc{Cons})$$

$$\frac{|\overline{A}_i| = |\overline{x}_i| \qquad B = \mu X.\{c_i : \overline{A}_i\} \qquad \Gamma \vdash^{p}_{p'} e_0 : B \qquad \Gamma,\ \overline{x}_i{:}\overline{A}_i[B/X] \vdash^{p'}_{p''} e_i : C}{\Gamma \vdash^{p}_{p''} \mathsf{match}\ e_0\ \mathsf{with}\ \{c_i(\overline{x}_i)\texttt{->}e_i\} : C} \qquad (\textsc{Match})$$

Fig. 3. Syntax directed type rules

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

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

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

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

Fig. 4. Structural type rules

$$\frac{}{X <: X}$$

$$\frac{|\overline{A}_i| = |\overline{B}_i| \qquad A_{ij} <: B_{ij}}{\mu X.\{c_i : \overline{A}_i\}_{i=1}^{n} <: \mu X.\{c_i : \overline{B}_i\}_{i=1}^{n}}$$

$$\frac{A' <: A \qquad B <: B' \qquad p \leq q}{A \xrightarrow{p} B <: A' \xrightarrow{q} B'}$$

$$\frac{A <: B \qquad p \leq q}{\mathsf{T}^{p}(A) <: \mathsf{T}^{q}(B)}$$

Fig. 5. Subtyping relation

$$\frac{}{A \lhd A}$$

$$\frac{|\overline{A}_i| = |\overline{B}_i| \qquad A_{ij} \lhd_X B_{ij}}{\mu X.\{c_i : \overline{A}_i\}_{i=1}^{n} \lhd \mu X.\{c_i : \overline{B}_i\}_{i=1}^{n}}$$

$$\frac{}{A \lhd_X A}$$

$$\frac{p \leq q}{\mathsf{T}^{p}(X) \lhd_X \mathsf{T}^{q}(X)}$$

Fig. 6. Lowering thunk costs

identical result type and that estimated resources after execution of any of the branches are equal; fulfilling such a condition may require relaxing type and/or cost information using the structural rules below.

A significant difference from our previous work [**?**] lies in the typing of let-expressions. Typing $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ allows lower costs for the bound variable x within the recursively-defined expression e_1. Specifically, it allows zero costs for both the thunk itself and also for its recursive references; this is justified because any recursive access to the defined value cannot incur evaluation costs, since either the thunk is already in *whnf*, or the access would cause a self-referential loop (which is prevented by the "black-holing" in the operational semantics).

The type rule Let uses an auxiliary ordering relation $\lhd$ on annotated types defined in Figure 6. Note that relation $\lhd$ is different from subtyping ($<:$); the former is used in Let for lowering *only* the cost for the recursively-defined thunks in a μ-type (thereby increasing precision) while the latter is used in a structural rule to lower *any* thunk or arrow costs to allow common types for branches with distinct costs. Note also that it would be enough for $\lhd$ to lower μ-thunks costs to zero (since that will always lead to a more precise analysis), but because the type system is only constraining admissible type annotations, we choose to express this as an ordering relation and let the constraint solver choose suitable annotations in order to minimize costs.

The structural rules of Figure 4 allow the analysis to be relaxed in various ways: Relax allows the relaxing of cost bounds. Subtype and Supertype allow subtyping in the conclusion and supertyping in a hypothesis, respectively; these make use of an separate relation defined in Figure 5. Informally, $A <: B$ means that the A and B have identical type structure but A has lower cost annotations in both thunk and function types.

The crucial rule Prepay allows (part of) the cost of a thunk to be paid in advance, thus reducing the cost of further uses of the same variable. Rule Var requires the cost of the thunk to be paid for *every* use, as in call-by-name evaluation. However, Prepay allows the cost of a thunk to be shared, which models the effect of memoization in call-by-need evaluation.

Note that weakening and contraction are implicitly allowed without any restrictions, so type assumptions may be freely duplicated without requiring the application of an explicit type rule.[7]

4 Experimental results

We have constructed a prototype implementation of our analysis as an inference algorithm for the type system of Section 3. A publicly accessible web version with several editable examples (including the ones presented here) is available at `http://kashmir.dcc.fc.up.pt/cgi/lazy.cgi`. The implementation combines the analysis presented in this paper with our earlier amortised analysis [**?**]. These techniques complement each other: amortisation deals with recursive definitions over finite data, while our new system deals with co-recursive definitions on infinite data. In this paper, of course, we focus only on examples of co-recursive definitions here.

The analysis is fully automatic, i.e. it does not require type annotations from the programmer and either produces an annotated typing or fails when no cost bounds can be found. Inference for a whole program is currently performed in three steps:

1. We first perform Damas-Milner type inference to obtain an unannotated version of the type derivation. The unannotated types form a free algebra and can be determined using standard first-order unification.
2. We then decorate types with fresh variables and perform a traversal of the type derivation gathering linear constraints among annotations following the type rules.
3. Finally, we feed the linear constraints to a standard linear programming solver[8] with the objective of minimizing the overall expression cost on the turnstile. Any solution gives rise to a valid annotated typing derivation.

As in Standard ML or Haskell, we associate constructors with specific data types (e.g. `Cons` and `Nil` with lists). This ensures that the use of the Cons rule is syntax-directed. Also, the implementation includes some minor language extensions, namely, primitive integers and associated arithmetic operations.

It remains to explain how to decide the use of the structural rules from Figure 4. Prepay is used immediately whenever bound variables are introduced, namely, in the body of a lambda, let-expression or match alternative. This can be done uniformly because the rule allows any part of the cost to be paid (including zero); hence, we defer to the LP solver the choice of how much should each individual thunk be prepaid 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).

[7] This is again quite different to [**?**], where restrictions on weakening and contraction are needed because of the amortisation technique.

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

4.1 Zipping streams

Our first example is a co-recursive *zipWith* function that combines two infinite lists by applying a function to corresponding elements:

```
let zipWith = \f xs ys -> match xs with
    Cons(x,xs’) -> match ys with
      Cons(y,ys’) -> let t = f x y
                     in let r = zipWith f xs’ ys’
                     in let s = Cons(t,r) in s
```

The analysis infers the following annotated type:

```
zipWith : T(T(a) -> T(b) -> c) ->
    T(Rec{Cons:(T(a),T(#)) | Nil:()}) ->
      T(Rec{Cons:(T(b),T(#)) | Nil:()}) ->@3
        Rec{Cons:(T(c),T@3(#)) | Nil:()}
```

Some remarks on the analysis output: μ-types are written `Rec{...}` with an implicit bound type variable represented by an ‘`#`’-sign; annotations in thunk and arrow types are marked by an ‘`@`’-sign; for readability, zero annotations are omitted. Hence, the type above ensures that *zipWith* yields a list where each successive tail costs (at most) 3 allocations (`T@3(#)`) plus 3 for the application itself (`->@3`); thus the cost for obtaining n elements is bounded by $3 + 3n$.

Note that the inference algorithm outputs only one of an infinite set of admissible solutions. Because *zipWith* was analysed in isolation, we obtained a type with zero costs for the function argument and, therefore, where all costs of the result are assigned to the list spine. If *zipWith* was used in a context where the argument function requires positive costs, we might instead obtain a type with costs in both head and spine thunks, e.g.:

```
zipWith : T(T(a) -> T(b) ->@1 c) ->
    T(Rec{Cons:(T(a),T(#)) | Nil:()}) ->
      T(Rec{Cons:(T(b),T(#)) | Nil:()}) ->@3
        Rec{Cons:(T@1(c),T@3(#)) | Nil:()}
```

4.2 Fibonacci numbers

Our next example is the infinite list of Fibonacci numbers from the introduction; this can be defined using the *zipWith* function shown before:

```
let zero = 0 in
let one  = 1 in
let plus = \x y -> x+y in
let fibs = (let t = match fibs with
                    Cons(x,fibs’) -> zipWith plus fibs fibs’
            in let r = Cons(one,t)
            in let s = Cons(zero,r)
            in s)
```

Here we extend the language with a type for integers by adding suitable constructors for each constant and primitive arithmetic operators. As in the STG machine [?], operators must be fully applied; higher-order values can be obtained using explicit lambda-expressions (`plus` in the example). We also assume that arithmetic operations have no intrinsic allocation costs, but since arguments of applications are restricted to be variables, compound results have to be let-bound (and thus heap allocated) anyway. The type inferred for *fibs* is as follows:

```
fibs : Rec{Cons:(T(Int),T@3(#)) | Nil:()}
```

From the type above we see that the infinite list evaluating each successive of Fibonacci requires at most 3 allocations. This matches exactly the cost of *zipWith* because *plus* has zero cost in our model. Note that it is essential that `fibs'` is the tail of `fibs`, for otherwise one would have to pay twice for evaluating each argument of `zipWith`. Thanks to our novel LET typing rule, our analysis can recognise this reduction in cost due to aliasing.

4.3 The Hamming problem

Our final example is the *Hamming problem*: produce an infinite list of numbers in ascending order and without duplicates, starting with 1 and such that, whenever x occurs in the list, so do $2 \times x$, $3 \times x$ and $5 \times x$. One elegant Haskell solution (from Bird and Wadler's textbook [?]) uses a function that merges infinite lists in ascending order:

```
merge (x:xs) (y:ys) | x==y = x : merge    xs     ys
                    | x<y  = x : merge    xs  (y:ys)
                    | x>y  = y : merge (x:xs)    ys
```

The Hamming numbers can then be defined co-recursively using *merge* and the standard list *map*:

```
hamming = 1 : merge (map (2*) hamming)
                (merge (map (3*) hamming) (map (5*) hamming))
```

Using some informal reasoning about the sharing properties of the cyclic list above, Bird and Wadler argue that n elements can be computed with bounded $O(n)$ cost [?]. We will see that our analysis can confirm this with a precise bound.

However, a direct translation of the above definitions into our core language does *not* admit an annotated type in our system: the two uses of *merge* in the definition of *hamming* require different cost annotations. Because of this, the constraints generated by the reconstruction algorithm will not admit a solution.[9]

One work around for this limitation is to simply duplicate the definition of *merge* so that each use can be assigned a precise type[10]:

[9] This does not happen for *map* because, in this particular problem, all the uses have identical costs.

[10] A more general solution would be to extend the analysis to include *effect polymorphism* – we leave this as further work (see Section 7).

	evaluation demand	0	1	2	3	4	5	6	7	8	9	10
Fibs	analysis	8	8	11	14	17	20	23	26	29	32	35
	semantics	8	8	8	11	14	17	20	23	26	29	32
Hamming	analysis	17	17	30	43	56	69	82	95	108	121	134
	semantics	17	17	30	35	42	47	54	64	69	76	86

Table 1. Comparison of analysis with the operational semantics.

```
hamming = 1 : merge1 (map (2*) hamming)
                (merge2 (map (3*) hamming) (map (5*) hamming))
```

With this translation, the reconstruction algorithm is able to obtain the following annotated types:

```
merge1 : T@3(Rec{Cons:(T(Int),T@3(#)) | Nil:()}) ->
          T@3(Rec{Cons:(T(Int),T@3(#)) | Nil:()}) ->@8
           Rec{Cons:(T(Int),T@8(#)) | Nil:()}
merge2 : T@3(Rec{Cons:(T(Int),T@3(#)) | Nil:()}) ->
          T@8(Rec{Cons:(T(Int),T@8(#)) | Nil:()}) ->@13
           Rec{Cons:(T(Int),T@13(#)) | Nil:()}
hamming : Rec{Cons:(T(Int),T@13(#)) | Nil:()}
```

The type inferred for *hamming* confirms Bird and Wadler's reasoning and provides a precise bound: each successive Fibonacci number requires (at most) 13 allocations.

Finally, we note that the revised Let type rule that is presented in this paper is essential for obtaining annotated types for the *fibs* and *hamming* examples above. In fact, these two examples do not admit annotated types using just the amortised analysis described in [?].

4.4 Comparison with the instrumented semantics

Table 1 presents a short assessment of the quality of the upper-bounds obtained from our analysis by comparison with the exact costs obtained from an implementation of the operational semantics of Section 2. Figures are grouped by the evaluation demand from the result infinite lists, where 0 evaluates the list to *whnf* (i.e. just a `Cons` cell), 1 evaluates the first element to *whnf*, 2 evaluates the second, etc. We first note that the analysis is indeed producing upper-bounds; this is true in general as shown by the soundness theorem proved in Section 5.

The results for the *fibs* are quite accurate: the inferred cost of 3 allocation for each successive elements is exact. There is a fixed overestimation of 3 allocations because a recursive type `Rec{Cons:(T(Int),T@3(#))|...}` cannot distinguish the lower cost of the first two elements. The results for *hamming* are less accurate; this is because the exact cost for successive elements is not constant, instead varying between 0 and 10 allocations; however, our annotated types assign identical cost for the entire spine (`Rec{Cons:(T(Int),T@13(#)|...}` — i.e. 13 allocations), hence the overestimation.

$$\frac{\mathcal{H}@w \text{ is defined}}{\mathcal{H},\mathcal{S},\mathcal{L} \vdash^{m}_{m} w \Downarrow^{\mathrm{I}} w, \mathcal{H}} \tag{$\textsc{Whnf}_{\Downarrow^{\mathrm{I}}}$}$$

$$\frac{\ell \notin \mathcal{L} \qquad \mathcal{H}[\ell \mapsto e], \mathcal{S}, \mathcal{L} \cup \{\ell\} \vdash^{m}_{m'} e \Downarrow^{\mathrm{I}} w, \mathcal{H}'[\ell \mapsto e]}{\mathcal{H}[\ell \mapsto e], \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} \ell \Downarrow^{\mathrm{I}} w, \mathcal{H}'[\ell \mapsto w]} \tag{$\textsc{Var}_{\Downarrow^{\mathrm{I}}}$}$$

$$\frac{\ell, \ell' \text{ are fresh} \qquad \mathcal{H}[\ell \mapsto e_1[\ell'/x],\, \ell' \mapsto \mathsf{ind}(\ell)], \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e_2[\ell/x] \Downarrow^{\mathrm{I}} w, \mathcal{H}'}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m+1}_{m'} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow^{\mathrm{I}} w, \mathcal{H}'} \tag{$\textsc{Let}_{\Downarrow^{\mathrm{I}}}$}$$

$$\frac{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e \Downarrow^{\mathrm{I}} u, \mathcal{H}' \qquad \mathcal{H}'@u = \lambda x.\, e' \qquad \mathcal{H}', \mathcal{S}, \mathcal{L} \vdash^{m'}_{m''} e'[\ell/x] \Downarrow^{\mathrm{I}} w, \mathcal{H}''}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m''} e\ \ell \Downarrow^{\mathrm{I}} w, \mathcal{H}''} \tag{$\textsc{App}_{\Downarrow^{\mathrm{I}}}$}$$

$$\frac{\begin{array}{c}\mathcal{H}, \mathcal{S} \cup \left(\bigcup_{i=1}^{n} \{\overline{x}_i\} \cup \mathrm{BV}(e_i)\right), \mathcal{L} \vdash^{m}_{m'} e_0 \Downarrow^{\mathrm{I}} u, \mathcal{H}' \\ \mathcal{H}'@u = c_k(\overline{\ell}) \qquad \mathcal{H}', \mathcal{S}, \mathcal{L} \vdash^{m'}_{m''} e_k[\overline{\ell}/\overline{x}_k] \Downarrow^{\mathrm{I}} w, \mathcal{H}''\end{array}}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m''} \mathsf{match}\ e_0\ \mathsf{with}\ \{c_i(\overline{x}_i)\text{->}e_i\}_{i=1}^{n} \Downarrow^{\mathrm{I}} w, \mathcal{H}''} \tag{$\textsc{Match}_{\Downarrow^{\mathrm{I}}}$}$$

Fig. 7. Indirection semantics.

5 Soundness

In this section we formulate the soundness of our analysis from Section 3. The structure of our proof is as follows:

1. in Section 5.1, we define a variant of the operational semantics which uses *indirections*;
2. in Section 5.5, we establish the soundness of the type rules against the indirection semantics.
3. finally, in Section 5.6, we show the equivalence of the original semantics and the revised indirection semantics (including preservation of resource bounds).

5.1 Indirection semantics

To facilitate proving the soundness of the type analysis of Section 3 we will consider a variant of the operational semantics. We exploit a new syntactic form for *indirections*. These do not occur in the original program, but are used internally by the evaluation mechanism.

$$e ::= \cdots \ \mid\ \mathsf{ind}(x) \qquad\qquad w ::= \lambda x.e \ \mid\ c(\overline{x}) \ \mid\ \mathsf{ind}(x)$$

Operationally, an indirection $\mathsf{ind}(x)$ will be treated similarly to the variable x (i.e. it references some expression in the heap). However, evaluation of an indirection will *not* force the evaluation of a thunk; instead, it succeeds immediately. This will be crucial for establishing the soundness of the type rule Let.

Figure 7 presents the revised semantics as a relation $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e \Downarrow^{\mathrm{I}} w, \mathcal{H}'$ where the components play identical roles to the semantics of Section 2. Note that, in this revised judgment, both expressions and *whnfs* may be indirections; they may also occur in heaps, $\mathcal{H}$ or $\mathcal{H}'$.

The $\textsc{Whnf}_{\Downarrow\text{I}}$ rule is revised to allow indirections as results. Indirections are used to mark recursive self-references, and thus this revised rule allows justifying lower costs for such cases in the soundness proof. The $\textsc{Var}_{\Downarrow\text{I}}$ rule is identical to the previous one. The revised rule for let-expressions $\textsc{Let}_{\Downarrow\text{I}}$ substitutes the bound variable in e_1 by an indirection instead of a self-reference; this will allow the costs of (co-)recursive uses to be distinguished in the soundness proof.

The revised rules $\textsc{Whnf}_{\Downarrow\text{I}}$, $\textsc{App}_{\Downarrow\text{I}}$ and $\textsc{Match}_{\Downarrow\text{I}}$ make use of a auxiliary partial function $\mathcal{H}@w$ for de-referencing a result w with respect to a heap $\mathcal{H}$. For constructors and abstractions this function is the identity; and in the case of indirections it dereferences a heap location. It is undefined for other expressions.

$$\mathcal{H}@\lambda x.e \overset{\text{def}}{=} \lambda x.e \qquad \mathcal{H}@c(\overline{x}) \overset{\text{def}}{=} c(\overline{x}) \qquad \mathcal{H}@\mathsf{ind}(\ell) \overset{\text{def}}{=} \mathcal{H}(\ell) \text{ if } \ell \in \text{dom}(\mathcal{H})$$

5.2 Typing rule for indirections

We introduce the following typing rule Ind for indirections:

$$\frac{A' \lhd A}{\Gamma, x{:}\mathsf{T}^q(A) \vdash^{0}_{0} \mathsf{ind}(x) : A'} \qquad (\textsc{Ind})$$

This rule is similar to Var except that it allows lowering the thunk costs both on the judgment and on the recursive type; we use the relation $\lhd$ of Figure 6 for the latter. The rule will be needed in the soundness proof solely for establishing well-typing of intermediate heap configurations (since indirections may not occur within source programs).

5.3 Global types and balance

The *global types* are given by a mapping $\mathcal{M}$ from locations to (annotated) types. The intuition is that when $\mathcal{M}(\ell) = \mathsf{T}^q(A)$ then q is (an upper bound of) the cost of evaluating ℓ and the resulting *whnf* admits type A. Furthermore, we introduce an auxiliary *balance* function $\mathcal{B}$ mapping locations to non-negative numbers. This keeps track of the partial costs that have been paid in advance by uses of the Prepay rule. We also define the *balance sum* over a heap configuration as the sum of the balance associated with all thunks that are not under evaluation:

$$\textstyle\sum_{\mathcal{H},\mathcal{L}} \mathcal{B} \overset{\text{def}}{=} \sum\{\mathcal{B}(\ell) : \ell \in \text{dom}(\mathcal{H}) \text{ and } \ell \notin \mathcal{L} \text{ and } \mathcal{H}(\ell) \text{ is not a } \textit{whnf}\}$$

Note that the balance is needed to prove the soundness of the analysis, but is *not* part of the operational semantics — in particular, it does not incur runtime costs.

5.4 Consistency and compatibility

We can now define the principal soundness invariants of our analysis, namely, a *consistency* relation for typing heap configurations and a *compatibility* relation between global types and contexts. We proceed by first defining typing of a single location and then extend it to typing a heap configuration.

Definition 2 (Typing of locations). *We say that location ℓ admits type $\mathsf{T}^q(A)$ under context Γ, balance $\mathcal{B}$, heap configuration $(\mathcal{H}, \mathcal{L})$, and write $\Gamma, \mathcal{B}; \mathcal{H}, \mathcal{L} \vdash_{\mathrm{Loc}} \ell : \mathsf{T}^q(A)$ if one of the following cases holds:*

(Loc1) $\mathcal{H}(\ell)$ *is in* whnf *and* $\Gamma \vdash^{0}_{0} \mathcal{H}(\ell) : A$*;*

(Loc2) $\mathcal{H}(\ell)$ *is not in* whnf *and* $\Gamma \vdash^{q\,+\,\mathcal{B}(\ell)}_{0} \mathcal{H}(\ell) : A$*;*

The two cases above are mutually exclusive: Loc1 applies when the expression in the heap is already in *whnf*; otherwise Loc2 applies. 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. Note that in [**?**], we additionally distinguished whether a location was under evaluation. Since we do not use the notion of *potential* here, this is no longer needed.

Definition 3 (Typing of heap configurations). *We say that a heap configuration $(\mathcal{H}, \mathcal{L})$ is consistent with context Γ, global types $\mathcal{M}$ and balance $\mathcal{B}$, and write $\Gamma, \mathcal{B} \vdash_{\mathrm{Mem}} (\mathcal{H}, \mathcal{L}) : \mathcal{M}$, if and only if for all $\ell \in dom(\mathcal{H})$ we have $\Gamma, \mathcal{B}; \mathcal{H}, \mathcal{L} \vdash_{\mathrm{Loc}} \ell : \mathcal{M}(\ell)$.*

The compatibility relation enforces that the global types of locations are supertypes (i.e. have lower costs) of the types occurring in a context.

Definition 4 (Compatibility). *We say that a global types $\mathcal{M}$ are* compatible *with a context Γ, written $\mathcal{M} <: \Gamma$, if and only if $\mathcal{M}(\ell) <: A$ for all $\ell{:}A \in \Gamma$.*

5.5 Soundness of the proof system

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

Theorem 1 (Soundness). *Let $t \geq 0$ be fixed but arbitrary. If the following statements hold*

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

$$\Gamma, \mathcal{B} \vdash_{\mathrm{Mem}} (\mathcal{H}, \mathcal{L}) : \mathcal{M} \tag{2}$$

$$\mathcal{M} <: \Gamma \tag{3}$$

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash e \Downarrow^{\mathrm{I}} w, \mathcal{H}' \tag{4}$$

then for all m such that $m \geq t + p + \sum_{\mathcal{H},\mathcal{L}} \mathcal{B}$, there exists m', Γ', $\mathcal{B}'$ and $\mathcal{M}'$ such that

$$\Gamma' \vdash^{0}_{0} w : A \tag{5}$$

$$\Gamma', \mathcal{B}' \vdash_{\mathrm{Mem}} (\mathcal{H}', \mathcal{L}) : \mathcal{M}' \tag{6}$$

$$\mathcal{M}' <: \Gamma' \tag{7}$$

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e \Downarrow^{\mathrm{I}} w, \mathcal{H}' \tag{8}$$

$$m' \geq t + p' + \textstyle\sum_{\mathcal{H}',\mathcal{L}} \mathcal{B}' \tag{9}$$

Informally, the soundness theorem reads as follows: if an expression e admits type A (1), the heap can be typed (2) (3), and the evaluation is successful (4), then the result *whnf* also admits type A (5). Furthermore, the final heap can also be typed (6) (7) and the static bounds that are obtained from the typing of e give safe resource estimates for evaluation (8) (9). Because of space limitations, we will only present the cases that differ significantly from our previous work [**?**], particularly the revised typing rule for *let* and for indirections.

Proof. The proof is by induction on the lengths of the derivations of evaluation (4) and typing (1) ordered lexicographically, with the former taking priority over the later.

We proceed by case analysis of the typing rule used in premise (1), considering just some representative cases.

Case Let. The typing premise (1) instantiates as

$$\Gamma \;\vdash^{1+p}_{p}\; \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : C$$

By inversion of rule Let together with the substituition lemma, we get

$$\Gamma, \ell'{:}\mathsf{T}^0(A') \;\vdash^{q}_{0}\; e_1[\ell'/x] : A \tag{10}$$

$$\Gamma, \ell{:}\mathsf{T}^q(A) \;\vdash^{p}_{p'}\; e_2[\ell/x] : C \tag{11}$$

where $A' \lhd A$. The evaluation premise (4) instantiates as

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \;\vdash^{1+m}_{m'}\; \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow^{\mathrm{I}} w, \mathcal{H}'$$

from which we get $\mathcal{H}_0, \mathcal{S}, \mathcal{L} \;\vdash^{m}_{m'}\; e_2[\ell/x] \Downarrow^{\mathrm{I}} w, \mathcal{H}'$ where $\mathcal{H}_0 = \mathcal{H}[\ell \mapsto e_1[\ell'/x], \ell' \mapsto \mathsf{ind}(\ell)]$. Define:

$$\begin{aligned} \mathcal{B}_0 &= \mathcal{B}[\ell \mapsto 0, \ell' \mapsto 0] \\ \mathcal{M}_0 &= \mathcal{M}[\ell \mapsto \mathsf{T}^q(A), \ell' \mapsto \mathsf{T}^0(A')] \\ \Gamma_0 &= \Gamma,\ \ell{:}\mathsf{T}^q(A),\ \ell'{:}\mathsf{T}^0(A') \end{aligned}$$

To apply induction to the evaluation of $e_2[\ell/x]$ we first need to re-establish type consistency and compatibility. Type consistency for ℓ follows from (10) and (Loc2); and for ℓ' follows directly from the type rule Ind and (Loc1). Compatibility is immediate because the types for ℓ and ℓ' in Γ_0 are exactly $\mathcal{M}_0(\ell)$ and $\mathcal{M}_0(\ell')$. Applying induction to (11) then yields all required conclusions. Note that the lower thunk costs for A' are only allowed for the recursive reference ℓ' introduced in the let-expression; crucially this is sound only because the recursive reference is introduced in the heap as in indirection whose cost is ignored by the typing rule Ind. Otherwise compatibility would not hold.

Case Ind. This case is immediate: taking $\Gamma' = \Gamma$, $\mathcal{B}' = \mathcal{B}$, $\mathcal{M}' = \mathcal{M}$ and $m' = m$ yields all required conclusions.

Case Var. The typing premise is $\Gamma, \ell{:}\mathsf{T}^q(A) \vdash^{q}_{0} \ell : A$ and the evaluation premise is $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} \ell \Downarrow^{\mathrm{I}} w, \mathcal{H}'[\ell \mapsto w]$; by inversion of rule $\textsc{Var}_{\Downarrow^{\mathrm{I}}}$ we get $\mathcal{H}, \mathcal{S}, \mathcal{L} \cup \{\ell\} \vdash^{m}_{m'} \mathcal{H}(\ell) \Downarrow^{\mathrm{I}} w, \mathcal{H}'$ and $\ell \notin \mathcal{L}$. By the type compatibility hypothesis we get that

$$\mathcal{M}(\ell) <: \mathsf{T}^q(A)$$

We now distinguish the two applicable cases:

$\mathcal{H}(\ell)$ **is in *whnf*.** The evaluation succeeds immediately by either $\textsc{Whnf}_{\Downarrow^{\mathrm{I}}}$ and we have $w = \mathcal{H}(\ell)$, $\mathcal{H} = \mathcal{H}'$ and $m' = m$, i.e. the update is without effect. Taking $\Gamma' = \Gamma$, $\mathcal{B}' = \mathcal{B}$, $\mathcal{M}' = \mathcal{M}$. By type consistency, we get

$$\Gamma \vdash^{0}_{0} \mathcal{H}(\ell) : A$$

which is equivalent to the required conclusion

$$\Gamma' \vdash^{0}_{0} w : A$$

The remaining conclusions are immediate because $\mathcal{H}' = \mathcal{H}$.

$\mathcal{H}(\ell)$ **is not in *whnf*.** Let $\mathsf{T}^r(\widehat{A}) = \mathcal{M}(\ell)$. In this case, type consistency for ℓ requires (Loc2), which instantiates as

$$\Gamma, \ell{:}\mathsf{T}^q(A) \vdash^{r\,+\,\mathcal{B}(\ell)}_{0} \mathcal{H}(\ell) : \widehat{A}$$

Recall that from compatibility for location ℓ we get $\mathsf{T}^r(\widehat{A}) <: \mathsf{T}^q(A)$, which implies $\widehat{A} <: A$. By applying the type rule Subtype we obtain

$$\Gamma, \ell{:}\mathsf{T}^q(A) \vdash^{r\,+\,\mathcal{B}(\ell)}_{0} \mathcal{H}(\ell) : A$$

By inversion of the evaluation premise we get

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \cup \{\ell\} \vdash^{m}_{m'} \mathcal{H}(\ell) \Downarrow^{\mathrm{I}} w, \mathcal{H}'$$

and $\ell \notin \mathcal{L}$. We now apply the induction hypothesis to the evaluation of $\mathcal{H}(\ell)$. Note that according to rule $\textsc{Var}_{\Downarrow^{\mathrm{I}}}$, we apply the induction hypothesis for $\mathcal{L}' = \mathcal{L} \cup \{\ell\}$. Observe that $m \geq t + p + \mathcal{B}(\ell) + \sum_{\mathcal{H}, \mathcal{L} \cup \{\ell\}} \mathcal{B}$ holds as required. We thus obtain m', Γ', $\mathcal{B}'$ and $\mathcal{M}'$ such that

$$\Gamma' \vdash^{0}_{0} w : A \tag{12}$$

$$\Gamma', \mathcal{B}' \vdash_{\textsc{Mem}} (\mathcal{H}', \mathcal{L} \cup \{\ell\}) : \mathcal{M}' \tag{13}$$

$$\mathcal{M}' <: \Gamma' \tag{14}$$

$$m' \geq t + p' + \textstyle\sum_{\mathcal{H}', \mathcal{L} \cup \{\ell\}} \mathcal{B} \tag{15}$$

The only remaining proof obligations is to re-establish these statements for the updated heap $\mathcal{H}'' = \mathcal{H}'[\ell \mapsto w]$. In particular, we need

$$\Gamma', \mathcal{B}' \vdash_{\textsc{Mem}} (\mathcal{H}'', \mathcal{L}) : \mathcal{M}' \tag{16}$$

$$m' \geq t + p' + \textstyle\sum_{\mathcal{H}', \mathcal{L} \cup \{\ell\}} \mathcal{B} \tag{17}$$

The only location changed from (13) to (16) is ℓ were the applicable case changes from (Loc2) to (Loc1). But the latter is immediate from (12) because $\mathcal{H}''(\ell) = w$. Since the balance sum skips locations mapped to whnf, we have $\sum_{\mathcal{H}',\mathcal{L}\cup\{\ell\}} \mathcal{B} = \sum_{\mathcal{H}'',\mathcal{L}} \mathcal{B}$, thus establishing (17) as required.

Case App. The typing and evaluation premises in this case are

$$\Gamma, y{:}A \vdash^{p+q}_{p'} e\,y : C \tag{18}$$
$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m''} (e\ \ell) \Downarrow^{\mathrm{I}} u, \mathcal{H}'' \tag{19}$$

By inversion of the type rule (App) applied to (18) we obtain

$$\Gamma, y{:}A \vdash^{p}_{p'} e : A \xrightarrow{q} C \tag{20}$$

By inversion of the evaluation rule $\text{App}_{\Downarrow^{\mathrm{I}}}$ applied to (19) we get

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e \Downarrow^{\mathrm{I}} u, \mathcal{H}' \tag{21}$$
$$\mathcal{H}'@u = \lambda x.e' \tag{22}$$
$$\mathcal{H}', \mathcal{S}, \mathcal{L} \vdash^{m'}_{m''} e'[\ell/x] \Downarrow^{\mathrm{I}} w, \mathcal{H}'' \tag{23}$$

Taking $t' = t + q$, we show that we verify the conditions for applying induction to the evaluation of e because

$$m \geq \underbrace{(t+q)}_{t'} + p + \textstyle\sum_{\mathcal{H},\mathcal{L}} \mathcal{B} \tag{24}$$

By induction we obtain $\Gamma', \mathcal{B}', \mathcal{M}'$ such that

$$\Gamma' \vdash^{0}_{0} u : A \xrightarrow{q} C \tag{25}$$
$$\Gamma', \mathcal{B}' \vdash_{\text{Mem}} (\mathcal{H}', \mathcal{L}) : \mathcal{M}' \tag{26}$$
$$\mathcal{M}' <: \Gamma' \tag{27}$$
$$\begin{aligned} m' &\geq (t+q) + p' + \textstyle\sum_{\mathcal{H}',\mathcal{L}} \mathcal{B}' \\ &= \underbrace{(t+p')}_{t''} + q + \textstyle\sum_{\mathcal{H}',\mathcal{L}} \mathcal{B}' \end{aligned} \tag{28}$$

By $\mathcal{H}'@u = \lambda x.\,e'$ we either have $u = \lambda x.\,e'$ or $u = \mathsf{ind}(\kappa)$. In either case the type remains unchanged, due to rule Ind and due to $\lhd$ not altering function types, we thus get

$$\Gamma' \vdash^{0}_{0} \lambda x.e' : A \xrightarrow{q} C$$

Using a standard Abs inversion lemma, we get

$$\Gamma', x{:}A \vdash^{q}_{0} e' : C$$

By the substitution lemma we get

$$\Gamma', \ell{:}A \vdash^{q}_{0} e'[\ell/x] : C$$

We can now apply induction again to (23) (evaluation of $e'[\ell/x]$) and obtain $m'', \Gamma'', \mathcal{M}'', \mathcal{B}''$ satisfying all desired conclusions:

$$\Gamma'' \vdash^{0}_{0} w : C \tag{29}$$

$$\Gamma'', \mathcal{B}'' \vdash_{\textsc{Mem}} (\mathcal{H}'', \mathcal{L}) : \mathcal{M}'' \tag{30}$$

$$\mathcal{M}'' <: \Gamma'' \tag{31}$$

$$\begin{aligned} m'' &\geq (t + p') + 0 + \textstyle\sum_{\mathcal{H}'', \mathcal{L}} \mathcal{B}'' \\ &= t + p' + \textstyle\sum_{\mathcal{H}'', \mathcal{L}} \mathcal{B}'' \end{aligned} \tag{32}$$

Corollary 1. *If the evaluation of a closed expression e with an initially empty memory succeeds, and e is well-typed $\emptyset \vdash^{p}_{p'} e : A$, then the total amount of allocations during this evaluation is bounded by p.*

Proof. This is a direct consequence of Theorem 1, by choosing $t = 0$. Observe that preconditions 2 and 3 are trivial in an empty memory configuration. Furthermore, the sum over the balance is an empty sum and thus equal to zero. Thus, for all m such that $m \geq p$, there exists some $m', \mathcal{H}'$ with $\emptyset, \emptyset, \emptyset \vdash^{m}_{m'} e \Downarrow^{\mathrm{I}} w, \mathcal{H}'$

5.6 Relationship with Launchbury's semantics

In this section we sketch the correspondence between the indirection semantics and Launchbury's standard semantics, which justifies our cost model. More precisely, we prove for every evaluation in the standard semantics that there is a corresponding one in the indirection semantics and that the conversion preserves cost (Theorem 2). The reverse correspondence also holds, but is not required for the soundness result, so we do not pursue it here. The development of the relationship follows [?]. We start by defining an auxiliary function to remove indirections from a heap.

Definition 5. *Consider a heap $\mathcal{H}$ such that $\mathcal{H}(\ell) = \mathit{ind}(\ell')$. The* indirection erasure *of ℓ from $\mathcal{H}$, written $\mathcal{H} \ominus \ell$, is defined as follows:*

$$\emptyset[\ell \mapsto \mathit{ind}(\ell')] \ominus \ell \stackrel{def}{=} \emptyset$$

$$\mathcal{H}[\kappa \mapsto e,\, \ell \mapsto \mathit{ind}(\ell')] \ominus \ell \stackrel{def}{=} (\mathcal{H}[\ell \mapsto \mathit{ind}(\ell')] \ominus \ell)[\kappa \mapsto e[\ell'/\ell]]$$

Note that we remove not just the indirection $\ell \mapsto \mathsf{ind}(\ell')$ but also rename all occurrences of ℓ to ℓ' in the remaining heap expressions.

Using indirection erasure, we can now define a relation on heaps $\mathcal{H} \succcurlyeq \mathcal{H}'$ which informally says that we obtain $\mathcal{H}'$ from $\mathcal{H}$ by removing a sequence of indirections.

Definition 6. *We say that $\mathcal{H}$ is* indirection-related *to $\mathcal{H}'$ and write $\mathcal{H} \succcurlyeq \mathcal{H}'$ iff there exists a (possibly empty) sequence of locations $\overline{\ell}$ such that $\mathcal{H}(\ell_i)$ is an indirection and $\mathcal{H} \ominus \overline{\ell} = \mathcal{H}'$.*

The next two lemmas state some auxiliary results about $\succcurlyeq$; the proofs are similar to the corresponding results from [?].

Lemma 1. *$\succcurlyeq$ is reflexive and transitive (i.e. a pre-order relation on heaps).*

Lemma 2. *If $\mathcal{H} \succcurlyeq \mathcal{H}'$ then $dom(\mathcal{H}) \supseteq dom(\mathcal{H}')$.*

Because expressions have free variables which must be interpreted in the context of a heap, it is convenient to extend the indirection relation to pairs $(\mathcal{H}, e)$ of a heap and associated expression; we do so by simply introducing the expression in a fresh location.

Definition 7. *We say that $(\mathcal{H}, e)$ is indirection-related to $(\mathcal{H}', e')$ and write $(\mathcal{H},\, e) \succcurlyeq (\mathcal{H}',\, e')$ iff there exists $\ell \notin dom(\mathcal{H}) \cup dom(\mathcal{H}') \cup FV(\mathcal{H}) \cup FV(\mathcal{H}')$ such that $\mathcal{H}[\ell \mapsto e] \succcurlyeq \mathcal{H}'[\ell \mapsto e']$.*

Before presenting the correspondence result we state some auxiliary lemmas; for space restrictions we omit most proofs.

Lemma 3. *If $(\mathcal{H}, e) \succcurlyeq (\mathcal{H}', e')$ then e' is a renaming of e i.e. there exist variables $\overline{x}$ and $\overline{y}$ such that $e[\overline{y}/\overline{x}] = e'$.*

Lemma 4. *If $(\mathcal{H}, e) \succcurlyeq (\mathcal{H}', e')$ and $\ell \in dom(\mathcal{H}')$ and $\ell \in FV(e')$ then $\ell \in FV(e)$.*

Lemma 5. *If $(\mathcal{H}, u) \succcurlyeq (\mathcal{H}', w)$ and w is in* whnf*, then $\mathcal{H}@u$ is defined (e.g. u is either a* whnf *or an indirection from which a* whnf *can be reached in a finite number of steps).*

Proof (Sketch.). By the definition of $\succcurlyeq$, there is a sequence of locations $\overline{\ell}$ such that $\mathcal{H} \ominus \overline{\ell} = \mathcal{H}'$. The proof is by induction on the length of $\overline{\ell}$.

Lemma 6. *If $(\mathcal{H},\, \textsf{let}\ x = e_1\ \textsf{in}\ e_2) \succcurlyeq (\mathcal{H}', \textsf{let}\ x = e_1'\ \textsf{in}\ e_2')$ and ℓ is a fresh location then*

$$(\mathcal{H}[\ell \mapsto e_1[\ell/x]],\, e_2[\ell/x]) \succcurlyeq (\mathcal{H}'[\ell \mapsto e_1'[\ell/x]],\, e_2'[\ell/x])\ .$$

We can now finally establish the correspondence between $\Downarrow$ and $\Downarrow^{\mathrm{I}}$.

Theorem 2. *If $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e \Downarrow w, \mathcal{H}'$ then for all $\widehat{\mathcal{H}}$ such that $\widehat{\mathcal{H}} \succcurlyeq \mathcal{H}$ there exists $\widehat{\mathcal{H}}'$ and $\widehat{w}$ such that:*

$$\widehat{\mathcal{H}}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e \Downarrow^{\mathrm{I}} \widehat{w}, \widehat{\mathcal{H}}'$$
$$(\widehat{\mathcal{H}}', \widehat{w}) \succcurlyeq (\mathcal{H}', w)$$

In order to prove the above theorem by induction on the evaluation we need to strengthen the statement by allowing the evaluations to start from indirection-related heap-expression pairs.

Proposition 1. *If $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e \Downarrow w, \mathcal{H}'$ then for all $\widehat{\mathcal{H}}$ and $\widehat{e}$ with*

$$(\widehat{\mathcal{H}},\, \widehat{e}) \succcurlyeq (\mathcal{H},\, e) \tag{33}$$

there exists $\widehat{\mathcal{H}}'$ and $\widehat{w}$ such that:

$$\widehat{\mathcal{H}}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} \widehat{e} \Downarrow^{\mathrm{I}} \widehat{w}, \widehat{\mathcal{H}}' \tag{34}$$

$$(\widehat{\mathcal{H}}',\, \widehat{\mathcal{H}}'@\widehat{w}) \succcurlyeq (\mathcal{H}',\, w) \tag{35}$$

Proof. By induction on the derivation of the evaluation $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e \Downarrow w, \mathcal{H}'$; we proceed by case-analysis of the last rule used. Due to space limitations, we only present selected cases.

Case WHNF$_{\Downarrow}$. The premises are $\mathcal{H}, \mathcal{S}, \mathcal{L},\ \vdash^{m}_{m} w \Downarrow w, \mathcal{H}$ and $(\widehat{\mathcal{H}}, u) \succcurlyeq (\mathcal{H}, w)$ for some $\widehat{\mathcal{H}}$ and expression u. By Lemma 5, we get that $\widehat{\mathcal{H}}@u$ is defined. This satisfies the side condition of rule WHNF$_{\Downarrow^{\mathrm{I}}}$. Hence, we application of this evaluation rule yields the required conclusion (34). Conclusion (35) follows by transitivity of $\succcurlyeq$.

Case VAR$_{\Downarrow}$. The evaluation premise is $\mathcal{H}[\ell \mapsto e], \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} \ell \Downarrow w, \mathcal{H}'[\ell \mapsto w]$. By inversion of rule VAR$_{\Downarrow}$ we get $\ell \notin \mathcal{L}$ and

$$\mathcal{H}[\ell \mapsto e], \mathcal{S}, \mathcal{L} \cup \{\ell\} \vdash^{m}_{m'} e \Downarrow w, \mathcal{H}'[\ell \mapsto e] \tag{36}$$

The premise (33) is $(\widehat{\mathcal{H}}_0,\, \widehat{e}_0) \succcurlyeq (\mathcal{H}[\ell \mapsto e],\, \ell)$; by Lemma 2 this implies $\widehat{\mathcal{H}}_0 = \widehat{\mathcal{H}}[\ell \mapsto \widehat{e}]$ for some $\widehat{e}$; and by Lemmas 3 and 4 we get $\widehat{e}_0 = \ell$. Thus the premise instantiates in this case as:

$$(\widehat{\mathcal{H}}[\ell \mapsto \widehat{e}],\, \ell) \succcurlyeq (\mathcal{H}[\ell \mapsto e],\, \ell) \tag{37}$$

We can now apply induction to (36) and (37) and obtain

$$\widehat{\mathcal{H}}[\ell \mapsto \widehat{e}], \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} \widehat{e} \Downarrow^{\mathrm{I}} \widehat{w}, \widehat{\mathcal{H}}'[\ell \mapsto \widehat{e}] \tag{38}$$

$$(\widehat{\mathcal{H}}'[\ell \mapsto \widehat{e}],\, \widehat{\mathcal{H}}'[\ell \mapsto \widehat{e}]@\widehat{w}) \succcurlyeq (\mathcal{H}'[\ell \mapsto e],\, w) \tag{39}$$

Conclusion (39) fulfils proof obligation (35). Applying VAR$_{\Downarrow^{\mathrm{I}}}$ to (38) yields the remaining obligation (34). This concludes the proof of the VAR$_{\Downarrow}$ case.

Case LET$_{\Downarrow}$. The evaluation premise is $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{1+m}_{m'} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow w, \mathcal{H}'$. By inversion of rule LET$_{\Downarrow}$ we get for some fresh ℓ:

$$\mathcal{H}[\ell \mapsto e_1[\ell/x]], \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e_2[\ell/x] \Downarrow w, \mathcal{H}' \tag{40}$$

By Lemma 3, the premise (33) instantiates as

$$(\widehat{\mathcal{H}}, \mathsf{let}\ x = \widehat{e}_1\ \mathsf{in}\ \widehat{e}_2) \succcurlyeq (\mathcal{H}, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) \tag{41}$$

By Lemma 6 and (41) we get

$$(\widehat{\mathcal{H}}[\ell \mapsto \widehat{e}_1[\ell/x]],\, \widehat{e}_2[\ell/x]) \succcurlyeq (\mathcal{H}[\ell \mapsto e_1[\ell/x],\, e_2[\ell/x]) \tag{42}$$

By the definition of $\succcurlyeq$ (erasing the indirection $\ell' \mapsto \mathsf{ind}(\ell)$) it is immediate that:

$$\widehat{\mathcal{H}}[\ell \mapsto \widehat{e}_1[\ell'/x], \ell' \mapsto \mathsf{ind}(\ell)] \succcurlyeq \widehat{\mathcal{H}}[\ell \mapsto \widehat{e}_1[\ell/x]] \tag{43}$$

By transitivity of $\succcurlyeq$ (Lemma 1) and (43) plus (42) we get

$$\begin{aligned}&(\widehat{\mathcal{H}}[\ell \mapsto \widehat{e}_1[\ell'/x], \ell' \mapsto \mathsf{ind}(\ell)],\, \widehat{e}_2[\ell/x])\\ &\qquad \succcurlyeq (\mathcal{H}[\ell \mapsto e_1[\ell/x],\, e_2[\ell/x])\end{aligned}$$

which is the premise needed for applying induction to the evaluation (40). As a result of induction we get

$$\widehat{\mathcal{H}}[\ell \mapsto \widehat{e}_1[\ell'/x], \ell' \mapsto \mathsf{ind}(\ell)], \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} \widehat{e}_2[\ell/x] \Downarrow^{\mathrm{I}} \widehat{w}, \widehat{\mathcal{H}}' \tag{44}$$

$$(\widehat{\mathcal{H}}', \widehat{\mathcal{H}}'@\widehat{w}') \succcurlyeq (\mathcal{H}', w) \tag{45}$$

Applying rule $\textsc{Let}_{\Downarrow^{\mathrm{I}}}$ to (44) together with (45) yields the required conclusions.

6 Related work

This paper extends our previous work on type-based static analysis of resource bounds for lazy functional programs using amortisation [?]. Unlike that system, here we focus on co-recursive infinite data structures and show that a simpler type-and-effect system without amortisation suffices to obtain static resource bounds. As described in Section 4, this type-and-effect system successfully produces resource bounds for examples that could not previously be analysed.

Our cost model is based on Launchbury's natural semantics for lazy evaluation [?], as subsequently refined by Sestoft [?], de la Encina and Peña-Marì [?,?]. The proof technique used in Section 5.6 for establishing correspondence between the indirections semantics and the standard one is based on work by Sánchez-Gil, Hidalgo-Herrero and Ortega-Mallén [?]. The first work on cost analysis for lazy evaluation of higher-order functional programs was by Sands [?,?]. This used *evaluation contexts* [?] and *projections* [?] to capture the degree of evaluation of data structures. This was intended to aid manual reasoning about program costs but is not directly automatable for use in a compiler or static analysis tool.

Several authors have proposed *symbolic profiling* approaches, where programs are annotated with additional cost parameters. For example, Wadler [?] has used a state monad to count reduction costs through a tick-counting operation. Danielsson extends this work using a cost-annotated monad [?] that allows expressing machine-checkable complexity annotations through dependent types in the Agda programming language. Unlike the work presented here, this system allows checking but not automatic inference of complexity annotations.

Turner's *elementary strong functional programming* [?] explores issues of guaranteed termination in a purely functional programming language. Turner's approach separates inductive data structures from *co-data* structures such as streams. This ensures that functions on both finite and infinite structures are total by construction using only primitive recursive definitions. However, this work does not consider evaluation costs, and does not provide an analysis.

Hughes, Pareto and Sabry [?] describe a *sized type* system for a simple higher-order, non-strict functional language, that guarantees *termination* and *productivity* of recursive and co-recursive definitions. This work was subsequently developed to ensure bounded space usage in the strict functional language Embedded ML [?], which lacks co-recursion. Brady and Hammond [?] have also developed an embedding of sized types in a dependently typed framework. However, all three approaches require the programmer to provide explicit size information, that is checked rather than inferred. Finally, a combination of sized types with memory regions has been suggested by Peña and Segura [?], building on information provided by ancillary analyses on termination and safe destruction [?]. However, this does not deal with co-recursive costs.

7 Conclusions and further work

This paper presents a type-and-effect system for predicting upper-bounds on allocation costs for co-recursive definitions in a lazy functional language. The analysis is formally based on a standard operational semantics for lazy evaluation and we present a detailed proof sketch of soundness. We have also implemented this type system as a fully automatic static analysis. Initial experimental results show it can deal with non-trivial examples (the Fibonacci sequence and the Hamming problem). We are not aware of any previous automatic analysis that is capable of dealing with these examples.

A number of future research directions are left open by this work. For simplicity we presented a type system without either type polymorphism or effect polymorphism. This limits the compositionality of the analysis (cf. the duplication of definitions required in Hamming example of Section 4.3). Let-bound polymorphism could, in principle, be added simply by capturing constraints in type schemes as in [?,?,?]. It then remains an open question whether our system admits a notion of principal types schemes [?] (although this concerns only the completeness of the inference algorithm and not soundness).

Again for simplicity we chose a uniform cost model (each *let*-expression costs one unit). It should be straightforward to extend this to a more realistic cost model by allowing variable costs, derived from e.g. the STG abstract machine [?,?]. Another option would be to focus on resources other than heap, e.g. time or stack usage.

We have considered annotated types that express linear bounds for co-recursion, i.e. where the cost for each successive value is bounded by a constant. Hoffmann *et al.* have previously demonstrated successful extensions to multivariate polynomial bounds, in the context of amortised cost analysis for recursion [?]. It would be interesting to explore whether their techniques could also be applied in our work, to allow for non-linear costs with respect to evaluation depth.

Finally, the presented analysis and cost model do not yet consider *deallocation* of resources. In order to reason about memory residency in a type-based system, we would need, for example, to express deallocation using some syntax-directed primitives. It should be possible to extend our language and type-system with a deallocation primitive (e.g. the deallocating match in [?] or a region-based mechanism [?,?]) to accommodate this. This would pave the way for an intermediate language for compiling lazily evaluated programs with static residency guarantees.

References

1. Barwise, J., Moss, L.: Vicious Circles. CSLI Publications (1996)
2. Bird, R., Wadler, P.: Introduction to Functional Programming. Prentice Hall, New York (1988)
3. Coutts, D.: Stream Fusion: Practical shortcut fusion for coinductive sequence types. PhD thesis, Worcester College, University of Oxford (2010)
4. Simoes, H., Vasconcelos, P., Jost, S., Hammond, K., Florido, M.: Automatic amortised analysis of dynamic memory allocation for lazy functional programs. In: Proc. of ACM Intl. Conf. Func. Programming (ICFP'12), ACM (2012) 165–176
5. Sestoft, P.: Deriving a Lazy Abstract Machine. J. Functional Programming **7**(3) (1997) 231–264
6. Launchbury, J.: A Natural Semantics for Lazy Evaluation. In: Proc. POPL '93: Symp. on Princ. of Prog. Langs. (1993) 144–154
7. Jost, S., Loidl, H.W., Hammond, K., Scaife, N., Hofmann, M.: "Carbon Credits" for Resource-Bounded Computations Using Amortised Analysis. In: Proc. FM 2009: Intl. Conf. on Formal Methods, Springer LNCS 5850 (2009) 354–369
8. de la Encina, A., Peña-Marí, R.: Proving the Correctness of the STG Machine. In: Proc. IFL '01: Impl. of Functional Langs., Stockholm, Sweden, Sept. 24-26, 2001, Springer LNCS 2312 (2002) 88–104
9. Peyton Jones, S.L.: Implementing Lazy Functional Languages on Stock Hardware – the Spineless Tagless G-machine. J. Functional Programming **2**(2) (April 1992) 127–202
10. Sánchez-Gil, L., Hidalgo-Herrero, M., Ortega-Mallén, Y.: The role of indirections in lazy natural semantics. Technical Report TR-13-13, Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid (2013)
11. de la Encina, A., Peña-Marí, R.: 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, ACM (2003) 102–112
12. de la Encina, A., Peña-Marí, R.: From Natural Semantics to C: a Formal Derivation of two STG Machines. J. Funct. Program. **19**(1) (2009) 47–94
13. Sands, D.: Calculi for Time Analysis of Functional Programs. PhD thesis, Imperial College, University of London (September 1990)
14. Sands, D.: Complexity Analysis for a Lazy Higher-Order Language. In: Proc. ESOP '90: European Symposium on Programming, Copenhagen, Denmark. Springer LNCS 432 (1990) 361–376
15. Wadler, P.: Strictness Analysis aids Time Analysis. In: Proc. POPL '88: ACM Symp. on Princ. of Prog. Langs. (1988) 119–132
16. Wadler, P., Hughes, J.: Projections for Strictness Analysis. In: Proc. FPCA'87: Intl. Conf. on Functional Prog. Langs. and Comp. Arch. Springer LNCS 274 (September 1987) 385–407
17. Wadler, P.: The Essence of Functional Programming. In: Proc. POPL '92: ACM Symp. on Principles of Prog. Langs. (January 1992) 1–14
18. Danielsson, N.A.: 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, ACM (2008) 133–144
19. Turner, D.: Elementary Strong Functional Programming. In: Proc. 1995 Symp. on Funct. Prog. Langs. in Education — FPLE '95. LNCS, Springer-Verlag (December 1995)

20. Hughes, R., Pareto, L., Sabry, A.: Proving the Correctness of Reactive Systems Using Sized Types. In: ACM Symp. on Principles of Prog. Langs. (POPL'96), St. Petersburg Beach, USA, ACM (January 1996) 410–423
21. Hughes, R., Pareto, L.: Recursion and Dynamic Data Structures in Bounded Space: Towards Embedded ML Programming. In: Proc. 1999 ACM Intl. Conf. on Functional Programming (ICFP '99). (1999) 70–81
22. Brady, E., Hammond, K.: A Dependently Typed Framework for Static Analysis of Program Execution Costs. In: IFL, Springer-Verlag LNCS 4015 (2005) 74–90
23. Pena, R., Segura, C.: A First-Order Functl. Lang. for Reasoning about Heap Consumption. In: Draft Proc. Intl. Workshop on Impl. and Appl. of Functl. Langs. (IFL '04). (2004) 64–80
24. Montenegro, M., Pena, R., Segura, C.: An Inference Algorithm for Guaranteeing Safe Destruction. In: Draft Proc. Trends in Functional Programming (TFP07), New York, 2–4 April (2007)
25. Talpin, J.P., Jouvelot, P.: Polymorphic type, region and effect inference. J. Funct. Program. **2**(3) (1992) 245–271
26. Nielson, F., Nielson, H.R., Amtoft, T.: Polymorphic subtyping for effect analysis: The algorithm. In: Logical and Operational Methods in the Analysis of Programs and Systems. (1996) 207–243
27. Nielson, H.R., Nielson, F., Amtoft, T.: Polymorphic subtyping for effect analysis: The static semantics. In: Logical and Operational Methods in the Analysis of Programs and Systems. (1996) 141–171
28. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: ACM Symp. on Principles of Prog. Langs. (POPL'82). POPL '82, New York, NY, USA, ACM (1982) 207–212
29. Marlow, S., Jones, S.P.: Making a fast curry: push/enter vs. eval/apply for higher-order languages. In: Proc. of the ACM SIGPLAN 2004 Intl. Conf. on Functional Programming (ICFP'04). Volume 16., ACM Press (January 2004) 4–15
30. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate Amortized Resource Analysis. ACM Trans. Program. Lang. Syst. **34**(3) (November 2012) 14:1–14:62
31. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: ACM Symp. on Principles of Prog. Langs. (POPL'03), ACM (January 2003) 185–197
32. Tofte, M., Talpin, J.P.: Region-based memory management. Information and Computation **132**(2) (1997) 109–176
33. Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N., Olesen, T., Sestoft, P.: Programming with regions in the ml kit (April 2002) IT University of Copenhagen `http://www.itu.dk/research/mlkit/`.

A Appendix

A.1 Detailed Proof

Proof. The proof is by induction on the lengths of the derivations of evaluation (4) and typing (1) ordered lexicographically, with the former taking priority over the later.

Case Let. The typing premise (1) instantiates as

$$\mathit{\Gamma} \vdash^{1+p}_{p} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : C$$

By inversion of rule Let together with the substituition lemma, we get

$$\mathit{\Gamma}, \ell'{:}\mathsf{T}^0(A') \vdash^{q}_{0} e_1[\ell'/x] : A \tag{46}$$

$$\mathit{\Gamma}, \ell{:}\mathsf{T}^q(A) \vdash^{p}_{p'} e_2[\ell/x] : C \tag{47}$$

where $A' \lhd A$. The evaluation premise (4) instantiates as

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{1+m}_{m'} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow^{\mathrm{I}} w, \mathcal{H}'$$

from which we get $\mathcal{H}_0, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e_2[\ell/x] \Downarrow^{\mathrm{I}} w, \mathcal{H}'$ where $\mathcal{H}_0 = \mathcal{H}[\ell \mapsto e_1[\ell'/x], \ell' \mapsto \mathsf{ind}(\ell)]$. Define:

$$\begin{aligned} \mathcal{B}_0 &= \mathcal{B}[\ell \mapsto 0, \ell' \mapsto 0] \\ \mathcal{M}_0 &= \mathcal{M}[\ell \mapsto \mathsf{T}^q(A), \ell' \mapsto \mathsf{T}^0(A')] \\ \mathit{\Gamma}_0 &= \mathit{\Gamma},\ \ell{:}\mathsf{T}^q(A),\ \ell'{:}\mathsf{T}^0(A') \end{aligned}$$

To apply induction to the evaluation of $e_2[\ell/x]$ we first need to re-establish type consistency and compatibility. Type consistency for ℓ follows from (10) and (Loc2); and for ℓ' follows directly from the type rule Ind and (Loc1). Compatibility is immediate because the types for ℓ and ℓ' in $\mathit{\Gamma}_0$ are exactly $\mathcal{M}_0(\ell)$ and $\mathcal{M}_0(\ell')$. Applying induction to (11) then yields all required conclusions. Note that the lower thunk costs for A' are only allowed for the recursive reference ℓ' introduced in the let-expression; crucially this is sound only because the recursive reference is introduced in the heap as in indirection whose cost is ignored by the typing rule Ind. Otherwise compatibility would not hold.

Case Ind. This case is immediate: taking $\mathit{\Gamma}' = \mathit{\Gamma}$, $\mathcal{B}' = \mathcal{B}$, $\mathcal{M}' = \mathcal{M}$ and $m' = m$ yields all required conclusions.

Case Var. The typing premise is $\mathit{\Gamma}, \ell{:}\mathsf{T}^q(A) \vdash^{q}_{0} \ell : A$ and the evaluation premise is $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} \ell \Downarrow^{\mathrm{I}} w, \mathcal{H}'[\ell \mapsto w]$; by inversion of rule $\textsc{Var}_{\Downarrow^{\mathrm{I}}}$ we get $\mathcal{H}, \mathcal{S}, \mathcal{L} \cup \{\ell\} \vdash^{m}_{m'} \mathcal{H}(\ell) \Downarrow^{\mathrm{I}} w, \mathcal{H}'$ and $\ell \notin \mathcal{L}$. By the type compatibility hypothesis we get that

$$\mathcal{M}(\ell) <: \mathsf{T}^q(A)$$

We now distinguish the two applicable cases:

$\mathcal{H}(\ell)$ is in *whnf*. The evaluation succeeds immediately by either $\textsc{Whnf}_{\Downarrow^{\mathrm{I}}}$ and we have $w = \mathcal{H}(\ell)$, $\mathcal{H} = \mathcal{H}'$ and $m' = m$, i.e. the update is without effect. Taking $\Gamma' = \Gamma$, $\mathcal{B}' = \mathcal{B}$, $\mathcal{M}' = \mathcal{M}$. By type consistency, we get

$$\Gamma \vdash^{0}_{0} \mathcal{H}(\ell) : A$$

which is equivalent to the required conclusion

$$\Gamma' \vdash^{0}_{0} w : A$$

The remaining conclusions are immediate because $\mathcal{H}' = \mathcal{H}$.

$\mathcal{H}(\ell)$ is not in *whnf*. Let $\mathsf{T}^r(\widehat{A}) = \mathcal{M}(\ell)$. In this case, type consistency for ℓ requires ($\textsc{Loc}2$), which instantiates as

$$\Gamma, \ell{:}\mathsf{T}^q(A) \vdash^{r\,+\,\mathcal{B}(\ell)}_{0} \mathcal{H}(\ell) : \widehat{A}$$

Recall that from compatibility for location ℓ we get $\mathsf{T}^r(\widehat{A}) <: \mathsf{T}^q(A)$, which implies $\widehat{A} <: A$. By applying the type rule $\textsc{Subtype}$ we obtain

$$\Gamma, \ell{:}\mathsf{T}^q(A) \vdash^{r\,+\,\mathcal{B}(\ell)}_{0} \mathcal{H}(\ell) : A$$

By inversion of the evaluation premise we get

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \cup \{\ell\} \vdash^{m}_{m'} \mathcal{H}(\ell) \Downarrow^{\mathrm{I}} w, \mathcal{H}'$$

and $\ell \notin \mathcal{L}$. We now apply the induction hypothesis to the evaluation of $\mathcal{H}(\ell)$. Note that according to rule $\textsc{Var}_{\Downarrow^{\mathrm{I}}}$, we apply the induction hypothesis for $\mathcal{L}' = \mathcal{L} \cup \{\ell\}$. Observe that $m \geq t + p + \mathcal{B}(\ell) + \sum_{\mathcal{H},\mathcal{L}\cup\{\ell\}} \mathcal{B}$ holds as required. We thus obtain m', Γ', $\mathcal{B}'$ and $\mathcal{M}'$ such that

$$\Gamma' \vdash^{0}_{0} w : A \tag{48}$$

$$\Gamma', \mathcal{B}' \vdash_{\textsc{Mem}} (\mathcal{H}', \mathcal{L} \cup \{\ell\}) : \mathcal{M}' \tag{49}$$

$$\mathcal{M}' <: \Gamma' \tag{50}$$

$$m' \geq t + p' + \textstyle\sum_{\mathcal{H}',\mathcal{L}\cup\{\ell\}} \mathcal{B} \tag{51}$$

The only remaining proof obligations is to re-establish these statements for the updated heap $\mathcal{H}'' = \mathcal{H}'[\ell \mapsto w]$. In particular, we need

$$\Gamma', \mathcal{B}' \vdash_{\textsc{Mem}} (\mathcal{H}'', \mathcal{L}) : \mathcal{M}' \tag{52}$$

$$m' \geq t + p' + \textstyle\sum_{\mathcal{H}',\mathcal{L}\cup\{\ell\}} \mathcal{B} \tag{53}$$

The only location changed from (13) to (16) is ℓ were the applicable case changes from ($\textsc{Loc}2$) to ($\textsc{Loc}1$). But the latter is immediate from (12) because $\mathcal{H}''(\ell) = w$. Since the balance sum skips locations mapped to whnf, we have $\sum_{\mathcal{H}',\mathcal{L}\cup\{\ell\}} \mathcal{B} = \sum_{\mathcal{H}'',\mathcal{L}} \mathcal{B}$, thus establishing (17) as required.

Case App. The typing and evaluation premises in this case are

$$\Gamma, y{:}A \vdash^{p+q}_{p'} e\,y : C \tag{54}$$

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m''} (e\ \ell) \Downarrow^{\mathrm{I}} u, \mathcal{H}'' \tag{55}$$

By inversion of the type rule (App) applied to (18) we obtain

$$\Gamma, y{:}A \vdash^{p}_{p'} e : A \xrightarrow{q} C \tag{56}$$

By inversion of the evaluation rule $\textsc{App}_{\Downarrow^{\mathrm{I}}}$ applied to (19) we get

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e \Downarrow^{\mathrm{I}} u, \mathcal{H}' \tag{57}$$

$$\mathcal{H}'@u = \lambda x.e' \tag{58}$$

$$\mathcal{H}', \mathcal{S}, \mathcal{L} \vdash^{m'}_{m''} e'[\ell/x] \Downarrow^{\mathrm{I}} w, \mathcal{H}'' \tag{59}$$

Taking $t' = t + q$, we show that we verify the conditions for applying induction to the evaluation of e because

$$m \geq \underbrace{(t+q)}_{t'} + p + \textstyle\sum_{\mathcal{H},\mathcal{L}} \mathcal{B} \tag{60}$$

By induction we obtain $\Gamma', \mathcal{B}', \mathcal{M}'$ such that

$$\Gamma' \vdash^{0}_{0} u : A \xrightarrow{q} C \tag{61}$$

$$\Gamma', \mathcal{B}' \vdash_{\textsc{Mem}} (\mathcal{H}', \mathcal{L}) : \mathcal{M}' \tag{62}$$

$$\mathcal{M}' <: \Gamma' \tag{63}$$

$$\begin{aligned} m' &\geq (t+q) + p' + \textstyle\sum_{\mathcal{H}',\mathcal{L}} \mathcal{B}' \\ &= \underbrace{(t+p')}_{t''} + q + \textstyle\sum_{\mathcal{H}',\mathcal{L}} \mathcal{B}' \end{aligned} \tag{64}$$

By $\mathcal{H}'@u = \lambda x.\, e'$ we either have $u = \lambda x.\, e'$ or $u = \mathsf{ind}(\kappa)$. In either case the type remains unchanged, due to rule Ind and due to $\lhd$ not altering function types, we thus get

$$\Gamma' \vdash^{0}_{0} \lambda x.e' : A \xrightarrow{q} C$$

Using a standard Abs inversion lemma, we get

$$\Gamma', x{:}A \vdash^{q}_{0} e' : C$$

By the substitution lemma we get

$$\Gamma', \ell{:}A \vdash^{q}_{0} e'[\ell/x] : C$$

We can now apply induction again to (23) (evaluation of $e'[\ell/x]$) and obtain $m'', \Gamma'', \mathcal{M}'', \mathcal{B}''$ satisfying all desired conclusions:

$$\Gamma'' \vdash^{0}_{0} w : C \tag{65}$$

$$\Gamma'', \mathcal{B}'' \vdash_{\textsc{Mem}} (\mathcal{H}'', \mathcal{L}) : \mathcal{M}'' \tag{66}$$

$$\mathcal{M}'' <: \Gamma'' \tag{67}$$

$$\begin{aligned} m'' &\geq (t+p') + 0 + \textstyle\sum_{\mathcal{H}'',\mathcal{L}} \mathcal{B}'' \\ &= t + p' + \textstyle\sum_{\mathcal{H}'',\mathcal{L}} \mathcal{B}'' \end{aligned} \tag{68}$$

Case Prepay. The typing premise is

$$\Gamma,\ \ell{:}\mathsf{T}^{q_0+q_1}(A)\ \vdash^{p\,+\,q_1}_{p'}\ e : C \tag{69}$$

$$m \geq t + (p + q_1) + \textstyle\sum_{\mathcal{H},\mathcal{L}} \mathcal{B} \tag{70}$$

By inversion of rule Prepay we get

$$\Gamma,\ \ell{:}\mathsf{T}^{q_0}(A)\ \vdash^{p}_{p'}\ e : C \tag{71}$$

Assume that $\mathcal{H}(\ell)$ is not in *whnf*, for otherwise the case is trivial (including indirections). By type consistency (2) and compatibility (3) there exists $\mathcal{M}(\ell) = \mathsf{T}^q(\widehat{A})$ such that $\widehat{A} <: A$ and $q \leq q_0 + q_1$. Let $r = \max(q - q_1, 0)$, $\mathcal{M}' = \mathcal{M}[\ell \mapsto \mathsf{T}^r(\widehat{A})]$ and $\mathcal{B}' = \mathcal{B}[\ell \mapsto q_1 + \mathcal{B}(\ell)]$. In order to apply induction we now re-establish the theorem's premises for these updated mappings. To establish type consistency for location ℓ by (Loc2) we then need to show that $\Gamma, \ell{:}\mathsf{T}^{q_0}(A) \vdash^{r\,+\,\mathcal{B}'(\ell)}_{0} \mathcal{H}(\ell) : \widehat{A}$ holds. Originally, type consistency (2) yields

$$\Gamma, \ell{:}\mathsf{T}^{q_0+q_1}(A)\ \vdash^{q\,+\,\mathcal{B}(\ell)}_{0}\ \mathcal{H}(\ell) : \widehat{A}$$

by rule Supertype we have

$$\Gamma, \ell{:}\mathsf{T}^{q_0}(A)\ \vdash^{q\,+\,\mathcal{B}(\ell)}_{0}\ \mathcal{H}(\ell) : \widehat{A}$$

By Definition of r we have $r \geq q - q_1$ which is equivalent to $r + q_1 \geq q$ and thus by rule Relax we deduce

$$\Gamma, \ell{:}\mathsf{T}^{q_0}(A)\ \vdash^{r\,+\,q_1\,+\,\mathcal{B}(\ell)}_{0}\ \mathcal{H}(\ell) : \widehat{A}$$

which is equivalent by our Definition of $\mathcal{B}'$ above to

$$\Gamma, \ell{:}\mathsf{T}^{q_0}(A)\ \vdash^{r\,+\,\mathcal{B}'(\ell)}_{0}\ \mathcal{H}(\ell) : \widehat{A}$$

We also need to re-establish type compatibility $\mathcal{M}' <: \Gamma, \ell{:}\mathsf{T}^{q_0}(A)$ We originally have

$$\mathcal{M} <: \Gamma, \ell{:}\mathsf{T}^{q_0+q_1}(A)$$

For location ℓ this gives

$$\mathcal{M}(\ell) <: \mathsf{T}^{q_0+q_1}(A)$$

We have $\mathcal{M}(\ell) = \mathsf{T}^q(\widehat{A})$; by the definition of subtyping this gives $q \leq q_0 + q_1$ which is equivalent to $q - q_1 \leq q_0$; by non-negativity of annotations we have $0 \leq q_0$. Combining both inequalities gives $\max(q - q_1, 0) \leq q_0$ hence $r \leq q_0$ which establish the subtyping

$$\mathcal{M}'(\ell) <: \mathsf{T}^{q_0}(A)$$

This establishes the compatibility for ℓ; the types for other locations are unchanged and thus we get

$$\mathcal{M}' <: \mathit{\Gamma}, \ell{:}\mathsf{T}^{q_0}(A)$$

It remains to show that $m \geq t + p + \sum_{\mathcal{H},\mathcal{L}} \mathcal{B}'$ holds. Originally, we had

$$m \geq t + (p + q_1) + \textstyle\sum_{\mathcal{H},\mathcal{L}} \mathcal{B} \tag{72}$$

which is equivalent to the desired inequality since $q_1 + \sum_{\mathcal{H},\mathcal{L}} \mathcal{B} = \sum_{\mathcal{H},\mathcal{L}} \mathcal{B}'$ by the definition of $\mathcal{B}'$. Applying induction yields all required results for this case.