Submission Draft

Appendix for
Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs

Hugo Simoes
Pedro Vasconcelos
Mario Florido
Universidade do Porto, Porto, Portugal
{hrsimoeh,pbv,amf}@dcc.fc.up.pt

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

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

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

This is the appendix for the paper “Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs”, containing all detailed proofs.

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

This paper makes the following novel contributions:

a) we present the first successful attempt, of which we are aware, to produce an automatic, type-based, static analysis of resource bounds for lazy evaluation;

b) we introduce a cost model for heap allocations for a minimal lazy functional language based on Launchbury’s natural semantics for lazy evaluation [30], and use this as the basis for developing a resource analysis;

c) we present a proof sketch of the soundness of our analysis with respect to the cost-instrumented semantics; and

d) we provide results from a prototype implementation† to show the applicability of our analysis to some non-trivial examples.

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

2. A Cost Model for Lazy Evaluation
Our cost model is built on Sestoft’s revision [40] of Launchbury’s natural semantics for lazy evaluation [30]. Launchbury’s semantics forms one of the earliest and most widely-used operational accounts of lazy evaluation for the λ-calculus. De la Encina and Peña-Mari [13, 14] subsequently proved that the Spineless Tagless

* The complete proof is included in Appendix A.
† http://www.dcc.fc.up.pt/~pbv/cgi/aalazy.cgi
G-Machine [24] is complete and almost sound w.r.t. one of Sestoft’s abstract machines. We therefore have a high degree of confidence that the cost model for lazy evaluation developed here is not just theoretically sound, but also that it could, in principle, be extended to model real implementations of lazy evaluation.

2.1 Syntax

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

\[ e ::= x \mid \lambda x. e \mid e e \mid \text{let } x = e_1 \text{ in } e_2 \mid \text{letcon}s x = c(y) \text{ in } e \mid \text{match } e_0 \text{ with } c(\vec{x}) \rightarrow e_1 \text{ otherwise } e_2 \]

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

\[ \vec{e} ::= e \mid c(\vec{e}) \]

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

\[ w ::= \lambda x. e \mid c(\vec{e}) \]

In the remainder of this paper we will use lowercase letters \(x, y\) for bound variables in initial expressions and \(\ell, \vec{\ell}\) for “fresh” variables (designated locations) that are introduced through evaluation of let- and letcon-expressions.

2.2 Cost-instrumented operational semantics

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

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

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

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

§ Due to de La Encina and Peña-Marí [13].
Definition 2.1 (Freshness). A variable \( x \) is fresh in judgement \( \mathcal{H}, \mathcal{S}, \mathcal{L} \vdash \overset{\ell}{\varphi} w, \mathcal{N} \) if \( x \) does not occur in either \( \text{dom}(\mathcal{H}) \), \( \mathcal{L} \) or \( \mathcal{S} \) or it does not occur in both in either \( \ell \) or \( \text{ran}(\mathcal{H}) \).

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

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

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

Consider the expression below, which includes a divergent term:

\[
2.3 \text{ Example: call-by-need versus call-by-value/call-by-name}
\]

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

A crucial difference between classic amortised analysis [35, 42] and our automatic type-based amortised analysis is that we assign potential on a per-reference basis. The advantage is that our potential can now reflect how often a data structure is processed, regardless of whether it is persistent or ephemeral. The disadvantage is that (fully evaluated) cyclic data can only be assigned either zero or infinite potential, and that our type system becomes slightly more complicated, in that it now requires structural type rules.

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

3.1 Annotated types and contexts

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

\[
A, B, C ::= X \mid A \rightarrow^\gamma B \mid T^\nu(A) \mid \mu X.\{c_1 : (q_{\text{nil}}, \hat{B}_1) \ldots ; c_n : (q_{\text{nil}}, \hat{B}_n)\}
\]

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

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

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

### 3.2 Sharing and Subtyping

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

### 3.3 Typing judgements

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

---

**Figure 2. Sharing Relation**

\[
\begin{align*}
\gamma(#A \mid \emptyset) \\
\gamma(X \mid X, \ldots, X) \\
B_i = \mu X. \{ c_1 : (q_{i1}, B_{i1}) \} \cdots \{ c_m : (q_{im}, B_{im}) \} \\
\gamma(A_j \mid B_{1j}, \ldots, B_{nj}) p_j \geq \sum_{i=1}^{n} q_{ij} \quad (1 \leq i \leq n, 1 \leq j \leq m) \\
\gamma(\mu X. \{ c_1 : (p_1, A_1) \} \cdots \{ c_m : (p_m, A_m) \} \mid B_1, \ldots, B_n) \\
\gamma(A_i \mid A) \quad \gamma(B_i \mid B) \quad q_i \geq q \quad q_i - q \geq q' - q' \quad (1 \leq i \leq n) \\
\gamma(A_i \mid A_1, \ldots, A_n) \quad q_i \geq q \quad q_i - q \geq q' - q' \quad (1 \leq i \leq n) \\
\gamma(T^p_\theta(\mathcal{A}) \mid T^p_\theta(A_1), \ldots, T^p_\theta(A_n)) \\
\gamma(A_i \mid B_{1j}, \ldots, B_{nj}) m = |\bar{X}| = |\bar{B}| \quad (1 \leq i \leq n, 1 \leq j \leq m) \\
\gamma(A_j \mid B_{1}, \ldots, B_{n}) \\
\gamma(\Gamma \mid \emptyset) \\
\gamma(\Gamma \mid \Delta) \\
\gamma(x : A, \Gamma \mid x : B_1, \ldots, x : B_n, \Delta)
\end{align*}
\]
allow the analysis to be relaxed in various ways: WEAK allows the
introduction of an extra hypothesis in the typing context; RELAX
allows argument costs to be relaxed; PREPAY allows (part of)
the cost of a thunk to be paid for, so reducing the cost of further uses;
SUPERTYPE and SUBTYPE allow supertyping in a hypothesis and
subtyping in the conclusion, respectively; finally, SHARE allows
the use of sharing to split potential in a hypothesis.

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

### 3.4 Worked examples

We now present type derivations for the examples from Section 2.3
in order to illustrate how the type rules of Figures 3 and 4 model
the costs of our operational semantics. Recall example (2.1) which
demonstrates that unneeded redexes are not reduced (i.e., that the
operational semantics succeeds and does not require one heap cell
for allocating the thunk named by $\lambda x. \lambda y. y$):

\[
\Gamma \vdash T^\lambda_p \lambda x. \lambda y. y
\]

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

\[
\mathcal{H}, S, L \vdash z = z \in (\lambda x. \lambda y. y) z \downarrow \lambda y. y, \mathcal{H}'
\]

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

\[
\emptyset \vdash z = z \in (\lambda x. \lambda y. y) z : T^{\lambda}_{q'}(B) \xrightarrow{\Delta} B
\]
The implementation allows some trivial syntactical extensions to the term language, namely, multiple constructor branches in match-expressions and omission of the default alternative. Also, as in the ML and Haskell languages, we require that data constructors are associated with a single data type; this ensures that the use of the CONS rule is syntax-directed.

It remains to explain how to decide when to use the structural rules from Figure 4. We use SHARE to split the context \( \Gamma \) into two \( \Gamma_1, \Gamma_2 \) when typing sub-expressions (e.g. when typing \( e_1 \) and \( e_2 \) in let \( x = e_1 \) in \( e_2 \)); note that this does not lose precision unnecessarily, since the unused types can be assigned zero potential. We consequently delegate the task of finding the best assignment (i.e. one yielding the least cost) to the LP solver. We use \( \text{WEAK} \) depending on the remaining free variables in the sub-expressions. We allow \( \text{PREPAY} \) to be used for the body \( e_2 \) of any let-expression let \( x = e_1 \) in \( e_2 \). Once again, this does not lose precision because the rule can be used to pay any part of the cost (possibly zero); hence, we allow the LP solver to decide how to use it for each individual thunk in order to achieve an overall optimal solution. Finally, we allow the use of \( \text{RELAX} \) at every node of the derivation and \( \text{SUBTYPE} \) at the application rule (to enforce compatibility between the function and its argument) and at the MATCH rule (to obtain a compatible result type). This may generate more constraints and variables for intermediate types than necessary; the resulting increase in size has negligible cost for current LP-solvers (in fact, all our examples were solved by a typical desktop computer in less than one second). Hoffman and Jost have shown that the LP problems that are generated for the eager amortised analysis exhibit regularities that allow lower complexity than general LP solving [19]. We conjecture that this should also be true for our analysis.

4. Experimental results

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

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

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

c) Finally, we feed the linear constraints to a standard linear programming solver\footnote{We use the GLPK library: \url{http://www.gnu.org/software/glpk}.} with the objective to minimize the overall expression cost. Any solution gives rise to a valid annotated typing derivation, and hence to a concrete formula bounding evaluation costs in terms of the program’s input data sizes.

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

It remains to explain how to decide when to use the structural rules from Figure 4. We use SHARE to split the context \( \Gamma \) into two \( \Gamma_1, \Gamma_2 \) when typing sub-expressions (e.g. when typing \( e_1 \) and \( e_2 \) in let \( x = e_1 \) in \( e_2 \)); note that this does not lose precision unnecessarily, since the unused types can be assigned zero potential. We consequently delegate the task of finding the best assignment (i.e. one yielding the least cost) to the LP solver. We use \( \text{WEAK} \) depending on the remaining free variables in the sub-expressions. We allow \( \text{PREPAY} \) to be used for the body \( e_2 \) of any let-expression let \( x = e_1 \) in \( e_2 \). Once again, this does not lose precision because the rule can be used to pay any part of the cost (possibly zero); hence, we allow the LP solver to decide how to use it for each individual thunk in order to achieve an overall optimal solution. Finally, we allow the use of \( \text{RELAX} \) at every node of the derivation and \( \text{SUBTYPE} \) at the application rule (to enforce compatibility between the function and its argument) and at the MATCH rule (to obtain a compatible result type). This may generate more constraints and variables for intermediate types than necessary; the resulting increase in size has negligible cost for current LP-solvers (in fact, all our examples were solved by a typical desktop computer in less than one second). Hoffman and Jost have shown that the LP problems that are generated for the eager amortised analysis exhibit regularities that allow lower complexity than general LP solving [19]. We conjecture that this should also be true for our analysis.
The analysis can now yield an informative type. If we abbreviate the type of lists of $A$ as:

$$L(q_c, q_n, A) \overset{def}{=} \mu X. \{ \text{Cons} : (q_c, (T^0_0(A), T^0_0(X))) | \text{Nil} : (q_n, ()) \}$$

then we obtain:

$$\text{rev\_acc} : T^0_0(L(0, 0, A)) \overset{0}{\rightarrow} T^0_0(L(1, 0, A)) \overset{0}{\rightarrow} L(0, 0, A)$$

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

### 4.2 Functional queues

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

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

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

- $\text{mkqueue} = \lambda f r \rightarrow \text{match} f \text{ with}
  \begin{align*}
  \text{Nil()} & \rightarrow \text{let} f’ = \text{reverse} r \\
  \text{in} & \text{letcons} r’ = \text{reverse} r’ \\
  \text{in} & \text{cons} x r’ \\
  \text{otherwise} & \text{let} v = f i \text{ in} f v : B,
  \end{align*}$
- $\text{dequeue} = \lambda q \rightarrow \text{match} q \text{ with}
  \begin{align*}
  \text{Pair}(f, r) & \rightarrow \text{letcons} r’ = \text{Cons}(x, r) \\
  \text{in} & \text{mkqueue} f r’ \\
  \text{otherwise} & \text{let} v = f i \text{ in} f v : B,
  \end{align*}$
Lemma 5.1

By induction on the height of derivation of $\Gamma$. We now present some auxiliary proof lemmas for our type system.

5.2 Global Types, Contexts and Balance

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

$$C_{\ell} = \{\ell_1 \mapsto \Gamma_1, \ldots, \ell_n \mapsto \Gamma_n\} \overset{\text{def}}{=} (\Gamma_1, \ldots, \Gamma_n)_{\ell}$$

We also extend subtyping to global types in the natural way, namely $M \ll M'$ if and only if $\text{dom}(M) \subseteq \text{dom}(M')$ and for all $\ell \in \text{dom}(M)$ we have $M(\ell) \ll M'(\ell)$. This relation will be used to assert that the potential assigned to global types is always non-increasing during execution. Furthermore, we introduce an auxiliary balance (or lazy potential) mapping $B$ from locations to non-negative rational numbers. This keeps track of the partial costs of thunks that have been paid in advance by applications of the PREPAY rule. Note that these auxiliary mappings are needed only in the soundness proof of the analysis for bookkeeping purposes, but are not part of the operational semantics—in particular, they do not incur runtime costs.

5.3 Potential

We define the potential of an augmented expression with respect to a heap and an annotated type. The potential of unevaluated expressions (i.e. thunks) and $\lambda$-expressions is always zero. For data constructors, the potential is obtained by summing the type annotation with the (recursive) potential contributed by each of the arguments. Note that for cyclic data structures, the potential is only defined if all the type annotations of all nodes encountered along a cycle are zero (the overall potential must therefore also be zero).

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

Equation (5.2) extends the definition to typing contexts in the natural way. Equation (5.3) defines potential for global contexts, but considers only thunks that are not under evaluation. Finally, (5.4)
5.4 Consistency and Compatibility

defines a convenient shorthand notation for a similar summation

\[
\phi_\Delta(\bar{e}:A) \triangleq \begin{cases} 
  p + \sum_i \phi_\Delta(\mathcal{H}(\ell_i):B_i[A/X]) & \text{if } A = \mu X.\{\cdots 1c:(p, B)]\cdots \text{ and } \bar{c} = c(\bar{\ell}) \\
  0 & \text{otherwise}
\end{cases}
\]

\[\phi_\Delta(\Gamma) \triangleq \sum\{ \phi_\Delta(\mathcal{H}(x):A) \mid x:A \in \Gamma \} \]

\[\Phi^k(\mathcal{C}) \triangleq \sum\{ \phi_\Delta(\mathcal{C}(\ell)) \mid \ell \in \text{dom}(\mathcal{H}) \text{ and } \ell \notin \mathcal{L} \text{ and } \mathcal{H}(\ell) \text{ is not a whnf} \} \]

\[\Phi^k(\mathcal{B}) \triangleq \sum\{ \mathcal{B}(\ell) \mid \ell \in \text{dom}(\mathcal{H}) \text{ and } \ell \notin \mathcal{L} \text{ and } \mathcal{H}(\ell) \text{ is not a whnf} \} \]

5.5 Soundness of the proof system

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

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

\[\Gamma \vdash^\omega e : A\]  
(1.A)

\[\mathcal{C}, \mathcal{B} \vdash^\text{Mem} (\mathcal{H}, \mathcal{L}) : \mathcal{M}\]  
(1.B)

\[\mathcal{Y}(\mathcal{M}[\Gamma, \Theta], \mathcal{C})\]  
(1.C)

\[\mathcal{H}, \mathcal{S}, \ell \vdash e \downarrow w, \mathcal{H}'\]  
(1.D)

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

\[ m \geq t + p + \phi_\Delta(\Gamma) + \phi_\Delta(\Theta) + \Phi^k(\mathcal{C}) + \Phi^k(\mathcal{B}) \]  
(1.E)
there exist $m', \Gamma', \ell', B'$ and $M'$ such that

$$\begin{align*}
m & \leq m' + t + p' + \phi_M(w; A) + \phi_{\Theta}(\Theta) + \Phi_{M'}(\ell') + \Phi_{\ell'}(B') \\
\Gamma' & \vdash t : A (1.F) \\
\ell' & \vdash \text{M} (1.G) \\
\ell, B' & \vdash \text{M} (1.H) \\
\ell, \ell, B' & \vdash \text{N} (1.I) \\
\ell, \ell, B' & \vdash \text{N} (1.J) \\
\ell, \ell, B' & \vdash \text{N} (1.K)
\end{align*}$$

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

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

Case $\text{VAR}$: The typing premise $\ell : T^p_p(A) \vdash t : A$ is an axiom. By inversion of the evaluation premise we obtain $\ell, \ell, L \cup \{\ell\} \vdash H(\ell) \vdash w, \ell'$. In order to apply induction to the evaluation of the thunk $H(\ell)$, we take the typing context from the hypothesis of type consistency for the location $\ell$. We apply induction to a typing with the global type $M(\ell)$ rather than the local type $T^p_p(A)$ in the local context. This gives us a stronger conclusion with a context that we can then split using Lemma 5.6 to justify type consistency for the heap update and the local context answer for the answer. Finally, we require an auxiliary result to ensure that if the update introduces a cycle, the locations on the cycle can be assigned a type with zero potential (Lemma A.4 in Appendix A).

Case $\text{LET}$: The typing premise is $\Gamma, \Delta \vdash e_1 : C$ and evaluation premise gives $\Theta_0, S, L, \ell, \ell \vdash e_2[\ell/x] \vdash w, \ell'$ where $\Theta_0 = H[\ell \mapsto e_1[\ell/x]]$ is the heap extended with a new location $\ell$ and thunk. To apply induction to the evaluation of $e_2[\ell/x]$ we reestablish the consistency to the new location $\ell$; this is done using $\Gamma$ from the typing hypothesis together with an idempotent type for self-references to $\ell$. Applying induction then yields all required conclusions.

Case $\text{MATCH}$: The typing premise is:

$$\begin{align*}
\Gamma, \Delta & \vdash e_0 \text{ match } c(\ell) \rightarrow e_1 \text{ otherwise } e_2 : C
\end{align*}$$

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

6. Related Work

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

Bakewell and Runciman [6] have previously defined an operational semantics for Core Haskell that gives time and space execution costs in terms of Sestoft’s semantics for his Mark 1 abstract machine. The work has subsequently been extended to give a model that can be used to determine space leaks by comparing the space usage for two evaluators using a bisimulation approach [5]. Gustavsson and Sands [17] have similarly defined a space-improvement relation that guarantees that some optimisation can never lead to asymptotically worse space behaviour for call-by-need programs and Moran and Sands [33] have defined an improvement relation for call-by-need programs that can be used to determine whether one terminating program improves another in all possible contexts. Finally, like de la Encina and Peña-Mari, Mountjoy [34] derived an operational semantics for the Spineless Tagless G-Machine from the natural semantics of Launchbury and Sestoft, including poly-applicative A-expressions. The main differences between these approaches are that de la Encina and Peña-Mari correct some mistakes in Mountjoy’s presentation, that they provide correctness proofs, that their semantics correctly deals with partial applications in the Spineless Tagless G-Machine, that they deal with partial applications as normal forms, and that they consider two distinct implementation variants, based on push/enter versus apply/eval. Our own work differs from this body of earlier work in that we not only provide an operational semantics to model lazy evaluation, but also provide a corresponding cost semantics from which we derive a static analysis to automatically determine upper bounds on the memory requirements of lazily evaluated programs.

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

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

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

The amortised analysis approach has been previously studied by a number of authors, but has never previously been used to automatically determine the costs of lazy evaluation. Tarjan [42] first described amortised analysis, but as a manual technique. Okasaki [35] subsequently described how Tarjan’s approach could be applied to (lazy) data structures, but again as a manual technique. While there has subsequently been significant interest in the use of amortised analysis for automatic resource usage analysis, using an advanced per-reference potential, none of this newer work, however, considers lazy evaluation. Hofmann and Jost [19] were the first to develop an automatic amortised analysis for heap consumption, exploiting a difference metric similar to that used by Crary and Weirich [11] (the latter, however, only check bounds, and therefore do not perform an automatic static analysis of the kind we require); Hofmann et al. have extended their method to cover a comprehensive subset of Java, including imperative updates, inheritance and type casts [20, 21]; Shkaravska et al. [41] subsequently considered heap consumption inference for first-order polymorphic lists; and Campbell [9] has developed the ideas of depth-based and temporary credit uses to give better results for stack usage. Hofmann et al. [18] achieved another breakthrough by extending the technique to infer multivariate polynomial cost functions, still only requiring efficient LP solving.

Finally, several authors have recently studied analyses for heap usage in eager languages, without considering lazy evaluation. For example, Albert et al. [2] present a fully automatic, live heap-space analysis for an object-oriented bytecode language with a scoped-memory manager, and have subsequently extended this to consider garbage collection [3], but, unlike our system, data-dependencies cannot be expressed. Braberman et al. [8] infer polynomial bounds on the live heap usage for a Java-like language with automatic memory management, but do not cover general recursive methods. Finally, Chu et al. [10] present a linearly-bounded heap and stack analysis for a low-level (assembler) language with explicit (de)-allocation, but do not cover lazy evaluation or high-level functional programming constructs.

7. Conclusions and Further Work

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

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

References


A. Detailed Proofs

This appendix includes the detailed proof of the soundness theorem for our analysis. We begin with an auxiliary definition and lemma that will be used in the proof of soundness of the analysis in the case VAR for updating a location with a wnhf.

A.1 Minor Lemmas

Lemma A.1 ($<:$ is a partial order).

Proof. Straightforward by induction on the type structure and the definition of sharing (Figure 2).

Lemma A.2. If $\gamma(A | A, A')$ then $\gamma(A' | A', A')$ as well.

A.2 Idempotent cycles

Definition A.3 (Reachability). The one-step reachability relation $\ell \sim_1\ell'$ between two locations $\ell, \ell'$ in a heap $H$ holds if and only if $H(\ell) = c(H')$ and $\ell' \in K$. The many-step reachability relation $\sim^*_\ell$ is defined as the transitive closure of the one-step reachability relation.

Note that reachability only traverses constructors, but not unevaluated thunks or lambda expressions. This mimics the definition of potential (Def. 5.5).

The following lemma shows that, in a consistent configuration, locations within cycles can be assigned global types with zero potential.

Lemma A.4 (Idempotent Cycles). Let $(H, L)$ be a heap configuration consistent with global types, contexts and balance $\Gamma, c, B,$ that is, such that $c, B \vdash \text{typing}(H, L) : \Gamma$ and $\gamma(H|\Gamma, c)$. Then there exist $c', \Gamma'$ such that $\Gamma \prec \Gamma'$ with $c', B \vdash \text{typing}(H, L) : \Gamma'$ and $\gamma(H' | \Gamma', c')$ such that for all $\ell$ with $\ell \sim^*_\ell$ we have $\gamma(H(\ell) | \Gamma'(\ell), \Gamma'(\ell))$ as well.

Proof. Consider a cycle consisting of the locations $\ell_0, \ell_1, \ldots, \ell_n$ with $\ell_i \sim_1 \ell_{i+1}$ and $\ell_{n+1} = \ell_0$. By Def. 3.1 (reachability) each $H(\ell_i)$ must be a constructor of the form $c_i \ldots (\ldots c_i)$, where $c_i$ is a constructor, and the length of the typing derivation does decrease.

The proof is by induction on the lengths of the derivations of (1.D) and (1.A) ordered lexicographically, with the derivation of the evaluation rule 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 derivation does decrease, allowing an induction over lexicographically ordered lengths of both derivations. We proceed by case analysis of the rule used in premise (1.A).

Case VAR: We have $[c : \ell : A]$ from the typing hypothesis (1.A). From the compatibility hypothesis (1.C) we then get $\gamma(N(\ell) | T^p_{m}(A), \Theta_{\ell}, c_{\ell})$ which implies $N(\ell) = T^p_{\ell}(\tilde{A})$ and $\gamma(A | \tilde{A})$ for some types $\tilde{A}, \tilde{A}$ and annotations $q, q'$ such that $\gamma(q, q' : T^p_{\ell} : p)$.

The evaluation premise (1.D) reads as $\{a, \tilde{a} \in \ell \notin \nu w, \tilde{a} \tilde{a} \nu w \tilde{a}$ for some intermediate heap $H$; by inversion of the only applicable evaluation rule VAR, we obtain $\ell \notin \nu L$ and

$$\nu H \cup \{a, \tilde{a} \notin \nu w, H$$

(A.4)

By (1.B) for location $\ell$ we get $\ell \notin \nu L$, by the premise of the VAR rule. The remaining cases are then LOC 1 and LOC 2, which apply according to whether $H(\ell)$ is in wnhf or not, respectively.
If $\mathfrak{H}(\ell)$ is in whnf: The evaluation $(A.4)$ terminates immediately by \textsc{WHNF} and we have $w = \mathfrak{H}(\ell)$ and $\mathfrak{H} = \mathfrak{H}' = \mathfrak{H}'[\ell \mapsto w]$, i.e. the update is without effect. By \textsc{Loc1} we get $\mathcal{C}(\ell) \vdash w : \bar{A}$. By $\mathcal{Y}(\bar{A} \mid A, \bar{A})$ established earlier and Lem. 5.4 we get $\mathcal{Y}(\mathcal{C}(\ell) \mid \Gamma_1', \Gamma_2')$ and $\Gamma_1' \vdash w : A$ as required for $(1.G)$, as well as $\Gamma_2' \vdash w : A$. Let $M' = M[\ell \mapsto T_{\mathcal{I}}'(\bar{A})]$ and $C' = C[\ell \mapsto \Gamma_2']$. By the previous results together with $(1.B)$ we get $\mathcal{C}', \mathcal{B} \vdash_{\text{Mem}} (\mathcal{H}, \ell) : M'$ as required for $(1.H)$. From the compatibility premise $(1.C)$ together with $\mathcal{Y}(\mathcal{C}(\ell) \mid \Gamma_1', \Gamma_2')$ established earlier we can conclude $\mathcal{Y}(M', \Gamma_1', \Theta, C')$ as required for $(1.J)$. Conclusion $(1.J)$ with $m' = m$ follows directly from an application of rule \textsc{WHNF}. It is known to show that the bound $(1.K)$ is satisfied for the choice $m' = m$. Our proof obligation is:

$$t + p + \phi_{\mathcal{Y}}(\ell; T_{\mathcal{I}}'(A)) + \phi_{\mathcal{Y}}(\Theta) + \Phi_{\mathcal{H}}(E) + \Phi_{\mathcal{H}}(B) \geq t + p' + \phi_{\mathcal{Y}}(w; A) + \phi_{\mathcal{Y}}(\Theta) + \Phi_{\mathcal{Y}}(E') + \Phi_{\mathcal{Y}}(B)$$

The above inequality holds because $p \geq q \geq q' \geq p'$ by type consistency for location $\ell$ as seen earlier; $\phi_{\mathcal{Y}}(\ell; T_{\mathcal{I}}'(A)) = \phi_{\mathcal{Y}}(w; A)$ by Def. 5.5 (potential); and $\Phi_{\mathcal{H}}(E') = \Phi_{\mathcal{H}}(E)$ because $E'$ differs from $E$ only for $\ell$ which is in whnf, and therefore its context does not contribute to the potential.

If $\mathfrak{H}(\ell)$ is not in whnf: By \textsc{Loc2} we get $\mathcal{C}(\ell) \vdash_{\text{Mem}} \mathfrak{H}(\ell) : \bar{A}$. Let $M' = M[\ell \mapsto T_{\mathcal{I}}'(\bar{A})]$ and $C' = C[\ell \mapsto \emptyset]$. We observe that $E', B \vdash_{\text{Mem}} (H, \mathcal{L} \cup \{l\}) : M'$ must hold by hypothesis $(1.B)$ together with the case for \textsc{Loc3} for location $\ell$. Furthermore $\mathcal{Y}(M', \ell, C', \ell')$ holds, since for all $\ell'$ we have $\mathcal{Y}(\ell, \ell') \cup E' \cup C' \ell'$ by definition.

We will now apply the induction hypothesis to the evaluation of $\mathfrak{H}(\ell)$ with type $\bar{A}$. We first show that $m$ can be chosen as required for the induction; the proof obligation is:

$$m \geq t + (q + \mathcal{B}(\ell)) + \phi_{\mathcal{Y}}(\mathcal{E}(\ell)) + \phi_{\mathcal{Y}}(\Theta) + \Phi_{\mathcal{H}}(E) + \Phi_{\mathcal{H}}(B)$$

(A.5)

Starting from the hypothesis (1.E) we obtain:

$$m \geq t + p + \phi_{\mathcal{Y}}(\ell; T_{\mathcal{I}}'(A)) + \phi_{\mathcal{Y}}(\Theta) + \Phi_{\mathcal{H}}(E) + \Phi_{\mathcal{H}}(B)$$

$$\geq t + q + 0 + (\phi_{\mathcal{Y}}(\mathcal{E}(\ell)) + \phi_{\mathcal{Y}}(\Theta) + \Phi_{\mathcal{H}}(E)) + (\Phi_{\mathcal{H}}(B) + \mathcal{B}(\ell))$$

$$= t + (q + \mathcal{B}(\ell)) + \phi_{\mathcal{Y}}(\mathcal{E}(\ell)) + \phi_{\mathcal{Y}}(\Theta) + \Phi_{\mathcal{H}}(E) + \Phi_{\mathcal{H}}(B)$$

The inequality follows, since $q \leq p$ from above; $\phi_{\mathcal{Y}}(\ell; T_{\mathcal{I}}'(A)) = 0$ because $\mathfrak{H}(\ell)$ is not in whnf; $\Phi_{\mathcal{H}}(E) = \Phi_{\mathcal{Y}}(\ell; T_{\mathcal{I}}'(A))$ by definition; and $\Phi_{\mathcal{H}}(B) = B(\ell) + \Phi_{\mathcal{Y}}(\ell; T_{\mathcal{I}}'(A))$ by definition.

We can now apply the induction hypothesis to the evaluation of $\mathfrak{H}(\ell)$ with type $\bar{A}$ and obtain:

$$\mathcal{M}' \vdash_{\mathcal{M}''}$$

$$\Gamma' \vdash_{\mathcal{M}''} w : \bar{A}$$

$$\mathcal{C}', \mathcal{B} \vdash_{\text{Mem}} (\mathcal{H}', \mathcal{L} \cup \{\ell\}) : \mathcal{M}''$$

$$\mathcal{Y}(\mathcal{M}'', \Gamma_1', \Theta, \mathcal{C}')$$

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \cup \{\ell\} \vdash_{\text{Mem}} \mathcal{H}(\ell) \Downarrow w, \mathcal{H}$$

(A.10)

By applying the induction hypothesis to the global type, we obtained a stronger typing $(A.7)$ for the resulting whnf as well. We now recover the required typing for $\bar{A}$ by the lemma for splitting contexts and the remaining potential associated through $\bar{A}$ allows us to establishment consistency for the remaining aliases. So by $\mathcal{Y}(\bar{A} \mid A, \bar{A})$ from above, $(A.7)$ and Lem. 5.4 we get $\mathcal{Y}(\Gamma_1' \mid \Gamma_1, \Gamma_2')$ and $\Gamma_1' \vdash w : \bar{A}$ as required for $(1.G)$, as well as $\Gamma_2' \vdash w : \bar{A}$; this together with $(A.8)$ and the case Loc1 of Def. 5.8 gives us $\mathcal{Y}(\ell; \ell \mapsto \Gamma_2'); \mathcal{B} \vdash_{\text{Mem}} (\mathcal{H}'[\ell \mapsto w], \mathcal{C}'; \ell') : \mathcal{M}'$ as required for $(1.H)$.

Conclusion $(1.F)$ follows by the transitivity of subtyping from $\mathcal{M} : \mathcal{M}''$ by the definition of $\mathcal{M}$. From $(A.9)$ we have $\mathcal{Y}(\mathcal{M}'', \Gamma_1', \Gamma_2', \Theta, \mathcal{C}')$ which by definition is equivalent to $\mathcal{Y}(\mathcal{M}', \Gamma_1', \Theta, \mathcal{C}'[\ell \mapsto \Gamma_2'])$, as required for $(1.I)$. Conclusion $(1.J)$ follows directly from $(A.10)$ by application of \textsc{Var}. It remains to show that $m'$ obtained from $(1.I)$ satisfies the requirements of $(1.K)$; our proof obligation is:

$$t + q' + \phi_{\mathcal{Y}}(w; \bar{A}) + \phi_{\mathcal{Y}}(\Theta) + \Phi_{\mathcal{Y}}(E) + \Phi_{\mathcal{Y}}(B') \geq$$

$$t + p' + \phi_{\mathcal{Y}}(w; A) + \phi_{\mathcal{Y}}(\Theta) + \Phi_{\mathcal{Y}}(E'[\ell \mapsto \Gamma_2']) + \Phi_{\mathcal{Y}}(B')$$

(A.12)

We first argue that we can assume without loss of generality that the potentials above are all defined (i.e. finite): these would be undefined only if the update $\mathfrak{H}'[\ell \mapsto w]$ introduces new cycles. In this case we apply Lem. A-4 (Idempotent Cycles) and obtain new global contexts and global types that still satisfy the three conclusions $(1.F)$, $(1.H)$ and $(1.I)$ proved earlier. Furthermore, any new cycles must include the updated location $\ell$, for which the refined global type assigns zero potential by Lem. 5.4. That implies $(A.12)$ is then an equality, since $\phi_{\mathcal{Y}}(w; A), \phi_{\mathcal{Y}}(w; A), \phi_{\mathcal{Y}}(w; A), \phi_{\mathcal{Y}}(w; A)$ and $\Phi_{\mathcal{Y}}(w; A)$ must then all be zero, and likewise $\Phi_{\mathcal{Y}}(E)[\ell \mapsto \Gamma_2']$ by definition.

In the remaining case where the update did not create a cycle, we have $\phi_{\mathcal{Y}}(w; \bar{A}) = \phi_{\mathcal{Y}}(w; A) = \phi_{\mathcal{Y}}(w; A)$, as required for $(1.J)$ with $m' = m$. Thus it remains to be shown that

$$\phi_{\mathcal{Y}}(w; \bar{A}) \geq \Phi_{\mathcal{Y}}(E)[\ell \mapsto \Gamma_2'] - \Phi_{\mathcal{Y}}(E)[\ell \mapsto \Gamma_2']$$

(A.13)
which follows by the compatibility concluded earlier, and applying Lem. 5.7 for $T^A_y(A) \prec \Delta$ to conclude the proof of the VAR case.

**Case** Let: The premises instantiate as $\Gamma, \Delta \vdash \ell \rightarrow e_1 \rightarrow C \rightarrow C \rightarrow e_2$, which implies $\Delta, \ell : T^0_y(A) \downarrow e_2[\ell/x] : C$ through the Substitution Lemma 5.1; and $\Phi, \Sigma, \ell \vdash \ell \rightarrow e_1 \rightarrow C \rightarrow C \rightarrow e_2$, requiring $\Phi, \Sigma, \ell \vdash e_2[\ell/x] : C$, which establishes the required type consistency for $C$. This follows from the induction hypothesis for these statements, 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 executed at this point, but just stored within the heap.

In order to establish type consistency for the extended memory $\mathcal{H}_0$, we set $B_0 = 0, \mathcal{M}_0 = \mathcal{M}[\ell \mapsto T^0_y(A)]$ and $\mathcal{E}_0 = \mathcal{E}[\ell \mapsto \ell, e_1[\ell/x]]$ for some idempotent type $A'$ with $\gamma(A | A, A')$. By definition of $\mathcal{E}_{A,A}$, the premises instantiate now as $\Gamma, \ell \vdash T^0_y(A)$, $\Sigma, \ell \vdash \ell \rightarrow e_1[\ell/x] : A$. The required global compatibility $\gamma(\mathcal{H}_0 | \Delta, \ell \rightarrow e_1[\ell/x])$ follows from (1.C) and (1.D) for the application of the Substitution Lemma 5.1, requiring $\ell \rightarrow e_1[\ell/x]$ such that $\ell$ is fresh and for $\mathcal{H}_0 = \mathcal{H}[\ell \mapsto e_1[\ell/x]]$, we intend to apply the induction hypothesis for these statements, so we must establish the required premises first. Note that we do not invoke the induction hypothesis for the subterm $e_1[\ell/x]$, since it is not executed at this point, but just stored within the heap.

In the case that $e_1[\ell/x] = 0$ in whnf, the required global compatibility $\gamma(\mathcal{H}_0 | \Delta, \ell \rightarrow e_1[\ell/x])$ follows from (1.C) and $\gamma(T^0_y(A) | T^0_y(A), T^0_y(A'))$, where the latter follows $\gamma(A | A, A')$ from the premises of (1.A) also.

**Premise (1.E)** reads as

$$m + 1 \geq \ell + 1 + p + \phi_{\Delta}(\ell, \Delta) + \phi_{\Theta}(\Theta) + \Phi_{c\ell}(\ell) + \Phi_{\Sigma}(B)$$

By our earlier observations, $\mathcal{H}_0(\ell)$ is either a $\lambda$-expression or not in whnf, hence $\phi_{\Delta}(\ell, T^0_y(A)) = 0$; by subtyping $\phi_{\Delta}(\ell, T^0_y(A)) = 0$ and thus $\phi_{\Delta}(\ell) = \phi_{\Delta}(\mathcal{E}_0(\ell))$ by definition of $\mathcal{E}_0$, and hence $\Phi_{c\ell}(\mathcal{E}_0) \leq \phi_{\Delta}(\ell) + \Phi_{\Sigma}(B)$ (note that this is an equality if $\mathcal{H}_0(\ell)$ is not in whnf, and $\Gamma$ is minimal, and strict inequality if $\mathcal{H}_0(\ell)$ is a $\lambda$-expression and $\phi_{\Delta}(\Gamma) > 0$); furthermore $\Phi_{c\ell}(B) = \Phi_{\Sigma}(\mathcal{E}_0)$ by definition. Combining these three with the inequality before yields as required

$$m \geq \ell + p + \phi_{\Delta}(\ell, T^0_y(A)) + \phi_{\Delta}(\Theta) + \Phi_{c\ell}(\mathcal{E}_0) + \Phi_{\Sigma}(B)$$

since the other statements are unaffected by the fresh $\ell$ extending $\mathcal{H}$ to $\mathcal{H}_0$.

Applying the induction hypothesis then yields all required conclusions directly without any alterations, except for (1.F) which follows by the transitivity of subtyping and $\mathcal{M} \prec \mathcal{H}_0$ by definition.

**Case** LetCons: This case shares many similarities with the previous case; the crucial difference lies in establishing (1.E) for the application of the induction hypothesis, which requires dealing with CONS directly.

The premises instantiate now as $\Gamma, \Delta \vdash \ell \rightarrow e_1 \rightarrow C \rightarrow C \rightarrow e_2$, which implies $\Delta, \ell : T^0_y(A) \downarrow e_2[\ell/x] : C$ by the Substitution Lemma 5.1; and $\Phi, \Sigma, \ell \vdash \ell \rightarrow e_1 \rightarrow C \rightarrow C \rightarrow e_2$, requiring $\Phi, \Sigma, \ell \vdash e_2[\ell/x] : C$, which is either a $\lambda$-term or not.

For the new location $\ell$ we require that $\Gamma, \ell : T^0_y(A'), \Sigma, \ell \vdash \ell \rightarrow e_1[\ell/x] : A$ holds. The required location $\ell$ may be in whnf and below the turnstile as required by (LOC1). Note that instead of $\Gamma, \ell \rightarrow e_1[\ell/x], \Sigma, \ell \vdash \ell \rightarrow e_1[\ell/x]$ for the new location $\ell$.

By definition of $\mathcal{E}_{A,A}$, the premises instantiate now as $\Gamma, \ell \vdash T^0_y(A)$, $\Sigma, \ell \vdash \ell \rightarrow e_1[\ell/x] : A$. The required global compatibility $\gamma(\mathcal{H}_0 | \Delta, \ell \rightarrow e_1[\ell/x])$ follows from (1.C) and $\gamma(T^0_y(A) | T^0_y(A), T^0_y(A'))$, where the latter follows $\gamma(A | A, A')$ from the premises of (1.A) again.

**Premise (1.E)** reads as

$$m + 1 \geq \ell + 1 + p + \phi_{\Delta}(\ell, \Delta) + \phi_{\Theta}(\Theta) + \Phi_{c\ell}(\ell) + \Phi_{\Sigma}(B)$$

By our earlier observations, $\mathcal{H}_0(\ell)$ is either a $\lambda$-expression or not in whnf, hence $\phi_{\Delta}(\ell, T^0_y(A)) = 0$; by subtyping $\phi_{\Delta}(\ell, T^0_y(A)) = 0$ and thus $\phi_{\Delta}(\ell) = \phi_{\Delta}(\mathcal{E}_0(\ell))$ by definition of $\mathcal{E}_0$, and hence $\Phi_{c\ell}(\mathcal{E}_0) \leq \phi_{\Delta}(\ell) + \Phi_{\Sigma}(B)$ (note that this is an equality if $\mathcal{H}_0(\ell)$ is not in whnf, and $\Gamma$ is minimal, and strict inequality if $\mathcal{H}_0(\ell)$ is a $\lambda$-expression and $\phi_{\Delta}(\Gamma) > 0$); furthermore $\Phi_{c\ell}(B) = \Phi_{\Sigma}(\mathcal{E}_0)$ by definition. Combining these three with the inequality before yields as required

$$m \geq \ell + p + \phi_{\Delta}(\ell, T^0_y(A)) + \phi_{\Delta}(\Theta) + \Phi_{c\ell}(\mathcal{E}_0) + \Phi_{\Sigma}(B)$$

since the other statements are unaffected by the fresh $\ell$ extending $\mathcal{H}$ to $\mathcal{H}_0$.

Applying the induction hypothesis then yields all required conclusions directly without any alterations, except for (1.F) which follows by the transitivity of subtyping and $\mathcal{M} \prec \mathcal{H}_0$ by definition.
Case ABS: The typing premise (1.A) is $\Gamma \vdash^p_\ell \lambda x.e : A \xrightarrow{\phi} C$. The evaluation premise (1.D) is $\mathcal{H}, S, \mathcal{L} \vdash \lambda x.e \Downarrow \lambda x.e, \mathcal{H}$. Assume $m$ satisfying (1.E); let $\Gamma'' = \Gamma, \ell' = \ell, \mathcal{M}' = \mathcal{M}, \mathcal{B}' = \mathcal{B}$ and $m' = m$ we trivially obtain (1.F), (1.G), (1.H), (1.I), (1.J). It remains to show that the bound (1.K) is satisfied when $m' = m$. From the premise (1.E) we know

$$m \geq t + \phi_{\beta}(\Gamma) + \phi_{\beta}(\Theta) + \Phi_{\nu}(E) + \Phi_{\nu}(B)$$

(A.14)

By inversion of rule ABS we obtain $\gamma(\Gamma | \Gamma, \Gamma)$ which by Lemma 5.6 implies $\phi_{\beta}(\Gamma) = 0$. By Def. 5.5 (potential) we also obtain $\phi_{\beta}(\lambda x.e : A \xrightarrow{\phi} C) = 0$, therefore $\phi_{\beta}(\Gamma) = \phi_{\beta}(\lambda x.e : A \xrightarrow{\phi} C)$; substituting in (A.14) gives

$$m \geq t + \phi_{\beta}(\lambda x.e : A \xrightarrow{\phi} C) + \phi_{\beta}(\Theta) + \Phi_{\nu}(E) + \Phi_{\nu}(B)$$

as required. This concludes the proof of the ABS case.

Case APP: The typing and evaluation premises (1.A) and (1.D) instantiate as $\Gamma, \ell : A \xrightarrow{p,q} e \ell : C$ and $\mathcal{H}, S, L \vdash e \ell : w, \mathcal{H}'$, respectively. By inversion of rules APP and APP1 we obtain

$$\mathcal{H}, S, L \vdash e' \ell' : w, \mathcal{H}''$$

(A.17)

By premise (1.E) we assume

$$m \geq t + p + q + \phi_{\beta}(\Gamma, \ell : A) + \phi_{\beta}(\Theta) + \Phi_{\nu}(E) + \Phi_{\nu}(B)$$

(A.18)

Inequality (A.18) shows that the bound for $m$ satisfies the requirements for applying induction for expression $e$ using judgments (A.15) and (A.16); we obtain $\Gamma', M', e', B'$ and $m'$ such that:

$$\Gamma' \vdash^p_\ell \lambda x.e' : A \xrightarrow{\phi} C$$

(A.19)

$$\mathcal{H}', S', L \vdash e' \ell' : \mathcal{H}''$$

(A.20)

$$\Gamma', B' \vdash_{\mu_{\text{MEM}}} (\mathcal{H}', L) : M'$$

(A.21)

$$\gamma(M' | \Gamma', \ell : A, \Theta, e')$$

(A.22)

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

(A.23)

$$m' \geq t + p' + \phi_{\beta}(\lambda x.e' : A \xrightarrow{\phi} C) + \phi_{\beta}(\ell : A, \Theta) + \Phi_{\nu}(E') + \Phi_{\nu}(B')$$

(A.24)

By Lemma 5.3 (ABS inversion) applied to judgement (A.20) we can assume without loss of generality that $\gamma(\Gamma' | \Gamma', \Gamma')$ and $\Gamma' : A \xrightarrow{p,q} e' : C$; applying Lemma 5.1 (substitution) we get

$$\Gamma', e : A \xrightarrow{p,q} e'[\ell'/x] : C$$

(A.25)

In order to apply the induction hypothesis to $e'[\ell'/x]$ it remains to show that the bound (A.24) for $m'$ satisfies the premise (1.E). By $\gamma(\Gamma' | \Gamma', \Gamma')$ established earlier and Lemma 5.6 (potential splitting) we know $\phi_{\beta}(\Gamma') = 0$ and therefore $\phi_{\beta}(\Gamma, \ell : A) = \phi_{\beta}(\ell : A)$. By Def. 5.5 (potential) we know $\phi_{\beta}(\lambda x.e' : A \xrightarrow{\phi} C) = 0$; substituting in (A.24) gives:

$$m' \geq t + q + p' + \phi_{\beta}(\lambda x.e' : A \xrightarrow{\phi} C) + \phi_{\beta}(\ell : A, \Theta) + \Phi_{\nu}(E') + \Phi_{\nu}(B')$$

(A.26)

$$= t + q + p' + \phi_{\beta}(\ell : A, \Theta) + \Phi_{\nu}(E') + \Phi_{\nu}(B')$$

(A.27)

$$= t + p' + q + \phi_{\beta}(\Gamma', \ell : A) + \phi_{\beta}(\Theta) + \Phi_{\nu}(E') + \Phi_{\nu}(B')$$

(A.28)

Hence we are in condition to apply the induction or $e'[\ell'/x]$ and obtain:

$$\mathcal{M}' \vdash \mathcal{M}'''$$

(A.29)

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

(A.30)

From (A.19) and (A.26) and the transitivity of subtyping we conclude $\mathcal{M} \vdash \mathcal{M}'''$. From (A.23) and (A.30) and rule APP3 we obtain $\mathcal{H}, S, L \vdash e \ell : w, \mathcal{H}'''$. Equations (A.27), (A.28), (A.29) and (A.31) establish the remaining proof obligations. This concludes the proof of the APP case.

Case CONS: This case cannot occur because the theorem applies only to initial expressions (not augmented expressions).
**Case MATCH**: The typing and evaluation premises are

\[ \Gamma, \Delta \vdash_{\text{pr}} e_0 \text{ with } c(\bar{x}) \rightarrow e_1 \text{ otherwise } e_2 : C \quad (A.32) \]

\[ \mathcal{H}, S, \mathcal{L} \vdash e_0 \text{ with } c(\bar{x}) \rightarrow e_1 \text{ otherwise } e_2 \Downarrow w, \mathcal{H}'' \quad (A.33) \]

From (A.32) by inversion of the rule MATCH we get:

\[ B = \mu X. \{ \ldots \mathcal{L} : (q, \bar{A}) \ldots \} \quad (A.34) \]

\[ \Gamma \vdash_{\text{pr}} e_0 : B \quad (A.35) \]

\[ \Delta, x_1 : A_1[B/X], \ldots, x_n : A_n[B/X] \vdash_{\text{pr}} e_1 : C \quad (A.36) \]

\[ \Delta \vdash_{\text{pr}} e_2 : C \quad (A.37) \]

We proceed by case analysis on the rule used for evaluation (A.33). For the case MATCH\_0 we obtain by inversion:

\[ \mathcal{H}, S \cup BV(e_1), \mathcal{L} \vdash e_0 \Downarrow c(\bar{y}), \mathcal{H}' \quad (A.38) \]

\[ \mathcal{H}', S, \mathcal{L} \vdash e_1[\bar{y}/\bar{x}] \Downarrow w, \mathcal{H}'' \quad (A.39) \]

From (A.36) together with Lemma 5.1 (substitution) we obtain

\[ \Delta, \ell_1 : A_1[B/X], \ldots, \ell_n : A_n[B/X] \vdash_{\text{pr}} e_1[\bar{y}/\bar{x}] : C \quad (A.40) \]

Let \( m \) be such that

\[ m \geq t + p + \phi_{\mathcal{H}}(\Delta, \Delta) + \phi_{\mathcal{H}}(\Theta) + \Phi_{\mathcal{H}}(\Theta') + \Phi_{\mathcal{H}}(\Theta') \]

\[ = t + p + \phi_{\mathcal{H}}(\Gamma) + \phi_{\mathcal{H}}(\Delta, \Theta) + \Phi_{\mathcal{H}}(\Theta') + \Phi_{\mathcal{H}}(\Theta') \quad (A.41) \]

We are now in condition to apply the induction hypothesis for expression \( e_0 \) using (A.43) and (A.38) and obtain:

\[ M < M' \quad (A.42) \]

\[ \Gamma' \vdash_{\text{pr}} c(\bar{y}) : B \quad (A.43) \]

\[ \mathcal{E}', \mathcal{B}' \vdash_{\text{Match}} (\mathcal{E}', \mathcal{L}) : M' \quad (A.44) \]

\[ \gamma(M' \mid (\Gamma', \Delta, \Theta), \mathcal{E}') \quad (A.45) \]

\[ \mathcal{H}', S, \mathcal{L} \vdash_{\text{pr}} e_0 \Downarrow c(\bar{y}), \mathcal{H}' \quad (A.46) \]

\[ m' \geq t + p' + \phi_{\mathcal{H}'}(c(\bar{y})), B) + \phi_{\mathcal{H}'}(\Delta, \Theta) + \Phi_{\mathcal{H}'}(\Theta') + \Phi_{\mathcal{H}'}(\Theta') \quad (A.47) \]

We now apply induction again, this time for expression \( e_1[\bar{y}/\bar{x}] \) using typing (A.40) and evaluation (A.39). It remains to show that the bound (A.47) satisfies the premise (1.E). By Def. 5.5 (potential) and (A.34) we know \( \phi_{\mathcal{H}'}(c(\bar{y})), B) = \sum_{i=1}^{n} \phi_{\mathcal{H}'}(\ell_i : A_i[B/X]) \); substituting in (A.47) yields:

\[ m' \geq t + p' + q + \sum_{i=1}^{n} \phi_{\mathcal{H}'}(\ell_i : A_i[B/X]) + \phi_{\mathcal{H}'}(\Delta, \Theta) + \Phi_{\mathcal{H}'}(\Theta') + \Phi_{\mathcal{H}'}(\Theta') \]

\[ = t + p' + q + \phi_{\mathcal{H}'}(\Delta, \ell_1 : A_1[B/X], \ldots, \ell_n : A_n[B/X]) + \phi_{\mathcal{H}'}(\Theta) + \Phi_{\mathcal{H}'}(\Theta') + \Phi_{\mathcal{H}'}(\Theta') \]

Hence we can apply induction and obtain:

\[ M' < M'' \quad (A.48) \]

\[ \Gamma'' \vdash_{\text{pr}} w : C \quad (A.49) \]

\[ \mathcal{E}'', \mathcal{B}'' \vdash_{\text{Match}} (\mathcal{E}'', \mathcal{L}) : M'' \quad (A.50) \]

\[ \gamma(M'' \mid (\Gamma'', \Theta), \mathcal{E}'') \quad (A.51) \]

\[ \mathcal{H}'', S, \mathcal{L} \vdash_{\text{pr}} e_1[\bar{y}/\bar{x}] \Downarrow w, \mathcal{H}'' \quad (A.52) \]

\[ m'' \geq t + p'' + \phi_{\mathcal{H}''}(w), C) + \phi_{\mathcal{H}''}(\Theta) + \Phi_{\mathcal{H}''}(\Theta') + \Phi_{\mathcal{H}''}(\Theta') \quad (A.53) \]

The required results follow by transitivity of subtyping and the MATCH\_0 rule. This concludes the case MATCH\_0.

For the subcase FAIL\_0 we get by inversion of the evaluation rule:

\[ \mathcal{H}, S \cup BV(e_1), \mathcal{L} \vdash e_0 \Downarrow w', \mathcal{H}' \quad (A.54) \]

\[ \mathcal{H}', S, \mathcal{L} \vdash e_2 \Downarrow w, \mathcal{H}'' \quad (A.55) \]

Again we proceed by assuming \( m \) as in (A.41) and apply induction for expression \( e_0 \), obtaining:

\[ M < M' \quad (A.56) \]

\[ \Gamma' \vdash_{\text{pr}} w' : B \quad (A.57) \]

\[ \mathcal{E}', \mathcal{B}' \vdash_{\text{Match}} (\mathcal{E}', \mathcal{L}) : M' \quad (A.58) \]

\[ \gamma(M' \mid (\Gamma', \Delta, \Theta), \mathcal{E}') \quad (A.59) \]

\[ \mathcal{H}, S, \mathcal{L} \vdash_{\text{pr}} e_0 \Downarrow w', \mathcal{H}' \quad (A.60) \]

\[ m' \geq t + p' + \phi_{\mathcal{H}'}(w', B) + \phi_{\mathcal{H}'}(\Delta, \Theta) + \Phi_{\mathcal{H}'}(\Theta') + \Phi_{\mathcal{H}'}(\Theta') \quad (A.61) \]
In order to apply induction for expression $e_2$ we first need to show that the bound (A.60) for $m'$ satisfies the premise (1.E):

$$m' \geq t + p' + \phi_{\beta_i}(w':B) + \phi_{\beta_i}(\Delta, \Theta) + \Phi_{\beta_i}(c') + \Phi_{\beta_i}(B')$$

$$\geq t + p' + \phi_{\beta_i}(\Delta) + \phi_{\beta_i}(\Theta) + \Phi_{\beta_i}(c') + \Phi_{\beta_i}(B')$$

The inequation above holds because potential $\phi_{\beta_i}(w':B)$ is always non-negative. Applying induction for $e_2$ gives:

$$\mathcal{M}' < \mathcal{M}''$$

$$\Gamma'' \mathrel{\vdash_{\mathcal{M}''}} w : C$$

$$\mathcal{C}'', \mathcal{L} \vdash_{\mathcal{M}''} e_2 \downarrow w, \mathcal{C}''$$

$$\mathcal{C}'', \mathcal{L} \vdash_{\mathcal{M}''} e_2 \downarrow w, \mathcal{C}''$$

$$m'' \geq t + p'' + \phi_{\beta_i}(w:C) + \phi_{\beta_i}(\Theta) + \Phi_{\beta_i}(c'') + \Phi_{\beta_i}(B'')$$

Again the required results follow from the above by transitivity of subtyping and the application of rule \textsc{Fail}_3. This concludes the proof of the \textsc{Match} case.

**Case Weak**: The typing premise (1.A) reads $\Gamma, x:A \vdash_{\mathcal{M}} e : C$. By inversion of rule \textsc{Weak} we get $\Gamma \vdash_{\mathcal{M}} e : C$. In order to apply the induction hypothesis for this judgment, we note that premise (1.B) (type consistency) holds unchanged; and because $\gamma(M | \Gamma, x:A, \Theta, c)$ implies $\gamma(M | \Gamma, \Theta, c)$ so does (1.C) (global compatibility). The bound (1.E) for the induction also holds because $\phi_{\beta_i}(\Gamma, x:A) \geq \phi_{\beta_i}(\Gamma)$. We can therefore apply induction to $e$ with the typing $\Gamma \vdash_{\mathcal{M}} e : C$ and obtain all required results for this case.

**Case Relax**: By the second premise of \textsc{Relax} follows $p - p_0 \geq 0$ and thus we can choose $t = t + p - p_0$. We apply the induction hypothesis to $\Gamma \vdash_{\mathcal{M}} e : A$ for this $t$. Since \textsc{Relax} is a structural rule, all statements apart from (1.A) and (1.E) remain unchanged. The induction hypothesis thus yields all required conclusions verbatim, except for (1.K). Instead, the induction yields $m' \geq t' + p' + r$, with $r = \phi_{\beta_i}(w:A) + \phi_{\beta_i}(\Theta) + \Phi_{\beta_i}(c') + \Phi_{\beta_i}(B')$. Unfolding our choice for $t'$ yields $m' \geq (t + p - p_0) + p_0 = r$. By the third premise of \textsc{Relax} follows $p - p_0 + p_0 \geq p'$ and thus $m' \geq t + p' + r$ as required to conclude this case.

**Case Prep**: The typing premise is

$$\Gamma, \ell:T_{\beta_i}^{q_0 + q_1}(A) \vdash_{\mathcal{M}, \ell} e : C$$

By inversion of the rule \textsc{Prep} we get

$$\Gamma, \ell:T_{\beta_i}^{q_0 + q_1}(A) \vdash_{\mathcal{M}, \ell} e : C$$

and $q_0 \geq q'$. Let $B' = B[\ell \mapsto q_1 + B(\ell)]$, i.e. $B'$ is equal to $B$ except for location $\ell$ where it increases by $q_1$. Assuming $m$ as in premise (1.E), we show that it satisfies the requirements for applying induction to (A.68) with the modified $B'$:

$$0 = t + p + q_1 + \phi_{\beta_i}(\Gamma, \ell:T_{\beta_i}^{q_0 + q_1}(A)) + \phi_{\beta_i}(\Theta) + \Phi_{\beta_i}(c') + \Phi_{\beta_i}(B')$$

The last inequality holds because $\phi_{\beta_i}(\ell:T_{\beta_i}^{q_0 + q_1}(A)) = \phi_{\beta_i}(\ell:T_{\beta_i}^{q_0 + q_1}(A))$ (potential) and $q_1 + \Phi_{\beta_i}(B) \geq \Phi_{\beta_i}(B')$; note that the later is an equality when $\mathcal{H}(\ell)$ is not a whnf.

We need to reestablish both global compatibility and type consistency in order to apply the induction hypothesis. Let $T_{\beta_i}^{q_1}(A') = M(\ell)$. By the definition of sharing and global compatibility (1.C) we have $\gamma(T_{\beta_i}^{q_1}(A') | T_{\beta_i}^{q_0 + q_1}(A))$ and hence $q_0 + q_1 \geq r$ and $q_0 + q_1 - r \geq q' - r'$. Define $k = \max(r - q_1, 0), k' = \min(r', k)$ and $\mathcal{M}' = M(\ell \mapsto T_{\beta_i}^{q_1}(A'))$.

To establish consistency for $\mathcal{M}'$, note that only the global type of location $\ell$ changes. Assume that (Loc2) applies, i.e. $\mathcal{H}(\ell)$ is not in whnf and $\ell \notin \mathcal{L}$, otherwise the claim is trivial. From the consistency premise (1.B) we have

$$\mathcal{C}(\ell) \vdash_{\mathcal{M}} T_{\beta_i}(A')$$

By the definition of $k$ and $k'$ we have $k + q_1 = \max(r - q_1, 0) + q_1 \geq r$ and $k' = \min(r', k) \leq r'$. Hence we can apply rule \textsc{Relax} to (A.69) and obtain

$$\mathcal{C}(\ell) \vdash_{\mathcal{M}} T_{\beta_i}(A')$$

By definition of $\mathcal{B}'$ the later is equivalent to the required

$$\mathcal{C}(\ell) \vdash_{\mathcal{M}} T_{\beta_i}(A')$$

To establish compatibility for $\mathcal{M}'$ we need to show

$$\gamma(T_{\beta_i}(A') | T_{\beta_i}^{q_0 + q_1}(A), \mathcal{C}(\ell))$$

From the compatibility premise (1.C) we know

$$\gamma(T_{\beta_i}(A') | T_{\beta_i}^{q_0 + q_1}(A), \mathcal{C}(\ell))$$

First we show that $\gamma(T_{\beta_i}(A') | T_{\beta_i}^{q_0 + q_1}(A))$: by definition of sharing, we need to show $q_0 \geq k$ and $q_0 - k \geq q' - k'$. By definition of $k$, we have $q_0 \geq k \iff q_0 \geq \max(r - q_1, 0) \iff q_0 \geq r - q_1 \land q_0 \geq 0 \iff q_0 + q_1 \geq r \land q_0 \geq 0$; the later holds by non-negativity.
Appling S can apply induction hypothesis directly and obtain:

\[ q_0 - k \geq q' - k' \]

\[ \iff q_0 + k' \geq q' + k \]  \hspace{1cm} \{ \text{definition of } k' \}

\[ \iff q_0 + \min(r', k) \geq q' + k \]  \hspace{1cm} \{ \text{definition of } \min \}

\[ \iff q_0 + r' \geq q' + k \land q_0 + k \geq q' + k \]

\[ \iff q_0 + r' \geq q' + k \land q_0 \geq q' \]  \hspace{1cm} \{ q_0 \geq q' \text{ by hyp. of rule PREPAY} \}

\[ \iff q_0 + r' \geq q' + k \]  \hspace{1cm} \{ \text{definition of } k \}

\[ \iff q_0 + r' \geq q' + \max(r - q_1, 0) \]  \hspace{1cm} \{ \text{definition of } \max \}

\[ \iff q_0 + r' \geq q' + r - q_1 \land q_0 + r' \geq q' \]  \hspace{1cm} \{ q_0 \geq q' \text{ and } r' \geq 0 \}

\[ \iff q_0 + q_1 - r \geq q' - r' \]

The later holds by the compatibility premise (A.70). For other types \( T_\alpha ^<(A''') \) in either \( \Gamma |_\ell \) or \( \mathcal{C}_\ell \), observe that \( T_\alpha ^<(A') <: T_\beta ^<(A') \) by construction and \( T_\alpha ^<(A') <: T_\beta ^<(A'') \) by the original compatibility (A.70). By transitivity we get the desired result.

Since the other premises remain unchanged, we can therefore apply induction and obtain precisely the results required for the conclusion of this case.

**Case SHARE:** The typing hypothesis is \( \Gamma, \ell: A \vdash T e : C \). By inversion of rule SHARE we obtain \( \Gamma, \ell: A_1, \ell: A_2 \vdash T e : C \) and \( \Upsilon (A | A_1, A_2) \). Assuming \( m \) as in premise (1.E), we get:

\[ m \geq t + \phi y (\Gamma, \ell: A) + \phi y (\Theta) + \Phi \phi (\ell: C) + \Phi \phi (\ell: \mathcal{B}) \geq t + \phi y (\Gamma, \ell: A_1, \ell: A_2) + \phi y (\Theta) + \Phi \phi (\ell: C) + \Phi \phi (\ell: \mathcal{B}) \]

The last inequality holds because by Lemma 5.6 (potential splitting) \( \phi y (\gamma (\ell): A) \geq \phi y (\gamma (\ell): A_1) + \phi y (\gamma (\ell): A_2) \). We can therefore apply the induction hypothesis to \( e \) with typing premise \( \Gamma, \ell: A_1, \ell: A_2 \vdash T e : C \) and obtain as result the required conclusions for the case SHARE. This concludes the proof of this case.

**Case SUPERTYPE:** The type rules gives \( \Gamma, x: A \vdash T e : C \) and \( A < B \). We show that we can apply induction for the premise \( \Gamma, x: B \vdash T e : C \). Type consistency holds unchanged for the induction; by \( A < B \) and the compatibility premise (1.C) \( \Upsilon (M | \Gamma, x: A, \Theta, C') \), which implies \( \Upsilon (M | \Gamma, x: B, \Theta, C') \). The bounds (1.E) also holds because \( \phi y (x: A) \geq \phi y (x: B) \) by \( A < B \) and Lemma 5.7. Applying the induction gives the required conclusions for the case SUPERTYPE.

**Case SUBTYPE:** The type rule gives us \( \Gamma \vdash T e : C \); by inversion we get \( \Gamma \vdash T e : B \) and \( B < C \). Because the context is unchanged, we can apply induction hypothesis directly and obtain:

\[ M < M' \]

\[ \Gamma' \vdash w : B \]

\[ \ell', \mathcal{B}' \vdash \mathcal{M} (\ell', \mathcal{L}) : M' \]

\[ \Upsilon (M' | (\Gamma', \Theta), \ell') \]

\[ \mathfrak{S}, \mathcal{L} \vdash \mathcal{M} (\ell', \mathcal{L}) : C \]

\[ m' \geq t + p' + \phi y (w: B) + \phi y (\Theta) + \Phi \phi (\ell': C') + \Phi \phi (\ell': \mathcal{B}') \]

Applying SUBTYPE to (A.72) gives \( \Gamma' \vdash w : C \) as required for (1.G). Lemma 5.7 with \( B < C \) gives \( \phi y (w: B) \geq \phi y (w: C) \); substituting in (A.76) establishes the bound (1.K). Results (A.71), (A.73), (A.74) and (A.75) directly establish the remaining proof obligations for this case.