Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Type-Based Cost Analysis for Lazy Functional Languages

Steffen Jost · Pedro Vasconcelos · Mário Florido · Kevin Hammond

Accepted: 2016

Abstract We present a static analysis for determining the execution costs of lazily evaluated functional languages, such as Haskell. Time- and space-behaviour of lazy functional languages can be hard to predict, creating a significant barrier to their broader acceptance. This paper applies a type-based analysis employing *amortisation* and *cost effects* to statically determine upper bounds on evaluation costs. While amortisation performs well with finite recursive data, we significantly improve the precision of our analysis for co-recursive programs (i.e. dealing with potentially infinite data structures) by tracking self-references. Combining these two approaches gives a fully automatic static analysis for both recursive and co-recursive definitions. The analysis is formally proven correct against an operational semantic that features an exchangeable parametric cost-model. An arbitrary measure can be assigned to all syntactic constructs, allowing to bound, for example, evaluation steps, applications, allocations, etc. Moreover, automatic inference only relies on first-order unification and standard linear programming solving. Our publicly available implementation demonstrates the practicability of our technique on editable non-trivial examples.

Keywords Automated Static Analysis · Lazy Evaluation · Corecursion · Amortised Analysis · Type Systems · Functional Programming

Author's version. The final publication is available at Springer via `http://dx.doi.org/10.1007/s10817-016-9398-9`.

This work has been partially supported by EU grants 288570 (ParaPhrase) and 644235 (RePhrase) under the Seventh Framework and Horizon 2020 Programmes.

S. Jost
LMU Munich, Munich, Germany

P. Vasconcelos · M. Florido
LIACC, DCC/Faculdade de Ciências, Universidade do Porto, Porto, Portugal

K. Hammond
University of St Andrews, St Andrews, UK

1 Introduction

While offering important benefits for modularity, abstraction and composability [9], a key obstacle to the broad adoption of non-strict functional programming languages, such as Haskell [18], is the difficulty in predicting operational properties, such as execution time or space usage. Non-strict semantics is typically implemented by *lazy evaluation*: delaying evaluation of sub-expressions until they are actually needed for the overall result. This allows defining structures that are potentially infinite, but for which only a part is needed to determine the final program result. Furthermore, if an expression is needed more than once, the result value from the first evaluation can be reused instead of re-evaluating the expression (implementing *graph reduction*). The disadvantage is that reasoning about evaluation costs becomes *less* compositional than in the usual eager setting. In the latter, the costs of composition can be approximated by adding up the costs of all individual components. For lazy evaluation, however, simply adding together the separate costs of producing and consuming lazy data structures would give a gross overestimation of costs and often fail to give a finite bound.

This paper describes a type-based approach for obtaining static cost bounds for lazily evaluated functional programs. The costs of individual program fragments are expressed by type annotations derived by an augmented type system. We show that this system can infer costs of recursive and co-recursive definitions that are linear on the number of of constructors in input or output, given assumptions for the costs of primitives and free variables. The analysis presented here combines two previous independent analyses that respectively considered the allocation costs of recursive [24] and co-recursive programs [29]. The significant simplification of the soundness invariants and proof, presented here for the first time, enabled a smooth combination of the two analyses into a single coherent whole. Furthermore, we generalize the combined analysis to a parametric cost model that allows bounds to be determined for any syntactically derivable measure, e.g. evaluation steps, function calls, allocations, etc.

This paper is structured as follows: Section 2 introduces the analysis by presenting cost bound results for simple examples; Section 3 defines our language and operational semantics; Section 4 presents type rules for deriving annotated types and discusses the prototype implementation; Section 5 provides key invariants and a detailed proof sketch of soundness; Section 6 describes an improvement for co-recursive programs using a technique developed previously [29]; Section 8 concludes and provides some directions for further work.

2 Motivating examples

In order to give an overview of the the analysis, we begin by presenting some simple example programs with cost bounds inferred for them. For ease of understanding, we use Haskell syntax for the example code; the translation into our intermediate term syntax (shown later) is mechanical. We provide the annotated typings produced by our prototype implementation verbatim and discuss the obtained bounds informally.

Infinite lists Consider two Haskell definitions for a function generating an infinite list of identical values:

```
repeat  x = let xs = x:xs in xs
repeat' x = x : repeat' x
```

The two definitions yield identical streams of values; the first one, however, is more efficient: `repeat` creates a single cyclic heap node, while `repeat'` will allocate many (identical) nodes as the result stream is traversed. Hence, `repeat` has constant cost while `repeat'` has a cost linear in the number of elements demanded from the result. This difference in cost is not immediately apparent, as pointed out in, e.g. [2]. We can observe this difference through the annotated types for *allocation costs* as inferred by our prototype analysis:

```
repeat  : T(a) ->@1 Rec{Cons:(T(a),T(#)) | Nil:()}
repeat' : T(b) ->@2 Rec{Cons:(T(b),T@2(#)) | Nil:()}
```

Some remarks on the output:

- recursive types `Rec{...}` are written structurally, i.e. listing all constructors and using `#` for recursive references;
- arguments to functions and constructors are wrapped in *thunk types* `T` representing possibly delayed evaluations;
- functions and thunk types are annotated with *evaluation costs* marked by `@`; for readability, zeros annotations are omitted.

The type of `repeat` is annotated with cost of 1 on the function (for allocating a single node); the result infinite list incurs no further costs (both head and tail thunks are annotated with zeros). The type of `repeat'`, however, shows that evaluation costs 2 for the first application plus 2 for every tail element that is accessed later on.[1] We can therefore read the cost bounds: for some `x`, evaluating n elements of `repeat x` has constant cost $1+0\cdot n=1$ whereas evaluating n elements of `repeat' x` costs $2+2\cdot n$.

Folding lists The next example uses the standard recursive definition of a higher-order fold function on lists to sum a list of integers:

```
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
sum = foldl (+) 0
```

In a cost model where every reduction step (and not just allocations) costs one unit, both `foldl` and also `sum` should exhibit linear cost on the size of the input list. Our analysis offers the following annotated types in this case:[2]

```
foldl : T(T(Int) -> T(Int) ->@1 Int) ->
          T(Int) -> T(Rec{Cons@11:(T(Int),T(#)) | Nil@1:()}) ->@2 Int
sum   : T(Rec{Cons@11:(T(Int),T(#)) | Nil:()}) ->@7 Int
```

The type of `foldl` is annotated with a positive cost on the outermost arrow. This indicates that fully applying this function has cost upper-bound of 2. Applying just the first two arguments incurs no cost (we simply write `->` instead of `->@0`). This is simply because the curried function requires the three arguments.

Furthermore, the third argument has `T@0` everywhere (again `@0` is omitted), which means that the evaluation of the list and each element within costs nothing, i.e. for this function type to be used, the list and its elements should be cost-free for the underlying metric (e.g. are already fully evaluated). However, unlike the previous example, `Cons` and `Nil` are annotated

[1] The extra cost per node accurately models the operational behaviour, but this is only visible at the lower-level of the intermediate notation (cf. Sect. 3).

[2] Other annotated types are possible. Our whole-program analysis only considers the use of `foldl` within `sum` here, leading to the shown annotated type.

with a positive cost that covers the processing of each constructor within that list. Since a finite list of length n has n `Cons` and a single `Nil`, this means that the cost of applying `foldl` is $11 \cdot n + 1 \cdot 1 + 2$. Thus the evaluation cost for applying `foldl` to a list of length n is $3 + 11n$. For `sum` we get the same linear coefficient but a higher a constant: $7 + 11n$ for a list of n elements; this is expected since `sum` is just a wrapper over the recursive worker `foldl`.

List fusion The next example illustrates how our analysis exposes the cost reduction of an optimization using the law $map\ f \circ map\ g = map\ (f \circ g)$. Replacing the left-hand side for the right-hand one results in an equivalent program that performs fewer allocations by avoiding the construction of an intermediate list. We define each side as separate functions for analysis:

```
lhs f g xs = map f (map g xs)
rhs f g xs = map (\x -> f (g x)) xs
```

The types inferred are as follows:

```
lhs : T(T(a) -> b) -> T(T(c) -> a) ->
          T(Rec{Cons@6:(T(c),T(#)) | Nil:()}) ->@5
              Rec{Cons:(T(b),T(#)) | Nil:()}
rhs : T(T(a) -> b) -> T(T(c) -> a) ->
          T(Rec{Cons@4:(T(c),T(#)) | Nil:()}) ->@2
              Rec{Cons:(T(b),T(#)) | Nil:()}
```

It is immediate from the bounds $5 + 6n$ for `lhs` and $2 + 4n$ for `rhs` that the latter has lower coefficients and thus consumes fewer resources for all list lengths n. Note that comparing the *total number of allocations* (as is done here) is actually desirable for analysing such optimizations [3] because the intermediate list is could immediately be deallocated as results are consumed, and thus the residency of the two programs could be identical.

Finally, we remark that we have shown only one of many admissible annotated types in the examples above. For example, while the total cost remains the same, an arbitrary part of the cost may be shifted to the result type, as shown by this alternative typing:

```
lhs : T(T(a) -> b) -> T(T(c) -> a) ->
          T(Rec{Cons@3:(T(c),T(#)) | Nil:()}) ->@5
              Rec{Cons:(T(b),T@3(#)) | Nil:()}
```

3 Language and cost semantics

We consider syntactical terms e for *initial expressions* (the λ-calculus extended with local bindings and pattern matching); *full expressions* $\widehat{e}$; and *weak head normal forms* w:[3]

$$
\begin{array}{rcl}
e & ::= & x \mid \lambda x.e \mid e\ x \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{letcons}\ x = c(\mathbf{y})\ \mathsf{in}\ e \\
& \mid & \mathsf{match}\ e_0\ \mathsf{with}\ \{c_1(\mathbf{x}_1)\texttt{->}e_1 \mid \cdots \mid c_n(\mathbf{x}_n)\texttt{->}e_n\} \\
\widehat{e} & ::= & e \mid c(\mathbf{x}) \\
w & ::= & \lambda x.e \mid c(\mathbf{x})
\end{array}
$$

[3] Boldface symbols denote possibly-empty sequences of the underlying syntactic categories, e.g. $\mathbf{x}$ and $\mathbf{y}$ are sequences of variables.

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

$$\frac{\ell \notin \mathcal{L} \qquad \mathcal{H}[\ell \mapsto \widehat{e}],\mathcal{S},\mathcal{L} \cup \{\ell\} \vdash^{m}_{m'} \widehat{e} \Downarrow w,\mathcal{H}'[\ell \mapsto \widehat{e}]}{\mathcal{H}[\ell \mapsto \widehat{e}],\mathcal{S},\mathcal{L} \vdash^{m+\mathtt{Kvar}}_{m'} \ell \Downarrow w,\mathcal{H}'[\ell \mapsto w]} \quad (\textsc{Var}_{\Downarrow})$$

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

$$\frac{\ell \text{ is fresh} \qquad \mathcal{H}[\ell \mapsto c(\mathbf{y}[\ell/x])],\mathcal{S},\mathcal{L} \vdash^{m}_{m'} e[\ell/x] \Downarrow w,\mathcal{H}'}{\mathcal{H},\mathcal{S},\mathcal{L} \vdash^{m+\mathtt{Kletcons}}_{m'} \mathsf{letcons}\ x = c(\mathbf{y})\ \mathsf{in}\ e \Downarrow w,\mathcal{H}'} \quad (\textsc{Letcons}_{\Downarrow})$$

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

$$\frac{\begin{array}{c}\mathcal{H},\mathcal{S} \cup (\bigcup_{i=1}^{n}\{\mathbf{x}_i\} \cup \mathrm{BV}(e_i)),\mathcal{L} \vdash^{m}_{m'} e_0 \Downarrow c_k(\boldsymbol{\ell}),\mathcal{H}' \\ \mathcal{H}',\mathcal{S},\mathcal{L} \vdash^{m'}_{m''} e_k[\boldsymbol{\ell}/\mathbf{x}_k] \Downarrow w,\mathcal{H}''\end{array}}{\mathcal{H},\mathcal{S},\mathcal{L} \vdash^{m+\mathtt{Kmatch}}_{m''} \mathsf{match}\ e_0\ \mathsf{with}\ \{c_i(\mathbf{x}_i)\text{->}e_i\}_{i=1}^{n} \Downarrow w,\mathcal{H}''} \quad (\textsc{Match}_{\Downarrow})$$

Fig. 1 Instrumented operational semantics

Our cost model is based on Sestoft's revision [23] of Launchbury's operational semantics for lazy evaluation [15], since Launchbury's semantics forms one of the earliest and most widely-used operational accounts of lazy evaluation for the λ-calculus. The main change in our presentation is a separate letcons-expression, using a similar notation as La Encina and Peña [14]; this is done to easily distinguish *allocation* from simply *referencing* constructors.

As in Launchbury's semantics, arguments of function and constructor applications must be variables. When necessary, complex arguments can be explicitly named using let-bindings; as in e.g. the STG machine [17], this restriction is required to make in-place update and sharing of results explicit.

Let-expressions bind variables to possibly recursive terms. For simplicity, we consider only single-variable bindings: multiple bindings can be encoded, if needed, using pairs and projections.[4]

Note that constructor applications $c(\mathbf{x})$ are *not* initial expressions; instead they are introduced through evaluation of a specialized letcons-expression. The operational semantics is defined for *full expressions* $\widehat{e}$ (or simply *expressions*), that are either initial expressions or constructor applications. The result of evaluation is an expression w in *weak head normal form* (*whnf*), i.e. it is a λ-abstraction or a constructor application.

3.1 Operational semantics

Figure 1 defines an operational semantics for lazy evaluation instrumented with a simple cost counting mechanism, against which we will define and prove our cost analysis. The rules define an evaluation relation $\mathcal{H},\mathcal{S},\mathcal{L} \vdash^{m}_{m'} \widehat{e} \Downarrow w,\mathcal{H}'$, where $\widehat{e}$ is a full expression, w is the evaluation result (in *weak head normal form*); $\mathcal{H}$ and $\mathcal{H}'$ are the initial and final *heaps*,

[4] Depending on the cost model, the encoding could incur some additional operational costs, but these would also be reflected in the analysis output.

i.e. mappings from variables to possibly-unevaluated expressions (*thunks*):

$$\mathcal{H} \quad ::= \quad \emptyset \quad | \quad \mathcal{H}[x \mapsto \widehat{e}]$$

We assume that heaps assign variables at most once, i.e. the notation $\mathcal{H}[x \mapsto \widehat{e}]$ implies that $(x \mapsto \widehat{e}') \notin \mathcal{H}$ for all $\widehat{e}'$. This allows writing heaps as sets without ambiguity, i.e. we consider $\mathcal{H}[x \mapsto \widehat{e}, y \mapsto \widehat{e}']$ and $\mathcal{H}[y \mapsto \widehat{e}', x \mapsto \widehat{e}]$ (where $x \neq y$) as equivalent.

Symbols ℓ, ℓ' (designated *locations*) denote variables introduced by the evaluation of let- and letcons-expressions. The set $\mathcal{L}$ contains locations of heap expressions ("thunks") that are under evaluation and is used to prevent cyclic evaluation (similar to the "black-hole" technique used by Launchbury).

Following Sestoft, we also keep track of bound variables throughout evaluation in a set $\mathcal{S}$ to prevent variable capture; this is used in the *freshness condition* (to be made precise later) used in rules $\text{LET}_{\Downarrow}$ and $\text{LETCONS}_{\Downarrow}$. We use $\text{BV}(\widehat{e})$ to denote the set of variables bound by lambda, let or match expressions in $\widehat{e}$; similarly, $\text{FV}(\widehat{e})$ denotes the set of free (i.e. unbound) variables in $\widehat{e}$.

Finally, parameters m, m' are non-negative integers representing the available resources before and after evaluation; thus, the difference $m - m'$ is the net evaluation cost. The use of two annotations instead of a single one (the net cost) simplifies the threading of resources in composite statements; furthermore, it should also simplify extensions for resource deallocation (cf. future work in Sect. 8).

Our semantics is parametrized by constants `Kvar`, `Kapp`, `Klet`, `Kletcons` and `Kmatch`, representing the cost assigned to each reduction rule. Specific evaluation rules require that enough resources are available by constraining the annotations on the turnstile, e.g. $\vdash \frac{m+\texttt{Kvar}}{m'}$ requires at least `Kvar` resources. Different instantiations of these constants allow modeling different costs, for example:

- setting all constants to one measures the number of evaluation steps;
- setting $\texttt{Klet} = \texttt{Kletcons} = 1$ and all other constants to zero measures the total number of constructors and thunks allocated;
- setting $\texttt{Kapp} = 1$ and all other constants to zero measures the number of applications.

The purpose of the analysis of Section 4 is to obtain static approximations for m and 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 $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash e \Downarrow w, \mathcal{H}'$ instead of $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e \Downarrow w, \mathcal{H}'$.

For the sake of completeness, we state an auxiliary definition, due to La Encina and Peña [14], formalising the notion of variable freshness.

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

Discussion of the evaluation rules $\text{LET}_{\Downarrow}$ and $\text{LETCONS}_{\Downarrow}$ are the only rules that augment the heap with a new expressions bound to a "fresh" location.

The $\text{WHNF}_{\Downarrow}$ rule for weak-head normal forms (λ-expressions and constructors) incurs no cost. The $\text{VAR}_{\Downarrow}$ and $\text{APP}_{\Downarrow}$ rules are identical to the equivalent ones in Launchbury's semantics. The $\text{VAR}_{\Downarrow}$ rule is restricted to locations that are not marked as being under evaluation (enforcing "black-holing" that explicitly excludes some non-terminating evaluations).

The $\text{MATCH}_{\Downarrow}$ 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 $\textsc{Let}_{\Downarrow}$ and $\textsc{LetCons}_{\Downarrow}$ rules.

Note that evaluating $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ does *not* evaluate e_1; instead it allocates a thunk and proceeds the evaluation of the body e_2. Thus, the cost of the let-expression as a whole is that that of e_2 plus a fixed constant `Klet`. The only rules that evaluate two expressions are $\textsc{App}_{\Downarrow}$ and $\textsc{Match}_{\Downarrow}$; in both cases the evaluation costs are threaded through the annotations m, m', m'' in the judgments with a suitable constant `Kapp` or `Kmatch` added.

4 Type-based analysis

Our type-based analysis combines an *effect system* [25,20] for higher-order functions and delayed computations and *amortisation* [26,16] for recursive functions. Amortised bounds for recursive functions are obtained by associating *potential* (i.e. a non-negative number) to data structures. The key objective is to choose the potential assignment so that it simplifies the amortised costs, e.g. so that the change in potential by evaluation offsets any variability in actual costs, thus making the amortised cost constant.

We assign potential to data structures using type annotations: the constructors of (recursive) data types are annotated with positive coefficients that specify the contribution of each constructor to the overall potential for that data structure. Values of function types never have any potential assigned to themselves. The annotations contained within argument and result types indicate only how potential is spent and transformed by applying the function to the data structures that carry potential. An *affine type system* [19] then ensures the crucial soundness property that potential is used at most once. The principal advantage of our type-based approach is that we can use efficient linear constraint solvers to determine suitable type annotations, thus automatically inferring the potential function and hence the amortised bounds.

4.1 Annotated types

The syntax of annotated types includes type variables, function types, thunk types and possibly-recursive algebraic data types.

$$A \quad ::= \quad X \quad | \quad A \xrightarrow{p} B \quad | \quad \mathsf{T}^p(A) \quad | \quad \mu X.\{\, c_1 : (q_1, \mathbf{A}_1) \mid \cdots \mid c_n : (q_n, \mathbf{A}_n) \,\}$$

Meta-variables A, B, C stand for types, X, Y for type variables, p, q for *cost annotations* (which are non-negative rational numbers) and n a non-negative integer. A vector of (possibly zero) types is denoted by $\mathbf{A}$. Function types $A \xrightarrow{p} B$ are annotated with an upper bound p on the cost of evaluating the function application. Note that type variables are used solely for denoting recursive references in data types; in particular, the type system is not polymorphic; this is further clarified at the end of Section 4.3.

Thunk types $\mathsf{T}^p(A)$ denote delayed computations yielding a value of type A and are similarly annotated with an upper bound p on the cost of evaluation to weak head normal form. For example: assuming a suitable type N for natural numbers, then $\mathsf{T}^1(N)$ is the type of a thunk yielding a natural number whose evaluation costs one unit; and $\mathsf{T}^0(N)$ is the type of a thunk whose evaluation is free; for the step-counting cost model, this means that the thunk is in weak-head normal form.

Note that annotations p and q in, e.g., $A \xrightarrow{p} B$ and $A \xrightarrow{0} \mathsf{T}^q(B)$, are handled differently in our type rules; this is in order to model the behaviour of lazy evaluation: the cost of using a thunk can be shared (because of memoization), whereas the cost of an application must be paid for each use.

Algebraic data types $\mu X.\{\, c_1 : (q_1, \mathbf{A}_1) \mid \cdots \mid c_n : (q_n, \mathbf{A}_n) \,\}$ are (possibly recursive) labelled sum of products $\mathbf{A}_i$ where each alternative is identified by a constructor c_i and annotated with a *potential* q_i; these annotations are used to justify amortised cost bounds for recursive functions. For example, suitable types for pairs, sums, natural numbers and lists are as follows:

$$A \times B = \mu X.\{\, \mathsf{pair} : (q, (A,B)) \,\} \qquad N = \mu X.\{\, \mathsf{zero} : (q_z, ()) \mid \mathsf{succ} : (q_s, X) \,\}$$
$$A + B = \mu X.\{\, \mathsf{inl} : (q_l, A) \mid \mathsf{inr} : (q_r, B) \,\} \qquad L(A) = \mu X.\{\, \mathsf{nil} : (q_n, ()) \mid \mathsf{cons} : (q_c, (A, X)) \,\}$$

The annotations q, q_l, q_r, etc. in the above types are *not* fixed; instead they are parameters that, for specific programs, convey the cost assignment to data structures. For example, a function with argument type $\mu X.\{\, \mathsf{nil} : (q_{nil}, ()) \mid \mathsf{cons} : (q_{cons}, (A, X)) \,\}$ admits a cost bound $q_{nil} + n \times q_{cons} + c$ where n is the length of the argument list; the fixed cost c is determined by other annotations (e.g. in arrow and thunk type). The type system in the Section 4.3 will enforce sound derivations of such annotated types.

4.2 Sharing and subtyping

Figure 2 shows the syntactical rules for an auxiliary *sharing relation* $A \curlyvee \{B_1, \ldots, B_n\}$ between a type A and a finite multiset of types $\{B_1, \ldots, B_n\}$ that is used to limit contraction in our type system. Informally, sharing allows distributing the potential in A among $B_1, \ldots, B_n$, while preserving cost annotations of functions and thunks. Sharing also allows the relaxation of annotations to subsume subtyping (i.e. potential can decrease while cost may increase). This relation is used in side conditions to the type rules to constrain types and contexts and in the soundness proof in Section 5.

The SHAREEMPTY and SHAREVEC rules are trivial. Rule SHAREVAR allows the free duplication of a type variable. This simplifies the formulation of the rule SHAREDAT for algebraic data types. Note that this could not be allowed if variables could be instantiated with arbitrary types.[5] However, in our case this is unproblematic because type variables are not quantified.

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 cost for functions and thunks, respectively, to be replicated. Rules SHAREEMPTYCTX and SHARECTX extend sharing to a binary relation between typing contexts: a context Γ shares to another context Δ if, for each variable x the types of x in Γ share to the types of x in Δ.

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 $A \curlyvee \{B\}$. This relation expresses the relaxation of potentials and costs: $A <: B$ 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 $A \curlyvee \{A, A\}$ (respectively $\Gamma \curlyvee \{\Gamma, \Gamma\}$) require that the potential of a value of type A be zero

[5] A polymorphic system would need to track sharing constraints for type variables alongside the types.

$$\frac{}{A \curlyvee \emptyset} \qquad \text{(SHAREEMPTY)}$$

$$\frac{}{X \curlyvee \{X, \ldots, X\}} \qquad \text{(SHAREVAR)}$$

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

$$\frac{A_i \curlyvee \{A\} \qquad B \curlyvee \{B_i\} \qquad q_i \geq p \qquad (1 \leq i \leq n)}{(A \xrightarrow{p} B) \curlyvee \{A_1 \xrightarrow{q_1} B_1, \ldots, A_n \xrightarrow{q_n} B_n\}} \qquad \text{(SHAREFUN)}$$

$$\frac{A \curlyvee \{A_1, \ldots, A_n\} \qquad q_i \geq p \qquad (1 \leq i \leq n)}{\mathsf{T}^p(A) \curlyvee \{\mathsf{T}^{q_1}(A_1), \ldots, \mathsf{T}^{q_n}(A_n)\}} \qquad \text{(SHARETHUNK)}$$

$$\frac{A_j \curlyvee \{B_{1j}, \ldots, B_{nj}\} \qquad m = |\mathbf{A}| = |\mathbf{B_i}| \qquad (1 \leq i \leq n,\ 1 \leq j \leq m)}{\mathbf{A} \curlyvee \{\mathbf{B}_1, \ldots, \mathbf{B}_n\}} \qquad \text{(SHAREVEC)}$$

$$\frac{}{\Gamma \curlyvee \emptyset} \qquad \text{(SHAREEMPTYCTX)}$$

$$\frac{A \curlyvee \{B_1, \ldots, B_n\} \qquad \Gamma \curlyvee \Delta}{(x : A, \Gamma) \curlyvee (x : B_1, \ldots, x : B_n, \Delta)} \qquad \text{(SHARECTX)}$$

Fig. 2 Sharing relation

(respectively, for values of types in Γ). The variant $A \curlyvee \{A, A'\}$ requires that A' is a subtype of A with no potential. We conclude this subsection by a simple observation on sharing:

Lemma 1 *Given a type A, we can always find A' such that $A \curlyvee \{A, A'\}$.*

Proof By induction on the structure of the type A and the rules of Figure 2: if A is a function type then we can choose $A' = A$; if A is a thunk type $\mathsf{T}^q(B)$ then choose $A' = \mathsf{T}^q(B')$ such that $B \curlyvee \{B, B'\}$; finally, if A is a recursive data types we can choose A' with identical structure and zero potential annotations and with sharing arguments.

4.3 Typing rules

Our analysis is presented in Figures 3 and 4 as a proof system that derives judgements of the form $\Gamma \vdash^{p}_{p'} \widehat{e} : A$, where Γ is a typing context, which is a multimap from variables to types[6], $\widehat{e}$ is a full expression, A is an annotated type and p, p' are non-negative rational 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. The type rules use the lower annotations in the turnstile for threading the amount of available resources through sub-evaluations in rules APP and MATCH. Because our semantics does not deallocate resources, we do not need lower annotations for left-over or freed resources on function and thunk types.

[6] 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 multiset union. Note that contexts are multimaps, as usual in an affine type system, in order to track each use of a variable.

$$\frac{}{x{:}\mathsf{T}^q(A) \vdash^{q+\mathtt{Kvar}}_{0} x : A} \quad (\textsc{Var})$$

$$\frac{\Gamma, x{:}\mathsf{T}^q(A') \vdash^{q}_{0} e_1 : A \qquad \Delta, x{:}\mathsf{T}^q(A) \vdash^{p}_{p'} e_2 : C \qquad x \notin \Gamma,\Delta \qquad A \between \{A,A'\}}{\Gamma, \Delta \vdash^{p+\mathtt{Klet}}_{p'} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : C} \quad (\textsc{Let})$$

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

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

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

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

$$\frac{\begin{array}{c} |\mathbf{A}_i| = |\mathbf{x}_i| \qquad B = \mu X.\{\, c_i : (q_i,\mathbf{A}_i)\,\}_{i=1}^{n} \\ \Gamma \vdash^{p}_{p'} e_0 : B \qquad \Delta, \mathbf{x}_i{:}\mathbf{A}_i[B/X] \vdash^{p'+q_i}_{p''} e_i : C \quad (1 \le i \le n) \end{array}}{\Gamma, \Delta \vdash^{p+\mathtt{Kmatch}}_{p''} \mathsf{match}\ e_0\ \mathsf{with}\ \{c_i(\mathbf{x}_i)\text{->}e_i\}_{i=1}^{n} : C} \quad (\textsc{Match})$$

Fig. 3 Syntax directed type rules

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

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

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

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

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

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

Fig. 4 Structural type rules

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 assumptions with a thunk type, i.e. of the form $x : \mathsf{T}^q(A)$ while LET and LETCONS introduce assumptions of a thunk type.

Rules LET and LETCONS allow recursive uses of the bound variable x; the side condition $A \curlyvee \{A, A'\}$ ensures that the type A' assigned to recursive references is a subtype of the result A with zero potential, since re-using potential through the self-reference would be unsound. Note also that rule LET requires that the full thunk cost q of e_1 to be paid by recursive references; this is a sound approximation, but can prevent typing many productive co-recursive definitions. Fortunately, an improved rule has been presented in an earlier work dealing with co-recursion alone [29]. However, for clarity of presentation, we present the simpler rule and soundness proof first, and defer the refinement to Section 6.

Rule LETCONS deals with the *allocation* of a new constructor; this requires paying its associated potential q. However, merely *referencing* a constructor incurs no cost (because constructors are whnfs); hence, rule CONS does not consume resources.

Rule ABS ensures that the cost q of the abstracted expression is captured in the type annotation for the function. Rule APP requires that this cost is paid for each application. The side condition $\Gamma \curlyvee \{\Gamma, \Gamma\}$ to ABS requires that the context shares to itself; this is to ensure that Γ cannot be assigned potential that could be used multiple times through repeated applications, since we chose to allow function types to be shared freely (cf. Lemma 5). A consequence of this side-condition is that only the last argument of a curried function is allowed non-zero potential.

The MATCH rule deals with pattern-matching over an expression of an algebraic data type. The rule requires that all branches admit an identical result type C and that resources p'' available after execution of any of the branches are equal; fulfilling such conditions may require relaxing potential and/or cost annotations using the structural rules described below. Note also that the typing judgement for each branch e_i of the match gains the excess potential q_i associated with constructor c_i in the type B.

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; and finally, the crucial rule PREPAY allows (whole or part of) the cost of a thunk to be paid in advance, so reducing the cost of further uses of the same thunk. Note the rule VAR requires the cost of the thunk to be paid for *every* use, as in call-by-name evaluation; it is rule PREPAY that allows the cost of a thunk to be shared, capturing the memoization in call-by-need evaluation.

Finally, we remark that our type system is not polymorphic; in particular, there is no notion of type quantification, hence free type variables in a judgment cannot be freely instantiated. However, we would surmise that, because of the SHAREVAR rule explained earlier, replacing free type variables with types A that freely share to themselves $A \curlyvee \{A, A\}$ (such as atomic types, function types or data types with zero potential) would lead to another provable type judgement.

4.4 Worked example

We now present a worked example of a type derivation illustrating how the analysis deals with lazy evaluation; in particular, we show how the rule PREPAY allows sharing thunk costs. Consider the following expression (of the λ-calculus enriched with let expressions):

$$\mathsf{let}\ f = ((\lambda x.x)\,(\lambda x.x))\ \mathsf{in}\ \lambda x.\, f\,(f\,x) \qquad \text{(EX1)}$$

The application of (EX1) to some argument forces evaluation of the sub-expression bound to f *once* even though it is used *twice*: first, the outer-most application reduces $((\lambda x.x)\,(\lambda x.x))$ to $\lambda x.x$ and updates the result for f; then the inner-most application re-use the memoized result. We start by re-writing (EX1) translating nested sub-terms in applications using let-expressions as required for our language (cf. Section 3):

$$\mathsf{let}\ f = (\mathsf{let}\ i = \lambda x.x\ \mathsf{in}\ (\lambda x.x)\,i)\ \mathsf{in}\ \lambda x.\,(\mathsf{let}\ y = f\,x\ \mathsf{in}\ f\,y) \qquad (\text{EX1}')$$

Considering the metric for counting *allocations* (i.e. uses of rules $\text{LET}_{\Downarrow}$ and $\text{LETCONS}_{\Downarrow}$), we argue that (EX1′) costs 2 plus the cost of its argument: one use of $\text{LET}_{\Downarrow}$ for the binding of y and one for the binding of i; because of lazy evaluation, the latter is accounted only once. Using the type rules of Fig. 3 and 4, we can derive an annotated typing that justifies our reasoning. We first derive a type just for the sub-term of (EX1′) named f. Since f evaluates to the identity function, we expect it to admit a type $\mathsf{T}^q(A) \xrightarrow{p} A$ for some annotations $p,q \geq 0$ and type A; the derivation is as follows:

$$x{:}\mathsf{T}^0(\mathsf{T}^q(A) \xrightarrow{q} A) \vdash^{0}_{0} x : \mathsf{T}^q(A) \xrightarrow{q} A \qquad \text{VAR} \qquad (1)$$

$$\vdash^{0}_{0} \lambda x.x : \mathsf{T}^0(\mathsf{T}^q(A) \xrightarrow{q} A) \xrightarrow{0} \mathsf{T}^q(A) \xrightarrow{q} A \qquad \text{ABS}(1) \qquad (2)$$

$$i{:}\mathsf{T}^0(\mathsf{T}^q(A) \xrightarrow{q} A) \vdash^{0}_{0} (\lambda x.x)\,i : \mathsf{T}^q(A) \xrightarrow{q} A \qquad \text{APP}(2) \qquad (3)$$

$$x{:}\mathsf{T}^q(A) \vdash^{q}_{0} x : A \qquad \text{VAR} \qquad (4)$$

$$i{:}\mathsf{T}^0(\mathsf{T}^q(A) \xrightarrow{q} A) \vdash^{0}_{0} \lambda x.x : \mathsf{T}^q(A) \xrightarrow{q} A \qquad \text{ABS}(4), \text{WEAK} \qquad (5)$$

$$\vdash^{1}_{0} \mathsf{let}\ i = \lambda x.x\ \mathsf{in}\ (\lambda x.x)\,i : \mathsf{T}^q(A) \xrightarrow{q} A \qquad \text{LET}(5,3) \qquad (6)$$

In the derived type $\mathsf{T}^q(A) \xrightarrow{q} A$ the costs of argument thunk and application are both q; this is because the function needs the argument.[7] The annotation on the turnstile accounts the cost of evaluating the let binding i (one use of $\text{LET}_{\Downarrow}$). Note that is accounted in the *judgement* rather than in the *function type*; this is crucial to allow sharing this cost.

We now continue deriving a type for the complete expression (EX1′);

$$f{:}\mathsf{T}^0(\mathsf{T}^q(A) \xrightarrow{q} A) \vdash^{0}_{0} f : \mathsf{T}^q(A) \xrightarrow{q} A \qquad \text{VAR} \qquad (7)$$

$$f{:}\mathsf{T}^0(\mathsf{T}^q(A) \xrightarrow{q} A), x{:}\mathsf{T}^q(A) \vdash^{q}_{0} f\,x : A \qquad \text{APP}(7) \qquad (8)$$

$$f{:}\mathsf{T}^0(\mathsf{T}^q(A) \xrightarrow{q} A), x{:}\mathsf{T}^q(A), y{:}\mathsf{T}^q(A') \vdash^{q}_{0} f\,x : A \qquad \text{WEAK}(8) \qquad (9)$$

$$f{:}\mathsf{T}^0(\mathsf{T}^q(A) \xrightarrow{q} A), y{:}\mathsf{T}^q(A) \vdash^{q}_{0} f\,y : A \qquad \text{VAR}, \text{APP} \qquad (10)$$

$$f{:}\mathsf{T}^0(\mathsf{T}^q(A) \xrightarrow{q} A),\ f{:}\mathsf{T}^0(\mathsf{T}^q(A) \xrightarrow{q} A), x{:}\mathsf{T}^q(A) \vdash^{1+q}_{0} \mathsf{let}\ y = f\,x\ \mathsf{in}\ f\,y : A \qquad \text{LET}(9,10) \qquad (11)$$

$$f{:}\mathsf{T}^0(\mathsf{T}^q(A) \xrightarrow{q} A), x{:}\mathsf{T}^q(A) \vdash^{1+q}_{0} \mathsf{let}\ y = f\,x\ \mathsf{in}\ f\,y : A \qquad \text{SHARE}(11) \qquad (12)$$

$$f{:}\mathsf{T}^1(\mathsf{T}^q(A) \xrightarrow{q} A), x{:}\mathsf{T}^q(A) \vdash^{2+q}_{0} \mathsf{let}\ y = f\,x\ \mathsf{in}\ f\,y : A \qquad \text{PREPAY}(12) \qquad (13)$$

$$f{:}\mathsf{T}^1(\mathsf{T}^q(A) \xrightarrow{q} A) \vdash^{0}_{0} \lambda x.\,\mathsf{let}\ y = f\,x\ \mathsf{in}\ f\,y : \mathsf{T}^q(A) \xrightarrow{2+q} A \qquad \text{ABS}(13) \qquad (14)$$

$$\vdash^{1}_{0} (\text{EX1}') : \mathsf{T}^q(A) \xrightarrow{2+q} A \qquad \text{LET}(\text{WEAK}(6), 14) \qquad (15)$$

For brevity, we omitted the side condition for the use of structural rule SHARE in line (12):

$$\mathsf{T}^0(\mathsf{T}^q(A) \xrightarrow{q} A) \curlyvee \{\mathsf{T}^0(\mathsf{T}^q(A) \xrightarrow{q} A), \mathsf{T}^0(\mathsf{T}^q(A) \xrightarrow{q} A)\}$$

[7] Note that, because of lazy evaluation, the function could discard the argument and thus the thunk cost need not always be accounted in the application cost. However, the function in this example is strict.

This is trivially satisfied because thunk type and functional type always shares to themselves (cf. rules SHARETHUNK and SHAREFUN in Fig. 2). Also, because the type rule LET always allows recursive uses, we must employ weakening to introduce a type for y in (9). By Lem. 1 we can choose a subtype A' of A such that $A \curlyvee \{A, A'\}$.

Proceeding backwards from the conclusion (15), rule LET introduces the assumption for f using the result of the previous judgment (6). Note that assumption $f{:}\mathsf{T}^1(\mathsf{T}^q(A) \xrightarrow{q} A)$ captures the delayed cost of evaluation: first f requires one unit to evaluate *and then* yields a function costing q units; thus, this could *not* be expressed as $f{:}\mathsf{T}^0(\mathsf{T}^q(A) \xrightarrow{1+q} A)$ or simply $f{:}\mathsf{T}^q(A) \xrightarrow{1+q} A$.

In line (13) we use the structural rule PREPAY to pay ahead the cost of f before sharing; the two uses of VAR to reference f in lines (7), (10) become free. Thus the cost of evaluating f is counted only once and we get a type $\mathsf{T}^q(A) \xrightarrow{2+q} A$ expressing the accurate cost of lazy evaluation: two allocations plus the cost of argument. We could have omitted the use of PREPAY and still derive an admissible, albeit less precise cost estimate: each use of f would cost 1 extra unit and the final judgment would be $\vdash^0_0 (\mathrm{EX1}') : \mathsf{T}^q(A) \xrightarrow{4+q} A$. This corresponds to the cost of call-by-name evaluation and thus a sound overestimation of lazy evaluation.

Because PREPAY is a structural rule, we could instead have employed it after ABS and before LET, i.e. after line (14); this would lead to the conclusion $\vdash^2_0 (\mathrm{EX1}') : \mathsf{T}^q(A) \xrightarrow{1+q} A$. The overall cost $2+1+q$ is the same as before, but is accounted differently: the cost of f is assigned to the judgment rather than the function type; thus, if this judgment was part of some larger derivation, the cost of f would be payed *once* instead of for *each application*.

4.5 Experimental results

We have constructed a prototype implementation of as a type reconstruction algorithm that takes a (closed) expression and either produces an annotated typing or fails (e.g. when the cost bounds are not linear). The implementation is fully automatic, i.e. it does not require any type annotations from the programmer. For convenience, constructors for standard algebraic data types such as booleans and lists, as well as primitive arithmetic operations on integers were also added to the implementation. 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`.

Type reconstruction is performed in three phases:

1. Damas-Milner type inference to obtain an unannotated version of the type derivation;
2. annotate types with fresh variables and traverse the type derivation gathering linear constraints according to the type rules of Sect. 4.3;
3. solve the collected constraints using a standard linear programming tool.[8]

Structural rules are used in the second phase only at specific points: PREPAY is applied immediately after bound variables are introduced, namely, in the body of a lambda, let-expression or match alternative;[9] this can be done uniformly because the rule allows any part of the cost to be paid (including zero). Hence, we defer to the constraint solver the choice of how much individual thunks should be prepaid in order to achieve an overall optimal solution. SUBTYPE is applied for the argument type of applications and for the

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

[9] This heuristic choice typically lowers cost overestimation by allowing paying ahead as early as possible; cf. last paragraph of Sect. 4.4.

result type of matches (losing precision if necessary to obtain compatible types). RELAX is also always applied after a match to join branches with possibly distinct costs. SHARE is used for variables that occur in two hypothesis before LET, LETCONS, APP, MATCH; as in for prepay, this is done uniformly and the choice of how to split potential between uses is delegated to the constraint solver. This allows applying the type rules of Sect. 4 syntax-directed way. The soundness of the resulting inference algorithm is straightforward; we conjecture that completeness should also hold. However, we have not formally stated and proved this.

Lastly, constraint solving using a linear programming tool requires an objective function; here we employ a simple heuristic: minimize the *sum of cost annotations* in the result type and the judgement. This means we do not report all solutions (i.e. an annotated typing plus the set of collected constraints) but only an admissible one, hence the implementation is a whole-program analysis. In practice, we found this heuristic gives good cost bounds for a number of small by representative examples.

The following presents some cost bounds inferred for recursive and co-recursive definitions. Instead of Haskell, we now use the concrete syntax of the term language of Sect. 3 (which is the actual input to our implementation). The final example of Fibonacci numbers serves as motivation for a refinement that will be presented in Sect. 6.

Example 1 (Zipping finite and infinite lists) Consider the standard `zipWith` function that combines two lists using an argument function:

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

Analysing `zipWith` for counting *applications* we obtain the following annotated type:

```
zipWith : T(T(a) -> T(b) -> c) ->
       T(Rec{Cons:(T(a),T(#)) | Nil:()}) ->
       T(Rec{Cons@5:(T(b),T(#)) | Nil:()}) ->
         Rec{Cons:(T(c),T(#)) | Nil:()}
```

This type expresses costs in terms of the length n of the second argument to `zipWith`: for a list of n `Cons` the cost is bounded by $5 \cdot n$. This corresponds to 2 applications for `f` and 3 applications for the recursive call to `zipWith`. This bound is tight if the second list is shorter than the first one, but in either case it is a sound upper bound.[10]

The type rules LET/LETCONS constrain the potential of self-referencing values to be zero (it would be unsound to allow otherwise). Thus, if the second argument of `zipWith` is an infinite list the above type would not be admissible and the linear solver would choose an alternative type, where costs are expressed in terms of the output type:

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

[10] Note that expressing the bound using the length of the first argument of a curried function is *not* allowed because of the ABS rule (cf. Sect. 4.3). This could be overcome simply by un-curring the definition [11].

```
T(Rec{Cons:(T(b),T(#)) | Nil:()}) ->
  Rec{Cons:(T@2(c),T@3(#)) | Nil:()}
```

The bound for fully evaluating n elements of the result is $2 \cdot n + 3 \cdot n$: evaluating each head cost 2 (for the curried applications to `f`) and each tail costs 3 (for the applications to `zipWith`). We also obtain bounds for partial evaluations; for example, evaluating just the n-th element requires traversing n tail thunks plus a single head, and hence costs $2 + 3 \cdot n$.

Example 2 (Fibonacci numbers) One well-known example of lazy evaluation in Haskell is the definition of the infinite sequence of Fibonacci numbers as a co-recursive definition.

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

By inlining the definition of `tail` and using `zipWith` as before, this translates as:

```
fibs = (let xs = match fibs with
                 Cons(x,fibs') -> zipWith plus fibs fibs'
         in letcons xs1 = Cons(1, xs)
         in letcons xs0 = Cons(0, xs1)
         in xs0)
```

For simplicity, consider again the metric counting applications (arithmetic operations are cost-free). Because of lazy evaluation, each successive Fibonacci number can then be obtained in bounded cost. However, a proof of this cannot be derived using the rules of Sect. 4 alone because this co-recursive use of `zipWith` does *not* admit a type. To understand why, let us reason about how the type rules constrain the thunks costs for `fibs`.

Assume the inputs of `zipWith` are lists with costs p, q for the tail thunks. Examining the definition of `fibs`, we see using rule APP for `zipWith plus fibs fibs'` requires the result tail to cost *at least* $p+q+3$: p for evaluating `fibs`, q for evaluating `fibs'` plus 3 for the curried applications. Combining these requirements generates unsatisfiable constraints ($p = q$ and $p = 3 + p + q$) and hence there is no admissible type. Note that this limitation is inherent in the type system rather than in the implementation.

In Section 6 we will see how to improve the analysis for such co-recursive definitions by *allowing recursive accesses to thunks to pay zero cost*. Informally, this is sound because such accesses must already be in normal form (or else they correspond to unproductive programs, e.g. let $x = x$ in $\ldots$). This improvement requires only a revision of the type rules LET and LETCONS. However, the formulation and proof of soundness becomes more complex, and so we chose to present the simpler system first.

Using the revised type rules, the implementation infers types[11] for `zipWith` and `fibs`, proving a cost bound for successive Fibonacci numbers:

```
fibs : Rec{Cons:(T(Int),T@5(#)) | Nil:()}
```

In our previous work [29] we show that this technique infers accurate bounds for other non-trivial co-recursive definitions e.g. the textbook solution to the *Hamming problem* [2].

Example 3 (Combining recursion and co-recursion) A common pattern in lazy functional programming is to express computations as composition of higher-order functions combining finite and infinite structures [9]. The next example shows the use `zipWith` to sum Fibonacci numbers with values from some other list:

[11] The reconstruction algorithm may always use the revised rules, since they subsume the previous ones.

```
zipWith' =  ...
sumWithFibs xs = let f = \x y -> x+y in zipWith' f fibs xs
```

Here `zipWith'` is an identical definition to `zipWith` that we duplicate simply to allow it to be assigned a distinct annotated type. In particular, it can be assigned a type with potential in the second argument:

```
zipWith' : T(T(Int) -> T(a) -> Int) ->
           T(Rec{Cons:(T(Int),T@5(#)) | Nil:()}) ->
           T(Rec{Cons@10:(T(a),T(#)) | Nil:()}) ->@5
             Rec{Cons:(T(Int),T(#)) | Nil:()}
sumWithFibs : T(Rec{Cons@10:(T(a),T(#)) | Nil:()}) ->@5
                Rec{Cons:(T(Int),T(#)) | Nil:()}
```

The cost bound for `sumWithFibs` is given in terms of potential assigned to the input list `xs`. Hence, if the input list has length k, the cost is $5+10\cdot k$ applications. Note that the choice of transferring potential of the recursive input data to pay the thunk costs of co-recursive results is performed automatically by the linear solver.

Finally, we remark that such an example could *not* be analysed by either systems presented in [24] and [29] alone: it requires the combination of potential and the improved rule for co-recursion.

5 Soundness

This section establishes the soundness of our analysis with respect to the operational semantics of Section 3. We begin by stating some auxiliary lemmas and preliminary definitions, notably formalizing the notion of *potential* from Section 4. We conclude with the soundness result proper (Theorem 1) and its much simpler conclusion (Corollary 2).

5.1 Auxiliary lemmas

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 2 (Substitution) *If* $\Gamma, x{:}A \vdash^{p}_{p'} \widehat{e} : C$ *and* $y \notin \Gamma \cup \mathrm{FV}(\widehat{e})$ *then* $\Gamma, y{:}A \vdash^{p}_{p'} \widehat{e}[y/x] : C$.

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

The following two lemmas establish inversion properties for type derivation of constructors and λ-abstractions.

Lemma 3 (CONS inversion) *If* $\Gamma \vdash c(\mathbf{y}) : B$ *then also* $B = \mu X.\{\ldots / c : (q, \mathbf{A}) / \ldots\}$ *and* $\Gamma \curlyvee \{\mathbf{y}{:}\mathbf{A}[B/X]\}$.

Lemma 4 (ABS inversion) *If* $\Gamma \vdash \lambda x.e : A \xrightarrow{q} C$ *then there exists* Γ' *such that* $\Gamma \curlyvee \Gamma'$, $\Gamma' \curlyvee \{\Gamma', \Gamma'\}$, $x \notin \mathrm{dom}(\Gamma')$ *and* $\Gamma', x{:}A \vdash^{q}_{0} e : C$.

Proof (Sketch for both lemmas) A typing with conclusion $\Gamma \vdash c(\mathbf{y}) : B$ must result from axiom CONS followed by (possibly zero) uses of structural rules. Similarly, a typing $\Gamma \vdash \lambda x.e : A \xrightarrow{q} C$ 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 next auxiliary lemma allows splitting contexts used for typing expressions in *whnf* according to a split of the result type.

Lemma 5 (Context Splitting) *If $A \curlyvee \{A_1, A_2\}$ and $\Gamma \vdash^{0}_{0} w : A$, where w is an expression in whnf; then there exists Γ_1, Γ_2 such that $\Gamma \curlyvee \{\Gamma_1, \Gamma_2\}$, $\Gamma_1 \vdash^{0}_{0} w : A_1$ and $\Gamma_2 \vdash^{0}_{0} w : A_2$.*

Proof (Sketch) The proof follows from an application of Lemma 3 (if w is a constructor) or Lemma 4 (if w is an abstraction) together with the definition of sharing. Note that in the latter case, the side condition $\Gamma \curlyvee \{\Gamma, \Gamma\}$ to the ABS type rule ensures that Γ is has no potential (for otherwise it would be unsound to duplicate it).

Lemma 6 (Transitivity of sharing) *If $A \curlyvee (B \cup \mathbf{S})$ and $B \curlyvee \mathbf{R}$ holds for types A, B and multisets of types $\mathbf{S}$, $\mathbf{R}$, then $A \curlyvee (\mathbf{R} \cup \mathbf{S})$ holds as well.*

Proof (Sketch) The proof is by induction on the derivations of the two sharing relations; in the non-trivial cases SHAREDAT, SHAREFUN and SHARETHUNK the conclusion follows directly from the transitivity of $\geq$.

5.2 Potential

Definition 2 (Potential) The (shallow) potential of an expression $\widehat{e}$ of type A, written $\phi(\widehat{e} : A)$, is defined as follows:

$$\phi(\widehat{e} : A) \stackrel{\text{def}}{=} \begin{cases} p, & \text{if } A = \mu X.\{\cdots \mid c{:}(p, \mathbf{B}) \mid \cdots\} \text{ and } \widehat{e} = c(\boldsymbol{\ell}) \\ 0, & \text{otherwise.} \end{cases}$$

For data constructors potential is given by the corresponding annotation in the type. For unevaluated expressions (i.e. thunks) and λ-abstractions, the potential is always zero.

Note that ϕ accounts only for the contribution of a single constructor rather than the accumulated potential that is accessible through the data structure. This is a notable difference compared to our earlier work on amortised analysis [24, 12, 13, 8], where (accumulated) potential was defined recursively. We recover the accumulated potential for a complete data structure by collecting all contributions (through all references to its locations) using the notion of *global types* (cf. Sect. 5.3). This change allows a significant simplification of the soundness proof in the lazy setting compared to our earlier work [24].

The next lemma formalizes the intuition that sharing distributes the potential associated with a type.

Lemma 7 (Potential splitting) *If $A \curlyvee \{A_1, \ldots, A_n\}$ then $\phi(\widehat{e} : A) \geq \sum_i \phi(\widehat{e} : A_i)$.*

Proof The result follows immediately from the definitions of sharing (Fig. 2) and potential (Def. 2).

This lemma has two important special cases that justify the previously-stated intuition that a type that shares with itself has no potential.

Corollary 1 *For types A,A' and all $\widehat{e}$, we have:*

1. *If $A \between \{A,A\}$ then $\phi(\widehat{e}:A) = 0$;*
2. *If $A \between \{A,A'\}$ then $\phi(\widehat{e}:A') = 0$.*

Proof Both equalities follow immediately from Lem. 7.

5.3 Global types, contexts and balance

In order to formulate the soundness invariants of our type system, we introduce three auxiliary mappings. Let *Type* and *Ctx* be the set of annotated types and contexts, respectively, and *Loc* be the set of locations. We consider the partial functions

$$\begin{aligned} \mathcal{M} &: \mathit{Loc} \longrightarrow \mathit{Type} \\ \mathcal{C} &: \mathit{Loc} \longrightarrow \mathit{Ctx} \\ \mathcal{B} &: \mathit{Loc} \longrightarrow \mathbb{Q}^+ \end{aligned}$$

which associate locations ℓ with:

1. the *global type* $\mathcal{M}(\ell)$ that accounts for the thunk cost and all potential associated with that location; note that $\mathcal{M}(\ell)$ is always of the form $\mathsf{T}^q(A)$ for some cost q and type A;
2. the *global context* $\mathcal{C}(\ell)$ that is used for typing the expression $\mathcal{H}(\ell)$ with the global type;
3. a non-negative rational number *balance* $\mathcal{B}(\ell)$ that keeps track of the thunk cost of ℓ that has been paid in advance by applications of the PREPAY rule.

These auxiliary mappings are needed only in the soundness proof of the analysis for bookkeeping purposes, but are *not* part of either the type system nor the operational semantics — in particular, they are not used for performing the analysis nor do they incur runtime costs.

The *projection* operation $\restriction_x$ for a context Γ is the multiset of types associated with x in Γ, i.e. $\Gamma\restriction_x = \{A \mid x{:}A \in \Gamma\}$. Projections extend to global contexts $\mathcal{C}$ in the natural way:

$$\mathcal{C}\restriction_x = \{\ell_1 \mapsto \Gamma_1, \ldots, \ell_n \mapsto \Gamma_n\}\restriction_x \stackrel{\text{def}}{=} \Gamma_1\restriction_x, \ldots, \Gamma_n\restriction_x$$

Definition 3 (Global subtyping) We extend subtyping to global types and write $\mathcal{M} <: \mathcal{M}'$ if and only if $\mathrm{dom}(\mathcal{M}) \subseteq \mathrm{dom}(\mathcal{M}')$ and for all $\ell \in \mathrm{dom}(\mathcal{M})$ we have $\mathcal{M}(\ell) = \mathsf{T}^q(A)$, $\mathcal{M}'(\ell) = \mathsf{T}^{q'}(A')$ and $A <: A'$.

This relation will be used to assert a soundness invariant, namely that potential assigned to global types is preserved by evaluation. However, because of the PREPAY rule, thunk costs may decrease; thus the definition above ignores such thunk annotations q, q'. Global subtyping inherits transitivity from Lem. 6 and our definition of subtyping on types.

We can now formulate the principal soundness invariants of our analysis, namely, *consistency* and *compatibility* relations between a heap configuration and the global types, contexts, and balance.

Definition 4 (Type consistency) A heap state $(\mathcal{H},\mathcal{L})$ is said to be consistent with global contexts $\mathcal{C}$, global types $\mathcal{M}$ and balance $\mathcal{B}$, written $\mathcal{C},\mathcal{B} \vdash (\mathcal{H},\mathcal{L}) : \mathcal{M}$, if and only if for all $\ell \in \mathrm{dom}(\mathcal{H}) \setminus \mathcal{L}$ we have $\mathcal{M}(\ell) = \mathsf{T}^q(A)$ and $\mathcal{C}(\ell) \vdash^{\frac{q+\mathcal{B}(\ell)}{0}} \mathcal{H}(\ell) : A$. Furthermore if $\mathcal{H}(\ell)$ is in whnf, then $q = 0$ and $\mathcal{B}(\ell) = 0$.

Informally, the above definition requires that, for every location ℓ that is *not* under evaluation, the global type $\mathcal{M}(\ell)$ is justified by a typing of the expression $\mathcal{H}(\ell)$ using the global context $\mathcal{C}(\ell)$ and the prepaid balance $\mathcal{B}(\ell)$. Note that locations under evaluation are trivially considered consistent.

Definition 5 (Compatibility) A global type $\mathcal{M}$ is *compatible* with context Γ and global contexts $\mathcal{C}$, written $\mathcal{M} \Yup (\Gamma; \mathcal{C})$, if and only if $\mathcal{M}(\ell) \Yup (\Gamma\restriction_\ell, \mathcal{C}\restriction_\ell)$ for all $\ell \in \mathrm{dom}(\mathcal{M})$.

Informally, this definition ensures that the global type $\mathcal{M}(\ell)$ of each location ℓ accounts for the joint potential of all references to it in either the local or global contexts.

Note that, although rules LET and LETCONS require assigning types with zero potential to recursive references, the type consistency and compatibility invariants0 do not prevent initial configurations containing cyclic heap structures with non-zero potential. However, the invariants do prevent any external references to such cyclic data from accessing this potential, as the following example illustrates.[12]

Example 4 Consider a heap $\mathcal{H} \stackrel{\text{def}}{=} \{\ell_1 \mapsto \mathsf{succ}(\ell_1)\}$ with a single location ℓ_1 inicialized with a self-referencing constructor. Let $N(q) \stackrel{\text{def}}{=} \mu X.\{\mathsf{zero} : (0, ()) \mid \mathsf{succ} : (q, \mathsf{T}^0(X)\}$ be the type of Peano naturals where the parameter q is the potential assigned to the successor.

We show that type consistency and compatibility hold for the following configuration which assigns arbitrary potential $q_1 \geq 0$ to the cyclic reference:

$$\begin{aligned}\mathcal{M}(\ell_1) &\stackrel{\text{def}}{=} \mathsf{T}^0(N(q_1))\\ \mathcal{C}(\ell_1) &\stackrel{\text{def}}{=} \{\ell_1{:}\mathsf{T}^0(N(q_1))\}\\ \mathcal{B}(\ell_1) &\stackrel{\text{def}}{=} 0\end{aligned}$$

For type consistency, it is enough to show that

$$\mathcal{C}(\ell_1) \vdash \mathsf{succ}(\ell_1) : N(q_1)$$

which follows directly by rule CONS.

Let Γ be a typing context for some expression that references location ℓ_1, i.e. $\Gamma = \{\ell_1{:}\mathsf{T}^0(N(q_2))\}$ where the potential q_2 is to be determined. For compatibility to hold, we require that

$$\mathcal{M}(\ell_1) \Yup \{\Gamma\restriction_{\ell_1}, \mathcal{C}\restriction_{\ell_1}\}$$

Substituting the types defined above gives

$$\mathsf{T}^0(N(q_1)) \Yup \{\mathsf{T}^0(N(q_2)), \mathsf{T}^0(N(q_1))\}$$

By rule SHARETHUNK followed by SHAREDAT (Fig. 2), we get

$$q_1 \geq q_2 + q_1$$

which together with non-negativity of annotations implies $q_2 = 0$. Thus, the potential q_1 inside the cycle is unconstrained, but any external reference that can be used by the program must have zero potential.

[12] Unlike our earlier work [24], we no longer require a technical lemma for replacing such pathological configurations. This is because the revised definition of potential is no longer recursive.

Finally, we define two shorthand aggregation notations; first, for summing the total potential of a heap with respect to the global types and second, for summing the balance of all locations:

$$\Phi_{\mathcal{H}}\,\mathcal{M} \stackrel{\text{def}}{=} \sum\big\{\phi(\mathcal{H}(\ell):A) \mid \ell \in \mathrm{dom}(\mathcal{H}) \text{ and } \mathcal{M}(\ell) = \mathsf{T}^q(A)\big\}$$
$$\sum\mathcal{B} \stackrel{\text{def}}{=} \sum\{\mathcal{B}(\ell) \mid \ell \in \mathrm{dom}(\mathcal{B})\}$$

5.4 Soundness of the proof system

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

Theorem 1 (Soundness) *Let $t \in \mathbb{Q}$ with $t \geq 0$ be fixed, but arbitrary; similarly, let Δ be an arbitrary context. If the following statements hold*

$$\Gamma \vdash^{p}_{p'} e : A \tag{H1}$$
$$\mathcal{C}, \mathcal{B} \vdash (\mathcal{H}, \mathcal{L}) : \mathcal{M} \tag{H2}$$
$$\mathcal{M} \Vdash (\Gamma, \Delta; \mathcal{C}) \tag{H3}$$
$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash e \Downarrow w, \mathcal{H}' \tag{H4}$$

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

$$m \geq t + p + \Phi_{\mathcal{H}}\,\mathcal{M} + \sum\mathcal{B} \tag{H5}$$

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

$$\mathcal{M} <: \mathcal{M}' \tag{C1}$$
$$\Gamma' \vdash^{0}_{0} w : A \tag{C2}$$
$$\mathcal{C}', \mathcal{B}' \vdash (\mathcal{H}', \mathcal{L}) : \mathcal{M}' \tag{C3}$$
$$\mathcal{M}' \Vdash (\Gamma', \Delta; \mathcal{C}') \tag{C4}$$
$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e \Downarrow w, \mathcal{H}' \tag{C5}$$
$$m' \geq t + p' + \phi(w:A) + \Phi_{\mathcal{H}'}\,\mathcal{M}' + \sum\mathcal{B}' \tag{C6}$$

Starting from an empty configuration, the theorem simplifies as follows.

Corollary 2 *If $\vdash^{p}_{p'} e : A$ and $\emptyset, \emptyset, \emptyset \vdash e \Downarrow w, \mathcal{H}'$ hold, then for all $m \in \mathbb{N}$ with $m \geq p$ there exists $m' \in \mathbb{N}$ with $m - m' \leq p - p'$ and the resource-bounded evaluation $\emptyset, \emptyset, \emptyset \vdash^{m}_{m'} e \Downarrow w, \mathcal{H}'$ succeeds as well.*

Proof (Corollary 2) We invoke Theorem 1 with $t = m - p$ and receive m' such that the resource-bounded evaluation succeeds (C5) and by (C6) also $m' \geq t + p' = (m - p) + p'$; rearranging the terms gives $m - m' \leq p' - p$ as required.

Informally, the soundness theorem reads as follows: if an expression e admits a type A (H1), the heap can be typed (H2) (H3) and the evaluation is successful (H4), then the result *whnf* also admits type A (C2). Furthermore, potential in global types is preserved (C1), the final heap can also be typed (C3) (C4) and the static bounds that are obtained from the typing of e give safe resource estimates for evaluation (H5) (C5) (C6). The arbitrary value t is used in the APP and RELAX cases to carry over excess potential which is not used immediately

but will be needed for subsequent evaluations. Similarly, the context Δ is used in the LET, LETCONS, APP and MATCH cases to preserve types for variables that are not in the current scope but are necessary for subsequent evaluations. Conclusion (C1) is crucial for the VAR case; while the contribution of $\phi(w:A)$ within (C6) is used in the MATCH case.

Note that our analysis infers a safety guarantee rather than a liveness one: as in the underlying type system for the λ-calculus with general recursion, a type derivation in our system is a proof of partial correctness and does not imply termination/productivity. Hence, the premise (H4) requires a terminating evaluation (but ignores the resource bounds).

Proof (Theorem 1) The proof is by induction on the lengths of the derivations of (H4) and (H1) ordered lexicographically, with the derivation of the evaluation taking priority over the typing derivation. This is required since an induction on the length of the typing derivation alone would fail for the case of unevaluated thunks, which prolongs the length of the typing derivation by a typing judgment for the thunk, granted through the type consistency hypothesis. On the other hand, the length of the derivation for the term evaluation never increases, but may remain unchanged where the last step of the typing derivation was obtained by a structural rule. In these cases, the length of the typing derivation does decrease, allowing an induction over lexicographically ordered lengths of both derivations. We proceed by case analysis of the typing rule used in premise (H1).

Case VAR*:* We have $\ell{:}\mathsf{T}^q(A) \vdash^{q+\mathtt{Kvar}}_{0} \ell : A$ from the typing hypothesis (H1). From the compatibility hypothesis (H3) we get $\mathcal{M}(\ell) \between (\mathsf{T}^q(A), \Delta\restriction_\ell, \mathcal{C}\restriction_\ell)$ which implies $\mathcal{M}(\ell) = \mathsf{T}^r(A_0)$ and $A_0 \between \{A, A'\}$ for some types A_0, A' and annotation r with $q \geq r$.

The evaluation premise (H4) reads as $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash \ell \Downarrow w, \mathcal{H}'[\ell \mapsto w]$ for some intermediate heap $\mathcal{H}'$; by inversion of the only applicable evaluation rule $\text{VAR}_\Downarrow$, we obtain $\ell \notin \mathcal{L}$ and

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \cup \{\ell\} \vdash \mathcal{H}(\ell) \Downarrow w, \mathcal{H}' \tag{16}$$

From $\ell \notin \mathcal{L}$ together with type consistency for ℓ (H2) we get $\mathcal{C}(\ell) \vdash^{r+\mathcal{B}(\ell)}_{0} \mathcal{H}(\ell) : A_0$. We proceed by case analysis on whether $\mathcal{H}(\ell)$ is in whnf or not.

If $\mathcal{H}(\ell)$ is in whnf: The evaluation (16) reduces immediately by $\text{WHNF}_\Downarrow$ and we have $w = \mathcal{H}(\ell)$ and $\mathcal{H} = \mathcal{H}' = \mathcal{H}'[\ell \mapsto w]$, i.e. the update is without effect. By the type consistency hypothesis for ℓ we get $r = 0$, $\mathcal{B}(\ell) = 0$ and $\mathcal{C}(\ell) \vdash^{0}_{0} w : A_0$. Let $\mathcal{M}' = \mathcal{M}[\ell \mapsto \mathsf{T}^0(A')]$; conclusion (C1) follows directly from $\mathcal{M}(\ell) = \mathsf{T}^r(A_0)$ and $A_0 \between \{A, A'\}$ established earlier. Again by $A_0 \between \{A, A'\}$ and Lem. 5 we get $\mathcal{C}(\ell) \between \{\Gamma_1', \Gamma_2'\}$ and $\Gamma_1' \vdash^{0}_{0} w{:}A$ as required for (C2), as well as $\Gamma_2' \vdash^{0}_{0} w : A'$. By the premise (H2) together with $\mathcal{C}(\ell) \between \{\Gamma_1', \Gamma_2'\}$ and Lem. 6 we get $\mathcal{C}', \mathcal{B} \vdash (\mathcal{H}, \mathcal{L}) : \mathcal{M}'$ as required for conclusion (C3). From the compatibility premise (H3) together with $\mathcal{C}(\ell) \between \{\Gamma_1', \Gamma_2'\}$ established earlier we can conclude $\mathcal{M}' \between (\Gamma_1', \Delta; \mathcal{C}')$ as required for (C4). Conclusion (C5) with $m' = m - \mathtt{Kvar}$ follows directly from an application of rule $\text{WHNF}_\Downarrow$ and $\text{VAR}_\Downarrow$. It remains to show that (C6) is satisfied for m' as mandated by the operational semantics here.

$$\begin{aligned} m - \mathtt{Kvar} &\geq (t + (q + \mathtt{Kvar}) + \Phi_{\mathcal{H}} \mathcal{M} + \textstyle\sum \mathcal{B}) - \mathtt{Kvar} \\ &= t + q + \left(\Phi_{\mathcal{H}\setminus\ell} \mathcal{M} + \phi(w : A_0)\right) + \textstyle\sum \mathcal{B} \\ &\geq t + q + \Phi_{\mathcal{H}\setminus\ell} \mathcal{M} + \left(\phi(w : A) + \phi\left(w : A'\right)\right) + \textstyle\sum \mathcal{B} \\ &\geq t + \phi(w : A) + \Phi_{\mathcal{H}} \mathcal{M}' + \textstyle\sum \mathcal{B} \end{aligned}$$

The first inequality holds by Lem. 7; while the second inequality holds by dropping q, as the evaluation of w has already been paid for.

If $\mathcal{H}(\ell)$ is not in whnf: By type consistency for ℓ we get $\mathcal{C}(\ell) \vdash^{r+\mathcal{B}(\ell)}_{0} \mathcal{H}(\ell) : A_0$. Let $\mathcal{M}_0 = \mathcal{M}[\ell \mapsto \mathsf{T}^r(A')]$, $\mathcal{B}_0 = \mathcal{B}[\ell \mapsto 0]$ and $\mathcal{C}_0 = \mathcal{C}[\ell \mapsto \emptyset]$. By definition of $\mathcal{M}_0$ and subtyping, it is immediate that $\mathcal{M} <: \mathcal{M}_0$. We observe also that $\mathcal{C}_0, \mathcal{B}_0 \vdash (\mathcal{H}, \mathcal{L} \cup \{\ell\}) : \mathcal{M}_0$ follows from the premise (H2) and because type consistency holds trivially for locations under evaluation. Furthermore $\mathcal{M}_0 \between (\mathcal{C}(\ell); \Delta, \mathcal{C}_0)$ holds, since for all x we have $\mathcal{C}\restriction_x = \mathcal{C}(\ell)\restriction_x \cup \mathcal{C}_0\restriction_x$ by definition.

Hypothesis (H5) instantiates as follows:

$$\begin{aligned} m - \mathtt{Kvar} &\geq \Big(t + (q + \mathtt{Kvar}) + \Phi_{\mathcal{H}}\,\mathcal{M} + \textstyle\sum \mathcal{B}\Big) - \mathtt{Kvar} \\ &= t + q + \Phi_{\mathcal{H}}\,\mathcal{M} + \textstyle\sum \mathcal{B} \\ &\geq t + \big(r + \mathcal{B}(\ell)\big) + \Phi_{\mathcal{H}}\,\mathcal{M}_0 + \textstyle\sum \mathcal{B}_0 \end{aligned}$$

which follows by $q \geq r$ and re-arranging the parcels.

We can now apply the induction to the evaluation of $\mathcal{H}(\ell)$ with type A_0 and obtain:

$$\mathcal{M}_0 <: \mathcal{M}' \tag{17}$$

$$\Gamma' \vdash^{0}_{0} w : A_0 \tag{18}$$

$$\mathcal{C}', \mathcal{B}' \vdash (\mathcal{H}', \mathcal{L} \cup \{\ell\}) : \mathcal{M}' \tag{19}$$

$$\mathcal{M}' \between (\Gamma', \Delta; \mathcal{C}') \tag{20}$$

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \cup \{\ell\} \vdash^{m-\mathtt{Kvar}}_{m'} \mathcal{H}(\ell) \Downarrow w, \mathcal{H}' \tag{21}$$

$$m' \geq t + 0 + \phi(w : A_0) + \Phi_{\mathcal{H}'}\,\mathcal{M}' + \textstyle\sum \mathcal{B}' \tag{22}$$

Note that applying induction to the global type gave a stronger typing than required for the result (18). We will now recover the required typing using Lem. 7 for splitting contexts; the remaining potential associated through A' allows us to establish memory consistency for the remaining aliases. So by $A_0 \between \{A, A'\}$ from above, (18) and Lem. 5 we get $\Gamma' \between \{\Gamma_1', \Gamma_2'\}$ and $\Gamma_1' \vdash^{0}_{0} w{:}A$ as required for (C2), as well as $\Gamma_2' \vdash^{0}_{0} w{:}A'$. From the later together with (19) we get the required for (C3):

$$\mathcal{C}'[\ell \mapsto \Gamma_2'], \mathcal{B}'[\ell \mapsto 0] \vdash (\mathcal{H}'[\ell \mapsto w], \mathcal{L}) : \mathcal{M}'$$

Conclusion (C1) follows by transitivity of subtyping from $\mathcal{M} <: \mathcal{M}_0$ established earlier and $\mathcal{M}_0 <: \mathcal{M}'$ given by (17). From (20) we have $\mathcal{M}' \between (\Gamma_1', \Gamma_2', \Delta; \mathcal{C}')$ which by definition is equivalent to $\mathcal{M}' \between (\Gamma_1', \Delta; \mathcal{C}'[\ell \mapsto \Gamma_2'])$, as required for (C4). Conclusion (C5) follows directly from (21) by application of $\textsc{Var}_{\Downarrow}$.

It remains to show that the bound for m' obtained from (22) satisfies the requirements of conclusion (C6); our proof obligation is:

$$t + 0 + \phi(w : A_0) + \Phi_{\mathcal{H}'}\,\mathcal{M}' + \textstyle\sum \mathcal{B}' \geq t + 0 + \phi(w : A) + \Phi_{\mathcal{H}'[\ell \mapsto w]}\,\mathcal{M}' + \sum \mathcal{B}'[\ell \mapsto 0]$$

which follows in two parts: First, we have $\sum \mathcal{B}' = \sum \mathcal{B}'[\ell \mapsto 0] + \mathcal{B}'(\ell) \geq \sum \mathcal{B}'[\ell \mapsto 0]$ because the balance is non-negative. Second, let $\mathcal{M}'(\ell) = \mathsf{T}(A'')$ for some type A'' and some unimportant thunk cost annotation, then we observe

$$\begin{aligned} \phi(w : A_0) + \Phi_{\mathcal{H}'}\,\mathcal{M}' &\geq \phi(w : A) + \phi\big(w : A'\big) + \Phi_{\mathcal{H}'}\,\mathcal{M}' \\ &\geq \phi(w : A) + \phi\big(w : A''\big) + \Phi_{\mathcal{H}'}\,\mathcal{M}' = \phi(w : A) + \Phi_{\mathcal{H}'[\ell \mapsto w]}\,\mathcal{M}' \end{aligned}$$

where the first inequality follows from $A_0 \between \{A, A'\}$ and Lem. 7. The second inequality follows, since by (17) we have $A' <: A''$; thus using the definition of subtying and Lem. 7 we therefore gain $\phi(w : A') \geq \phi(w : A'')$. The final equality then follows because $\mathcal{H}'(\ell)$ is not a whnf, hence does not contribute to the sum of potential.

Case LET*:* The typing and evaluation premises (H1) and (H4) instantiate as

$$\Gamma, \Delta' \vdash^{p+\mathtt{Klet}}_{p'} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : C \tag{23}$$

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m+\mathtt{Klet}}_{m'} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow w, \mathcal{H}' \tag{24}$$

From (23) and Substitution we get $\Delta', \ell{:}\mathsf{T}^q(A) \vdash^{p}_{p'} e_2[\ell/x] : C$. By (24) and inversion of rule $\text{LET}_{\Downarrow}$ we get $\mathcal{H}_0, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e_2[\ell/x] \Downarrow w, \mathcal{H}'$ where ℓ is a fresh location and for $\mathcal{H}_0 = \mathcal{H}\big[\ell \mapsto e_1[\ell/x]\big]$. We intend to apply the induction hypothesis for the evaluation of $e_2[\ell/x]$, so we must establish the required premises first. Note that we do not invoke the induction hypothesis for the subterm e_1, since it is not evaluated at this point, but just stored within the heap.

Let $\mathcal{B}_0 = \mathcal{B}[\ell \mapsto 0]$, $\mathcal{M}_0 = \mathcal{M}[\ell \mapsto \mathsf{T}^q(A)]$ and $\mathcal{C}_0 = \mathcal{C}[\ell \mapsto (\Gamma, \ell{:}\mathsf{T}^q(A'))]$ for some type A' such that provided by the premises of (H1), effectively having all potential removed. In order to establish type consistency $\mathcal{C}_0, \mathcal{B}_0 \vdash (\mathcal{H}_0, \mathcal{L}) : \mathcal{M}_0$ for the extended heap, note that existing locations are unaffected, since ℓ is fresh. For the new location ℓ, it is enough to show that $\Gamma, \ell{:}\mathsf{T}^q(A') \vdash^{q+\mathcal{B}_0(\ell)}_{0} e_1[\ell/x] : A$. This follows directly from (H1) by the Substitution Lemma 2, replacing x with the fresh name ℓ throughout.

The required compatibility $\mathcal{M}_0 \curlyvee (\Delta', \ell{:}\mathsf{T}^q(A), \Delta; \mathcal{C}_0)$ follows from premise (H3) and $\mathsf{T}^q(A) \curlyvee \{\mathsf{T}^q(A), \mathsf{T}^q(A')\}$, where the latter follows $A \curlyvee \{A, A'\}$ from the premises of (H1) again. Premise (H5) reads as

$$m + \mathtt{Klet} \geq t + (p + \mathtt{Klet}) + \Phi_{\mathcal{H}}\,\mathcal{M} + \sum \mathcal{B}$$

whereas for applying the induction hypothesis we need

$$m \geq t + p + \Phi_{\mathcal{H}_0}\,\mathcal{M}_0 + \sum \mathcal{B}_0$$

This follows in three steps: First, subtract $\mathtt{Klet}$ from both sides. Second, we have $\Phi_{\mathcal{H}_0}\,\mathcal{M}_0 = \phi(e_1[\ell/x] : A) + \Phi_{\mathcal{H}}\,\mathcal{M}$, but note that $\phi(e_1[\ell/x] : A) = 0$, since e_1 cannot be a constructor. and the potential is zero in all other cases by Def. 2; hence $\Phi_{\mathcal{H}_0}\,\mathcal{M}_0 = \Phi_{\mathcal{H}}\,\mathcal{M}$. Third, we have $\sum \mathcal{B} = \sum \mathcal{B}_0$ by definition.

Applying the induction hypothesis then yields all required conclusions directly without any alterations, except for (C1); the latter follows by transitivity of subtyping from $\mathcal{M} <: \mathcal{M}_0$ (by definition of $\mathcal{M}_0$) and $\mathcal{M}_0 <: \mathcal{M}'$ (by the induction result).

Case LETCONS*:* This case shares many similarities with the previous case; the crucial difference lies in establishing (H5) for the application of the induction hypothesis, which requires dealing with CONS directly.

Premise (H1) now instantiates as $\Gamma, \Delta' \vdash^{p+q+\mathtt{Kletcons}}_{p'} \mathsf{letcons}\ x = c(\mathbf{y})\ \mathsf{in}\ e : C$ which implies $\Delta', \ell{:}\mathsf{T}^0(A) \vdash^{p}_{p'} e[\ell/x] : C$ by the Substitution Lemma 2; and premise (H4) instantiates as $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m+\mathtt{Kletcons}}_{m'} \mathsf{letcons}\ x = c(\mathbf{y})\ \mathsf{in}\ e \Downarrow w, \mathcal{H}'$ (H4), with rule $\text{LETCONS}_{\Downarrow}$ requiring $\mathcal{H}_0, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e[\ell/x] \Downarrow w, \mathcal{H}'$ to hold, where ℓ is a fresh location and $\mathcal{H}_0 = \mathcal{H}\big[\ell \mapsto c(\mathbf{y}[\ell/x])\big]$. We intend to apply the induction hypothesis for these statements, so we must establish the required premises first. Note that we cannot invoke the induction hypothesis for the constructor, since the theorem only applies to initial expressions, but not augmented expressions.

To establish type consistency for the extended heap $\mathcal{H}_0$, we set $\mathcal{B}_0 = \mathcal{B}[\ell \mapsto 0]$, $\mathcal{M}_0 = \mathcal{M}[\ell \mapsto \mathsf{T}^0(A)]$ and $\mathcal{C}_0 = \mathcal{C}[\ell \mapsto (\Gamma, \ell{:}\mathsf{T}^0(A'))]$ for some type A' with $A \curlyvee \{A, A'\}$ provided

by the premises of (H1). Type consistency for existing locations is unaffected by these extensions, since ℓ is fresh. For the new location ℓ we require that $\Gamma, \ell{:}\mathsf{T}^0(A') \vdash^{0}_{0} c(\mathbf{y}[\ell/x]) : A$ holds, which we have from premise (H1) by the application of Substitution (Lem. 2), replacing x with ℓ throughout.

The required compatibility $\mathcal{M}_0 \between (\Delta', \ell{:}\mathsf{T}^0(A), \Delta; \mathcal{C}_0)$ follows from the premise (H3) and $\mathsf{T}^0(A) \between \{\mathsf{T}^0(A), \mathsf{T}^0(A')\}$, where the latter follows again by $A \between \{A, A'\}$ from (H1).

Premise (H5) reads as

$$m + \texttt{Kletcons} \geq t + (p + q + \texttt{Kletcons}) + \Phi_{\mathcal{H}}\,\mathcal{M} + \sum \mathcal{B}$$

By definition $\Phi_{\mathcal{H}_0}\,\mathcal{M}_0 = \phi(c(\mathbf{y}[\ell/x]) : A) + \Phi_{\mathcal{H}}\,\mathcal{M} = q + \Phi_{\mathcal{H}}\,\mathcal{M}$; furthermore, $\sum \mathcal{B}_0 = \sum \mathcal{B}$ by definition. Combining these three with the inequality before yields as required

$$m \geq t + p + \Phi_{\mathcal{H}_0}\,\mathcal{M}_0 + \sum \mathcal{B}_0$$

Applying the induction hypothesis then yields all required conclusions directly without any alterations, except for (C1); the latter follows by transitivity of subtyping from $\mathcal{M} <: \mathcal{M}_0$ (by definition of $\mathcal{M}_0$) and $\mathcal{M}_0 <: \mathcal{M}'$ (by the induction result).

Case ABS*:* The typing premise (H1) is $\Gamma \vdash^{0}_{0} \lambda x.\, e : A \xrightarrow{q} C$. The evaluation premise (H4) is $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash \lambda x.\, e \Downarrow \lambda x.\, e, \mathcal{H}$. Assume m satisfying (H5); let $\Gamma' = \Gamma$, $\mathcal{C}' = \mathcal{C}$, $\mathcal{M}' = \mathcal{M}$, $\mathcal{B}' = \mathcal{B}$ and $m' = m$ we trivially obtain (C1), (C2), (C3), (C4), (C5). It remains to show that the bound (C6) is satisfied when $m' = m$. From the premise (H5) we know

$$m \quad \geq \quad t + 0 + \Phi_{\mathcal{H}}\,\mathcal{M} + \sum \mathcal{B} \quad = \quad t + 0 + \phi(\lambda x.\, e : A \xrightarrow{q} C) + \Phi_{\mathcal{H}}\,\mathcal{M}' + \sum \mathcal{B}'$$

which follows by the Def. 2, which assigns zero potential for lambda expressions; and the above noted equalities. This already concludes the proof of the ABS case.

Case APP*:* The typing premise (H1) instantiates as $\Gamma, \ell{:}A \vdash^{p+q+\texttt{Kapp}}_{p'} e\ \ell : C$ and the evaluation premise (H4) as $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash e\ \ell \Downarrow w, \mathcal{H}''$. By inversion of rules APP and $\text{APP}_{\Downarrow}$ we obtain

$$\Gamma \vdash^{p}_{p'} e : A \xrightarrow{q} C \tag{25}$$

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash e \Downarrow \lambda x.\, e', \mathcal{H}' \tag{26}$$

$$\mathcal{H}', \mathcal{S}, \mathcal{L} \vdash e'[\ell/x] \Downarrow w, \mathcal{H}'' \tag{27}$$

By premise (H5) we assume

$$m + \texttt{Kapp} \geq t + (p + q + \texttt{Kapp}) + \Phi_{\mathcal{H}}\,\mathcal{M} + \sum \mathcal{B}$$

which we can rearrange to $m \geq (t + q) + p + \Phi_{\mathcal{H}}\,\mathcal{M} + \sum \mathcal{B}$ in order to apply the induction hypothesis for expression e, also using judgments (25) and (26). We thereby obtain $\Gamma', \mathcal{M}', \mathcal{C}', \mathcal{B}'$ and m' such that:

$$\mathcal{M} <: \mathcal{M}' \tag{28}$$

$$\Gamma' \vdash^{0}_{0} \lambda x.\, e' : A \xrightarrow{q} C \tag{29}$$

$$\mathcal{C}', \mathcal{B}' \vdash (\mathcal{H}', \mathcal{L}) : \mathcal{M}' \tag{30}$$

$$\mathcal{M}' \between (\Gamma', \ell{:}A, \Delta; \mathcal{C}') \tag{31}$$

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e \Downarrow \lambda x.\, e', \mathcal{H}' \tag{32}$$

$$m' \geq (t + q) + p' + \Phi_{\mathcal{H}'}\,\mathcal{M}' + \sum \mathcal{B}' \tag{33}$$

Applying Lem. 4 (ABS inversion) to judgement (29), we get Γ_0' such that $\Gamma_0', x{:}A \vdash^{q}_{0} e' : C$ and $\Gamma' \Cap \Gamma_0'$. By substitution (Lem. 2) we get $\Gamma_0', \ell{:}A \vdash^{q}_{0} e'[\ell/x] : C$. Compatibility for Γ_0' still holds from (31) together with the side condition $\Gamma' \Cap \Gamma_0'$ and transitivity of sharing (Lem. 6). Rearranging (33) we have $m' \geq (t+p') + q + \Phi_{\mathcal{H}'}\,\mathcal{M}' + \sum \mathcal{B}'$ and therefore we can apply the induction hypothesis to the evaluation of $e'[\ell/x]$ and obtain:

$$\mathcal{M}' <: \mathcal{M}'' \tag{34}$$
$$\Gamma'' \vdash^{0}_{0} w : C \tag{35}$$
$$\mathcal{C}'', \mathcal{B}'' \vdash (\mathcal{H}'', \mathcal{L}) : \mathcal{M}'' \tag{36}$$
$$\mathcal{M}'' \Cap (\Gamma'', \Delta; \mathcal{C}'') \tag{37}$$
$$\mathcal{H}', \mathcal{S}, \mathcal{L} \vdash^{m'}_{m''} e'[\ell/x] \Downarrow w, \mathcal{H}'' \tag{38}$$
$$m'' \geq (t+p') + 0 + \phi(w : C) + \Phi_{\mathcal{H}''}\,\mathcal{M}'' + \sum \mathcal{B}'' \tag{39}$$

From (28) and (34) and the transitivity of subtyping we conclude $\mathcal{M} <: \mathcal{M}''$. From (32) and (38) and rule $\text{APP}_{\Downarrow}$ we obtain $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m+\texttt{Kapp}}_{m''} e\,\ell \Downarrow w, \mathcal{H}''$. Statements (35), (36), (37) and (39) establish the remaining proof obligations. This concludes the proof of the APP case.

Case CONS*:* This case does not apply because the theorem applies to initial expressions only (not full expressions).

Case MATCH*:* The typing and evaluation premises are

$$\Gamma, \Delta' \vdash^{p+\texttt{Kmatch}}_{p''} \mathsf{match}\ e_0\ \mathsf{with}\ \{c_i(\mathbf{x}_i)\texttt{->}e_i\}_{i=1}^{n} : C \tag{40}$$
$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash \mathsf{match}\ e_0\ \mathsf{with}\ \{c_i(\mathbf{x}_i)\texttt{->}e_i\}_{i=1}^{n} \Downarrow w, \mathcal{H}'' \tag{41}$$

From (40) by inversion of the rule MATCH we get:

$$B = \mu X.\{\, c_i : (q_i, \mathbf{A}_i)\,\}_{i=1}^{n} \tag{42}$$
$$\Gamma \vdash^{p}_{p'} e_0 : B \tag{43}$$
$$\Delta', \mathbf{x}_i{:}\mathbf{A}_i[B/X] \vdash^{p'+q_i}_{p''} e_i : C \tag{44}$$

By inversion of rule $\text{MATCH}_{\Downarrow}$ we get $1 \leq k \leq n$ such that:

$$\mathcal{H}, \mathcal{S} \cup \textstyle\bigcup_i \{\mathbf{x}_i\} \cup \mathbf{BV}(e_i), \mathcal{L} \vdash e_0 \Downarrow c_k(\boldsymbol{\ell}), \mathcal{H}' \tag{45}$$
$$\mathcal{H}', \mathcal{S}, \mathcal{L} \vdash e_k[\boldsymbol{\ell}/\mathbf{x}_k] \Downarrow w, \mathcal{H}'' \tag{46}$$

We apply induction for expression e_0 using (43) and (45) for $m \geq t + p + \Phi_{\mathcal{H}}\,\mathcal{M} + \sum \mathcal{B}$ as given by the premise to the MATCH case and subtracting `Kmatch` on both sides. Type consistency remains unaltered and compatibility follows immediately by associativity[13] of multiset union for contexts. We obtain:

$$\mathcal{M} <: \mathcal{M}' \tag{47}$$
$$\Gamma' \vdash^{0}_{0} c_k(\boldsymbol{\ell}) : B \tag{48}$$
$$\mathcal{C}', \mathcal{B}' \vdash (\mathcal{H}', \mathcal{L}) : \mathcal{M}' \tag{49}$$
$$\mathcal{M}' \Cap (\Gamma', \Delta', \Delta; \mathcal{C}') \tag{50}$$
$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e_0 \Downarrow c_k(\boldsymbol{\ell}), \mathcal{H}' \tag{51}$$
$$m' \geq t + p' + \phi(c_k(\boldsymbol{\ell}) : B) + \Phi_{\mathcal{H}'}\,\mathcal{M}' + \sum \mathcal{B}' \tag{52}$$

[13] Treating Δ' as a part of Δ instead of Γ, in order to preserve Δ' for the second induction.

From (44) for $i = k$ together with Lemma 2 (substitution) we obtain

$$\Delta', \boldsymbol{\ell}{:}\mathbf{A}_k[B/X] \vdash^{\frac{p'+q_k}{p''}} e_k[\boldsymbol{\ell}/\mathbf{x}_k] : C \tag{53}$$

In order to apply induction for expression $e_k[\boldsymbol{\ell}/\mathbf{x}_k]$ using typing (53) and evaluation (46) we need to show that the bound (52) satisfies the subsequent premise (H5). By (42) and the definition of potential we get $\phi(c_k(\boldsymbol{\ell}) : B) = q_k$; substituting in (52) yields $m' \geq (t + p') + q_k + \Phi_{\mathcal{H}'}\mathcal{M}' + \sum\mathcal{B}'$ as required. Type consistency follows directly from the results of the previous induction. From (48) and Lem. 3 (CONS inversion) we get $\Gamma' \curlyvee \{\boldsymbol{\ell}{:}\mathbf{A}_k[B/X]\}$; together with (50) establishes compatibility. The second induction step yields:

$$\mathcal{M}' <: \mathcal{M}'' \tag{54}$$

$$\Gamma'' \vdash^{\frac{0}{0}} w : C \tag{55}$$

$$\mathcal{C}'', \mathcal{B}'' \vdash (\mathcal{H}'', \mathcal{L}) : \mathcal{M}'' \tag{56}$$

$$\mathcal{M}'' \curlyvee (\Gamma'', \Delta'; \mathcal{C}'') \tag{57}$$

$$\mathcal{H}', \mathcal{S}, \mathcal{L} \vdash^{\frac{m'}{m''}} e_k[\boldsymbol{\ell}/\mathbf{x}] \Downarrow w, \mathcal{H}'' \tag{58}$$

$$m'' \geq t + p'' + \phi(w : C) + \Phi_{\mathcal{H}''}\mathcal{M}'' + \sum\mathcal{B}'' \tag{59}$$

The required results follow by transitivity of subtyping and application of the $\text{MATCH}_{\Downarrow}$ rule. This concludes the subcase MATCH.

Case PREPAY*:* The typing premise instantiates as $\Gamma, \ell{:}\mathsf{T}^{q_0+q_1}(A) \vdash^{\frac{p+q_1}{p'}} e : C$. By inversion of the rule PREPAY we get:

$$\Gamma, \ell{:}\mathsf{T}^{q_0}(A) \vdash^{\frac{p}{p'}} e : C \tag{60}$$

Let $\mathsf{T}^r(A') = \mathcal{M}(\ell)$. By the definition of sharing and compatibility (H3) we have $\mathsf{T}^r(A') \curlyvee \{\mathsf{T}^{q_0+q_1}(A)\}$ and hence $q_0 + q_1 \geq r$. Define $k = \max(r - q_1, 0)$, and $\mathcal{M}_0 = \mathcal{M}[\ell \mapsto \mathsf{T}^k(A')]$.[14] We have $\Phi_{\mathcal{H}}\mathcal{M} = \Phi_{\mathcal{H}}\mathcal{M}_0$ because potential ignores thunk annotations. Furthermore, let $\mathcal{B}_0 = \mathcal{B}[\ell \mapsto q_1 + \mathcal{B}(\ell)]$, i.e. $\mathcal{B}_0$ is equal to $\mathcal{B}$ except for location ℓ where it increases by q_1. Assuming m as in premise (H5), it still satisfies the requirements for applying induction to (60) with the modified $\mathcal{M}_0$ and $\mathcal{B}_0$:

$$m \;\geq\; t + (p + q_1) + \Phi_{\mathcal{H}}\mathcal{M} + \sum\mathcal{B} \;=\; t + p + \Phi_{\mathcal{H}}\mathcal{M}_0 + \sum\mathcal{B}_0$$

In order to reestablish both global compatibility and type consistency for $\mathcal{M}_0$ and $\mathcal{B}_0$, note that only the global type of location ℓ changes. Assume that $\ell \notin \mathcal{L}$, since otherwise the claim is satisfied trivially. From the consistency premise (H2) we have

$$\mathcal{C}(\ell) \vdash^{\frac{r+\mathcal{B}(\ell)}{0}} \mathcal{H}(\ell) : A' \tag{61}$$

By the definition of k and $\mathcal{B}_0$, we get $k + \mathcal{B}_0(\ell) = \max(r - q_1, 0) + (\mathcal{B}(\ell) + q_1) \geq r + \mathcal{B}(\ell)$; hence by RELAX we get the required consistency for ℓ.

Compatibility remains unchanged for all locations besides ℓ; for ℓ we only need to show that the reference $\ell{:}\mathsf{T}^{q_0}(A)$ satisfies the invariant. From $q_0 + q_1 \geq r$, as noted earlier, we get $q_0 \geq r - q_1$ which, together with non-negativity of annotations, implies $q_0 \geq \max(r - q_1, 0) = k$. From this we recover compatibility for the context $\Gamma, \ell{:}A$. Since the other premises remain unchanged, we can therefore apply induction and obtain precisely the results required for the conclusion of this case.

[14] Note that it is possible to prepay a location more than once or more than necessary; hence we use truncated subtraction to ensure that the thunk annotation remains non-negative.

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

$$\frac{A = \mu X.\{\cdots \mid c : (q, \mathbf{B}) \mid \cdots\} \qquad A \curlyvee \{A, A'\} \qquad x \notin \Gamma, \Delta \qquad A'' \lhd A' \qquad \Gamma, x{:}\mathsf{T}^0(A'') \vdash^{0}_{0} c(\mathbf{y}) : A \qquad \Delta, x{:}\mathsf{T}^0(A) \vdash^{p}_{p'} e : C}{\Gamma, \Delta \vdash^{p+q+\mathtt{Kletcons}}_{p'} \mathsf{letcons}\ x = c(\mathbf{y})\ \mathsf{in}\ e : C} \ (\textsc{Letcons}\dagger)$$

Fig. 5 Revised let and letcons type rules

Case RELAX*:* By the second premise of RELAX follows $q - p \geq 0$ and thus we can choose $t' = t + q - p$. We apply the induction hypothesis to $\Gamma \vdash^{p}_{p'} e : A$ for this t'. Since RELAX is a structural rule, all statements apart from (H1) and (H5) remain unchanged. The induction thus yields all required conclusions verbatim, except for (C6). Instead, the induction yields

$$\begin{array}{lll} m' & \geq t' + p' & + \phi(w : A) + \Phi_{\mathcal{G}'}\mathcal{M}' + \sum \mathcal{B}' \\ & = (t + q - p) + p' & + \phi(w : A) + \Phi_{\mathcal{G}'}\mathcal{M}' + \sum \mathcal{B}' \\ & \geq t + q' & + \phi(w : A) + \Phi_{\mathcal{G}'}\mathcal{M}' + \sum \mathcal{B}' \end{array}$$

with the last inequality following from the last premise of the RELAX rule: $q - p \geq q' - p'$.

The few remaining cases are straightforward. We thus conclude the proof of Theorem 1.

6 Improving precision for co-recursion

To improve the precision of the analysis for co-recursive programs we can employ the technique of our previous work [29]. We now present a combined system which merges both our analysis techniques for recursion (Section 4) and co-recursion in one unified system.

6.1 Revised type rules

The combination modifies only the type rules for let- and letcons-expressions allowing lower thunk costs for self-references (Fig. 5). The intuition is that self-references in recursive definitions should already be evaluated (otherwise it would correspond to a non-productive computation), and are never assigned potential (since they are always a part of a cyclic structure). Hence, the self-reference $x{:}\mathsf{T}^q(A')$ in the context for the newly bound expression e_1 is replaced in the revised type rules by $x{:}\mathsf{T}^0(A'')$, where A'' allows lowering the thunk cost for the recursive type; this is expressed by a separate type relation $\lhd$, defined as follows.

Definition 6 $A \lhd B$ iff $A = B$ or there exists a recursive type C with free variable Y such that $A = \mu X.C[\mathsf{T}^q(X)/Y]$, $B = \mu X.C[\mathsf{T}^{q'}(X)/Y]$ and $q \leq q'$.

For example, given a suitable type N for naturals, consider the types of infinite streams $A = \mu X.\{\mathsf{cons} : (q, (N, \mathsf{T}^0(X)))\}$ and $B = \mu X.\{\mathsf{cons} : (q, (N, \mathsf{T}^1(X)))\}$, i.e. the two types are identical except that A has lower thunk cost annotation for the recursive reference X; we then have $A \lhd B$.

Note that the revised rules apply both to infinite structures (co-data) and recursive definitions (functions). For function types, $\lhd$ is simply the identity and then the LET† rule behaves as the previous LET. Thus the revised rules strictly subsume the previous ones.

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

$$\frac{\ell \notin \mathcal{L} \qquad \mathcal{H}[\ell \mapsto \widehat{e}],\mathcal{S},\mathcal{L}\cup\{\ell\} \vdash^{m}_{m'} \widehat{e} \Downarrow^{\mathrm{I}} w,\mathcal{H}'[\ell \mapsto \widehat{e}]}{\mathcal{H}[\ell \mapsto \widehat{e}],\mathcal{S},\mathcal{L} \vdash^{m+\mathtt{Kvar}}_{m'} \ell \Downarrow^{\mathrm{I}} w,\mathcal{H}'[\ell \mapsto w]} \quad (\textsc{Var}_{\Downarrow^{\mathrm{I}}})$$

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

$$\frac{\ell,\ell' \text{ are fresh} \qquad \mathcal{H}\big[\ell \mapsto c(\mathbf{y}[\ell'/x]),\, \ell' \mapsto \mathsf{ind}(\ell)\big],\mathcal{S},\mathcal{L} \vdash^{m}_{m'} e[\ell/x] \Downarrow^{\mathrm{I}} w,\mathcal{H}'}{\mathcal{H},\mathcal{S},\mathcal{L} \vdash^{m+\mathtt{Kletcons}}_{m'} \mathsf{letcons}\ x = c(\mathbf{y})\ \mathsf{in}\ e \Downarrow^{\mathrm{I}} w,\mathcal{H}'} \quad (\textsc{LetCons}_{\Downarrow^{\mathrm{I}}})$$

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

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

Fig. 6 Indirection semantics

6.2 Soundness

Proving the soundness of the revised type rules requires distinguishing self-references from ordinary ones, in particular, justifying lower costs for the former. However, the compatibility invariant of the proof of Sect. 5 requires that every use of a variable pays the cost specified in its global type. To reconcile these requirements, we can apply the approach of our previous work [29]: use an intermediate operational semantics to establish soundness of the type system; and show a cost correspondence between the intermediate and original semantics. For brevity, we do not repeat the proof of correspondence between the two operational semantics here.

The intermediate operational semantics uses *indirections* for self-references. An indirection behaves similarly to a variable (i.e. it refers a heap expression). However, evaluation of an indirection will *not* trigger the evaluation of a thunk: instead, it succeeds immediately if and only if the referred expression is already in whnf; thus indirections are always cost-free.

We augment the syntax of initial expressions and results of evaluations with an extra form for indirections:

$$\begin{array}{rcl} \widehat{e} & ::= & \cdots \mid \mathsf{ind}(x) \\ w & ::= & \lambda x.e \mid c(\mathbf{x}) \mid \mathsf{ind}(x) \end{array}$$

Figure 6 presents the intermediate operational 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 as the semantics of Section 3. Note that indirections are allowed both as full expressions $\widehat{e}$ or results w; they may also occur in heaps $\mathcal{H}$ or $\mathcal{H}'$.

The $\textsc{Whnf}_{\Downarrow^{\mathrm{I}}}$ 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 $\textsc{Var}_{\Downarrow^{\mathrm{I}}}$ rule is identical to the previous one. The revised rule for let-expressions $\textsc{Let}_{\Downarrow^{\mathrm{I}}}$ 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 $\textsc{Whnf}_{\Downarrow \mathrm{I}}$, $\textsc{App}_{\Downarrow \mathrm{I}}$ and $\textsc{Match}_{\Downarrow \mathrm{I}}$ 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.

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

We also introduce a typing rule IND for indirections:

$$\frac{A \between \{A, A'\} \qquad A'' \lhd A'}{x{:}\mathsf{T}^q(A) \vdash^{0}_{0} \mathsf{ind}(x) : A''} \tag{IND}$$

The following lemma establishes a useful property of indirection typing, namely, that the result type shares to itself and hence has zero potential.

Lemma 8 *If $\Gamma \vdash \mathsf{ind}(x) : B$ then $\phi(\widehat{e} : B) = 0$.*

Proof The typing derivation of $\Gamma \vdash \mathsf{ind}(x) : B$. must result from an application of rule IND followed by zero or more structural rules. From rule IND we get $\phi(\widehat{e} : A'') = 0$ from the side-conditions $A \between \{A, A'\}$ and $A'' \lhd A'$ together with Corollary 1. For the structural rule SUBTYPE the conclusion type B must be a subtype of the hypothesis A, hence by Lem. 7 if $\phi(\widehat{e} : A) = 0$ then $\phi(\widehat{e} : B) = 0$. All the remaining structural rules do not modify the conclusion type, thus the result follows by induction.

This will be needed in the soundness proof solely for establishing well-typing of intermediate heap configurations (since indirections do not occur in the initial expression). The type rule is similar to VAR except for allow lower costs both on the judgment and in self-references to a recursive type; and requires zero potential[15]. Consequently, we extend Def. 2 (potential) for indirections by $\phi(\mathsf{ind}(x) : B) = 0$, since indirections are never assigned any potential.

The soundness statement for the revised system remains exactly the same as Theorem 1, except for replacing the evaluation relation $\Downarrow$ by $\Downarrow^{\mathrm{I}}$; still assuming a terminating evaluation as before. We therefore just refer to the original statements, if these remain unchanged.

Theorem 2 (Revised soundness) *Let $t \in \mathbb{Q}$ with $t \geq 0$ be fixed, but arbitrary; similarly, let Δ be an arbitrary context. If statements* (H1) (H2) (H3) *and the revised hypothesis*

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash e \Downarrow^{\mathrm{I}} w, \mathcal{H}' \tag{H4$'$}$$

hold, then for all $m \in \mathbb{N}$ such that (H5) *holds, there exists m', Γ', $\mathcal{C}'$, $\mathcal{B}'$ and $\mathcal{M}'$ such that follows* (C1) (C2) (C3) (C4) (C6) *and the revised conclusion*

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e \Downarrow^{\mathrm{I}} w, \mathcal{H}' \tag{C5$'$}$$

Proof (Sketch) The proof is largely similar to the previous one so we present a sketch that highlights the differences in crucial cases.

[15] This requirement is new, since [29] did not consider potential.

Case LET*:* The typing and evaluation premises (H1) and (H4′) instantiate as

$$\Gamma,\Delta' \vdash^{p+\mathtt{Klet}}_{p'} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : C \tag{62}$$

$$\mathcal{H},\mathcal{S},\mathcal{L} \vdash^{m+\mathtt{Klet}}_{m'} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow^{\mathrm{I}} w,\mathcal{H}' \tag{63}$$

From (62) and Substitution we get $\Delta',\ell{:}\mathsf{T}^q(A) \vdash^{p}_{p'} e_2[\ell/x] : C$. By (63) and inversion of rule $\textsc{Let}_{\Downarrow\mathrm{I}}$ we get $\mathcal{H}_0,\mathcal{S},\mathcal{L} \vdash^{m}_{m'} e_2[\ell/x] \Downarrow w,\mathcal{H}'$ where ℓ,ℓ' are fresh locations and for $\mathcal{H}_0 = \mathcal{H}\big[\ell \mapsto e_1[\ell'/x], \ell' \mapsto \mathsf{ind}(\ell)\big]$. We intend to apply the induction hypothesis for the evaluation of $e_2[\ell/x]$, so we must establish the required premises first. We define:

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

where the types A',A'' are provided by the premises of (H1) with $A \curlyvee \{A,A'\}$ and $A'' \lhd A'$. Note that $A' \curlyvee \{A',A'\}$ and $A'' \curlyvee \{A'',A''\}$ are a trivial consequences. In order to establish type consistency for the extended heap, note that existing locations are unaffected, since ℓ and ℓ' are fresh. For the new location ℓ, it is enough to show that $\Gamma,\ell'{:}\mathsf{T}^0(A'') \vdash^{q+\mathcal{B}_0(\ell)}_{0} e_1[\ell'/x] : A$. This follows by inversion of (H1) and by substitution of x with the fresh name ℓ'. To establish consistency for ℓ' it suffices to show $\ell{:}\mathsf{T}^q(A') \vdash^{\mathcal{B}_0(\ell')}_{0} \mathsf{ind}(\ell) : A''$ which follows from the IND type rule and $A'' \lhd A'$ and $A' \curlyvee \{A',A''\}$ as already noted above.

The compatibility for ℓ requires that $\mathsf{T}^q(A) \curlyvee \{\mathsf{T}^q(A), \mathsf{T}^q(A')\}$ which trivially follows from $A \curlyvee \{A,A'\}$. For ℓ' compatibility requires just $\mathsf{T}^0(A'') \curlyvee \{\mathsf{T}^0(A'')\}$ which is immediate.

Note $\Phi_{\mathcal{H}_0}\mathcal{M}_0 = \phi(e_1[\ell/x] : A) + \phi(\mathsf{ind}(\ell) : A'') + \Phi_{\mathcal{H}}\mathcal{M}$; by definition, $\phi(e_1[\ell/x] : A) = \phi(\mathsf{ind}(\ell) : A'') = 0$ (since neither of these expressions are constructors). Hence $\Phi_{\mathcal{H}_0}\mathcal{M}_0 = \Phi_{\mathcal{H}}\mathcal{M}$. Similarly, we have $\sum\mathcal{B} = \sum\mathcal{B}_0$ (because the new locations have zero balance). The remaining steps for this case are then identical to the previous proof of Theorem 1.

Case MATCH*:* The typing and evaluation premises (H1) and (H4′) instantiate as

$$\begin{aligned}
&\Gamma,\Delta' \vdash^{p+\mathtt{Kmatch}}_{p''} \mathsf{match}\ e_0\ \mathsf{with}\ \{c_i(\mathbf{x}_i)\text{->}e_i\}_{i=1}^n : C\\
&\mathcal{H},\mathcal{S},\mathcal{L} \vdash \mathsf{match}\ e_0\ \mathsf{with}\ \{c_i(\mathbf{x}_i)\text{->}e_i\}_{i=1}^n \Downarrow^{\mathrm{I}} w,\mathcal{H}''
\end{aligned} \tag{64}$$

From (64) by inversion of the rule MATCH we get:

$$B = \mu X.\{\, c_i : (q_i,\mathbf{A}_i)\,\}_{i=1}^n \tag{65}$$

$$\Gamma \vdash^{p}_{p'} e_0 : B \tag{66}$$

$$\Delta',\ \mathbf{x}_i{:}\mathbf{A}_i[B/X] \vdash^{p'+q_i}_{p''} e_i : C \tag{67}$$

By inversion of rule $\textsc{Match}_{\Downarrow\mathrm{I}}$ we get $1 \le k \le n$ such that:

$$\mathcal{H},\mathcal{S} \cup \big(\textstyle\bigcup_i\{\mathbf{x}_i\} \cup \mathrm{BV}(e_i)\big),\mathcal{L} \vdash e_0 \Downarrow^{\mathrm{I}} u,\mathcal{H}' \tag{68}$$

$$\mathcal{H}'@u = c_k(\boldsymbol{\ell}) \tag{69}$$

$$\mathcal{H}',\mathcal{S},\mathcal{L} \vdash e_k[\boldsymbol{\ell}/\mathbf{x}_k] \Downarrow^{\mathrm{I}} w,\mathcal{H}'' \tag{70}$$

We apply induction for expression e_0 using (66) and (68) for $m \ge t + p + \Phi_{\mathcal{H}}\mathcal{M} + \sum\mathcal{B}$ as given by (H5) and subtracting `Kmatch` on both sides. Type consistency remains unaltered and compatibility follows immediately by associativity of multiset union for contexts. We

obtain several important statments, of which the following differ as compared to the previous proof:

$$\Gamma' \vdash^{0}_{0} u : B \tag{71}$$

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e_0 \Downarrow^{\mathrm{I}} u, \mathcal{H}' \tag{72}$$

$$m' \geq t + p' + \phi(u : B) + \Phi_{\mathcal{H}'} \mathcal{M}' + \sum \mathcal{B}' \tag{73}$$

From (67) for $i = k$ together with Lemma 2 (substitution) we obtain

$$\Delta', \boldsymbol{\ell}{:}\mathbf{A}_k[B/X] \vdash^{p'+q_k}_{p''} e_k[\boldsymbol{\ell}/\mathbf{x}_k] : C \tag{74}$$

In order to apply induction for expression $e_k[\boldsymbol{\ell}/\mathbf{x}_k]$ using typing (74) and evaluation (70) we need to show that the bound (73) satisfies the subsequent premise (H5). By (69), u is either a constructor or an indirection referencing a constructor. In the first case, the proof proceeds exactly as in the proof of Theorem 1. Otherwise, assume that u is an indirection; applying Lem. 8 to the typing (71) yields $\phi(c_k(\ell) : B) = 0$. Therefore by (65) and the definition of sharing (Fig. 2), we get that $q_k = 0$. Also, by definition of potential for indirections, $\phi(u : B) = 0$; hence:

$$\begin{aligned} m' &\geq t + p' + \phi(u : B) + \Phi_{\mathcal{H}'} \mathcal{M}' + \sum \mathcal{B}' \\ &= t + p' + 0 + \Phi_{\mathcal{H}'} \mathcal{M}' + \sum \mathcal{B}' \qquad = t + p' + q_k + \Phi_{\mathcal{H}'} \mathcal{M}' + \sum \mathcal{B}' \end{aligned}$$

This allows us to apply the induction hypothesis to (67) once more, just as in the previous proof of the MATCH case, then proceeding exactly as before.

The remaining cases are similar to the corresponding ones in the proof of Theorem 1; this concludes the proof sketch for Theorem 2.

7 Related work

Pioneering work on cost analysis for higher-order functional programs with lazy evaluation was done by Sands [21,22]. This approach used *evaluation contexts* [30] and *projections* [32] to capture the degree of evaluation of data structures; it can be used to aid manual reasoning about program costs but is not directly automatable for use in a compiler or static analysis tool.

The use of amortisation for complexity analysis of imperative data structures goes back to Tarjan [26]. Okasaki [16] extended this technique to functional data structures, in particular, showing how lazy evaluation can be used to combine amortised bounds with persistence. The prepay rule in our system plays an analogous role to Okasaki's notion of "discharging debits". The main difference is that our analysis is automatic and applies to complete programs rather than data structures in isolation; this is achieved by assigning potential in a type-directed way and by a sub-structural type system (e.g. an affine system with the explicit sharing rule) to ensure that potential is used at most once.

Benzinger [1] developed an automated complexity analysis for functional programs synthesized using the Nuprl proof system. However, this system deals only with call-by-name rather than lazy evaluation/call-by-need. The system uses type-based decomposition and polynomialization to express the costs of higher-order arguments and generates recurrence equations; these are solved using the *Mathematica* computer algebra system (CAS). Using a general-purpose CAS allows deriving non-linear (and even non-polynomial) bounds, but

obtaining simplified results may require interaction with an expert user (unlike the linear programming techniques we employ). Thus, the approach is not as automatable as the one presented here.

Several authors have proposed *symbolic profiling* approaches, where programs are annotated with additional cost parameters. For example, Wadler [31] has used a state monad to count reduction costs through a tick-counting operation. Building on this work, Danielsson [4] developed a library for expressing complexity costs using dependent types in Agda. Our use of thunk cost annotations to express the cost of lazy data structures is inspired by this work; the idea of pre-paying costs also appears here (although in Danielsson's system it is a manually-introduced term rather than a structural type rule whose effect is automatically decided later by the linear solver). Unlike our system, no potential is associated to data structures (ensuring its safe use would require a linear or affine type system); instead the costs of recursive functions must be expressed in terms of explicit size annotations rather than changes in potential. Finally, unlike the work presented here, this system allows checking complexity but not automatic inference.

Hughes and Pareto [10] have combined a *sized type system* with *cost effects* to statically check space bounds for a small functional language. The system allows checking (but not inferring) stack and heap resource bounds of an abstract machine using dynamically-allocated *regions*. Vasconcelos [28] developed a size and cost analysis for *Core Hume*, a minimal functionally-inspired language, using abstract interpretation techniques to compute a safe approximation to size and cost bounds automatically. Both these approaches considered only strict and first-order languages.

Hofmann and Jost [8] proposed a type system for static prediction of heap resources for a strict first-order functional language; this was only later recognized as an *automatic* amortised analysis. The key contribution that enabled automation was the assignment of potential in a type-directed way, using weights for constructors. Potential functions are thus expressed as linear combinations of the number of uses of each constructor and allows finding suitable coefficients using a standard linear programming solver. The approach was extended to higher-order functions and limited polymorphism in later work [12]. Our earlier work [24] builds on this line of research extending it to a lazily-evaluated language rather than a strict one; a subsequent development [29] focused on improving the precision of analysis for the co-recursive fragment of the language.

An important extension of the technique to infer multivariate polynomial bounds was achieved by Hoffmann et al. in [5]. It is crucial to note that a standard linear programming solver is still sufficient to infer asymptotically tight polynomial bounds for a large number of programs; a realistic case study examining OCaml standard libraries can be found in a preprint available from Hoffmann's homepage. Furthermore, Hoffmann and Shao [6] showed how to apply the technique to parallel first-order programs.

8 Conclusions and further work

This paper has described a type-based automatic analysis that is capable of inferring upper bounds on the costs for lazily evaluated programs. This allows us to understand and reason about resource usage of non-strict functional languages, which has historically been a key obstacle to their wider use. The main contributions of this paper are:

1. the combination of two previous type-based analyses [24, 29] that increases the range of programs that can be effectively analysed;

2. the generalization to a parametric cost model;
3. a detailed soundness proof based on a novel approach for accounting potential of heaps per location rather than recursively.

The key aspects of our approach are amortisation (especially its application to avoid duplication of thunk evaluation costs) and the tracking of self-references. This is essential to model the *graph reduction* techniques that are used in typical lazy functional language implementations. Our experimental results show accurate and effective bounds on the resource usage for a number of non-trivial examples, including higher-order recursive and co-recursive programs on finite and infinite lists.

Directions for further work: first, we have not considered any issues of resource deallocation. Previous work in the strict setting [8, 12] indicate that this should be possible. A key issue is how to expose deallocation in a syntax-directed way, e.g. using regions [27].

Second, we have only considered a simply-typed system here. This means that our analysis is *monovariant*, i.e. distinct uses are aliased by a single (simple) annotated type. This suffices for analysing whole-programs, provided we re-run the analysis for different uses for best precision (e.g. duplicating definitions as in the `sumWithFibs` example of Sect. 4.5). This could easily be avoided by explicitly capturing and copying constraints for each use as was done in [11]. Extending our system with *type and effect polymorphism* would make the analysis fully modular. While technically more difficult, we believe that the standard approaches of type and effect systems [25] should be applicable to our setting. A modular analysis would also pave the way for dealing with a larger programs in a real programming language such as Haskell.

Third, only linear cost functions were considered here. Work by Hoffmann et al. for strict languages [5] shows that it is possible to extend amortised analysis approaches to cover multivariate polynomial cost bounds, extending the utility of the work. Adopting these techniques to lazy evaluation ought to be relatively straightforward, albeit technically complex.

Lazy evaluation can be seen as allowing a controlled form of mutation in a functional setting; it would therefore be interesting to explore whether the approach taken here for assigning potential to heaps would also apply to amortised cost analyses for mutation in the strict setting, e.g. ML-like references or mutable arrays [7]. Finally, work shows that the used techniques extend to parallel programs [6] as well, so it would be exciting to combine this with the lazy setting as well.

References

1. Benzinger, R.: Automated higher-order complexity analysis. Theor. Comput. Sci. **318**(1-2), 79–103 (2004). DOI 10.1016/j.tcs.2003.10.022. URL `http://dx.doi.org/10.1016/j.tcs.2003.10.022`
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. Ph.D. thesis, Worcester College, University of Oxford (2010)
4. Danielsson, N.A.: Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In: Proc. POPL 2008: 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, pp. 133–144. ACM, San Francisco, USA (2008)
5. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate Amortized Resource Analysis. In: 38th Symp. on Principles of Prog. Langs. (POPL'11), pp. 357–370 (2011)
6. Hoffmann, J., Shao, Z.: Automatic static cost analysis for parallel programs. In: Proc. of the 24th European Symposium on Programming (ESOP'15), pp. 132–157. Springer (2015)
7. Hoffmann, J., Shao, Z.: Type-based amortized resource analysis with integers and arrays. Journal of Functional Programming **25**, e17 (2015)

8. Hofmann, M., Jost, S.: Static Prediction of Heap Space Usage for First-Order Functional Programs. In: Proc.of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 185–197. ACM, ACM Press, New Orleans, LA, USA (2003)
9. Hughes, J.: Why Functional Programming Matters. Computer Journal **32**(2), 98–107 (1989)
10. Hughes, J., 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), *ACM Sigplan Notices*, vol. 34, pp. 70–81. ACM Press (1999)
11. Jost, S.: Automated Amortised Analysis. Ph.D. thesis, LMU Munich
12. Jost, S., Loidl, H.W., Hammond, K., Hofmann, M.: Static Determination of Quantitative Resource Usage for Higher-Order Programs. In: Proc. ACM Symp. on Prin. of Prog. Langs. (POPL), pp. 223–236 (2010)
13. Jost, S., Loidl, H.W., Hammond, K., Scaife, N., Hofmann, M.: "Carbon Credits" for Resource-Bounded Computations Using Amortised Analysis. In: A. Cavalcanti, D.R. Dams (eds.) FM 2009: Formal Methods, *Lecture Notes in Computer Science*, vol. 5850, pp. 354–369. Springer, Heidelberg (2009)
14. La Encina, A., Peña, R.: Proving the Correctness of the STG Machine. In: Proc. IFL '01: Impl. of Functional Langs., Stockholm, Sweden, Sept. 24-26, 2001, pp. 88–104. Springer LNCS 2312 (2002)
15. Launchbury, J.: A Natural Semantics for Lazy Evaluation. In: Proc. POPL '93: Symp. on Princ. of Prog. Langs., pp. 144–154 (1993)
16. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press, Cambridge, UK (1998)
17. Peyton Jones, S., Marlow, S., Reid, A.: The STG runtime system (revised). Available from `http://www.haskell.org/ghc/documentation.html` (1999)
18. Peyton Jones (ed.), S., Augustsson, L., Boutel, B., Burton, F., Fasel, J., Gordon, A., Hammond, K., Hughes, R., Hudak, P., Johnsson, T., Jones, M., Peterson, J., Reid, A., Wadler, P.: Report on the Non-Strict Functional Language, Haskell (Haskell98). Tech. rep., Yale University (1999)
19. Pierce, B.C.: Advanced Topics in Types and Programming Languages. The MIT Press (2004)
20. Reistad, B., Gifford, D.: Static Dependent Costs for Estimating Execution Time. In: Proc. ACM Conf. on Lisp and Functional Programming, pp. 65–78. ACM Press, Orlando, Florida, June 27–29 (1994)
21. Sands, D.: Calculi for Time Analysis of Functional Programs. Ph.D. thesis, Imperial College, University of London (1990)
22. Sands, D.: Complexity Analysis for a Lazy Higher-Order Language. In: Proc. ESOP '90: European Symposium on Programming, Copenhagen, Denmark, Springer LNCS 432, pp. 361–376 (1990)
23. Sestoft, P.: Deriving a Lazy Abstract Machine. J. Functional Programming **7**(3), 231–264 (1997)
24. Simões, H., Jost, S., Vasconcelos, P., Florido, M., Hammond, K.: Automatic amortised analysis of dynamic memory allocation for lazy functional programs. In: Proceedings of the 17th ACM SIGPLAN international conference on Functional Programming (ICFP'12), pp. 165–176. ACM (2012)
25. Talpin, J.P., Jouvelot, P.: The type and effect discipline. In: A. Scedrov (ed.) Proc. 1992 Logics in Computer Science Conf., vol. 111, pp. 245–271. IEEE (1992)
26. Tarjan, R.E.: Amortized computational complexity . SIAM Journal on Algebraic and Discrete Methods **6**(2), 306–318 (1985)
27. Tofte, M., Talpin, J.P.: Region-based memory management. Information and Computation **132**(2), 109–176 (1997). URL `citeseer.ist.psu.edu/tofte97regionbased.html`
28. Vasconcelos, P.B.: Space cost analysis using sized types. Ph.D. thesis, School of Computer Science, University of St Andrews (2008). URL `http://hdl.handle.net/10023/564`
29. Vasconcelos, P.B., Jost, S., Florido, M., Hammond, K.: Type-Based Allocation Analysis for Co-recursion in Lazy Functional Languages. In: Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings, pp. 787–811 (2015)
30. Wadler, P.: Strictness Analysis aids Time Analysis. In: Proc. POPL '88: ACM Symp. on Princ. of Prog. Langs., pp. 119–132 (1988)
31. Wadler, P.: The Essence of Functional Programming. In: Proc. POPL '92: ACM Symp. on Principles of Prog. Langs., pp. 1–14 (1992)
32. Wadler, P., Hughes, J.: Projections for Strictness Analysis. In: Proc. FPCA'87: Intl. Conf. on Functional Prog. Langs. and Comp. Arch., Springer LNCS 274, pp. 385–407 (1987)