Multivariate Amortized Resource Analysis

JAN HOFFMANN, Yale University
KLAUS AEHLIG and MARTIN HOFMANN, Ludwig-Maximilians-Universität München

We study the problem of automatically analyzing the worst-case resource usage of procedures with several arguments. Existing automatic analyses based on amortization or sized types bound the resource usage or result size of such a procedure by a sum of unary functions of the sizes of the arguments.

In this article we generalize this to arbitrary multivariate polynomial functions thus allowing bounds of the form mn which had to be grossly overestimated by $m^2 + n^2$ before. Our framework even encompasses bounds like $\sum_{i,j \leq n} m_i m_j$ where the m_i are the sizes of the entries of a list of length n.

This allows us for the first time to derive useful resource bounds for operations on matrices that are represented as lists of lists and to considerably improve bounds on other superlinear operations on lists such as longest common subsequence and removal of duplicates from lists of lists. Furthermore, resource bounds are now closed under composition which improves accuracy of the analysis of composed programs when some or all of the components exhibit superlinear resource or size behavior.

The analysis is based on a novel multivariate amortized resource analysis. We present it in form of a type system for a simple first-order functional language with lists and trees, prove soundness, and describe automatic type inference based on linear programming.

We have experimentally validated the automatic analysis on a wide range of examples from functional programming with lists and trees. The obtained bounds were compared with actual resource consumption. All bounds were asymptotically tight, and the constants were close or even identical to the optimal ones.

14

1. INTRODUCTION

A primary feature of a computer program is its quantitative performance characteristics: the amount of resources like time, memory, and power the program needs to perform its task.

Ideally, it should be possible for an experienced programmer to extrapolate from the source-code of a well-written program to its *asymptotic* worst-case behavior. But it is often insufficient to determine the asymptotic behavior of programs only. A conservative estimation of the resource consumption for a specific input or a comparison of two programs with the same asymptotic behavior require instead concrete upper bounds for specific hardware, that is to say, closed functions in the sizes of the program's inputs that bound the number of clock cycles or memory cells used by the program for inputs of these sizes on a given system.

Concrete worst-case bounds are particularly useful in the development of embedded systems and hard real-time systems. In the former, one wants to use hardware that is just good enough to accomplish a task in order to produce a large number of units at lowest possible cost. In the latter, one needs to guarantee specific worst-case running times to ensure the safety of the system.

The manual determination of such bounds is very cumbersome. See, for example, the careful analyses carried out by Knuth in *The Art of Computer Programming* where he pays close attention to the concrete and best possible values of constants for the MIX architecture. Not everyone commands the mathematical ease of Knuth and even he would run out of steam if he had to do these calculations over and over again while going through the debugging loops of program development. In short, derivation of precise bounds by hand appears to be unfeasible in practice in all but the simplest cases.

As a result, automatic methods for static resource analysis are highly desirable and have been the subject of extensive research. On the one hand, there is the large field of WCET (Worst-Case Execution Time) analysis [Wilhelm et al. 2008] that is focused on (yet not limited to) the runtime analysis of sequential code without loops taking into account low-level features like hardware caches and instruction pipelines. On the other hand, there is an active research community that employs type systems and abstract interpretation to deal with the analysis of loops, recursion, and data structures [Gulwani et al. 2009b; Albert et al. 2009b; Jost et al. 2010].[1]

In this article we continue our work [Hoffmann and Hofmann 2010b, 2010a] on the resource analysis of programs with recursion and inductive data structures. A previous version of this work appeared at a conference [Hoffmann et al. 2011]. Our approach is as follows.

(1) We consider *Resource-Aware ML* (RAML), a first-order fragment of OCamel that features integers, lists, binary trees, and recursion.

(2) We define a big-step operational semantics that formalizes the actual resource consumptions of programs. It is parametrized with a resource metric that can be directly related to the compiled assembly code for a specific system architecture [Jost et al. 2009].[2] To formalize diverging computations, we use a partial big-step operational semantics that we introduced earlier [Hoffmann and Hofmann 2010a].

(3) We describe an elaborated resource-parametric type system whose type judgments establish concrete worst-case bounds in terms of closed, easily understood formulas. The type system allows for an efficient and completely automatic inference algorithm that is based on linear programming.

(4) We prove the nontrivial soundness of the derived resource bounds with respect to the big-step operational semantics. The partial operational semantics enables a strong and concise soundness result: if the type analysis has established a resource

---

[1]See Section 10 for a detailed overview of the state-of-the-art.
[2]To obtain clock-cycle bounds for atomic steps one has to employ WCET tools [Jost et al. 2009].

bound for an expression then the resource consumption of its (possibly nonterminating) evaluation does not exceed the bound.

(5) We verify the practicability of our approach with a publically available implementation and a reproducible experimental evaluation.

Following Hofmann and Jost [2003], our type system relies on the potential method of amortized analysis to take into account the interactions between different parts of a computation. This technique has been successfully applied to the type-based resource analysis of object-oriented programs [Hofmann and Jost 2006; Hofmann and Rodriguez 2009], polymorphic and higher-order programs [Jost et al. 2010], Java-like bytecode by means of separation logic [Atkey 2010], and to generic resource metrics [Jost et al. 2009; Campbell 2009]. The main limitation shared by these analysis systems is their restriction to linear resource bounds which can be efficiently reduced to solving linear constraints.

A recently discovered technique [Hoffmann and Hofmann 2010b, 2010a] yields an automatic amortized analysis for polynomial bounds while still relying on linear constraint solving only. The resulting extension of the linear system [Hofmann and Jost 2003; Jost et al. 2009] efficiently computes resource bounds for first-order functional programs that are sums $\sum p_i(n_i)$ of univariate polynomials p_i. For instance, it automatically infers evaluation-step bounds for the sorting algorithms quick sort and insertion sort that exactly match the measured worst-case behavior of the functions [Hoffmann and Hofmann 2010a]. The computation of these bounds takes less than a second.

This analysis system for polynomial bounds has, however, two drawbacks that hamper the automatic computation of bounds for larger programs. First, many functions with multiple arguments that appear in practice have *multivariate* cost characteristics like $m \cdot n$. Second, if data from different sources is interlinked in a program then multivariate bounds like $(m+n)^2$ arise even if all functions have a univariate resource behavior. In these cases the analysis fails, or the bounds are hugely overapproximated by $3m^2 + 3n^2$.

To overcome these drawbacks, this article presents an automatic type-based amortized analysis for multivariate polynomial resource bounds. We faced three main challenges in the development of the analysis.

(1) There is the identification of multivariate polynomials that accurately describe the resource cost of typical examples. It is necessary that they are closed under natural operations to be suitable for local typing rules. Moreover, they must handle an unbounded number of arguments to tightly cope with nested data structures.

(2) There is the automatic relation of sizes of data structures in function arguments and results, even if data is scattered over different locations (like $n_1 + n_2 \leq n$ in the partitioning of quick sort).

(3) There is the smooth integration of the inference of size relations and resource bounds to deal with the interactions of different functions while keeping the analysis technically feasible in practice.

To address challenge one we define *multivariate resource polynomials* that are a generalization of the resource polynomials that we used earlier [Hoffmann and Hofmann 2010b]. To address challenges two and three we introduce a multivariate potential-based amortized analysis (Section 5 and Section 6). The local typing rules emit only simple linear constraints and are remarkably modest considering the variety of relations between different parts of the data that are taken into account.

Our experiments with a prototype implementation[3] (see Section 8) show that our system automatically infers tight multivariate bounds for complex programs that

---

[3]See http://raml.tcs.ifi.lmu.de for a Web interface, example programs, and the source-code.

involve nested data structures such as trees of lists. Additionally, it can deal with the same wide range of linear and univariate programs as the previous systems.

As representative examples we present in Section 8 the analyses of the dynamic programming algorithm for the length of the longest common subsequence of two lists and an implementation of insertion sort that lexicographically sorts a list of lists. Note that the latter example exhibits a worst-case running time of the form $O(n^2m)$, where n is the length of the outer list and m is the maximal length of the inner lists. The reason is that each of the $O(n^2)$ comparisons performed by insertion sort needs time linear in m.

Furthermore, we describe two slightly more involved programs that demonstrate interesting capabilities of the analysis. The first program shows that our multivariate resource polynomials are particularly advantageous for composed functions. It partitions a list into arbitrary many sublists and then sorts each sublist with quick sort. If n is the length of the input list then there can be n sublists and each sublist can have up to n elements after the partitioning. The worst-case time complexity of quick sort is $O(n^2)$. Our analysis recognizes that the number of sublists and the lengths of the sublists are related and computes a quadratic resource bound for the program.

The second example program demonstrates the benefits of the amortized method. We first implement matrix multiplication for lists of lists. We then consider a binary tree of matrices with possibly different (but fitting) dimensions and multiply the matrices in breadth-first order. The breadth-first traversal relies on a functional queue that is implemented with two lists. The prototype implementation computes an asymptotically tight polynomial bound of degree four.

The main contributions we make in this article are as follows.

(1) We give the definition of multivariate resource polynomials that generalize univariate resource polynomials [Hoffmann and Hofmann 2010b] (in Section 4).
(2) We provide an introduction of type annotation that correspond to global polynomial potential functions for amortized analysis which depend on the sizes of several parts of the input (in Section 5).
(3) We present local typing rules that modify type annotations for global potential functions (in Section 6).
(4) We implement an efficient type inference algorithm that relies on linear constraint solving only.

This journal version of the article extends and improves the conference version [Hoffmann et al. 2011]. The main enhancements are the following:

—Full proofs of the theorems.
—An improved soundness theorem that states that bounds hold for terminating and nonterminating computations.
—A detailed description of the inference algorithm.
—A more in-depth discussion of related work.
—Additional example programs.

Some parts of this article are part of the first author's PhD thesis [Hoffmann 2011].

2. BACKGROUND AND INFORMAL PRESENTATION

Amortized analysis with the potential method has been introduced [Tarjan 1985] to manually analyze the efficiency of data structures. The key idea is to incorporate a nonnegative potential into the analysis that can be used to pay (costly) operations.

To apply the potential method to statically analyze a program, one has to determine a mapping from machine states to potentials for every program point. Then one has to show that for every possible evaluation, the potential at a program point suffices

to cover the cost of the next transition and the potential at the succeeding program point. The initial potential is then an upper bound on the resource consumption of the program.

2.1. Linear Potential

One way to achieve such an analysis is to use linear potential functions [Hofmann and Jost 2003]. Inductive data structures are statically annotated with a positive rational numbers q to define nonnegative potentials $\Phi(n) = q \cdot n$ as a function of the size n of the data. Then a sound albeit incomplete type-based analysis of the program text statically verifies that the potential is sufficient to pay for all operations that are performed on this data structure during any possible evaluation of the program.

The analysis is best explained by example. Consider the function filter of type $(\text{int}, L(\text{int})) \rightarrow L(\text{int})$ that removes the multiples of a given integer from a list of integers.

```
filter(a,l) = match l with | nil -> nil
         | (x::xs) -> let xs' = filter(a,xs) in
                      if x mod a == 0 then xs' else x::xs'
```

Assume that we need two memory cells to create a new list cell. Then the heap-space usage of an evaluation of filter(a,ℓ) is at most $2|\ell|$. To infer an upper bound on the heap-space usage we enrich the type of filter with a priori unknown potential annotations[4] $q_{(0,i)}, p_i \in \mathbb{Q}_0^+$.

$$\text{filter}:((\text{int}, L(\text{int})), (q_{(0,0)}, q_{(0,1)})) \rightarrow (L(\text{int}), (p_0, p_1))$$

The intuitive meaning of the resulting type is as follows: to evaluate filter(a,ℓ) one needs $q_{(0,1)}$ memory cells per element in the list ℓ and $q_{(0,0)}$ additional memory cells. After the evaluation there are p_0 memory cells and p_1 cells per element of the returned list left. We say that the pair (a, ℓ) has potential $\Phi((a, \ell), (q_{(0,0)}, q_{(0,1)})) = q_{(0,0)} + q_{(0,1)} \cdot |\ell|$ and that $\ell' = \text{filter}(a, \ell)$ has potential $\Phi(\ell', (p_0, p_1)) = p_0 + p_1 \cdot |\ell'|$. A valid potential annotation would be, for instance, $q_{(0,0)} = p_0 = p_1 = 0$ and $q_{(0,1)} = 2$. Another valid annotation would be $q_{(0,0)} = p_0 = 0$, $p_1 = 2$, and $q_{(0,1)} = 4$. It can be used to type the inner call of filter in an expression like filter(a,filter(b,ℓ)). The resources left after the inner call to filter are consumed by the outer one.

To infer the potential annotations one can use a standard type inference in which simple linear constraints are collected as each typing rule is applied. For the heap-space consumption of filter the constraints would state that $q_{(0,0)} \geq p_0$ and $q_{(0,1)} \geq 2 + p_1$.

2.2. Univariate Polynomials

An automatic amortized analysis can be also used to derive potential functions of the form $\sum_{i=0}^{k} q_i \binom{n}{i}$ with $q_i \geq 0$ while still relying on solving linear inequalities only [Hoffmann and Hofmann 2010b]. These potential functions are attached to inductive data structures via type annotations of the form $\vec{q} = (q_0, \ldots, q_k)$ with $q_i \in \mathbb{Q}_0^+$. For instance, the typing $\ell:(L(\text{int}), (4, 3, 2, 1))$, defines the potential $\Phi(\ell, (4, 3, 2, 1)) = 4 + 3|\ell| + 2\binom{|\ell|}{2} + 1\binom{|\ell|}{3}$.

The use of the binomial coefficients rather than powers of variables has several advantages. In particular, the identity $\sum_{i=0,\ldots,k} q_i \binom{n+1}{i} = \sum_{i=0,\ldots,k-1} q_{i+1} \binom{n}{i} + \sum_{i=0,\ldots,k} q_i \binom{n}{i}$ gives rise to a local typing rule for *list match* which allows to type naturally both recursive calls and other calls to subordinate functions in branches of a pattern match.

---

[4]We use the naming scheme of the unknowns that arises from the more general method introduced in this article.

This identity forms the mathematical basis of the *additive shift* $\lhd$ of a type annotation which is defined by $\lhd(q_0, \ldots, q_k) = (q_0 + q_1, \ldots, q_{k-1} + q_k, q_k)$. For example, it appears in the typing tail:$(L(\text{int}), \vec{q}) \to (L(\text{int}), \lhd(\vec{q}))$ of the function tail that removes the first element from a list. The potential resulting from the contraction xs:$(L(\text{int}), \lhd(\vec{q}))$ of a list (x::xs):$(L(\text{int}), \vec{q})$, usually in a pattern match, suffices to pay for three common purposes: (i) to pay the constant costs q_1 after and before the recursive calls, (ii) to fund, by $(q_2, \ldots, q_k)$, calls to auxiliary functions, and (iii) to pay, by $(q_0, \ldots, q_k)$, for the recursive calls.

To see how the polynomial potential annotations are used, consider the function eratos:$L(\text{int}) \to L(\text{int})$ that implements the sieve of Eratosthenes. It successively calls the function filter to delete multiples of the first element from the input list. If eratos is called with a list of the form $[2, 3, \ldots, n]$ then it computes the list of primes p with $2 \le p \le n$.

```
eratos l = match l with | nil -> nil
              | (x::xs) -> x::eratos(filter(x,xs))
```

Note that it is possible in our system to implement the function filter with a destructive pattern match (just replace match with matchD). That would result in a filter function that does not consume heap-cells and in a linear heap-space consumption of eratos. But to illustrate the use of quadratic potential we use the filter function with linear heap-space consumption from the first example.[5] In an evaluation of eratos(ℓ) the function filter is called once for every sublist of the input list ℓ in the worst case. Then the calls to filter cause a worst-case heap-space consumption of $2\binom{|\ell|}{2}$. This is, for example, the case if ℓ is a list of pairwise distinct primes. Additionally, there is the creation of a new list element for every recursive call of eratos. Thus, the total worst-case heap-space consumption of the function is $2n + 2\binom{n}{2}$ if n is the size of the input list.

To bound the heap-space consumption of eratos, our analysis system automatically computes the following type.

$$\text{eratos:}(L(\text{int}), (0, 2, 2)) \to (L(\text{int}), (0, 0, 0))$$

Since the typing assigns the initial potential $2n + 2\binom{n}{2}$ to a function argument of size n, the analysis computes a tight heap-space bound for eratos. In the pattern match, the additive shift assigns the type $(L(\text{int}), (2, 4, 2))$ to the variable xs of size $n - 1$. The constant potential 2 is then used to pay for the cons operation (i). The nonconstant potential xs:$(L(\text{int}), (0, 4, 2))$ is used for two purposes. The potential xs:$(L(\text{int}), (0, 2, 0))$ is used to pay the cost of filter(xs) (ii) and the potential xs:$(L(\text{int})(0, 2, 2)$ is passed on to the result of filter(xs) to pay for the recursive call of eratos (iii). To this end, we use filter with the following type.

$$\text{filter:}((\text{int}, L(\text{int})), (0, 4, 2)) \to (L(\text{int}), (0, 2, 2))$$

It expresses the fact that filter can pass unused potential to its result since the returned list is at most as long as the input list.

To infer the typing, we start with an unknown potential annotation as in the linear case.

$$\text{eratos:}(L(\text{int}), (q_0, q_1, q_2)) \to (L(\text{int}), (p_0, p_1, p_2))$$

The syntax-directed type analysis then computes linear inequalities which state that $q_0 \ge p_0$, $q_1 \ge 2 + p_1$, and $q_2 \ge 2 + p_2$.

This analysis method works for many functions that admit a worst-case resource consumption that can be expressed by sums of univariate polynomials like $n^2 + m^2$.

---

[5]It is just more convenient to argue about heap space than to argue about evaluation steps.

However, it often fails to compute types for functions whose resource consumption is bounded by a mixed term like $n^2 \cdot m$. The reason is that the potential is attached to a single data structure and does not take into account relations between different data structures.

2.3. Multivariate Bounds

This article extends type-based amortized analysis to compute mixed resource bounds like $2n \cdot \binom{m}{2}$. To this end, we introduce a global polynomial potential annotation that can express a variety of relations between different parts of the input. To give a flavor of the basic ideas we informally introduce this global potential in this section for pairs of integer lists.

The potential of a single integer list can be expressed as a vector $(q_0, q_1, \ldots, q_k)$ that defines a potential function of the form $\sum_{i=0}^{k} q_i \binom{n}{i}$. To represent mixed terms of degree $\leq k$ for a pair of integer lists we use a triangular matrix $Q = (q_{(i,j)})_{0 \leq i+j \leq k}$. Then Q defines a potential function of the form $\sum_{0 \leq i+j \leq k} q_{(i,j)} \binom{n}{i} \binom{m}{j}$ where m and n are the lengths of the two lists.

This definition has the same advantages as the univariate version of the system. Particularly, we can still use the additive shift to assign potential to sublists. To generalize the additive shift of the univariate system, we use the identity $\sum_{0 \leq i+j \leq k} q_{(i,j)} \binom{n+1}{i} \binom{m}{j} = \sum_{0 \leq i+j \leq k-1} q_{(i+1,j)} \binom{n}{i} \binom{m}{j} + \sum_{0 \leq i+j \leq k} q_{(i,j)} \binom{n}{i} \binom{m}{j}$. It is reflected by two additive shifts $\lhd_1(Q) = (q_{(i,j)} + q_{(i+1,j)})_{0 \leq i+j \leq k}$ and $\lhd_2(Q) = (q_{(i,j)} + q_{(i,j+1)})_{0 \leq i+j \leq k}$ where $q_{(i,j)} := 0$ if $i + j > k$. The shift operations can be used like in the univariate case. For example, we derive the typing tail1: $((L(\text{int}), L(\text{int})), Q) \rightarrow ((L(\text{int}), L(\text{int})), \lhd_1(Q))$ for the function tail1(xs,ys) = (tail xs,ys).

To see how the mixed potential is used, consider the function dyad that computes the dyadic product of two lists.

```
mult(x,l) = match l with | nil -> nil
                          | (y::ys) -> x*y::mult(x,ys)

dyad(l,ys) = match l with | nil -> nil
                | (x::xs) -> (mult(x,ys))::dyad(xs,ys)
```

Similar to previous examples, mult consumes $2n$ heap cells if n is the length of input. We do not have to assign any potential to the output since it is not processed further in this example. This exact bound is represented by the typing

$$\text{mult: } ((\text{int}, L(\text{int})), (0, 2, 0)) \rightarrow (L(\text{int}), (0, 0, 0))$$

that states that the potential is $0 + 2n + 0\binom{n}{2}$ before and 0 after the evaluation of mult(x,ℓ) if ℓ is a list of length n.

The function dyad consumes $2n + 2nm$ heap cells if n is the length of first argument and m is the length of the second argument. This is why the following typing represents a tight heap-space bound for the function.

$$\text{dyad: } \left((L(\text{int}), L(\text{int})), \begin{pmatrix} 0 & 0 & 0 \\ 2 & 2 & \\ 0 & & \end{pmatrix} \right) \rightarrow (L(\text{int}, \text{int}), 0)$$

To verify this typing of dyad, the additive shift $\lhd_1$ is used in the pattern matching. This results in the potential

$$\text{(xs,ys): } \left((L(\text{int}), L(\text{int})), \begin{pmatrix} 2 & 2 & 0 \\ 2 & 2 & \\ 0 & & \end{pmatrix} \right)$$

that is used as in the function eratos: the constant potential 2 is used to pay for the cons operation (i), the linear potential $ys\text{:}(L(\text{int}), (0, 2, 0))$ is used to pay the cost of mult(ys) (ii), the rest of the potential is used to pay for the recursive call (iii).

Multivariate potential is also needed to assign a superlinear potential to the result of a function like append. This is, for example, needed to type an expression like eratos(append(ℓ_1, ℓ_2)). Here, append would have the type

$$\text{append:} \left((L(\text{int}), L(\text{int})), \begin{pmatrix} 0 & 2 & 2 \\ 4 & 2 \\ 2 \end{pmatrix}\right) \rightarrow (L(\text{int}), (0, 2, 2)).$$

The correctness of the bound follows from the convolution formula $\binom{n+m}{2} = \binom{n}{2} + \binom{m}{2} + nm$ and from the fact that append consumes $2n$ resources if n is the length of the first argument. The respective initial potential $4n + 2m + 2(\binom{n}{2} + \binom{m}{2} + mn)$ furnishes a tight bound on the worst-case heap-space consumption of the evaluation of eratos(append(ℓ_1, ℓ_2)), where $|\ell_1| = n, |\ell_2| = m$.

3. RESOURCE AWARE ML

RAML (Resource-Aware ML) is a first-order functional language with ML-style syntax, booleans, integers, pairs, lists, binary trees, recursion, and pattern match. In the implementation of RAML we already included a destructive pattern match that we could handle using the methods described here.

3.1. Syntax

To simplify typing rules and semantics, we define the following expressions of RAML to be in *let normal form*. In the implementation we transform unrestricted expressions into a let normal form with explicit sharing before the type analysis. The explicit sharing accounts for multiple occurrences of variables and simplifies the type inference. See Section 7 for details.

$$
\begin{aligned}
e ::=\ & () \mid \text{True} \mid \text{False} \mid n \mid x \mid x_1\ binop\ x_2 \mid f(x_1, \ldots, x_n) \\
& \mid \text{let } x = e_1 \text{ in } e_2 \mid \text{if } x \text{ then } e_t \text{ else } e_f \\
& \mid (x_1, x_2) \mid \text{nil} \mid \text{cons}(x_h, x_t) \mid \text{leaf} \mid \text{node}(x_0, x_1, x_2) \\
& \mid \text{match } x \text{ with } (x_1, x_2) \rightarrow e \\
& \mid \text{match } x \text{ with } \mid \text{nil} \rightarrow e_1 \mid \text{cons}(x_h, x_t) \rightarrow e_2 \\
& \mid \text{match } x \text{ with } \mid \text{leaf} \rightarrow e_1 \mid \text{node}(x_0, x_1, x_2) \rightarrow e_2 \\
binop ::=\ & + \mid - \mid * \mid \text{mod} \mid \text{div} \mid \text{and} \mid \text{or}
\end{aligned}
$$

We skip the standard definitions of integer constants $n \in \mathbb{Z}$ and variable identifiers $x \in \text{VID}$. For the resource analysis it is unimportant which ground operations are used in the definition of *binop*. In fact, one can use here every function that has a constant worst-case resource consumption. In our system we assume that we have integers of a fixed length, say 32 bits, to ensure this property of the integer operations.

3.2. Simple Types

We define the well-typed expressions of RAML by assigning a *simple type*, a usual ML type without resource annotations, to well-typed expressions. Simple types are data types and first-order types as given by the grammars that follow.

$$A ::= \text{unit} \mid \text{bool} \mid \text{int} \mid L(A) \mid T(A) \mid (A, A) \qquad\qquad F ::= A \rightarrow A$$

To each simple type A we assign a set of semantic values $[\![A]\!]$ in the obvious way. For example, $[\![T(\text{int}, \text{int})]\!]$ is the set of finite binary trees whose nodes are labeled with pairs

of integers. It is convenient to identify tuples like (A_1, A_2, A_3, A_4) with the pair type $(A_1, (A_2, (A_3, A_4)))$.

A *typing context* Γ is a partial, finite mapping from variable identifiers to data types. A *signature* Σ is a finite, partial mapping of function identifiers to first-order types. The typing judgment $\Gamma \vdash_\Sigma e : A$ states that the expression e has type A under the signature Σ in the context Γ. The typing rules that define the typing judgment are standard and a subset of the resource-annotated typing rules from Section 5 if the resource annotations are omitted.

3.3. Programs

Each RAML program consists of a signature Σ and a family $(e_f, y_f)_{f \in \mathrm{dom}(\Sigma)}$ of expressions with a distinguished variable identifier such that $y_f{:}A \vdash_\Sigma e_f{:}B$ if $\Sigma(f) = A \to B$.

We write $f(y_1, \ldots, y_k) = e'_f$ as an abbreviation for $\Sigma(f) = (A_1, (A_2, (\ldots, A_k) \cdots)) \to B$ and $y_1{:}A_1, \ldots, y_k{:}A_k \vdash_\Sigma e'_f{:}B$. In this case, f is defined by $e_f = $ match y_f with $(y_1, y'_f) \to$ match y'_f with $(y_2, y''_f) \ldots e'_f$. Of course, one can use such function definitions also in the implementation.

3.4. Big-Step Operational Semantics

To prove the correctness of our analysis, we define a big-step operational semantics that measures the quantitative resource consumption of programs. It is parametric in the resource of interest and can measure every quantity whose usage in a single evaluation step can be bounded by a constant. The actual constants for a step on a specific system architecture can be derived by analyzing the translation of the step in the compiler implementation for that architecture [Jost et al. 2009].

The semantics is formulated with respect to a stack and a heap as usual: Let Loc be an infinite set of *locations* modeling memory addresses on a heap. The set of RAML *values* Val is given by

$$v ::= \ell \mid b \mid n \mid \textsc{Null} \mid (v, v).$$

A value $v \in$ Val is either a location $\ell \in$ Loc, a boolean constant b, an integer n, a null value $\textsc{Null}$, or a pair of values (v_1, v_2). We identify the tuple $(v_1, \ldots, v_n)$ with the pair $(v_1, (v_2, \ldots) \cdots)$.

A *heap* is a finite partial mapping $\mathcal{H} :$ Loc $\to$ Val that maps locations to values. A *stack* is a finite partial mapping $\mathcal{V} :$ VID $\to$ Val from variable identifiers to values.

Since we also consider resources like memory that can become available during an evaluation, we have to track the *watermark* of the resource usage, that is, the maximal number of resource units that are simultaneously used during an evaluation. In order to derive a watermark of a sequence of evaluations from the watermarks of the subevaluations one has also to take into account the number of resource units that are available after each subevaluation.

The operational evaluation rules define an evaluation judgment of the form

$$\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (q, q')$$

expressing the following. If the stack $\mathcal{V}$ and the initial heap $\mathcal{H}$ are given then the expression e evaluates to the value v and the new heap $\mathcal{H}'$. To evaluate e one needs at least $q \in \mathbb{Q}_0^+$ resource units and after the evaluation there are $q' \in \mathbb{Q}_0^+$ resource units available. The actual resource consumption is then $\delta = q - q'$. The quantity δ is negative if resources become available during the execution of e.

Figure 1 shows the evaluation rules of the big-step semantics. There is at most one pair (q, q') such that $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (q, q')$ for a given expression e, a heap $\mathcal{H}$ and a stack $\mathcal{V}$. The nonnegative number q is the (high) watermark of resources that are used simultaneously during the evaluation.

$$\frac{x \in \mathrm{dom}(\mathcal{V})}{\mathcal{V}, \mathcal{H} \vdash x \rightsquigarrow \mathcal{V}(x), \mathcal{H} \mid K^{\mathrm{var}}} \quad \text{(E:Var)} \qquad\qquad \frac{}{\mathcal{V}, \mathcal{H} \vdash () \rightsquigarrow \mathrm{Null}, \mathcal{H} \mid K^{\mathrm{unit}}} \quad \text{(E:ConstU)}$$

$$\frac{\mathcal{V}(x) = v \qquad [y_f \mapsto v], \mathcal{H} \vdash e_f \rightsquigarrow v', \mathcal{H}' \mid (q, q')}{\mathcal{V}, \mathcal{H} \vdash f(x) \rightsquigarrow v', \mathcal{H}' \mid K_1^{\mathrm{app}} \cdot (q, q') \cdot K_2^{\mathrm{app}}} \quad \text{(E:App)} \qquad \frac{n \in \mathbb{Z}}{\mathcal{V}, \mathcal{H} \vdash n \rightsquigarrow n, \mathcal{H} \mid K^{\mathrm{int}}} \quad \text{(E:ConstI)}$$

$$\frac{x_1, x_2 \in \mathrm{dom}(\mathcal{V}) \qquad v = op(\mathcal{V}(x_1), \mathcal{V}(x_2))}{\mathcal{V}, \mathcal{H} \vdash x_1 \; op \; x_2 \rightsquigarrow v, \mathcal{H} \mid K^{op}} \quad \text{(E:BinOp)} \qquad \frac{b \in \{\mathit{True}, \mathit{False}\}}{\mathcal{V}, \mathcal{H} \vdash b \rightsquigarrow b, \mathcal{H} \mid K^{\mathrm{bool}}} \quad \text{(E:ConstB)}$$

$$\frac{\mathcal{V}(x) = \mathit{True} \qquad \mathcal{V}, \mathcal{H} \vdash e_t \rightsquigarrow v, \mathcal{H}' \mid (q, q')}{\mathcal{V}, \mathcal{H} \vdash \mathit{if} \; x \; \mathit{then} \; e_t \; \mathit{else} \; e_f \rightsquigarrow v, \mathcal{H}' \mid K_1^{\mathrm{conT}} \cdot (q, q') \cdot K_2^{\mathrm{conT}}} \quad \text{(E:CondT)}$$

$$\frac{\mathcal{V}(x) = \mathit{False} \qquad \mathcal{V}, \mathcal{H} \vdash e_f \rightsquigarrow v, \mathcal{H}' \mid (q, q')}{\mathcal{V}, \mathcal{H} \vdash \mathit{if} \; x \; \mathit{then} \; e_t \; \mathit{else} \; e_f \rightsquigarrow v, \mathcal{H}' \mid K_1^{\mathrm{conF}} \cdot (q, q') \cdot K_2^{\mathrm{conF}}} \quad \text{(E:CondF)}$$

$$\frac{\mathcal{V}, \mathcal{H} \vdash e_1 \rightsquigarrow v_1, \mathcal{H}_1 \mid (q, q') \qquad \mathcal{V}[x \mapsto v_1], \mathcal{H}_1 \vdash e_2 \rightsquigarrow v_2, \mathcal{H}_2 \mid (p, p')}{\mathcal{V}, \mathcal{H} \vdash \mathit{let} \; x = e_1 \; \mathit{in} \; e_2 \rightsquigarrow v_2, \mathcal{H}_2 \mid K_1^{\mathrm{let}} \cdot (q, q') \cdot K_2^{\mathrm{let}} \cdot (p, p') \cdot K_3^{\mathrm{let}}} \quad \text{(E:Let)}$$

$$\frac{\mathcal{V}(x) = (v_1, v_2) \qquad \mathcal{V}[x_1 \mapsto v_1, x_2 \mapsto v_2], \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (q, q')}{\mathcal{V}, \mathcal{H} \vdash \mathit{match} \; x \; \mathit{with} \; (x_1, x_2) \rightarrow e \rightsquigarrow v, \mathcal{H}' \mid K_1^{\mathrm{matP}} \cdot (q, q') \cdot K_2^{\mathrm{matP}}} \quad \text{(E:MatP)}$$

$$\frac{x_1, x_2 \in \mathrm{dom}(\mathcal{V}) \qquad v = (\mathcal{V}(x_1), \mathcal{V}(x_2))}{\mathcal{V}, \mathcal{H} \vdash (x_1, x_2) \rightsquigarrow v, \mathcal{H} \mid K^{\mathrm{pair}}} \quad \text{(E:Pair)} \qquad \frac{}{\mathcal{V}, \mathcal{H} \vdash \mathit{nil} \rightsquigarrow \mathrm{Null}, \mathcal{H} \mid K^{\mathrm{nil}}} \quad \text{(E:Nil)}$$

$$\frac{x_h, x_t \in \mathrm{dom}(\mathcal{V}) \qquad v = (\mathcal{V}(x_h), \mathcal{V}(x_t)) \qquad \ell \notin \mathrm{dom}(\mathcal{H})}{\mathcal{V}, \mathcal{H} \vdash \mathit{cons}(x_h, x_t) \rightsquigarrow \ell, \mathcal{H}[\ell \mapsto v] \mid K^{\mathrm{cons}}} \quad \text{(E:Cons)}$$

$$\frac{\mathcal{V}(x) = \mathrm{Null} \qquad \mathcal{V}, \mathcal{H} \vdash e_1 \rightsquigarrow v, \mathcal{H}' \mid (q, q')}{\mathcal{V}, \mathcal{H} \vdash \mathit{match} \; x \; \mathit{with} \mid \mathit{nil} \rightarrow e_1 \mid \mathit{cons}(x_h, x_t) \rightarrow e_2 \rightsquigarrow v, \mathcal{H}' \mid K_1^{\mathrm{matN}} \cdot (q, q') \cdot K_2^{\mathrm{matN}}} \quad \text{(E:MatNil)}$$

$$\frac{\mathcal{V}(x) = \ell \qquad \mathcal{H}(\ell) = (v_h, v_t) \qquad \mathcal{V}[x_h \mapsto v_h, x_t \mapsto v_t], \mathcal{H} \vdash e_2 \rightsquigarrow v, \mathcal{H}' \mid (q, q')}{\mathcal{V}, \mathcal{H} \vdash \mathit{match} \; x \; \mathit{with} \mid \mathit{nil} \rightarrow e_1 \mid \mathit{cons}(x_h, x_t) \rightarrow e_2 \rightsquigarrow v, \mathcal{H}' \mid K_1^{\mathrm{matC}} \cdot (q, q') \cdot K_2^{\mathrm{matC}}} \quad \text{(E:MatCons)}$$

$$\frac{}{\mathcal{V}, \mathcal{H} \vdash \mathit{leaf} \rightsquigarrow \mathrm{Null}, \mathcal{H} \mid K^{\mathrm{leaf}}} \quad \text{(E:Leaf)}$$

$$\frac{x_0, x_1, x_2 \in \mathrm{dom}(\mathcal{V}) \qquad v = (\mathcal{V}(x_0), \mathcal{V}(x_1), \mathcal{V}(x_2)) \qquad \ell \notin \mathrm{dom}(\mathcal{H})}{\mathcal{V}, \mathcal{H} \vdash \mathit{node}(x_0, x_1, x_2) \rightsquigarrow \ell, \mathcal{H}[\ell \mapsto v] \mid K^{\mathrm{node}}} \quad \text{(E:Node)}$$

$$\frac{\mathcal{V}(x) = \mathrm{Null} \qquad \mathcal{V}, \mathcal{H} \vdash e_1 \rightsquigarrow v, \mathcal{H}' \mid (q, q')}{\mathcal{V}, \mathcal{H} \vdash \begin{array}{l} \mathit{match} \; x \; \mathit{with} \mid \mathit{leaf} \rightarrow e_1 \\ \mid \mathit{node}(x_0, x_1, x_2) \rightarrow e_2 \end{array} \rightsquigarrow v, \mathcal{H}' \mid K_1^{\mathrm{matTL}} \cdot (q, q') \cdot K_2^{\mathrm{matTL}}} \quad \text{(E:MatLeaf)}$$

$$\frac{\mathcal{V}(x) = \ell \qquad \mathcal{H}(\ell) = (v_0, v_1, v_2) \qquad \mathcal{V}[x_0 \mapsto v_0, x_1 \mapsto v_1, x_2 \mapsto v_2], \mathcal{H} \vdash e_2 \rightsquigarrow v, \mathcal{H}' \mid (q, q')}{\mathcal{V}, \mathcal{H} \vdash \begin{array}{l} \mathit{match} \; x \; \mathit{with} \mid \mathit{leaf} \rightarrow e_1 \\ \mid \mathit{node}(x_0, x_1, x_2) \rightarrow e_2 \end{array} \rightsquigarrow v, \mathcal{H}' \mid K_1^{\mathrm{matTN}} \cdot (q, q') \cdot K_2^{\mathrm{matTN}}} \quad \text{(E:MatNode)}$$

Fig. 1.   Evaluation rules of the big-step operational semantics.

It is handy to view the pairs (q, q') in the evaluation judgments as elements of a monoid $\mathcal{Q} = (\mathbb{Q}_0^+ \times \mathbb{Q}_0^+, \cdot)$. The neutral element is $(0, 0)$ which means that resources are neither needed before the evaluation nor restituted after the evaluation. The operation $(q, q') \cdot (p, p')$ defines how to account for an evaluation consisting of evaluations whose resource consumptions are defined by (q, q') and (p, p'), respectively. We define

$$(q, q') \cdot (p, p') = \begin{cases} (q + p - q', \; p') & \text{if } q' \leq p \\ (q, \; p' + q' - p) & \text{if } q' > p \end{cases}.$$

If resources are never restituted (as with time) then we can restrict to elements of the form $(q, 0)$ and $(q, 0) \cdot (p, 0)$ is just $(q + p, 0)$.

We identify a rational number q with an element of $\mathcal{Q}$ as follows: $q \geq 0$ denotes $(q, 0)$ and $q < 0$ denotes $(0, -q)$. This notation avoids case distinctions in the evaluation rules since the constants K that appear in the rules might be negative.

The following facts are often used in proofs.

PROPOSITION 3.1. *Let* $(q, q') = (r, r') \cdot (s, s')$.

(1) $q \geq r$ *and* $q - q' = r - r' + s - s'$
(2) *If* $(p, p') = (\bar{r}, r') \cdot (s, s')$ *and* $\bar{r} \geq r$ *then* $p \geq q$ *and* $p' = q'$
(3) *If* $(p, p') = (r, r') \cdot (\bar{s}, s')$ *and* $\bar{s} \geq s$ *then* $p \geq q$ *and* $p' \leq q'$
(4) $(r, r') \cdot ((s, s') \cdot (t, t')) = ((r, r') \cdot (s, s')) \cdot (t, t')$

The evaluation rules are standard apart from the resource information that measures the resource consumption. These annotation are very similar in each rule and we explain them exemplary for the rules E:VAR and E:CONDT.

Assume that the resource cost for looking up the value of a variable on the stack and copying it to some register is $K^{\text{var}} \geq 0$. The rule E:VAR then states that the resource consumption of the evaluation of a variable is $(K^{\text{var}}, 0)$. So the watermark of the resource consumption is K^{var} and there are no resources left after the evaluation. If $K^{\text{var}} < 0$ then E:VAR states that the resource consumption of the evaluation of a variable is $(0, K^{\text{var}})$. So the watermark is zero and after the evaluation there are K^{var} resources available.

Now consider the rule E:CONDT. Assume that the resource cost of looking up the value of the variable x and jumping to the source-code of e_t is $K_1^{\text{conT}} \geq 0$. Assume furthermore that jump to the code after the conditional costs $K_2^{\text{conT}} \geq 0$ resources. The rule E:CONDT states that the cost for the evaluation is $(K_1^{\text{conT}}, 0) \cdot (q, q') \cdot (K_2^{\text{conT}}, 0)$ if the watermark for the evaluation of e_t is q and if there are q' resources left after the evaluation. There are two cases. If $q' \geq K_2^{\text{conT}}$ then the overall watermark of the evaluation is $q + K_1^{\text{conT}}$ and there are $q' - K_2^{\text{conT}}$ resources available after the evaluation. If $q' < K_2^{\text{conT}}$ then the overall watermark of the evaluation is $q + K_1^{\text{conT}} + K_2^{\text{conT}} - q'$ and there are zero resources available after the evaluation. The statement is similar for negative constants K_i^{conT}.

The values of the constants $K_i^x \in \mathbb{Q}$ in the rules depend on the resource, the implementation, and the system architecture. In fact, the value of a constant can also be a function of the type of a subexpression. For instance, the size of a cons cell depends on the size of the value that is stored in the cell in our implementation. Since the types of all subexpressions are available at compile time, this is a straightforward extension.

Actual constants for stack-space, heap-space, and clock-cycle consumption were determined for the abstract machine of the language Hume [Hammond and Michaelson 2003] on the Renesas M32C/85U architecture. A list can be found in the literature [Jost et al. 2009].

The following proposition states that heap cells are never changed during an evaluation after they have been allocated. This is a convenient property to simplify some of the

$$\frac{v \in \{True, False\}}{\mathcal{H} \vDash v \mapsto v : bool} \text{ (V:BOOL)} \qquad \frac{v \in \mathbb{N}}{\mathcal{H} \vDash v \mapsto v : int} \text{ (V:INT)} \qquad \frac{v = \text{NULL}}{\mathcal{H} \vDash v \mapsto () : unit} \text{ (V:UNIT)}$$

$$\frac{v = (v_1, v_2) \qquad \mathcal{H} \vDash v_1 \mapsto a_1 : A_1 \qquad \mathcal{H} \vDash v_2 \mapsto a_2 : A_2}{\mathcal{H} \vDash v \mapsto (a_1, a_2) : (A_1, A_2)} \text{ (V:PAIR)}$$

$$\frac{v = \text{NULL} \qquad A \in \mathcal{A}}{\mathcal{H} \vDash v \mapsto [] : L(A)} \text{ (V:NIL)} \qquad \frac{v = \text{NULL} \qquad A \in \mathcal{A}}{\mathcal{H} \vDash v \mapsto leaf : T(A)} \text{ (V:LEAF)}$$

$$\frac{v \in \text{Loc}}{\mathcal{H}(v) = (v_1, v_2) \qquad \mathcal{H}' = \mathcal{H} \backslash v \qquad \mathcal{H}' \vDash v_1 \mapsto a_1 : A \qquad \mathcal{H}' \vDash v_2 \mapsto [a_2, \dots, a_n] : L(A)}{\mathcal{H} \vDash v \mapsto [a_1, \dots, a_n] : L(A)} \text{ (V:CONS)}$$

$$\frac{v \in \text{Loc} \qquad \mathcal{H}(v) = (v_0, v_1, v_2)}{\mathcal{H}' = \mathcal{H} \backslash v \qquad \mathcal{H}' \vDash v_0 \mapsto a : A \qquad \mathcal{H}' \vDash v_1 \mapsto t_1 : T(A) \qquad \mathcal{H}' \vDash v_2 \mapsto t_2 : T(A)}{\mathcal{H} \vDash v \mapsto tree(a, t_1, t_2) : T(A)} \text{ (V:NODE)}$$

Fig. 2.   Relating heap cells to semantic values.

later proofs but it is not necessarily needed. In fact, we could also allow the deallocation of memory cells. How to formally deal with this is described in the literature [Jost et al. 2009].

PROPOSITION 3.2. *Let e be an expression, $\mathcal{V}$ be a stack, and $\mathcal{H}$ be a heap. If $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (q, q')$ then $\mathcal{H}'(\ell) = \mathcal{H}(\ell)$ for all $\ell \in dom(\mathcal{H})$.*

PROOF. The only rules that allocate new heap cells are E:CONS and E:NODE. And in these rules we have the side condition $\ell \notin \mathcal{H}$ that prevents an old location from being changed by assigning a value to ℓ. $\square$

3.5. Well-Formed Environments

If $\mathcal{H}$ is a heap, v is a value, A is a type, and $a \in \llbracket A \rrbracket$ then we write $\mathcal{H} \vDash v \mapsto a : A$ to mean that v defines the semantic value $a \in \llbracket A \rrbracket$ when pointers are followed in $\mathcal{H}$ in the obvious way. The judgment is formally defined in Figure 2.

We write [] for the empty list. For a nonempty list $[a_1, \dots, a_n]$ we write $[a_1, \dots, a_n] = a_1 :: [a_2, \dots, a_n]$. The tree with root a, left subtree t_1, and right subtree t_2 is denoted by $tree(a, t_1, t_2)$. The empty tree is denoted by $leaf$. For a heap $\mathcal{H}$, we write $\mathcal{H}' = \mathcal{H} \backslash \ell$ for the heap in which the location ℓ is removed. That is, $dom(\mathcal{H}') = dom(\mathcal{H}) \backslash \{\ell\}$ and $\mathcal{H}'(\ell') = \mathcal{H}(\ell')$ for all $\ell' \in dom(\mathcal{H}')$.

Note that for every heap $\mathcal{H}$ there exist three pairs of semantic values a and data types A such that $\mathcal{H} \vDash \text{NULL} \mapsto a : A$; namely $a = ()$, $a = []$, and $a = leaf$. However, if we fix a data type A and a heap $\mathcal{H}$ then there exists at most one semantic value a such that $\mathcal{H} \vDash \text{NULL} \mapsto a : A$.

PROPOSITION 3.3. *Let $\mathcal{H}$ be a heap, v be a value, and let A be a data type. If $\mathcal{H} \vDash v \mapsto a : A$ and $\mathcal{H} \vDash v \mapsto a' : A$ then $a = a'$.*

PROOF. We prove the claim by induction on the derivation of $\mathcal{H} \vDash v \mapsto a : A$.

Assume first that $\mathcal{H} \vDash v \mapsto a : A$ has been derived by the application of a single rule. Then the judgment has been derived by one of the rules V:BOOL, V:INT, V:UNIT, V:NIL, or V:LEAF. An inspection of the rules shows that for given A and v only one of the rules is applicable. Thus it follows that $a = a'$.

Assume now that the derivation of $\mathcal{H} \vDash v \mapsto a : A$ ends with an application of the rule V:CONS. Then $A = L(B)$, $a = [a_1, \ldots, a_n]$, $v \in \text{Loc}$, and $\mathcal{H}(v) = (v_1, v_2)$. It follows that the derivation of $\mathcal{H} \vDash v \mapsto a' : A$ also ends with an application of V:CONS. Thus we have $a' = [b_1, \ldots, b_m]$. From the premises of V:CONS it follows that

$$\mathcal{H}' \vDash v_1 \mapsto a_1 : A$$
$$\mathcal{H}' \vDash v_2 \mapsto [a_2, \ldots, a_n] : L(A)$$
$$\mathcal{H}' \vDash v_1 \mapsto b_1 : A$$
$$\mathcal{H}' \vDash v_2 \mapsto [b_2, \ldots, b_m] : L(A)$$

where $\mathcal{H}' = \mathcal{H} \backslash v$. It follows by induction that $n = m$ and $b_i = a_i$ for all $1 \le i \le n$.

The cases in which the derivation ends with the V:NODE or V:PAIR are similar. $\square$

Note that if $\mathcal{H} \vDash v \mapsto a : A$ then v may well point to a data structure with some aliasing, but no circularity is allowed since this would require infinity values a. We do not include them because in our functional language there is no way of generating such values.

We write $\mathcal{H} \vDash v : A$ to indicate that there exists a, necessarily unique, semantic value $a \in [\![A]\!]$ so that $\mathcal{H} \vDash v \mapsto a : A$. A stack $\mathcal{V}$ and a heap $\mathcal{H}$ are *well-formed* with respect to a context Γ if $\mathcal{H} \vDash \mathcal{V}(x) : \Gamma(x)$ holds for every $x \in \text{dom}(\Gamma)$. We then write $\mathcal{H} \vDash \mathcal{V} : \Gamma$.

Theorem 3.4 shows that the evaluation of a well-typed expression in a well-formed environment results in a well-formed environment.

THEOREM 3.4. *If* $\Sigma; \Gamma \vdash e : B$, $\mathcal{H} \vDash \mathcal{V} : \Gamma$ *and* $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (q, q')$ *then* $\mathcal{H}' \vDash \mathcal{V} : \Gamma$ *and* $\mathcal{H}' \vDash v : B$.

A proof of Theorem 3.4 is given in the first author's PhD thesis [Hoffmann 2011].

3.6. Partial Big-Step Operational Semantics

A notorious dissatisfying feature of classical big-step semantics is that it does not provide evaluation judgments for nonterminating evaluations. This is problematic if one intends to prove statements for all computations (divergent and convergent) that do not go wrong—that is, for all computations that result from the evaluation of well-typed programs.

A straightforward remedy is to use a small-step semantics to describe computations. But in the context of resource analysis, the use of big-step rules seems to be more favorable. Firstly, big-step rules can more directly axiomatize the resource behavior of compiled code on specific machines. Secondly, it allows for shorter and less syntactic proofs.

Another classic approach [Cousot and Cousot 1992; Leroy 2006] is to add divergence rules to the operational semantics that are interpreted coinductively. But then one loses the ability to prove statements by induction on the evaluation which is crucial for the proof of the soundness theorem (Theorem 6.7). It should also be possible to work with a coinductive definition in the style of Cousot or Leroy [Cousot and Cousot 1992; Leroy 2006]. However, coinductive semantics lends itself less well to formulating and proving semantic soundness theorems of the form "if the program is well-typed and the operational semantics says X then Y holds". For example, in Leroy's Lemmas 17 through 22 [Leroy 2006] the coinductive definition appears in the conclusion rather than as a premise.

That is why we use a different approach that we proposed in a companion paper [Hoffmann and Hofmann 2010a]. We define a *big-step semantics for partial evaluations* that directly corresponds to the rules of the big-step semantics in Figure 1. It defines a statement of the form

$$\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \mid q$$

$$\frac{}{\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \mid 0} \ \text{(P:Zero)} \qquad \frac{b \in \{\textit{True}, \textit{False}\}}{\mathcal{V}, \mathcal{H} \vdash b \rightsquigarrow \mid K^{\text{bool}}} \ \text{(P:ConstB)} \qquad \frac{}{\mathcal{V}, \mathcal{H} \vdash () \rightsquigarrow \mid K^{\text{unit}}} \ \text{(P:ConstU)}$$

$$\frac{n \in \mathbb{Z}}{\mathcal{V}, \mathcal{H} \vdash n \rightsquigarrow \mid K^{\text{int}}} \ \text{(P:ConstI)} \qquad \frac{x \in \text{dom}(\mathcal{V})}{\mathcal{V}, \mathcal{H} \vdash x \rightsquigarrow \mid K^{\text{var}}} \ \text{(P:Var)} \qquad \frac{x_1, x_2 \in \text{dom}(\mathcal{V})}{\mathcal{V}, \mathcal{H} \vdash (x_1, x_2) \rightsquigarrow \mid K^{\text{pair}}} \ \text{(P:Pair)}$$

$$\frac{\mathcal{V}(x) = v \quad [y_f \mapsto v], \mathcal{H} \vdash e_f \rightsquigarrow \mid q}{\mathcal{V}, \mathcal{H} \vdash f(x) \rightsquigarrow \mid K_1^{\text{app}} + q} \ \text{(P:App)} \qquad \frac{\mathcal{V}, \mathcal{H} \vdash e_1 \rightsquigarrow \mid q}{\mathcal{V}, \mathcal{H} \vdash \textit{let } x = e_1 \textit{ in } e_2 \rightsquigarrow \mid K_1^{\text{let}} + q} \ \text{(P:Let1)}$$

$$\frac{\begin{array}{c}\mathcal{V}, \mathcal{H} \vdash e_1 \rightsquigarrow v_1, \mathcal{H}_1 \mid (q, q') \\ \mathcal{V}[x \mapsto v_1], \mathcal{H}_1 \vdash e_2 \rightsquigarrow \mid p \quad K_1^{\text{let}} \cdot (q, q') \cdot K_2^{\text{let}} \cdot (p, 0) = (r, r')\end{array}}{\mathcal{V}, \mathcal{H} \vdash \textit{let } x = e_1 \textit{ in } e_2 \rightsquigarrow \mid r} \ \text{(P:Let2)}$$

$$\frac{\mathcal{V}(x) = \textit{True} \quad \mathcal{V}, \mathcal{H} \vdash e_t \rightsquigarrow \mid q}{\mathcal{V}, \mathcal{H} \vdash \textit{if } x \textit{ then } e_t \textit{ else } e_f \rightsquigarrow \mid K_1^{\text{conT}} + q} \ \text{(P:CondT)} \qquad \frac{x_1, x_2 \in \text{dom}(\mathcal{V})}{\mathcal{V}, \mathcal{H} \vdash x_1 \textit{ op } x_2 \rightsquigarrow \mid K^{\text{op}}} \ \text{(P:BinOp)}$$

$$\frac{\mathcal{V}(x) = \textit{False} \quad \mathcal{V}, \mathcal{H} \vdash e_f \rightsquigarrow \mid q}{\mathcal{V}, \mathcal{H} \vdash \textit{if } x \textit{ then } e_t \textit{ else } e_f \rightsquigarrow \mid K_1^{\text{conF}} + q} \ \text{(P:CondF)} \qquad \frac{x_h, x_t \in \text{dom}(\mathcal{V})}{\mathcal{V}, \mathcal{H} \vdash \textit{cons}(x_h, x_t) \rightsquigarrow \mid K^{\text{cons}}} \ \text{(P:Cons)}$$

$$\frac{\mathcal{V}(x) = (v_1, v_2) \quad \mathcal{V}[x_1 \mapsto v_1, x_2 \mapsto v_2], \mathcal{H} \vdash e \rightsquigarrow \mid q}{\mathcal{V}, \mathcal{H} \vdash \textit{match } x \textit{ with } (x_1, x_2) \to e \rightsquigarrow \mid K_1^{\text{matP}} + q} \ \text{(P:MatP)} \qquad \frac{}{\mathcal{V}, \mathcal{H} \vdash \textit{nil} \rightsquigarrow \mid K^{\text{nil}}} \ \text{(P:Nil)}$$

$$\frac{\mathcal{V}(x) = \textsc{Null} \quad \mathcal{V}, \mathcal{H} \vdash e_1 \rightsquigarrow \mid q}{\mathcal{V}, \mathcal{H} \vdash \textit{match } x \textit{ with } \mid \textit{nil} \to e_1 \mid \textit{cons}(x_h, x_t) \to e_2 \rightsquigarrow \mid K_1^{\text{matN}} + q} \ \text{(P:MatNil)}$$

$$\frac{\mathcal{V}(x) = \ell \quad \mathcal{H}(\ell) = (v_h, v_t) \quad \mathcal{V}[x_h \mapsto v_h, x_t \mapsto v_t], \mathcal{H} \vdash e_2 \rightsquigarrow \mid q}{\mathcal{V}, \mathcal{H} \vdash \textit{match } x \textit{ with } \mid \textit{nil} \to e_1 \mid \textit{cons}(x_h, x_t) \to e_2 \rightsquigarrow \mid K_1^{\text{matC}} + q} \ \text{(P:MatCons)}$$

$$\frac{}{\mathcal{V}, \mathcal{H} \vdash \textit{leaf} \rightsquigarrow \mid K^{\text{leaf}}} \ \text{(P:Leaf)} \qquad \frac{x_0, x_1, x_2 \in \text{dom}(\mathcal{V})}{\mathcal{V}, \mathcal{H} \vdash \textit{node}(x_0, x_1, x_2) \rightsquigarrow \mid K^{\text{node}}} \ \text{(P:Node)}$$

$$\frac{\mathcal{V}(x) = \textsc{Null} \quad \mathcal{V}, \mathcal{H} \vdash e_1 \rightsquigarrow \mid q}{\mathcal{V}, \mathcal{H} \vdash \textit{match } x \textit{ with } \mid \textit{leaf} \to e_1 \mid \textit{node}(x_0, x_1, x_2) \to e_2 \rightsquigarrow \mid K_1^{\text{matTL}} + q} \ \text{(P:MatLeaf)}$$

$$\frac{\mathcal{V}(x) = \ell \quad \mathcal{H}(\ell) = (v_0, v_1, v_2) \quad \mathcal{V}[x_0 \mapsto v_0, x_1 \mapsto v_1, x_2 \mapsto v_2], \mathcal{H} \vdash e_2 \rightsquigarrow \mid q}{\mathcal{V}, \mathcal{H} \vdash \textit{match } x \textit{ with } \mid \textit{leaf} \to e_1 \mid \textit{node}(x_0, x_1, x_2) \to e_2 \rightsquigarrow \mid K_1^{\text{matTN}} + q} \ \text{(P:MatNode)}$$

Fig. 3.   Partial big-step operational semantics.

where $\mathcal{V}$ is a stack, $\mathcal{H}$ is a heap, $q \in \mathbb{Q}_0^+$, and e is an expression. The meaning is that there is a partial evaluation of e with the initial stack $\mathcal{V}$ and the initial heap $\mathcal{H}$ that consumes q resources. Here, q is the watermark of the resource usage. We do not have to keep track of the restituted resources since a partial evaluation consists always of a complete evaluation followed by a partial evaluation.

Note that the rule P:ZERO is essential for the partiality of the semantics. It can be applied at any point to stop the evaluation and thus yields to a nondeterministic evaluation judgment.

The rule P:VAR can be understood as follows. To partially evaluate a variable, one can only do one evaluation step, namely evaluating the variable thereby producing the cost K^{var} if $K^{\mathrm{var}} > 0$ and zero cost otherwise.

The rule P:LET1 can be read as follows. If there is a partial evaluation of e_1 that needs q resources then one can partially evaluate let $x = e_1$ in e_2 by starting the evaluation of the let expression which costs $K_1^{\mathrm{let}} \geq 0$ or reimburses $K_1^{\mathrm{let}} < 0$ resources. Then one can partially evaluate e_1, deriving a partial evaluation of the let expression that produces the watermark $K_1^{\mathrm{let}} + q$.

Theorem 3.5 proves that if an expression converges in a given environment then the resource-usage watermark of the evaluation is an upper bound for the resource usage of every partial evaluation of the expression in that environment.

THEOREM 3.5. *If $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (q, q')$ and $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \mid p$ then $p \leq q$.*

PROOF. By induction on the derivation D of the judgment $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (q, q')$. To prove the *induction basis* let D consist of one step. Then e is a constant c, a variable x, a binary operation $x_1 \ op \ x_2$, a pair (x_1, x_2), the constant nil, leaf, $\mathrm{cons}(x_1, x_2)$, or $\mathrm{node}(x_1, x_2, x_3)$. Let e be, for instance, a variable x. Then by definition of E:VAR it follows that $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (K^{\mathrm{var}}, 0)$ or $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (0, -K^{\mathrm{var}})$. Thus $q = \max(0, K^{\mathrm{var}})$. The only P-rules that apply to x are P:VAR and P:ZERO. Thus it follows that if $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \mid p$ then $p = \max(0, K^{\mathrm{var}})$. The other cases are similar.

For the induction step assume that $|D| > 1$. Then e is a pattern match, a function application, a conditional, or a let expression. For instance, let e be the expression let $x = e_1$ in e_2. Then it follows from rule E:LET that $\mathcal{V}, \mathcal{H} \vdash e_1 \rightsquigarrow v_1, \mathcal{H}_1 \mid (q_1, q_1')$, $\mathcal{V}[x \mapsto v_1], \mathcal{H}_1 \vdash e_2 \rightsquigarrow v_2, \mathcal{H}_2 \mid (q_2, q_2')$ and

$$(q, q') = K_1^{\mathrm{let}} \cdot (q_1, q_1') \cdot K_2^{\mathrm{let}} \cdot (q_2, q_2') \cdot K_3^{\mathrm{let}}. \tag{1}$$

By induction we conclude

$$\text{if } \mathcal{V}, \mathcal{H} \vdash e_1 \rightsquigarrow \mid p_1 \text{ then } p_1 \leq q_1 \tag{2}$$

$$\text{if } \mathcal{V}[x \mapsto v_1], \mathcal{H}_1 \vdash e_2 \rightsquigarrow \mid p_2 \text{ then } p_2 \leq q_2. \tag{3}$$

Now let $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \mid p$. Then this judgment was derived via the rules P:LET1 or P:LET2. In the first case it follows by definition that $p = \max(p_1 + K_1^{\mathrm{let}}, 0)$ for some p_1 and $p_1 \leq q_1$ by (2). Therefrom it follows with (1) that $p \leq q$.

If $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \mid p$ was derived by P:LET2 then it follows that $(p, p') = K_1^{\mathrm{let}} \cdot (q_1, q_1') \cdot K_2^{\mathrm{let}} \cdot (p_2, 0)$ for some p', p_2. We conclude from (3) that $p_2 \leq q_2$ and hence from Proposition 3.1 and (1) $p \leq q$. The other cases are similar to the case P:LET1.   □

Theorem 3.9 states that, in a well-formed environment, every well-typed expression either diverges or evaluates to a value of the stated type. To this end we instantiate the resource constants in the rules to count the number of evaluation steps.

PROPOSITION 3.6. *Let the resource constants be instantiated by $K^x = 1$, $K_1^x = 1$ and $K_y^x = 0$ for all x and all $y > 1$. Let $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (q, q')$ and let the derivation of the judgment have n steps. Then $q = n$ and $q' = 0$.*

PROOF. By induction on the derivation D of $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (q, q')$.

If D consists of only one step ($|D| = 1$) then e is a constant c, a variable x, a binary operation $x_1 \ op \ x_2$, a pair (x_1, x_2), the constant nil, leaf, $\mathrm{cons}(x_1, x_2)$, or $\mathrm{node}(x_1, x_2, x_3)$. In each case, $q = 1$ and $q' = 0$ follows immediately from the respective evaluation rule.

Now let $|D| > 1$. Then e is a pattern match, a function application, a conditional, or a let expression. For instance, let e be the expression let $x = e_1$ in e_2. Then it follows from rule E:LET that $\mathcal{V}, \mathcal{H} \vdash e_1 \rightsquigarrow v_1, \mathcal{H}_1 \mid (q_1, q_1')$, $\mathcal{V}[x \mapsto v_1], \mathcal{H}_1 \vdash e_2 \rightsquigarrow v_2, \mathcal{H}_2 \mid (q_2, q_2')$ and

$$(q, q') = 1 \cdot (q_1, q_1') \cdot 0 \cdot (q_2, q_2') \cdot 0 = (1 + q_1, q_1') \cdot (q_2, q_2').$$

Let n_1 be the evaluation steps needed by e_1 and let n_2 be the number of evaluation steps needed by e_2. By induction it follows that $q_1 = n_1$, $q_2 = n_2$ and $q_1' = q_2' = 0$. Thus $q = n_1 + n_2 + 1 = n$.

The other cases are similar. □

The following lemma shows that if there is a complete evaluation that uses n steps then there are partial evaluations that use i steps for $0 \le i \le n$. It is used in the proof of Theorem 3.9 with $i = n$.

LEMMA 3.7. *Let the resource constants be instantiated by $K^x = 1$, $K_1^x = 1$ and $K_m^x = 0$ for all x and all $m > 1$. If $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (n, 0)$ then $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \mid i$ for every $0 \le i \le n$.*

PROOF. By induction on the derivation D of $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (n, 0)$. The proof is very similar to the proof of Theorem 3.5. □

Lemma 3.8 proves that one can always make one partial evaluation step for a well-typed expression in a well-formed environment. It is used in the induction basis of the proof of Theorem 3.9.

LEMMA 3.8. *Let the resource constants be instantiated by $K^x = 1$, $K_1^x = 1$ and $K_m^x = 0$ for all x and all $m > 1$. If $\Sigma; \Gamma \vdash e : A$, $\mathcal{H} \vDash \mathcal{V} : \Gamma$ then $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \mid 1$.*

PROOF. By case distinction on e. The proof is straightforward so we only demonstrate two characteristic cases.

Let e, for instance, be a variable x. Then it follows from $\Sigma; \Gamma \vdash x : A$ and $\mathcal{H} \vDash \mathcal{V} : \Gamma$ that $x \in \mathcal{V}$. Thus $\mathcal{V}, \mathcal{H} \vdash x \rightsquigarrow \mid 1$ by (P:VAR).

Let e now be a conditional if x then e_t else e_f. Then it follows from $\Sigma; \Gamma \vdash e : A$ and $\mathcal{H} \vDash \mathcal{V} : \Gamma$ that $\mathcal{V}(x) \in \{\text{True}, \text{False}\}$. Furthermore, we derive $\mathcal{V}, \mathcal{H} \vdash e_t \rightsquigarrow \mid 0$ and $\mathcal{V}, \mathcal{H} \vdash e_f \rightsquigarrow \mid 0$ with the rule P:ZERO. Thus we can use either P:CONDT or P:CONDF to derive $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \mid 1$. □

THEOREM 3.9. *Let the resource constants be instantiated by $K^x = 1$, $K_1^x = 1$ and $K_m^x = 0$ for all x and all $m > 1$. If $\Sigma; \Gamma \vdash e : A$ and $\mathcal{H} \vDash \mathcal{V} : \Gamma$ then $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (n, 0)$ for an $n \in \mathbb{N}$ or $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \mid m$ for every $m \in \mathbb{N}$.*

PROOF. We show by induction on n that if

$$\Sigma; \Gamma \vdash e : A, \qquad \mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \mid n \qquad \text{and} \qquad \mathcal{H} \vDash \mathcal{V} : \Gamma \qquad (4)$$

then $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (n, 0)$ or $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \mid n + 1$. Then Theorem 3.9 follows since $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \mid 0$ for every $\mathcal{V}, \mathcal{H}$ and e.

Induction basis $n = 0$: We use Lemma 3.8 to conclude from the well-formedness of the environment (4) that $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \mid 1$.

Induction step $n > 0$: Assume (4). If e is a constant c, a variable x, a binary operation x_1 op x_2, a pair (x_1, x_2), the constant nil, or cons(x_1, x_2), then $n = 1$ and we derive $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (1, 0)$ immediately from the corresponding evaluation rule.

If e is a pattern match, a function application, a conditional, or a let expression then we use the induction hypothesis. Since the other cases are similar, we provide the argument only for the case where e is a let expression let $x = e_1$ in e_2. Then $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \mid n$ was derived via P:LET1 or P:LET2. In the case of P:LET1 it follows that $\mathcal{V}, \mathcal{H} \vdash e_1 \rightsquigarrow \mid n - 1$. By the induction hypothesis we conclude that either $\mathcal{V}, \mathcal{H} \vdash e_1 \rightsquigarrow \mid n$ or $\mathcal{V}, \mathcal{H} \vdash e_1 \rightsquigarrow v_1, \mathcal{H}_1 \mid (n - 1, 0)$. In the first case we can use P:LET1 to derive $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \mid n + 1$.

In the second case it follows from Theorem 3.4 that $\mathcal{H}_1 \vDash \mathcal{V} : \Gamma$ and $\mathcal{H}_1 \vDash v_1{:}A$ and thus $\mathcal{H}_1 \vDash \mathcal{V}[x \mapsto v_1]{:}\Gamma, x{:}A$. We then apply Lemma 3.8 to obtain $\mathcal{V}[x \mapsto v_1], \mathcal{H}_1 \vdash e_2 \rightsquigarrow | 1$. Therefore we can apply P:LET2 to derive $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow | n+1$.

Assume now that e was derived by the use of P:LET2. Then it is true that $\mathcal{V}, \mathcal{H} \vdash e_1 \rightsquigarrow v_1, \mathcal{H}_1 | (n_1, 0)$ and $\mathcal{V}[x \mapsto v_1], \mathcal{H}_1 \vdash e_2 \rightsquigarrow | n_2$ for some n_1, n_2 with $n_1 + n_2 + 1 = n$. From Theorem 3.4 it follows that $\mathcal{H}_1 \vDash \mathcal{V}[x \mapsto v_1]{:}\Gamma, x{:}A$. Therefore we can apply the induction hypothesis to infer that $\mathcal{V}[x \mapsto v_1], \mathcal{H}_1 \vdash e_2 \rightsquigarrow v_2, \mathcal{H}_2 | (n_2, 0)$ or $\mathcal{V}[x \mapsto v_1], \mathcal{H}_1 \vdash e_2 \rightsquigarrow | n_2 + 1$. In the first case we apply E:LET and derive $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v_2, \mathcal{H}_2 | (n, 0)$. In the second case we apply P:LET2 and derive $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow | n+1$. □

3.7. The Cost-Free Resource Metric

The typing rules in Section 6 make use of the *cost-free* resource metric. This is the metric in which all constants K that appear in the rules are instantiated to zero. It follows that if $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' | (q, q')$ then $q = q' = 0$. We will use the cost-free metric in Section 6 to pass on potential in the typing rule for let expressions.

The following proposition is direct.

PROPOSITION 3.10. *Let all resource constants K be instantiated by $K = 0$. If $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' | (q, q')$ then $q = q' = 0$. If $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow | q$ then $q = 0$.*

4. RESOURCE POLYNOMIALS

A resource polynomial maps a value of some data type to a nonnegative rational number. Potential functions are always given by such resource polynomials.

In the case of an inductive tree-like data type, a resource polynomial will only depend on the list of entries of the data structure in preorder. Thus, if $D(A)$ is such a data type with entries of type A, for example, A-labeled binary trees, and v is a value of type $D(A)$ then we write $elems(v) = [a_1, \ldots, a_n]$ for this list of entries. For example, if $D(A)$ is the set of binary, node-labeled trees then we have $elems(leaf) = []$ and $elems(tree(a, t_1, t_2)) = a :: elems(t_1)elems(t_2)$.

An analysis of typical polynomial computations operating on a data structure v with $elems(v) = [a_1, \ldots, a_n]$ shows that it consists of operations that are executed for every k-tuple $(a_{i_1}, \ldots, a_{i_k})$ with $1 \leq i_1 < \cdots < i_k \leq n$. The simplest examples are linear map operations that perform some operation for every a_i. Other common examples are sorting algorithms that perform comparisons for every pair (a_i, a_j) with $1 \leq i < j \leq n$ in the worst case.

4.1. Base Polynomials

For each data type A we now define a set $\mathcal{P}(A)$ of functions $p : [\![A]\!] \to \mathbb{N}$ that map values of type A to natural numbers. The resource polynomials for type A are then given as nonnegative rational linear combinations of these *base polynomials*.

We use the notation $a \mapsto e(a)$ for the anonymous function that maps an argument a to the natural number that is defined by the expression $e(a)$. In the lambda calculus one would write $\lambda a \,.\, e(a)$ instead of $a \mapsto e(a)$. We define $\mathcal{P}(A)$ as follows.

$$\mathcal{P}(A) = \{a \mapsto 1\} \text{ if } A \text{ is an atomic type}$$
$$\mathcal{P}(A_1, A_2) = \{(a_1, a_2) \mapsto p_1(a_1) \cdot p_2(a_2) \mid p_i \in \mathcal{P}(A_i)\}$$
$$\mathcal{P}(D(A)) = \left\{ v \mapsto \sum_{1 \leq j_1 < \cdots < j_k \leq n} \prod_{i=1,\ldots,k} p_i(a_{j_i}) \,\middle|\, k \in \mathbb{N}, p_i \in \mathcal{P}(A) \right\}$$

In the last clause $[a_1, \ldots, a_n] = elems(v)$. Every set $\mathcal{P}(A)$ contains the constant function $v \mapsto 1$. In the case of $D(A)$ this arises for $k = 0$ (one element sum, empty product).

In the case of lists, the intuition is that—for a fixed k—we process all ordered k-tuples that can be formed from the elements of the list. The cost for processing one of the k-tuples is given by a product of base polynomials and we sum up these polynomials to obtain a base polynomial for the list. For example, the function $\ell \mapsto \binom{|\ell|}{k}$ is in $\mathcal{P}(L(A))$ for every $k \in \mathbb{N}$; simply take $p_1 = \cdots = p_k = 1$ in the definition of $\mathcal{P}(D(A))$. The function $(\ell_1, \ell_2) \mapsto \binom{|\ell_1|}{k_1} \cdot \binom{|\ell_2|}{k_2}$ is in $\mathcal{P}(L(A), L(B))$ for every $k_1, k_2 \in \mathbb{N}$ and $[\ell_1, \ldots, \ell_n] \mapsto \sum_{1 \le i < j \le n} \binom{|\ell_i|}{k_1} \cdot \binom{|\ell_j|}{k_2} \in \mathcal{P}(L(L(A)))$ for every $k_1, k_2 \in \mathbb{N}$.

4.2. Resource Polynomials

A *resource polynomial* $p : [\![A]\!] \to \mathbb{Q}_0^+$ for a data type A is a nonnegative linear combination of base polynomials, that is,

$$p = \sum_{i=1,\ldots,m} q_i \cdot p_i$$

for $m \in \mathbb{N}$, $q_i \in \mathbb{Q}_0^+$ and $p_i \in \mathcal{P}(A)$. We write $\mathcal{R}(A)$ for the set of resource polynomials for A.

An instructive, but not exhaustive, example is given by $\mathcal{R}_n = \mathcal{R}(L(\text{int}), \ldots, L(\text{int}))$. The set $\mathcal{R}_n$ is the set of linear combinations of products of binomial coefficients over variables $x_1, \ldots, x_n$, that is, $\mathcal{R}_n = \{\sum_{i=1}^m q_i \prod_{j=1}^n \binom{x_j}{k_{ij}} \mid q_i \in \mathbb{Q}_0^+, m \in \mathbb{N}, k_{ij} \in \mathbb{N}\}$. (As always, we have $0 \in \mathbb{N}$.) Here, the variable x_i represents the lengths of ith list. These expressions naturally generalize the polynomials used in our univariate analysis [Hoffmann and Hofmann 2010b] and meet two conditions that are important to efficiently manipulate polynomials during the analysis. First, the polynomials are nonnegative, and secondly, they are closed under the discrete difference operators Δ_i for every i. The discrete derivative $\Delta_i p$ is defined through $\Delta_i p(x_1, \ldots, x_n) = p(x_1, \ldots, x_i + 1, \ldots, x_n) - p(x_1, \ldots, x_n)$.

As in Hoffmann and Hofmann [2010b] it can be shown that $\mathcal{R}_n$ is the largest set of polynomials enjoying these closure properties. It would be interesting to have a similar characterization of $\mathcal{R}(A)$ for arbitrary A. So far, we know that $\mathcal{R}(A)$ is closed under sum and product (see Lemma 5.1) and are compatible with the construction of elements of data structures in a very natural way (see Lemmas 5.3 and 5.4). This provides some justification for their choice and canonicity. An abstract characterization would have to take into account the fact that our resource polynomials depend on an unbounded number of variables, for example, sizes of inner data structures, and are not invariant under permutation of these variables. See Section 9 for a more detailed discussion.

5. ANNOTATED TYPES

The resource polynomials described in Section 4 are nonnegative linear combinations of base polynomials. The rational coefficients of the linear combination are present as type annotations in our type system. To relate type annotations to resource polynomials we systematically describe base polynomials and resource polynomials for data of a given type.

If one considers only univariate polynomials then their description is straightforward. Every inductive data of size n admits a potential of the form $\sum_{1 \le i \le k} q_i \binom{n}{i}$. So we can describe the potential function with a vector $\vec{q} = (q_1, \ldots, q_k)$ in the corresponding recursive type. For instance, we can write $L^{\vec{q}}(A)$ for annotated list types. Since each annotation refers to the size of one input part only, univariately annotated types can be directly composed. For example, an annotated type for a pair of lists has the form $(L^{\vec{q}}(A), L^{\vec{p}}(A))$. See Hoffmann and Hofmann [2010b] for details.

Here, we work with multivariate potential functions, that is, functions that depend on the sizes of different parts of the input. For a pair of lists of lengths n and m we have, for instance, a potential function of the form $\sum_{0 \le i+j \le k} q_{ij} \binom{n}{i} \binom{m}{j}$ which can be described by the coefficients q_{ij}. But we also want to describe potential functions that refer to the sizes of different lists inside a list of lists, etc. That is why we need to describe a set of indexes $I(A)$ that enumerate the basic resource polynomials p_i and the corresponding coefficients q_i for a data type A. These type annotations can be, in a straightforward way, automatically transformed into usual easily understood polynomials. This is done in our prototype to present the bounds to the user at the end of the analysis.

5.1. Names For Base Polynomials

To assign a unique name to each base polynomial we define the *index set* $I(A)$ to denote resource polynomials for a given data type A. Interestingly, but as we find coincidentally, $I(A)$ is essentially the meaning of A with every atomic type replaced by unit. For example, the indices for pair types are pairs and the indices for list types are lists, etc.

$$I(A) = \{*\} \text{ if } A \in \{\text{int}, \text{bool}, \text{unit}\}$$
$$I(A_1, A_2) = \{(i_1, i_2) \mid i_1 \in I(A_1) \text{ and } i_2 \in I(A_2)\}$$
$$I(L(B)) = I(T(B)) = \{[i_1, \dots, i_k] \mid k \ge 0, i_j \in I(B)\}$$

The *degree* $\deg(i)$ of an index $i \in I(A)$ is defined as follows.

$$\deg(*) = 0$$
$$\deg(i_1, i_2) = \deg(i_1) + \deg(i_2)$$
$$\deg([i_1, \dots, i_k]) = k + \deg(i_1) + \cdots + \deg(i_k)$$

Define $I_k(A) = \{i \in I(A) \mid \deg(i) \le k\}$. The indexes $i \in I_k(A)$ are an enumeration of the base polonomials $p_i \in \mathcal{P}(A)$ of degree at most k. For each $i \in I(A)$, we define a base polynomial $p_i \in \mathcal{P}(A)$ as follows: If $A \in \{\text{int}, \text{bool}, \text{unit}\}$ then

$$p_*(v) = 1.$$

If $A = (A_1, A_2)$ is a pair type and $v = (v_1, v_2)$ then

$$p_{(i_1, i_2)}(v) = p_{i_1}(v_1) \cdot p_{i_2}(v_2).$$

If $A = D(B)$ (in our type system D is either lists or binary node-labeled trees) is a data structure and $elems(v) = [v_1, \dots, v_n]$ then

$$p_{[i_1, \dots, i_m]}(v) = \sum_{1 \le j_1 < \cdots < j_m \le n} p_{i_1}(v_{j_1}) \cdots p_{i_m}(v_{j_m}).$$

We use the notation 0_A (or just 0) for the index in $I(A)$ such that $p_{0_A}(a) = 1$ for all a. We have $0_{\text{int}} = *$ and $0_{(A_1, A_2)} = (0_{A_1}, 0_{A_2})$ and $0_{D(B)} = []$. If $A = D(B)$ for B a data type then the index $[0, \dots, 0] \in I(A)$ of length n is denoted by just n. We identify the index (i_1, i_2, i_3, i_4) with the index $(i_1, (i_2, (i_3, i_4)))$.

For a list $i = [i_1, \dots, i_k]$ we write $i_0::i$ to denote the list $[i_0, i_1, \dots, i_k]$. Furthermore, we write ii' for the concatenation of two lists i and i'.

Recall that $\mathcal{R}(A)$ denotes the set of nonnegative rational linear combinations of the base polynomials. For $p \in \mathcal{R}(A)$ we define $\deg(p) = \deg(i)$ where $i \in I(A)$ is the unique index such that $p_i = p$.

LEMMA 5.1. *If* $p, p' \in \mathcal{R}(A)$ *then* $p + p', p \cdot p' \in \mathcal{R}(A)$, *and* $\deg(p + p') = \max\{\deg(p), \deg(p')\}$ *and* $\deg(p \cdot p') = \deg(p) + \deg(p')$.

PROOF. By linearity it suffices to show this lemma for base polynomials, as no cancellation can occur since our linear combinations are nonnegative.

A simple induction on A shows the claim for the base polynomials. □

COROLLARY 5.2. *For every $p \in \mathcal{R}(A, A)$ there exists $p' \in \mathcal{R}(A)$ with $\deg(p') = \deg(p)$ and $p'(a) = p(a, a)$ for all $a \in [\![A]\!]$.*

PROOF. Since p is a linear combination of base polynomials it is sufficient to show the corollary for base polynomials. Assume thus that $p \in \mathcal{P}(A, A)$. By definition there are $p_i, p_j \in \mathcal{P}(A)$ such that $p(a, a') = p_i(a) \cdot p_j(a')$ all $a, a' \in [\![A]\!]$. From Lemma 5.1 it follows that $p' = p_i \cdot p_j \in \mathcal{P}(A)$. Per definition, we have for all $a \in [\![A]\!]$ that $p'(a) = p_i(a) \cdot p_j(a) = p(a, a)$ and $\deg(p') = \deg(p)$. □

LEMMA 5.3. *Let $a \in [\![A]\!]$ and $\ell \in [\![L(A)]\!]$. Let $i_0, \ldots, i_k \in I(A)$ and $k \geq 0$. Then $p_{[i_0, i_1, \ldots, i_k]}([]) = 0$ and $p_{[i_0, i_1, \ldots, i_k]}(a::\ell) = p_{i_0}(a) \cdot p_{[i_1, \ldots, i_k]}(\ell) + p_0(a) \cdot p_{[i_0, i_1, \ldots, i_k]}(\ell)$.*

PROOF. Recall that $p_0(a) = 1$ for all a and let $\ell = [v_1, \ldots, v_n]$. Writing v_0 for a we compute as follows, splitting the sum into the case where v_0 is chosen and the case where v_0 is not chosen.

$$
\begin{aligned}
p_{[i_0, i_1, \ldots, i_k]}(a::\ell) &= \sum_{0 \leq j_0 < j_1 < \cdots < j_m \leq n} p_{i_o}(v_{j_0}) \cdot p_{i_1}(v_{j_1}) \cdots p_{i_m}(v_{j_m}) \\
&= \sum_{1 \leq j_1 < \cdots < j_m \leq n} p_{i_o}(v_0) \cdot p_{i_1}(v_{j_1}) \cdots p_{i_m}(v_{j_m}) \\
&\quad + \sum_{1 \leq j_0 < j_1 < \cdots < j_m \leq n} p_{i_o}(v_{j_0}) \cdot p_{i_1}(v_{j_1}) \cdots p_{i_m}(v_{j_m}) \\
&= p_{i_o}(a) \cdot \sum_{1 \leq j_1 < \cdots < j_m \leq n} p_{i_1}(v_{j_1}) \cdots p_{i_m}(v_{j_m}) \\
&\quad + \sum_{1 \leq j_0 < j_1 < \cdots < j_m \leq n} p_{i_o}(v_{j_0}) \cdot p_{i_1}(v_{j_1}) \cdots p_{i_m}(v_{j_m}) \\
&= p_{i_0}(a) \cdot p_{[i_1, \ldots, i_k]}(\ell) + p_0(a) \cdot p_{[i_0, i_1, \ldots, i_k]}(\ell)
\end{aligned}
$$

The statement $p_{[i_0, i_1, \ldots, i_k]}([]) = 0$ is obvious as the sum in the definition of the corresponding base polynomial is over the empty index set. □

Lemma 5.4 characterizes concatenations of lists (written as juxtaposition) as they will occur in the construction of tree-like data.

LEMMA 5.4. *Let $\ell_1, \ell_2 \in [\![L(A)]\!]$. Then $\ell_1 \ell_2 \in [\![L(A)]\!]$ and $p_{[i_1, \ldots, i_k]}(\ell_1 \ell_2) = \sum_{t=0}^{k} p_{[i_1, \ldots, i_t]}(\ell_1) \cdot p_{[i_{t+1}, \ldots, i_k]}(\ell_2)$.*

This can be proved by induction on the length of ℓ_1 using Lemma 5.3 or else by a decomposition of the defining sum according to which indices hit the first list and which ones hit the second.

5.2. Annotated Types and Potential Functions

We use the indexes and base polynomials to define type annotations and resource polynomials. We then give examples to illustrate the definitions.

A *type annotation* for a data type A is defined to be a family:

$$Q_A = (q_i)_{i \in I(A)} \text{ with } q_i \in \mathbb{Q}_0^+.$$

We say Q_A is of *degree (at most) k* if $q_i = 0$ for every $i \in I(A)$ with $deg(i) > k$. An *annotated data type* is a pair (A, Q_A) of a data type A and a type annotation Q_A of some degree k.

Let $\mathcal{H}$ be a heap and let v be a value with $\mathcal{H} \vDash v \mapsto a : A$ for a data type A. Then the type annotation Q_A defines the *potential*:

$$\Phi_{\mathcal{H}}(v{:}(A, Q_A)) = \sum_{i \in I(A)} q_i \cdot p_i(a).$$

Usually, we define type annotations Q_A by only stating the values of the nonzero coefficients q_i. However, it is sometimes handy to write annotations $(q_0, \ldots, q_n)$ for a list of atomic types just as a vector. Similarly, we write annotations $(q_0, q_{(1,0)}, q_{(0,1)}, q_{(1,1)}, \ldots)$ for pairs of lists of atomic types sometimes as a triangular matrix.

If $a \in [\![A]\!]$ and Q_A is a type annotation for A then we also write $\Phi(a : (A, Q_A))$ for $\sum_i q_i \cdot p_i(a)$.

5.3. Examples

The simplest annotated types are those for atomic data types like integers. The indexes for int are $I(\text{int}) = \{*\}$ and thus each type annotation has the form (int, q_0) for a $q_0 \in \mathbb{Q}_0^+$. It defines the constant potential function $\Phi_{\mathcal{H}}(v{:}(\text{int}, q_0)) = q_0$. Similarly, tuples of atomic types feature a single index of the form $(*, \ldots, *)$ and a constant potential function defined by some $q_{(*,\ldots,*)} \in \mathbb{Q}_0^+$.

More interesting examples are lists of atomic types like, for example, $L(\text{int})$. The set of indexes of degree k is then $I_k(L(\text{int})) = \{[], [*], [*, *], \ldots, [*, \ldots, *]\}$ where the last list contains k unit elements. Since we identify a list of i unit elements with the integer i we have $I_k(L(\text{int})) = \{0, 1, \ldots, k\}$. Consequently, annotated types have the form $(L(\text{int}), (q_0, \ldots, q_k))$ for $q_i \in \mathbb{Q}_0^+$. The defined potential function is $\Phi([a_1, \ldots, a_n]{:}(L(\text{int}), (q_0, \ldots, q_n)) = \sum_{0 \leq i \leq k} q_i \binom{n}{i}$.

The next example is the type $(L(\text{int}), L(\text{int}))$ of pairs of integer lists. The set of indexes of degree k is $I_k(L(\text{int}), L(\text{int})) = \{(i, j) \mid i + j \leq k\}$ if we identify lists of units with their lengths as usual. Annotated types are then of the from $((L(\text{int}), L(\text{int})), Q)$ for a triangular $k \times k$ matrix Q with nonnegative rational entries. If $\ell_1 = [a_1, \ldots, a_n]$, $\ell_2 = [b_1, \ldots, b_m]$ are two lists then the potential function is $\Phi((\ell_1, \ell_2), ((L(\text{int}), L(\text{int})), (q_{(i,j)}))) = \sum_{0 \leq i + j \leq k} q_{(i,j)} \binom{n}{i} \binom{m}{j}$.

Finally, consider the type $A = L(L(\text{int}))$ of lists of lists of integers. The set of indexes of degree k is then $I_k(L(L(\text{int}))) = \{[i_1, \ldots, i_m] \mid m \leq k, i_j \in \mathbb{N}, \sum_{j=1,\ldots,m} i_j \leq k - m\} = \{0, \ldots, k\} \cup \{[1], \ldots, [k-1]\} \cup \{[0, 1], \ldots\} \cup \cdots$. Let $\ell = [[a_{11}, \ldots, a_{1m_1}], \ldots, [a_{n1}, \ldots, a_{nm_n}]]$ be a list of lists and $Q = (q_i)_{i \in I_k(L(L(\text{int})))}$ be a corresponding type annotation. The defined potential function is then $\Phi(\ell, (L(L(\text{int})), Q)) =$

$$\sum_{[i_1,\ldots,i_l] \in I_k(A)} \sum_{1 \leq j_1 < \cdots < j_l \leq n} q_{[i_1,\ldots,i_l]} \binom{m_{j_1}}{i_1} \cdots \binom{m_{j_l}}{i_l}.$$

In practice the potential functions are usually not very complex since most of the q_i are zero. Note that the resource polynomials for binary trees are identical to those for lists.

5.4. The Potential of a Context

For use in the type system we need to extend the definition of resource polynomials to typing contexts. We treat a context like a tuple type.

Let $\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n$ be a typing context and let $k \in \mathbb{N}$. The index set $I_k(\Gamma)$ is defined through

$$I_k(\Gamma) = \left\{ (i_1, \ldots, i_n) \,\middle|\, i_j \in I_{m_j}(A_j), \sum_{j=1,\ldots,n} m_j \leq k \right\}.$$

A *type annotation Q* of degree k for Γ is a family

$$Q = (q_i)_{i \in I_k(\Gamma)} \text{ with } q_i \in \mathbb{Q}_0^+.$$

We denote a *resource-annotated context* with $\Gamma; Q$. Let $\mathcal{H}$ be a heap and $\mathcal{V}$ be a stack with $\mathcal{H} \vDash \mathcal{V} : \Gamma$ where $\mathcal{H} \vDash \mathcal{V}(x_j) \mapsto a_{x_j} : \Gamma(x_j)$. The potential of $\Gamma; Q$ with respect to $\mathcal{H}$ and $\mathcal{V}$ is

$$\Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q) = \sum_{(i_1,\ldots,i_n) \in I_k(\Gamma)} q_{\vec{i}} \prod_{j=1}^{n} p_{i_j}(a_{x_j}).$$

Here, $\vec{i} = (i_1, \ldots, i_n)$. In particular, if $\Gamma = \emptyset$ then $I_k(\Gamma) = \{()\}$ and $\Phi_{\mathcal{V},\mathcal{H}}(\Gamma; q_{()}) = q_{()}$. We sometimes also write q_0 for $q_{()}$.

6. TYPING RULES

If $f : \llbracket A \rrbracket \to \llbracket B \rrbracket$ is a function computed by some program and $K(a)$ is the cost of the evaluation of $f(a)$ then our type system will essentially try to identify resource polynomials $p \in \mathcal{R}(A)$ and $\bar{p} \in \mathcal{R}(B)$ such that $p(a) \geq \bar{p}(f(a)) + K(a)$. The key aspect of such amortized cost accounting is that it interacts well with composition.

PROPOSITION 6.1. *Let $p \in \mathcal{R}(A)$, $\check{p} \in \mathcal{R}(B)$, $\bar{p} \in \mathcal{R}(C)$, $f : \llbracket A \rrbracket \to \llbracket B \rrbracket$, $g : \llbracket B \rrbracket \to \llbracket C \rrbracket$, $K_1 : \llbracket A \rrbracket \to \mathbb{Q}$, and $K_2 : \llbracket B \rrbracket \to \mathbb{Q}$. If $p(a) \geq \check{p}(f(a)) + K_1(a)$ and $\check{p}(b) \geq \bar{p}(g(b)) + K_2(b)$ for all a, b, c then $p(a) \geq \bar{p}(g(f(a))) + K_1(a) + K_2(f(a))$ for all a.*

Notice that if we merely had $p(a) \geq K_1(a)$ and $\check{p}(b) \geq K_2(b)$ then no bound could be directly obtained for the composition.

Interaction with parallel composition, that is, $(a, c) \mapsto (f(a), c)$, is more complex due to the presence of mixed multiplicative terms in the resource polynomials. Assume, for example, that $(f(a), c)$ is a pair of lists that is used as the argument of a function g in a larger program. If the resource consumption of $g(f(a), c)$ is, say, $|f(a)| \cdot |c|$ then we need to assign the corresponding potential to the pair $(f(a), c)$. But to express a global bound on the resource consumption of the program, this potential has to be derived from the potential of the pair (a, c) taking into account the different sizes of a and $f(a)$. The following proposition describes at a high level how this is handled in our type system.

PROPOSITION 6.2. *Let $p \in \mathcal{R}(A, C)$, $\bar{p} \in \mathcal{R}(B, C)$, $f : \llbracket A \rrbracket \to \llbracket B \rrbracket$, and $K : \llbracket A \rrbracket \to \mathbb{Q}$. For each $j \in I(C)$ let $p^{(j)} \in \mathcal{R}(A)$ and $\bar{p}^{(j)} \in \mathcal{R}(B)$ be such that $p(a, c) = \sum_j p^{(j)}(a) p_j(c)$ and $\bar{p}(b, c) = \sum_j \bar{p}^{(j)}(b) p_j(c)$.*

If $p^{(0)}(a) \geq \bar{p}^{(0)}(f(a)) + K(a)$ and $p^{(j)}(a) \geq \bar{p}^{(j)}(f(a))$ holds for all a and $j \neq 0$ then $p(a, c) \geq \bar{p}(f(a), c) + K(a)$.

In fact, the situation is more complicated due to our accounting for high watermarks as opposed to merely additive cost, and also due to the fact that functions are recursively defined and may be partial. Furthermore, we have to deal with contexts and not merely types. To gain an intuition for the development to come, the preceding simplified view should, however, prove helpful.

6.1. Type Judgments

The declarative typing rules for RAML expressions (see Figures 4 and 5) define a *resource-annotated typing judgment* of the form $\Sigma; \Gamma; Q \vdash e : (A, Q')$ where e is a RAML expression, Σ is a resource-annotated signature (see the following), $\Gamma; Q$ is a resource-annotated context, and (A, Q') is a resource-annotated data type. The intended meaning of this judgment is that if there are more than $\Phi(\Gamma; Q)$ resource units available then this is sufficient to evaluate e. In addition, there are at least $\Phi(v:(A, Q'))$ resource units left if e evaluates to a value v.

6.2. Programs with Annotated Types

Resource-annotated first-order types have the form $(A, Q) \to (B, Q')$ for annotated data types (A, Q) and (B, Q'). A *resource-annotated signature* Σ is a finite, partial mapping of function identifiers to *sets of* resource-annotated first-order types.

A RAML program with resource-annotated types consists of a resource-annotated signature Σ and a family of expressions with variables identifiers $(e_f, y_f)_{f \in \text{dom}(\Sigma)}$ such that $\Sigma; y_f{:}A; Q \vdash e_f : (B, Q')$ for every function type $(A, Q) \to (B, Q') \in \Sigma(f)$.

6.3. Notations

Families that describe type and context annotations are denoted with upper case letters $Q, P, R, \ldots$ with optional superscripts. We use the convention that the elements of the families are the corresponding lower case letters with corresponding superscripts, that is, $Q = (q_i)_{i \in I}$, $Q' = (q'_i)_{i \in I}$, and $Q^x = (q^x_i)_{i \in I}$.

Let Q, Q' be two annotations with the same index set I. We write $Q \le Q'$ if $q_i \le q'_i$ for every $i \in I$. For $K \in \mathbb{Q}$ we write $Q = Q' + K$ to state that $q_{\vec{0}} = q'_{\vec{0}} + K \ge 0$ and $q_i = q'_i$ for $i \ne \vec{0} \in I$. Let $\Gamma = \Gamma_1, \Gamma_2$ be a context, let $i = (i_1, \ldots, i_k) \in I(\Gamma_1)$, and $j = (j_1, \ldots, j_l) \in I(\Gamma_2)$. We write (i, j) to denote the index $(i_1, \ldots, i_k, j_1, \ldots, j_l) \in I(\Gamma)$.

We write $\Sigma; \Gamma; Q \vdash^{\text{cf}} e : (A, Q')$ to refer to cost-free type judgments where all constants K in the rules from Figures 4 and 5 are zero. We use it to assign potential to an extended context in the let rule. More explanations will follow later.

Let Q be an annotation for a context Γ_1, Γ_2. For $j \in I(\Gamma_2)$ we define the *projection* $\pi_j^{\Gamma_1}(Q)$ of Q to Γ_1 to be the annotation $Q' \in I(\Gamma_1)$ with $q'_i = q_{(i,j)}$. The essential properties of the projections are stated by Propositions 6.2 and 6.3; they show how the analysis of juxtaposed functions can be broken down into individual components and combined again after the size of one component has changed.

PROPOSITION 6.3. *Let $\Gamma, x{:}A; Q$ be an annotated context, $\mathcal{H} \vDash \mathcal{V}{:}\Gamma, x{:}A$, and $\mathcal{H} \vDash \mathcal{V}(x) \mapsto a : A$. Then it is true that $\Phi_{\mathcal{V},\mathcal{H}}(\Gamma, x{:}A; Q) = \sum_{j \in I(A)} \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; \pi_j^{\Gamma}(Q)) \cdot p_j(a)$.*

6.4. Additive Shift

A key notion in the type system is the *additive shift* that is used to assign potential to typing contexts that result from a pattern match or from the application of a constructor of an inductive data type. We first define the additive shift, then illustrate the definition with examples, and finally state the soundness of the operation.

Let $\Gamma, y{:}L(A)$ be a context and let $Q = (q_i)_{i \in I(\Gamma, y:L(A))}$ be a context annotation of degree k. The *additive shift for lists* $\lhd_L(Q)$ of Q is an annotation $\lhd_L(Q) = (q'_i)_{i \in I(\Gamma, x:A, xs:L(A))}$ of degree k for a context $\Gamma, x{:}A, xs{:}L(A)$ that is defined through

$$q'_{(i,j,\ell)} = \begin{cases} q_{(i,j::\ell)} + q_{(i,\ell)} & j = 0 \\ q_{(i,j::\ell)} & j \ne 0 \end{cases}.$$

Let $\Gamma, t{:}T(A)$ be a context and let $Q = (q_i)_{i \in I(\Gamma, t:T(A))}$ be a context annotation of degree k. The *additive shift for binary trees* $\lhd_T(Q)$ of Q is an annotation $\lhd_T(Q) = (q'_i)_{i \in I(\Gamma')}$ of

degree k for a context $\Gamma' = \Gamma, x{:}A, xs_1{:}T(A), xs_2{:}T(A)$ that is defined by

$$q'_{(i,j,\ell_1,\ell_2)} = \begin{cases} q_{(i,j::\ell_1\ell_2)} + q_{(i,\ell_1\ell_2)} & j = 0 \\ q_{(i,j::\ell_1\ell_2)} & j \neq 0 \end{cases}.$$

The definition of the additive shift is short but substantial. The shift operations are used in a situation where $y = x :: xs$ and $t = tree(x, xs_1, xs_2)$, respectively. We begin by illustrating its effect in some example cases.

To start with, consider a context $\ell{:}L(\mathrm{int})$ with a single integer list that features an annotation $(q_0, \ldots, q_k) = (q_{[]}, \ldots, q_{[0,\ldots,0]})$. The shift operation $\lhd_L$ for lists produces an annotation for a context of the form $x{:}\mathrm{int}, xs{:}L(\mathrm{int})$, namely $\lhd_L(q_0, \ldots, q_k) = (q_{(0,0)}, \ldots, q_{(0,k)})$ such that $q_{(0,i)} = q_i + q_{i+1}$ for all $i < k$ and $q_{(0,k)} = q_k$. This is exactly the additive shift that we introduced in our previous work for the univariate system [Hoffmann and Hofmann 2010b]. We use it in a context where ℓ points to a list of length $n + 1$ and xs is the tail of ℓ. It reflects the fact that $\sum_{i=0,\ldots,k} q_i \binom{n+1}{i} = \sum_{i=0,\ldots,k-1} q_{i+1} \binom{n}{i} + \sum_{i=0,\ldots,k} q_i \binom{n}{i}$.

Now consider the annotated context $t{:}T(\mathrm{int}); (q_0, \ldots, q_k)$ with a single variable t that points to a tree with $n + 1$ nodes. The additive shift $\lhd_T$ produces an annotation for a context of the form $x{:}\mathrm{int}, t_1{:}T(\mathrm{int}), t_2{:}T(\mathrm{int})$. We have $\lhd_T(q_0, \ldots, q_k) = (q_{(0,i,j)})_{i+j \leq k}$ where $q_{(0,i,j)} = q_{i+j} + q_{i+j+1}$ if $i + j < k$ and $q_{(0,i,j)} = q_{i+j}$ if $i + j = k$. The intention is that t_1 and t_2 are the subtrees of t which have n_1 and n_2 nodes, respectively ($n_1 + n_2 = n$). The definition of the additive shift for trees incorporates the convolution $\binom{n+m}{k} = \sum_{i+j=k} \binom{n}{i}\binom{m}{j}$ for binomials. It is true that $\sum_{i=0,\ldots,k} q_i \binom{n+1}{i} = \sum_{i=0,\ldots,k-1}(q_i + q_{i+1})\binom{n}{i} + q_k \binom{n}{k} = \sum_{i=0}^{k-1} \sum_{j_1+j_2=i}(q_i + q_{i+1})\binom{n_1}{j_1}\binom{n_2}{j_2} + \sum_{j_1+j_2=k} q_i \binom{n_1}{j_1}\binom{n_2}{j_2}$.

As a last example consider the context $l_1{:}L(\mathrm{int}), l_2{:}L(\mathrm{int}); Q$ where $Q = (q_{(i,j)})_{i+j \leq k}$, l_1 is a list of length m, and l_2 is a list of length $n + 1$. The additive shift results in an annotation for a context of the form $l_1{:}L(\mathrm{int}), x{:}\mathrm{int}, xs{:}L(\mathrm{int})$ and the intention is that xs is the tail of l_2, that is, a list of length n. From the definition it follows that $\lhd_L(Q) = (q_{(i,0,j)})_{i+j \leq k}$ where $q_{(i,0,j)} = q_{(i,j)} + q_{(i,j+1)}$ if $i + j < k$ and $q_{(i,0,j)} = q_{(i,j)}$ if $i + j = k$. The soundness follows from the fact that for every $i \leq k$ we have that $\sum_{j=1}^{k-i} q_{(i,j)} \binom{m}{i}\binom{n+1}{j} = \binom{m}{i}\left(\sum_{j=0}^{k-i-1}(q_{(i,j)} + q_{(i,j+1)})\binom{n}{i} + q_{(i,k-i)}\binom{n}{k}\right)$.

Lemmas 6.4 and 6.5 state the soundness of the shift operations for lists and trees, respectively.

LEMMA 6.4. *Let* $\Gamma, \ell{:}L(A); Q$ *be an annotated context,* $\mathcal{H} \vDash \mathcal{V} : \Gamma, \ell{:}L(A)$, $\mathcal{H}(\ell) = (v_1, \ell')$ *and let* $\mathcal{V}' = \mathcal{V}[x_h \mapsto v_1, x_t \mapsto \ell']$. *Then* $\mathcal{H} \vDash \mathcal{V}' : \Gamma, x_h{:}A, x_t{:}L(A)$ *and* $\Phi_{\mathcal{V},\mathcal{H}}(\Gamma, \ell{:}L(A); Q) = \Phi_{\mathcal{V}',\mathcal{H}}(\Gamma, x_h{:}A, x_t{:}L(A); \lhd_L(Q))$.

This is a consequence of Lemma 5.3. One takes the linear combination of instances of its second equation and regroups the right-hand side according to the base polynomials for the resulting context.

LEMMA 6.5. *Let* $\Gamma, t{:}T(A); Q$ *be an annotated context,* $\mathcal{H} \vDash \mathcal{V} : \Gamma, t{:}T(A)$, $\mathcal{H}(t) = (v_1, t_1, t_2)$, *and* $\mathcal{V}' = \mathcal{V}[x_0 \mapsto v_1, x_1 \mapsto t_1, x_2 \mapsto t_2]$. *If* $\Gamma' = \Gamma, x{:}A, x_1{:}T(A), x_2{:}T(A)$ *then* $\mathcal{H} \vDash \mathcal{V}' : \Gamma'$ *and* $\Phi_{\mathcal{V},\mathcal{H}}(\Gamma, t{:}T(A); Q) = \Phi_{\mathcal{V}',\mathcal{H}}(\Gamma'; \lhd_T(Q))$.

We remember that the potential of a tree only depends on the list of nodes in preorder. So, we can think of the context splitting as done in two steps. First the head is separated, as in Lemma 6.4, and then the list of remaining elements is split into two lists. Lemma 6.5 is then proved like the previous one by regrouping terms using Lemma 5.3 for the first separation and Lemma 5.4 for the second one.

6.5. Sharing

Let $\Gamma, x_1{:}A, x_2{:}A; Q$ be an annotated context. The *sharing operation* $\curlyvee(Q)$ defines an annotation for a context of the form $\Gamma, x{:}A$. It is used when the potential is split between multiple occurrences of a variable.

The following lemma show that sharing is a linear operation that does not lead to any loss of potential. This is a consequence of Corollary 5.2. However, we include a proof to show how the coefficient can be computed.

LEMMA 6.6. *Let A be a data type. Then there are nonnegative rational numbers $c_k^{(i,j)}$ for $i, j, k \in I(A)$ and $\deg(k) \leq \deg(i, j)$ such that the following holds. For every context $\Gamma, x_1{:}A, x_2{:}A; Q$ and every $\mathcal{H}, \mathcal{V}$ with $\mathcal{H} \vDash \mathcal{V} : \Gamma, x{:}A$ it holds that $\Phi_{\mathcal{V},\mathcal{H}}(\Gamma, x{:}A; Q') = \Phi_{\mathcal{V}',\mathcal{H}}(\Gamma, x_1{:}A, x_2{:}A; Q)$ where $\mathcal{V}' = \mathcal{V}[x_1, x_2 \mapsto \mathcal{V}(x)]$ and $q'_{(\ell,k)} = \sum_{i,j \in I(A)} c_k^{(i,j)} q_{(\ell,i,j)}$.*

PROOF. The task is to show that for every resource polynomial $p_{(i,j)}((v, v)) = p_i(v) \cdot p_i(v)$ can be written as a sum (possibly with repetitions) of $p_{i'}(v)$'s.

We argue by induction on A. If A is an atomic type bool, int, or unit, we can simply write $1 \cdot 1$ as 1. If A is a pair $A = (B, C)$ then we have $p_{(i,j)}((v, w)) \cdot p_{(i',j')}((v, w)) = p_i(v)p_j(w)p_{i'}(v)p_{j'}(w) = (p_i(v)p_{i'}(v))(p_j(w)p_{j'}(w))$. By induction hypothesis, $(p_i(v)p_{i'}(v))$ and $(p_j(w)p_{j'}(w))$ both are sums of elemtary resource polynomials for B or C, respectively. So the expression is a sum of terms of the form $p_{i''}(v)p_{j''}(w)$, which is $p_{(i'',j'')}((v, w))$. If A is a list $A = L(B)$ we have to consider

$$p_{[i_1,\ldots,i_k]}([v_1, \ldots, v_n])p_{[i'_1,\ldots,i'_{k'}]}([v_1, \ldots, v_n])$$
$$= \left(\sum_{1 \leq j_1 < \cdots < j_k \leq n} p_{i_1}(v_{j_1}) \ldots p_{i_k}(v_{j_k}) \right) \left(\sum_{1 \leq j'_1 < \cdots < j'_{k'} \leq n} p_{i'_1}(v_{j'_1}) \ldots p_{i'_{k'}}(v_{j'_{k'}}) \right).$$

Using the distributive law, this can be considered as the sum over all possible ways to arrange the $j_1, \ldots, j_k$ and $j'_1, \ldots, j'_{k'}$ relative to each other respecting their respective orders, including the case that some j_i coincide with some $j'_{i'}$. Each of the terms in this sum of fixed length (independent of the lists!) has the shape

$$\sum_{1 \leq j''_1 < \cdots < j''_\ell \leq n} q_1(v_{j''_1}) \ldots q_\ell(v_{j''_\ell})$$

where each $q_r(v_{j_r})$ is either a $p_{i_s}(v_{j_r})$, a $p_{i'_{s'}}(v_{j_r})$, or a product $p_{i_r}(v_{j_r})p_{i'_{s'}}(v_{j_r})$. The latter can, by induction hypothesis, be written as sum of $p_{i''}(v_{j_r})$'s. Again, this representation is independent of the actual value of v_{j_r}. Using distributivity again, we obtain a sum of expressions of the form

$$\sum_{1 \leq j''_1 < \cdots < j''_\ell \leq n} p_{i''_1}(v_{j''_1}) \ldots p_{i''_\ell}(v_{j''_\ell}) = p_{[i''_1,\ldots,i''_\ell]}.$$

The case of A being a tree $A = T(B)$ is reduced to the case of A being a list, as the potential for trees is defined to be that of a list—the preorder traversal of the tree. □

In fact, inspection of the argument of the underlying Lemma 5.1 shows that the coefficients $c_k^{(i,j)}$, are indeed *natural* numbers and can be computed effectively. For a context $\Gamma, x_1{:}A, x_2{:}A; Q$ we define $\curlyvee(Q)$ to be the Q' from Lemma 6.6.

For example, consider a function $f(\ell_1, \ell_2)$ with two list arguments that has a resource consumption of $2|\ell_1||\ell_2|$. This can be expressed by a resource annotation $Q = (q_i)_{i \in I(L(\mathrm{int}),L(\mathrm{int}))}$ with $q_{(1,1)} = 2$ and $q_i = 0$ for all $i \neq (1,1)$. Assume now that we define a function $g(x) = f(x, x)$. The sharing coefficients $c_k^{(1,1)}$ describe now how

we can express the potential $Q' = (q'_k)_{k \in I(L(I))}$ of the argument x. Namely, we have $q'_k = 2 \cdot c_k^{(1,1)}$. Furthermore, $c_1^{(1,1)} = 1$, $c_2^{(1,1)} = 2$ and $c_k^{(1,1)} = 0$ for $k \notin \{1, 2\}$. This reflects the identity $q \cdot n^2 = 2q \cdot \binom{n}{2} + q \cdot n$.

6.6. Typing Rules

Figures 4 and 5 show the annotated typing rules for RAML expressions. We assume a fixed global signature Σ that we omit from the rules. The last four rules are structural rules that apply to every expression. The other rules are syntax-driven and there is one rule for every construct of the syntax. In the implementation we incorporated the structural rules in the syntax-driven ones. The most interesting rules are explained next.

T:SHARE has to be applied to expressions that contain a variable twice (z in the rule). The sharing operation $\curlyvee(P)$ transfers the annotation P for the context $\Gamma, x{:}A, y{:}A$ into an annotation Q for the context $\Gamma, z{:}A$ without loss of potential (Lemma 6.6). This is crucial for the accuracy of the analysis since instances of T:SHARE are quite frequent in typical examples. The remaining rules are affine linear in the sense that they assume that every variable occurs at most once.

T:CONS assigns potential to a lengthened list. The additive shift $\lhd_L(Q')$ transforms the annotation Q' for a list type into an annotation for the context $x_h{:}A, x_t{:}L(A)$. Lemma 6.4 shows that potential is neither gained nor lost by this operation. The potential Q of the context has to pay for both the potential Q' of the resulting list and the resource cost K^{cons} for list cons.

T:MATL shows how to treat pattern matching of lists. The initial potential defined by the annotation Q of the context $\Gamma, x{:}L(A)$ has to be sufficient to pay the costs of the evaluation of e_1 or e_2 (depending on whether the matched list is empty or not) and the potential defined by the annotation Q' of the result type. To type the expression e_1 of the nil case we use the projection $\pi_0^\Gamma(Q)$ that results in an annotation for the context Γ. Since the matched list is empty in this case no potential is lost by the discount of the annotations $q_{(i,j)}$ of Q where $j \neq 0$. To type the expression e_2 of the cons case we rely on the shift operation $\lhd_L(Q)$ for lists that results in an annotation for the context $\Gamma, x_h{:}A, x_t{:}L(A)$. Again there is no loss of potential (see Lemma 6.4). The equalities relate the potential before and after the evaluation of e_1 or e_2, to the potential before and after the evaluation of the match operation by incorporating the respective resource cost for the matching.

T:NODE and T:MATT are similar to the corresponding rules for lists but use the shift operator $\lhd_T$ for trees (see Lemma 6.5).

T:LET comprises essentially an application of Proposition 6.2 (with $f = e_1$ and $C = \Gamma_2$) followed by an application of Proposition 6.1 (with f being the parallel composition of e_1 and the identity on Γ_2 and g being e_2). Of course, the rigorous soundness proof takes into account partiality and additional constant costs for dispatching a let. It is part of the inductive soundness proof for the entire type system (Theorem 6.7).

The derivation of the type judgment $\Gamma_1, \Gamma_2; Q \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : (B, Q')$ can be explained in two steps. The first one starts with the derivation of the judgment $\Gamma_1; P \vdash e_1 : (A, P')$ for the subexpression e_1. The annotation P corresponds to the potential that is attached exclusively to Γ_1 by the annotation Q plus some resource cost for the *let*, namely $P = \pi_0^{\Gamma_1}(Q) + K_1^{\mathrm{let}}$. Now we derive the judgment $\Gamma_2, x{:}A; R \vdash e_2 : (B, R')$. The potential that is assigned by R to $x{:}A$ is the potential that resulted from the judgment for e_1 plus some cost that might occur when binding the variable x to the value of e_1, namely $P' = \pi_0^{x:A}(R) + K_2^{\mathrm{let}}$. The potential that is assigned by R to Γ_2 is essentially the potential that is assigned to Γ_2 by Q, namely $\pi_0^{\Gamma_2}(Q) = \pi_0^{\Gamma_2}(R)$.

$$\frac{Q = Q' + K^{\mathrm{var}}}{x{:}B; Q \vdash x : (B, Q')} \ (\text{T:Var}) \qquad \frac{b \in \{\mathit{True}, \mathit{False}\} \quad q_0 = q_0' + K^{\mathrm{bool}}}{\emptyset; Q \vdash b : (\mathit{bool}, Q')} \ (\text{T:ConstB})$$

$$\frac{op \in \{\mathbf{or}, \mathbf{and}\} \quad q_{(0,0)} = q_0' + K^{op}}{x_1{:}\mathit{bool}, x_2{:}\mathit{bool}; Q \vdash x_1 \ op \ x_2 : (\mathit{bool}, Q')} \ (\text{T:OpBool}) \qquad \frac{n \in \mathbb{Z} \quad q_0 = q_0' + K^{\mathrm{int}}}{\emptyset; Q \vdash n : (\mathit{int}, Q')} \ (\text{T:ConstI})$$

$$\frac{op \in \{+, -, *, \mathbf{mod}, \mathbf{div}\} \quad q_{(0,0)} = q_0' + K^{op}}{x_1{:}\mathit{int}, x_2{:}\mathit{int}; Q \vdash x_1 \ op \ x_2 : (\mathit{int}, Q')} \ (\text{T:OpInt}) \qquad \frac{q_0 = q_0' + K^{\mathrm{unit}}}{\emptyset; Q \vdash () : (\mathit{unit}, Q')} \ (\text{T:ConstU})$$

$$\frac{P + K_1^{\mathrm{app}} = Q \quad P' = Q' + K_2^{\mathrm{app}} \quad (A, P) \to (B, P') \in \Sigma(f)}{x{:}A; Q \vdash f(x) : (B, Q')} \ (\text{T:App})$$

$$\frac{\begin{array}{cc} \Gamma_1; P \vdash e_1 : (A, P') & \Gamma_2, x{:}A; R \vdash e_2 : (B, R') \\ P + K_1^{\mathrm{let}} = \pi_{\vec{0}}^{\Gamma_1}(Q) \quad P' = \pi_{\vec{0}}^{x:A}(R) + K_2^{\mathrm{let}} \quad R' = Q' + K_3^{\mathrm{let}} \\ \forall \vec{0} \neq j \in I(\Gamma_2){:} \quad \Gamma_1; P_j \overset{\mathbf{cf}}{\vdash} e_1 : (A, P_j') \quad P_j = \pi_j^{\Gamma_1}(Q) \quad P_j' = \pi_j^{x:A}(R) \end{array}}{\Gamma_1, \Gamma_2; Q \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : (B, Q')} \ (\text{T:Let})$$

$$\frac{\begin{array}{ccc} \Gamma; P \vdash e_t : (B, P') & P + K_1^{\mathrm{conT}} = \pi_0^{\Gamma}(Q) & P' = Q' + K_2^{\mathrm{conT}} \\ \Gamma; R \vdash e_f : (B, R') & R + K_1^{\mathrm{conF}} = \pi_0^{\Gamma}(Q) & R' = Q' + K_2^{\mathrm{conF}} \end{array}}{\Gamma, x{:}\mathit{bool}; Q \vdash \mathbf{if} \ x \ \mathbf{then} \ e_t \ \mathbf{else} \ e_f : (B, Q')} \ (\text{T:Cond})$$

$$\frac{A = (A_1, A_2) \quad \Gamma, x_1{:}A_1, x_2{:}A_2; P \vdash e : (B, P') \quad P + K_1^{\mathrm{matP}} = Q \quad P' = Q' + K_2^{\mathrm{matP}}}{\Gamma, x{:}A; Q \vdash \mathbf{match} \ x \ \mathbf{with} \ (x_1, x_2) \to e : (B, Q')} \ (\text{T:MatP})$$

$$\frac{Q = Q' + K^{\mathrm{pair}}}{x_1{:}A_1, x_2{:}A_2; Q \vdash (x_1, x_2) : ((A_1, A_2), Q')} \ (\text{T:Pair}) \qquad \frac{q_0 = q_0' + K^{\mathrm{nil}}}{\emptyset; Q \vdash \mathbf{nil} : (L(A), Q')} \ (\text{T:Nil})$$

$$\frac{q_0 = q_0' + K^{\mathrm{leaf}}}{\emptyset; Q \vdash \mathbf{leaf} : (T(A), Q')} \ (\text{T:Leaf}) \qquad \frac{Q = \lhd_L(Q') + K^{\mathrm{cons}}}{x_h{:}A, x_t{:}L(A); Q \vdash \mathbf{cons}(x_h, x_t) : (L(A), Q')} \ (\text{T:Cons})$$

$$\frac{Q = \lhd_T(Q') + K^{\mathrm{node}}}{x_0{:}A, x_1{:}T(A), x_2{:}T(A); Q \vdash \mathbf{node}(x_0, x_1, x_2) : (T(A), Q')} \ (\text{T:Node})$$

$$\frac{\begin{array}{ccc} \Gamma; R \vdash e_1 : (B, R') & R + K_1^{\mathrm{matN}} = \pi_0^{\Gamma}(Q) & R' = Q' + K_2^{\mathrm{matN}} \\ \Gamma, x_h{:}A, x_t{:}L(A); P \vdash e_2 : (B, P') & P + K_1^{\mathrm{matC}} = \lhd_L(Q) & P' = Q' + K_2^{\mathrm{matC}} \end{array}}{\Gamma, x{:}L(A); Q \vdash \mathbf{match} \ x \ \mathbf{with} \mid \mathbf{nil} \to e_1 \mid \mathbf{cons}(x_h, x_t) \to e_2 : (B, Q')} \ (\text{T:MatL})$$

$$\frac{\begin{array}{cc} \Gamma; R \vdash e_1 : (B, R') & \Gamma, x_0{:}A, x_1{:}T(A), x_2{:}T(A); P \vdash e_2 : (B, P') \\ R + K_1^{\mathrm{matTL}} = \pi_0^{\Gamma}(Q) \quad R' = Q' + K_2^{\mathrm{matTL}} \quad P + K_1^{\mathrm{matTN}} = \lhd_T(Q) \quad P' = Q' + K_2^{\mathrm{matTN}} \end{array}}{\Gamma, x{:}T(A); Q \vdash \mathbf{match} \ x \ \mathbf{with} \mid \mathbf{leaf} \to e_1 \mid \mathbf{node}(x_0, x_1, x_2) \to e_2 : (B, Q')} \ (\text{T:MatT})$$

Fig. 4. Typing rules for annotated types (1 of 2).

$$\frac{\Gamma, x{:}A, y{:}A; P \vdash e : (B, Q') \qquad Q = \curlyvee(P)}{\Gamma, z{:}A; Q \vdash e[z/x, z/y] : (B, Q')} \text{ (T:SHARE)} \qquad \frac{\Gamma; \pi_0^{\Gamma}(Q) \vdash e : (B, Q')}{\Gamma, x{:}A; Q \vdash e : (B, Q')} \text{ (T:AUGMENT)}$$

$$\frac{\begin{array}{c} \Gamma; P \vdash e : (B, P') \\ Q \geq P \qquad Q' \leq P' \end{array}}{\Gamma; Q \vdash e : (B, Q')} \text{ (T:WEAKEN)} \qquad \frac{\begin{array}{c} \Gamma; P \vdash e : (B, P') \\ Q = P + c \qquad Q' = P' + c \end{array}}{\Gamma; Q \vdash e : (B, Q')} \text{ (T:OFFSET)}$$

Fig. 5.   Typing rules for annotated types (2 of 2).

The second step of the derivation is to relate the annotations in R that refer to mixed potential between $x{:}A$ and Γ_2 to the annotations in Q that refer to potential that is mixed between Γ_1 and Γ_2. To this end we remember that we can derive from a judgment $\Gamma_1; S \vdash e_1 : (A, S')$ that $\Phi(\Gamma_1; S) \geq \Phi(v{:}(A, S'))$ if e_1 evaluates to v. This inequality remains valid if multiplied with a potential for $\phi_{\Gamma_2} = \Phi(\Gamma_2; T)$, that is, $\Phi(\Gamma_1; S) \cdot \phi_{\Gamma_2} \geq \Phi(v{:}(A, S')) \cdot \phi_{\Gamma_2}$. To relate the mixed potential annotations we thus derive a cost-free judgment $\Gamma_1; P_j \vdash^{\underline{cf}} e_1 : (A, P'_j)$ for every $\vec{0} \neq j \in I(\Gamma_2)$. (We use cost-free judgments to avoid paying multiple times for the evaluation of e_1.) Then we equate P_j to the corresponding annotations in Q and equate P'_j to the corresponding annotations in R, that is, $P_j = \pi_j^{\Gamma_1}(Q)$ and $P'_j = \pi_j^{x:A}(R)$. The intuition is that j corresponds to ϕ_{Γ_2}. Note that we use a fresh signature Σ in the derivation of each cost-free judgment for e_1.

6.7. Soundness

The main theorem of this article states that type derivations establish correct bounds: an annotated type judgment for an expression e shows that if e evaluates to a value v in a well-formed environment then the initial potential of the context is an upper bound on the watermark of the resource usage and the difference between initial and final potential is an upper bound on the consumed resources.

The introduction of the partial evaluation rules enables us to formulate a stronger soundness theorem than in earlier works on amortized analysis; for instance, Hoffmann and Hofmann [2010b] and Jost et al. [2009]. It states that the bounds derived from an annotated type judgment also hold for nonterminating evaluations. Additionally, the novel way of cost monitoring in the operational semantics enables a more concise statement.

THEOREM 6.7 (SOUNDNESS).   *Let* $\mathcal{H} \vDash \mathcal{V}{:}\Gamma$ *and* $\Sigma; \Gamma; Q \vdash e{:}(B, Q')$.

(1) *If* $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (p, p')$ *then* $p \leq \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q)$ *and* $p - p' \leq \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q) - \Phi_{\mathcal{H}'}(v{:}(B, Q'))$.
(2) *If* $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \mid p$ *then* $p \leq \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q)$.

Theorem 6.7 is proved by a nested induction on the derivation of the evaluation judgment—$\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (p, p')$ or $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \mid p$, respectively—and the type judgment $\Gamma; Q \vdash e{:}(B, Q')$. The inner induction on the type judgment is needed because of the structural rules. There is one proof for all possible instantiations of the resource constants. It is technically involved but conceptually unsurprising. Compared to earlier works [Hoffmann and Hofmann 2010b, 2010a], further complexity arises from the new rich potential annotations. It is mainly dealt with in Lemmas 6.4, 6.5, and 6.6, and the concept of projections as explained in Propositions 6.2 and 6.3.

The proof can be found in the appendix of this article.

It follows from Theorem 6.7 and Theorem 3.9 that runtime bounds also prove the termination of programs. Corollary. 6.8 states this fact formally.

COROLLARY 6.8. *Let the resource constants be instantiated by $K^x = 1$, $K_1^x = 1$ and $K_m^x = 0$ for all x and all $m > 1$. If $\mathcal{H} \vDash \mathcal{V}{:}\Gamma$ and $\Sigma; \Gamma; Q \vdash e{:}(A, Q')$ then there is an $n \in \mathbb{N}, n \leq \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q)$ such that $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (n, 0)$.*

7. TYPE INFERENCE

The type-inference algorithm for RAML extends the algorithm that we have developed for the univariate polynomial system [Hoffmann and Hofmann 2010a]. It is not complete with respect to the typing rules in Section 6 but it works well for the example programs we tested.

Its basis is a classic constraint-based type inference [Pierce 2004] generating simple linear constraints for the annotations that are collected during the inference. The constraints can be solved later by linear programming. In order to obtain a finite set of constraints one has to provide a maximal degree of the resource bounds. If the degree is too low then the generated linear program has no solution. The maximal degree can either be specified by the user or can be incremented successively after an unsuccessful analysis.

A main challenge in the inference is the handling of resource-polymorphic recursion which we believe to be of very high complexity if not undecidable in general. To deal with it practically, we employ a heuristic that has been developed for the univariate system. In a nutshell, a function is allowed to invoke itself recursively with a type different from the one that is being justified (polymorphic recursion) provided that the two types differ only in lower-degree terms. In this way, one can successively derive polymorphic type schemes for higher and higher degrees.

The idea is based on the observation that any concrete typing for a given resource metric can be superposed with a *cost-free* typing to obtain another typing for the given resource metric (like solutions of inhomogeneous systems by superposition with homogeneous solutions in linear algebra). With respect to this declarative view, the inference algorithm can compute every set of types for a function f that has the form $\Sigma(f) = \{T + q \cdot T_i \mid q \in \mathbb{Q}_0^+, 1 \leq i \leq m\}$ for a resource-annotated function type T, cost-free function types T_i, and m recursive calls of f in its function body.

The technique is exemplified for the function append in Section 7.4. For further details, see Hoffmann and Hofmann [2010a]. The generalization of this approach to the multivariate setting poses no extra difficulties.

The number of multivariate polynomials our type system takes into account grows exponentially in the maximal degree (e.g., $nm, n\binom{m}{2}, n\binom{m}{3}, m\binom{n}{2}, m\binom{n}{3}, \binom{n}{2}\binom{m}{2}$ for a pair of integer lists if the maximal degree is 4). Thus the number of inequalities we collect for a fixed program grows also exponentially in the given maximal degree.

Moreover, one often has to analyze function applications context-sensitively with respect to the call stack. Recall, for example, the type derivation of the expression filter(a,filter(b,l)) from Section 2 in which we use two different types for filter. The function type $((int, L(int)), (0, 2)) \rightarrow (L(int), (0, 0))$ is used for the outer call of filter. However, the type $((int, L(int)), (0, 4)) \rightarrow (L(int), (0, 2))$ is used for the inner call of filter to assign potential to its output that is sufficient to match the argument type of the outer call.

In our prototype implementation we collapse the cycles in the call graph and analyze each function once for every path in the resulting graph. As a result, a (mutually) recursive definition can have a different type at every call site after its definition. The recursive calls of a function in its definition are treated with our algorithm for resource-polymorphic recursion as described before. This type inference for function types is explained more formally by the algorithm in Section 7.2.

$$\frac{\pi_0^{x:B}(Q) = Q' + K^{\mathrm{var}}}{\Gamma, x{:}B; Q \vdash^{\underline{k}} x : (B, Q')} \text{ (A:VAR)} \qquad \frac{\Gamma, x_1{:}A, x_2{:}A; P \vdash^{\underline{k}} e : (B, Q') \qquad Q = \curlyvee(P)}{\Gamma, x{:}A; Q \vdash^{\underline{k}} \mathit{share}\,(x, x_1, x_2)\,\mathit{in}\,e' : (B, Q')} \text{ (A:SHARE)}$$

$$\frac{\begin{array}{ccc} \Gamma; P \vdash^{\underline{k}} e_t : (B, P') & P + K_1^{\mathrm{conT}} = \pi_0^{\Gamma}(Q) & P' \geq Q' + K_2^{\mathrm{conT}} \\ \Gamma; R \vdash^{\underline{k}} e_f : (B, R') & R + K_1^{\mathrm{conF}} = \pi_0^{\Gamma}(Q) & R' \geq Q' + K_2^{\mathrm{conF}} \end{array}}{\Gamma, x{:}\mathit{bool}; Q \vdash^{\underline{k}} \mathit{if}\,x\,\mathit{then}\,e_t\,\mathit{else}\,e_f : (B, Q')} \text{ (A:COND)}$$

$$\frac{P + c + K_1^{\mathrm{app}} = \pi_0^{x:A}(Q) \qquad P' = Q' + c + K_2^{\mathrm{app}} \qquad \Sigma(f) = (A, P) \to (A', P')}{\Gamma, x{:}A; Q \vdash^{\underline{1}} f(x) : (A', Q')} \text{ (A:APP1)}$$

$$\frac{\begin{array}{c} P + P_{\mathrm{cf}} + K_1^{\mathrm{app}} = \pi_0^{x:A}(Q) \qquad P' + P'_{\mathrm{cf}} = Q' + K_2^{\mathrm{app}} \qquad \Sigma(f) = (A, P) \to (A', P') \\ \Sigma_{\mathrm{cf}}(f) = (A, P_{\mathrm{cf}}) \to (A', P'_{\mathrm{cf}}) \qquad y_f{:}A; P_{\mathrm{cf}} \vdash^{\underline{\mathrm{cf}(k-1)}} e_f : (A', P'_{\mathrm{cf}}) \qquad k > 1 \end{array}}{\Gamma, x{:}A; Q \vdash^{\underline{k}} f(x) : (A', Q')} \text{ (A:APP)}$$

<div align="center">Fig. 6.   Selected algorithmic typing rules.</div>

7.1. Algorithmic Type Rules

To obtain a type inference that produces linear constraints, we have to develop algorithmic versions of the typing rules from Section 6. We describe this in detail for the univariate system in another article [Hoffmann and Hofmann 2010a]. It works similar for the multivariate system in this article. Basically, the structural typing rules have to be integrated in the syntax-directed rules.

Other than in the declarative versions, in the algorithmic typing, signatures map function names to a single function type. The judgment

$$\Gamma; Q \vdash^{\underline{k}} e : (A, Q')$$

denotes that $\Gamma; Q \vdash e : (A, Q')$ and that all type annotations in the corresponding derivation are of maximal degree k. The judgment $\Gamma; Q \vdash^{\underline{\mathrm{cf}(k)}} e : (A, Q')$ states that we have $\Gamma; Q \vdash^{\underline{k}} e : (A, Q')$ for the cost-free resource metric.

The rule T:AUGMENT can be eliminated by formulating the ground rules such as T:VAR or T:CONSTU for larger contexts. As an example we present the rule A:VAR, which is the algorithmic version of the rule T:VAR in Figure 6.

If the syntax-directed rules implicitly assume that two resource annotations are equal or defer by a fixed constant in two branches of a computation, an integration of the rule T:WEAKEN enables the analysis of a wider range of programs. For example, T:WEAKEN is integrated into the rule A:COND in Figure 6 by replacing the two preconditions $P' = Q' + K_2^{\mathrm{conT}}$ and $R' = Q' + K_2^{\mathrm{conF}}$ of the rule T:COND with $P' \geq Q' + K_2^{\mathrm{conT}}$ and $R' \geq Q' + K_2^{\mathrm{conF}}$. This enables type inference for programs whose resource consumptions differ in the two branches of the conditional. The same adaption is needed in the rules for pattern matching.

A difference to standard type systems is the sharing rule T:SHARE that has to be applied if the same free variable is used more than once in an expression. The rule is not problematic for the type inference and there are several ways to deal with it in practice. In our implementation, we transform input programs into programs that make sharing explicit with a syntactic construct before the type inference. Such a transformation is straightforward: Each time a free variable x occurs twice in an expression e, we replace the first occurrence of x with x_1 and the second occurrence of x with x_2 obtaining a new expression e'. We then replace e with $\mathit{share}(x, x_1, x_2)\,\mathit{in}\,e'$. In this way, the sharing rule becomes a normal syntax-directed rule in the type inference. Another possibility would

be to integrate sharing directly into the typing rule for let expressions as we did in an earlier work [Hoffmann and Hofmann 2010a]. Then one has to ensure a variable only occurs once in each function or constructor call. The algorithmic type rule A:SHARE for the sharing construct is given in Figure 6.

Key rules for the type inference are the algorithmic versions of the rule T:APP in Figure 6. The rule A:APP1 is essentially the rule T:APP from Section 6. It is used if the maximal degree is one and leads to a resource-monomorphic typing of recursive calls. The rule A:APP1 is the only rule in which we integrate the structural rule T:OFFSET. In some sense, this is analogous to our treatment of function calls with maximal degree greater than one.

The rule A:APP is used if the maximal degree is greater than one. It enables resource-polymorphic recursion. More precisely, it states that one can add a cost-free typing of the function body to the function type that is given by the signature Σ. Note that $(e_f, y_f)_{f \in \text{dom}(\Sigma_{cf})}$ must be a valid RAML program with cost-free types of degree at most $k-1$. The signature Σ_{cf} is annotated with fresh resource variables on each application of the rule.

The idea behind the rule A:APP is as follows. In order to pay for the resource cost of a function call $f(x)$, the available potential $(\Phi(x{:}A; \pi_0^{x:A}(Q)))$ must meet the requirements of the functions' signature $(\Phi(x{:}A; P))$. Additionally available potential $(\Phi(x{:}A; P_{cf}))$ can be passed to a cost-free typing of the function body. The potential after the function call $(\Phi(f(x){:}(A', Q')))$ is then the sum of the potentials that are assigned by the cost-free typing $(\Phi(f(x){:}(A', P'_{cf})))$ and by the function signature $(\Phi(f(x){:}(A', P')))$. As a result, $f(x)$ can be used resource-polymorphically with a specific typing for each recursive call while the resource-monomorphic function signature enables an efficient type inference.

7.2. Iterative Inference of Function Types

The collection of constraints can be informally described as follows. For every strongly connected component F of functions in the call graph—that is, a set of mutually recursive definitions F—repeat the following.

(1) Annotate the signature of each function $f \in F$ with fresh resource variables.
(2) Use the algorithmic typing rules to type the corresponding expressions e_f. Introduce fresh resource variables for each type annotation in the derivation and collect the corresponding inequalities.
   (a) For a function application $g \in F$: if the maximal degree is 1 use the function resource-monomorphically with the signature from (1) using the rule A:APP1. If the maximal degree is greater than 1, go to (1) and derive a cost-free typing of e_g with a fresh signature. Store the arising inequalities and use the resource variables from the obtained typing together with the signature from (1) in the rule A:APP.
   (b) For a function application $g \notin F$: repeat the algorithm for the SCC of g. Store the arising inequalities and use the obtained annotated type of g.

In contrast to the univariate system [Hoffmann and Hofmann 2010a], cost-free type derivations also depend on resource-polymorphic recursion to assign superlinear potential to function results. A simple example is the function append whose type inference is explained in detail in Section 7.4.

7.3. LP Solving

The last step of the inference is to solve the generated constraints with an LP solver. The linear objective function that we use states that the annotations of the arguments of function types should be minimized. We use a heuristic that approximates that

the minimization coefficients of higher degree take priority over the minimization of coefficients of lower degree.

However, one can always construct programs with, say, maximal degree two, for which the LP solver would return a solution in which the quadratic coefficient is not the minimal one. The reason is that the objective function must be a linear expression in the current LP solvers. So we cannot state that it is preferable to minimize the quadratic coefficient q_2 at the cost of a larger linear coefficient q_1. This is nevertheless unproblematic in practice. Our experiments showed that the inferred types are quite robust to changes to the objective function. The reason is that it is, for example, not possible to use quadratic potential instead of a linear potential for many programs.

7.4. Type Inference for an Example Program

The inference algorithm is best illustrated by way of example. To this end, consider the functions append, pairs, and appPairs which are defined in the following program.

```
append : (L(int),L(int)) -> L(int)
append(l,ys) = match l with | nil -> ys
                      | (x::xs) -> let l' = append(xs,ys) in x::l';

attach : (int,L(int)) -> L(int,int)
attach(n,l) = match l with | nil -> nil
                   | (x::xs) -> (n,x)::attach(n,xs);

append2 : (L(int,int),L(int,int)) -> L(int,int)
append2(l,ys) = match l with | nil -> ys
                      | (x::xs) -> let l' = append2(xs,ys) in x::l';

pairs : L(int) -> L(int,int)
pairs(l) = match l with | nil  -> nil
                | (x::xs) -> append2(attach(x,xs),pairs xs);

appPairs : (L(int),L(int)) -> L(int,int)
appPairs (x,y) = let z = append(x,y) in
                    pairs(z);
```

The definition of the function append2 is identical to the definition of append. But since we have a monomorphic language we need two versions with different types. The function pairs computes the two-element subsets of a given set (representing sets as tuples or lists). The expression pairs([1,2,3]) evaluates for example to [(1,2),(1,3),(2,3)]. Finally, the function appPairs concatenates its two list arguments using append and then calls pairs on the resulting list.

We are now interested in the heap-space consumption of the function appPairs. For this purpose, assume that each list node occupies exactly one heap cell and that only lists are stored on the heap. The (worst-case) heap-space consumptions of the functions can then be described as follows by using the argument names in the respective definitions.

—append consumes $|\ell|$ heap cells
—attach consumes $|\ell|$ heap cells
—append2 consumes $|\ell|$ heap cells
—pairs consumes $2\binom{|\ell|}{2}$ heap cells
—appPairs consumes $|x| + 2\binom{|x|}{2} + 2\binom{|y|}{2} + 2|x||y|$ heap cells

To derive these bounds with our type-inference algorithm, we set $K^{\text{cons}} = 1$ and $K = 0$ for all other constants K. If we set the maximal degree to 2 then we infer the following

$$\dfrac{\begin{array}{c}((L(int), L(int), A) \to (L(int), A') \\ \in \Sigma(\text{append})\end{array}}{\begin{array}{c}x : L(int), y : L(int); P \vdash^{\underline{2}} \\ \text{append}(x,y) : (L(int), P')\end{array}} \text{(APPEXT)} \qquad \dfrac{(L(int), B) \to (L(int, int), B') \in \Sigma(\text{pairs})}{z : L(int); R \vdash^{\underline{2}} \text{pairs}(z) : (L(int, int), R')} \text{(APPEXT)}}{x : L(int), y : L(int); Q \vdash^{\underline{2}} \text{let } z = \text{append}(x,y) \text{ in pairs}(z) : (L(int, int), Q')} \text{(T:LET)}$$

$$
\begin{array}{cccc}
P \geq A + K_1^{\text{app}} & R \geq B + K_1^{\text{app}} & Q \geq P + K_1^{\text{let}} & R' \geq Q' + K_3^{\text{let}} \\
A' \geq P' + K_2^{\text{app}} & B' \geq R' + K_2^{\text{app}} & P' \geq R + K_2^{\text{let}} &
\end{array}
$$

Fig. 7. Type inference for the function appPairs.

types.

$$\text{append} : ((L(\text{int}), L(\text{int})), \begin{pmatrix} 0\ 0\ 2 \\ 1\ 2 \\ 2 \end{pmatrix}) \to (L(\text{int}), (0, 2, 0))$$

$$\text{attach} : ((\text{int}, L(\text{int})), (0, 2, 0)) \to (L(\text{int}, \text{int}), (0, 1, 0))$$

$$\text{append2} : ((L(\text{int}), L(\text{int})), \begin{pmatrix} 0\ 0\ 0 \\ 1\ 0 \\ 0 \end{pmatrix}) \to (L(\text{int}), (0, 0, 0))$$

$$\text{pairs} : (L(\text{int}), (0, 0, 2)) \to (L(\text{int}, \text{int}), (0, 0, 0))$$

$$\text{appPairs} : ((L(\text{int}), L(\text{int})), \begin{pmatrix} 0\ 0\ 2 \\ 1\ 2 \\ 2 \end{pmatrix}) \to (L(\text{int}, \text{int}), (0, 0, 0))$$

Note that the given function types are those that are needed to derive the typing for appPairs. For example, the type of append does not only express that the function consumes $|\ell|$ heap cells but also that the resulting list of length $|\ell| + |ys|$ has to carry the potential $2\binom{|\ell|+|ys|}{2} = 2|\ell||ys| + 2\binom{|\ell|}{2} + 2\binom{|ys|}{2}$ to pay for the resource consumption of pairs in appPairs.

In the remainder of this section, we illustrate how the inference algorithm derives the typing for appPairs. First, we set the maximal degree to 2. We then annotate all functions in the SCC of appPairs with fresh resource variables of maximal degree 2. Since appPairs is the only function in the SCC we obtain the following function type.

$$\text{appPairs} : (L(\text{int}), L(\text{int}), Q) \to (L(\text{int}, \text{int}), Q')$$

We have $Q = (q_{(0,0)}, q_{(1,0)}, q_{(2,0)}, q_{(1,1)}, q_{(0,1)}, q_{(0,2)})$ and $Q' = (q'_0, q'_1, q'_2)$ where all q_i and q'_j are fresh variables. In Figure 7, we then use the type rules to obtain a type derivation for the function body e_{appPairs}. We use the same resource variables as in the function type for the argument (namely Q) and the result (namely Q'). All other types are annotated with fresh resource variables and constraints are collected as indicated by the type rules. Note that the constraints in Figure 7 are given using the generic constants like K_1^{app}. However, in our current heap-space setting we have in fact $K_1^{\text{app}} = K_2^{\text{app}} = K_1^{\text{let}} = K_2^{\text{let}} = K_3^{\text{let}} = 0$. As before, we use abbreviations for the constraints. For example, $R \geq B + K_1^{\text{app}}$ stands for the constraints $r_0 \geq b_0 + K_1^{\text{app}}$, $r_1 \geq b_1$, and $r_2 \geq b_2$. Since there are no recursive calls to appPairs, we do not derive a cost-free typing for the function.

To establish relationships between the constraints A and A' in the function annotation for append and between the constraints B and B' in the annotation for pairs, we initiate another run of the algorithm for the respective SCCs of the functions. We express this by the use of a new type rule APPEXT.

In an analysis of the SCC of the function pairs, the inference algorithm would generate constraints that are equivalent to the following constraints.

$$\cfrac{\Sigma'(\text{append}) = ((L(int), L(int)), A) \to (L(int), A')}{\cfrac{\cfrac{\begin{array}{c}\Sigma_{cf}(\text{append}) = ((L(int), L(int)), C) \to (L(int), C')\\ xs : L(int), ys : L(int); T \vdash^2\end{array}}{\text{append}(xs,ys) : (L(int), T')}\;(\text{A:App})}{}}$$

$$\cfrac{\cfrac{x : int, \ell' : L(int); S \vdash^2}{x{::}\ell' : (L(int), S')}\;(\text{T:Cons}) \qquad \cdots}{\cfrac{x : int, xs : L(int), ys : L(int) ; U \vdash^2}{\text{let } \ell' = \text{append}(xs,ys) \text{ in } x{::}\ell' : (L(int), U')}\;(\text{T:Let})}$$

$$\cfrac{\cfrac{ys : L(int); V \vdash^2}{ys : (L(int), V')}\;(\text{T:Var})}{\begin{array}{l}\ell : L(int), ys : L(int); A \vdash^2 \quad \text{match } \ell \text{ with } | \text{ nil } \to ys\\ \qquad\qquad\qquad\qquad\qquad\qquad | \ (x{::}xs) \to \text{let } \ell' = \text{append}(xs,ys) \text{ in } x{::}\ell' : (L(int), A')\end{array}}\;(\text{T:MatL})$$

$U' \geq A' + K_2^{\text{matC}}$	$a_{(0,2)} \geq u_{(*,0,2)}$	$u_{(*,0,0)} \geq t_{(0,0)} + K_1^{\text{let}}$	$u_{(*,1,1)} \geq t_{(1,1)}$
$a_{(0,0)} + a_{(1,0)} \geq u_{(*,0,0)} + K_1^{\text{matC}}$	$V' \geq A' + K_2^{\text{matN}}$	$u_{(*,1,0)} \geq t_{(1,0)}$	$t_0' \geq s_{(*,0)} + K_2^{\text{let}}$
$a_{(1,0)} + a_{(2,0)} \geq u_{(*,1,0)}$	$v_0 \geq a_{(0,2)} + K_1^{\text{matN}}$	$u_{(*,2,0)} \geq t_{(2,0)}$	$t_1' \geq s_{(*,1)}$
$a_{(1,1)} \geq u_{(*,1,1)}$	$v_1 \geq a_{(0,2)}$	$u_{(*,0,1)} \geq t_{(0,1)}$	$t_2' \geq s_{(*,2)}$
$a_{(2,0)} \geq u_{(*,2,0)}$	$v_2 \geq a_{(0,2)}$	$u_{(*,0,2)} \geq t_{(0,2)}$	$S' \geq U' + K_3^{\text{let}}$
$T \geq A + C + K_1^{\text{app}}$	$A' + C' \geq T' + K_2^{\text{app}}$	$S \geq \triangleleft_L(S') + K^{\text{cons}}$	$V \geq V' + K^{\text{var}}$

Fig. 8. Type inference for the function append.

$$b_0 \geq b_0' \qquad\qquad b_1 \geq 0 \qquad\qquad b_2 \geq 2 + b_1' \qquad\qquad b_2' = 0$$

If pairs would be called twice in the body of appPairs then we would generate two copies of this constraint set with different resource variables, one for each application of function. In this way, we allow the LP solver to infer a different typing for every (nonrecursive) call site.

We do not study the generation of the constraints for pairs in detail but focus rather on the more interesting generation of the constraints for append whose type derivation uses polymorphic recursion.[6] The inference algorithm will in fact infer the following infinite set $\Sigma(\text{append})$ of types for append.

$$\Sigma(\text{append}) = \left\{ \left((L(\text{int}), L(\text{int})), \begin{pmatrix} 0 & & 2n & 2 \\ 1{+}2n & 2 & & \\ 2 & & & \end{pmatrix} \right) \to (L(\text{int}), (0, 2n, 2)) \mid n \in \mathbb{N} \right\}$$

To see why polymorphic recursion is required, consider a type derivation of the typing in $\Sigma(\text{append})$ for $n = 0$. This function type states that we have to assign the type $(L(\text{int}), (0, 0, 2))$ to the result of the function. The type rules then require the same type for the cons expression $x{::}\ell'$ in the cons part of the pattern match. That means that if $|\ell'| = m$ then $x{::}\ell'$ has potential $2\binom{m+1}{2}$. An inspection of the rule T:Cons shows that, according to the additive shift, the type of ℓ' has to be $(L(\text{int}), (0, 2, 2))$. This represents the potential $2m + 2\binom{m}{2} = 2\binom{m+1}{2}$. Since $\ell' = \text{append}(xs,ys)$ the expression append(xs,ys) has to be typed with type $(L(\text{int}), (0, 2, 2))$ too. However, the monomorphic function type ($n = 0$) of append states that append(xs,ys) is of type $(L(\text{int}), (0, 0, 2))$. To get a correct type derivation, we have thus to use the following function type in $\Sigma(\text{append})$ for $n = 2$.

To infer the types in $\Sigma(\text{append})$ we apply the inference algorithm like in the case of appPairs. For the function arguments we use the annotation A and for the result we use A' as needed in the function signature Σ in the type derivation of appPairs. To generate the constraints, we fix this typing in a monomorphic signature Σ' with $\Sigma'(\text{append}) = ((L(\text{int}), L(\text{int})), A) \to (L(\text{int}), A')$. Since append is the only function in the SCC, we do not have to annotate other functions with fresh resource variables and have $\text{dom}(\Sigma') = \{\text{append}\}$.

---

[6]Note that the constraint generation for the function pairs would also involve a cost-free typing of maximal degree one for pairs which is used in the recursive call. However, all coefficients in this typing can simply be zero.

The most interesting part of the type derivation is the function application of append in the rule A:App. The cost-free function type $\Sigma_{cf}(\text{append}) = ((L(\text{int}), L(\text{int})), C) \rightarrow (L(\text{int}), C')$ has only annotations of maximal degree 1. That is, $C = (c_{(0,0)}, c_{(1,0)}, c_{(0,1)})$ and $C' = (c'_0, c'_1)$. The type states how to pass linear potential from the argument to the results. To generate the constraints that relate C to C' we annotate all functions in the SCC of append with fresh resource variables of maximal degree 1 and infer a cost-free type derivation for append. Since the maximal degree is 1, we use the rule A:App1 instead of A:App using the function type that is given in Σ_{cf} only. The generated constraints are equivalent to the following ones.

$$c_0 \geq c'_0 \qquad\qquad c_{(0,1)} \geq c'_1 \qquad\qquad c_{(1,0)} \geq c'_1$$

Intuitively, the constraints $T \geq A + C + K_1^{\text{app}}$ and $A' + C' \geq T' + K_2^{\text{app}}$ that are generated during the application of A:App express that the available potential (given by T) must be sufficient to cover cost of the recursive call as stated by Σ' (given by A). Additional potential (given by C)), can be passed on to the result of the function using a cost-free type.

Finally, we solve the collected constraints with an LP solver. To obtain an optimal typing for the function appPairs, the solver minimizes the objective function $q_{(0,0)} + 100q_{(1,0)} + 100q_{(0,1)} + 10000q_{(0,2)} + 10000q_{(2,0)} + 10000q_{(1,1)}$. It states that quadratic coefficients are more expensive than linear ones and that linear coefficients are more expansive than constants.

8. EXPERIMENTAL EVALUATION

We implemented the analysis system in a prototype of Resource-Aware ML. It is written in Haskell and consists of a parser (546 lines of code), a standard type checker (490 lines of code), an interpreter (333 lines of code), an LP solver interface (301 lines of code), and the actual analyzer (1637 lines of code). So far, we successfully analyzed 2491 lines of RAML code with the prototype.

In the conference version of this article we experimented with an early version of the prototype without any performance optimizations. In fact, the computed constraint systems were sometimes infeasible for the LP solver[7] we used.

Meanwhile we improved the performance of the prototype by an order of magnitude. Mainly, this improvement is due to the use of a different LP solver. We currently use the solver Coin LP[8]. Further speedup would be possible by using a commercial LP solver and by optimizing our Haskell implementation. However, we decided that accessibility and maintainability take precedence over performance in the prototype implementation. Right now, the maximal degree for which the performance is acceptable seems to be five or six.

Further improvement is possible by finding a suitable heuristic that is in between the (maybe too) flexible method we use here and the inference for the univariate system that also works efficiently with high maximal degree for large programs. For example, we could set certain coefficients q_i to zero before even generating the constraints. Alternatively, we could limit the number of different types for each function.

Table I shows a compilation of the computation of evaluation-step bounds for several example functions. All computed bounds are asymptotically tight. The runtime of the analysis is less then one second for all examples on an 3.6 GHz Intel Core 2 Duo iMac with 4 GB RAM depending on the needed degree and the complexity of the source program.

---

[7]lp_solve version 5.5.0.1.

[8]Clp version 1.14. See https://projects.coin-or.org/Clp.

Table I. Computation of Experimental Results

Function / Type	Computed Evaluation-Step Bound / Simplified Computed Bound	Actual Behavior	Run Time / # Constr.
isortlist : $L(L(\text{int}))\to L(L(\text{int}))$	$\sum_{1\leq i<j\leq n} 16m_i+16\binom{n}{2}+12n+3$ $8n^2m+8n^2-8nm+4n+3$	$O(n^2m)$	0.19 s 7307
nub : $(L(L(\text{int})), L(L(\text{int})))\to L(L(\text{int}))$	$\sum_{1\leq i<j\leq n} 12m_i+18\binom{n}{2}+12n+3$ $6n^2m+9n^2-6nm+3n+3$	$O(n^2m)$	0.21 s 9170
transpose : $L(L(\text{int}))\to L(L(\text{int}))$	$\sum_{1\leq i\leq n} 32m_i+2n+13$ $32nm+2n+13$	$O(nm)$	0.10 s 4223
mmult : $(L(L(\text{int})), L(L(\text{int})))\to L(L(\text{int}))$	$(\sum_{1\leq i\leq x} y_i)(32 + 28n)+14n+2x+21$ $28xyn+32xy+2x+14n+21$	$O(nxy)$	0.32 s 12311
dyad : $(L(\text{int}), L(\text{int}))\to L(L(\text{int}))$	$10nx+14n+3$ $10nx+14n+3$	$O(nx)$	0.02 s 344
lcs : $(L(\text{int}), L(\text{int}))\to\text{int}$	$39nx + 6x + 21n + 19$ $39nx + 6x + 21n + 19$	$O(nx)$	0.10 s 2921
subtrees : $T(\text{int})\to L(T(\text{int}))$	$8\binom{n}{2} + 23n + 3$ $4n^2 + 19n + 3$	$O(n^2)$	0.06 s 854
eratos : $L(\text{int})\to L(\text{int})$	$16\binom{n}{2}+12n+3$ $8n^2+4n+3$	$O(n^2)$	0.04 s 288
splitandsort : $L(\text{int}, \text{int})\to L(L(\text{int}), \text{int})$	$42\binom{n}{2}+58n+9$ $21n^2 + 37n + 9$	$O(n^2)$	0.64 s 20550

The table shows analyzed functions together with the function types, the computed evaluation-step bounds, the actual worst-case time behavior, the runtime of the analysis in seconds, and the number of generated linear constraints. All computed bounds are asymptotically tight and the constant factors are close to the worst-case behavior. In the bounds n is the size of the first argument, m_i are the sizes of the elements of the first argument, x is the size of the second argument, y_i are the sizes of the elements of the second argument, $m = \max_{1\leq i\leq n} m_i$, and $y = \max_{1\leq i\leq x} y_i$.

Our experiments show that the constant factors in the computed bounds are generally quite tight and even match the measured worst-case running times of many functions. The univariate analysis [Hoffmann and Hofmann 2010b, 2010a] infers identical bounds for the functions subtrees and eratos. In contrast, it can infer bounds for the other functions only after manual source-code transformations. Even then, the resulting bounds are not asymptotically tight.

We present the experimental evaluation of five programs shortly. The source-code and the experimental validation for the other examples are available online.[9] It is also possible to download the source-code of the prototype and to analyze user-generated examples directly on the Web.

8.1. Lexicographic Sorting of Lists of Lists

The following RAML code implements the well-known sorting algorithm insertion sort that lexicographically sorts a list of lists. To lexicographically compare two lists one needs time linear in the length of the shorter one. Since insertion sort does quadratic many comparisons in the worst case it has a running time of $O(n^2m)$ if n is the length of the outer list and m is the maximal length of the inner lists.

---

[9]http://raml.tcs.ifi.lmu.de.

```
leq (l1,l2) = match l1 with | nil -> true
                  | (x::xs) -> match l2 with | nil -> false
                                  | (y::ys) -> (x<y) or ((x == y) and leq (xs,ys));

insert (x,l) = match l with | nil -> [x]
                  | (y::ys) -> if leq(x,y) then x::y::ys
                                  else y::insert(x,ys);

isortlist l = match l with | nil -> nil
                  | (x::xs) -> insert (x,isortlist xs);
```

What follows is the analysis' output for the function isortlist when instantiated to bound the number of needed evaluation steps. The computation needs less than a second on typical desktop computers.

```
isortlist: L(L(int)) --> L(L(int))
Positive annotations of the argument
0  --> 3.0        2 --> 16.0      1 --> 12.0        [1,0] --> 16.0

The number of evaluation steps consumed by isortlist is at most:
        8.0*n^2*m + 8.0*n^2 - 8.0*n*m + 4.0*n + 3.0
where
   n is the length of the input
   m is the length of the elements of the input
```

The more precise bound implicit in the positive annotations of the argument is presented in mathematical notation in Table I.

We manually identified inputs for which the worst-case behavior of isortlist emerges (namely reversely sorted lists with similar inner lists). Then we measured the needed evaluation steps and compared the results to our computed bound. Our experiments show that the computed bound exactly matches the actual worst-case behavior.

8.2. Longest Common Subsequence

An example of dynamic programming that can be found in many textbooks is the computation of (the length of) the Longest Common Subsequence (LCS) of two given lists (sequences). If the sequences $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$ are given then an $n \times m$ matrix (here a list of lists) A is successively filled such that $A(i, j)$ contains the length of the LCS of $a_1, \ldots, a_i$ and $b_1, \ldots, b_j$. The following recursion is used in the computation.

$$A(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ A(i-1, j-1) + 1 & \text{if } i, j > 0 \text{ and } a_i = b_j \\ \max(A(i, j-1), A(i-1, j)) & \text{if } i, j > 0 \text{ and } a_i \neq b_j \end{cases}$$

The runtime of the algorithm is thus $O(nm)$. The following is the RAML implementation of the algorithm.

```
lcs(l1,l2) = let m = lcstable(l1,l2) in
    match m with | nil -> 0
      | (l1::_) -> match l1 with | nil -> 0
                    | (len::_) -> len;

lcstable (l1,l2) =
    match l1 with | nil -> [firstline l2]
      | (x::xs) -> let m = lcstable (xs,l2) in
                    match m with | nil -> nil
                      | (l::ls) -> (newline (x,l,l2))::l::ls;
```
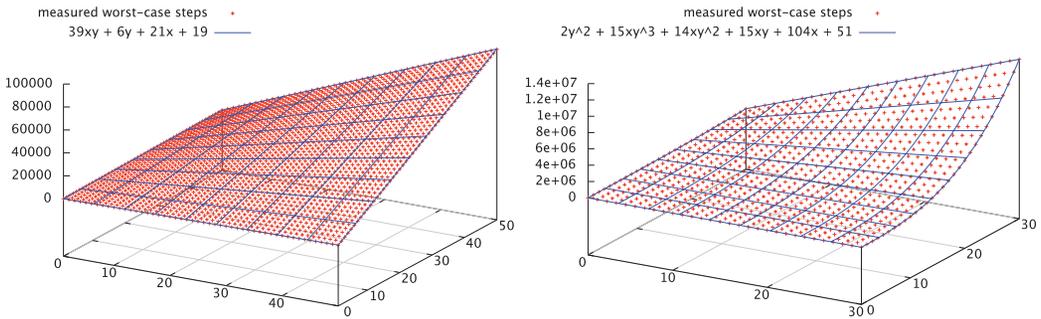
Fig. 9. The computed evaluation-step bound (lines) compared to the actual worst-case number of evaluation-steps for sample inputs of various sizes (crosses) used by lcs (on the left) and bftMult (on the right).

```
newline (y,lastline,l) =
   match l with | nil -> nil
     | (x::xs) -> match lastline with | nil -> nil
                     | (belowVal::lastline') ->
                         let nl = newline(y,lastline',xs) in
                         let rightVal = right nl in
                         let diagVal = right lastline' in
                         let elem = if x == y then diagVal+1
                                          else max(belowVal,rightVal)
                         in elem::nl;

firstline(l) = match l with | nil -> nil
                   | (x::xs) -> 0::firstline xs;

right l = match l with | nil -> 0
              | (x::xs) -> x;
```

The analysis of the program takes less than a second on a usual desktop computer and produces the following output for the function lcs.

```
lcs: (L(int),L(int)) --> int
Positive annotations of the argument
(0,0) --> 19.0      (1,0) --> 21.0
(0,1) --> 6.0       (1,1) --> 39.0

The number of evaluation steps consumed by lcs is at
most:    39.0*m*n + 6.0*m + 21.0*n + 19.0
where
   n is the length of the first component of the input
   m is the length of the second component of the input
```

Figure 9 shows that the computed bound is close to the measured number of evaluation steps needed. In the case of lcs the runtime exclusively depends on the lengths of the input lists.

8.3. Split and Sort

Our multivariate resource polynomials take into account the individual sizes of all inner data structures. In contrast to the approximation of, say, the lengths of inner lists by their maximal lengths, this approach leads to tight bounds when composing functions.

The function splitAndSort demonstrates this advantage.

```
splitAndSort : L(int,int) -> L(L(int),int)

splitAndSort l = sortAll (split l);
```

An input to the function is a list such as $\ell=[(1,0),(2,1),(3,0),(4,0),(5,1)]$ that contains integer pairs of the form code(value,key). The list is processed in two steps. First, split partitions the values according to their keys. For instance, we have split(ℓ)=[([2,5],1),([1,3,4],0)]. In the second step—implemented by sortAll—the inner lists are sorted with quick sort.

The function split is implemented as follows.

```
split : L(int,int) -> L(L(int),int)

split l = match l with | nil -> nil
            | (x::xs) -> insert( x, split xs);

insert : ((int,int),L(L(int),int)) -> L(L(int),int)

insert (x,l) = let (valX,keyX) = x in
  match l with | nil -> [([valX],keyX)]
    | (l1::ls) -> let (vals1,key1) = l1 in
              if key1 == keyX then (valX::vals1,key1)::ls
                else (vals1,key1)::insert(x,ls);
```

The prototype computes the tight quadratic bound $9n^2 + 9n + 3$ on the number of evaluation steps split needs for inputs of length n.

The second part of splitAndSort is sortAll. It uses the sorting algorithm quick sort to sort all the inner lists of its input. The function can be implemented as follows.

```
sortAll : L(L(int),int) -> L(L(int),int)

sortAll l = match l with | nil -> nil
              | (x::xs) -> let (vals,key) = x in
                        (quicksort vals,key)::sortAll(xs);

quicksort : L(int) -> L(int)

quicksort l = match l with | nil -> nil
              | (z::zs) -> let (xs,ys) = splitqs (z,zs) in
                        append(quicksort xs, z::(quicksort ys));

splitqs : (int,L(int)) -> (L(int),L(int))

splitqs(pivot,l) = match l with | nil -> (nil,nil)
                    | (x::xs) -> let (ls,rs) = splitqs (pivot,xs) in
                              if x > pivot then (ls,x::rs) else (x::ls,rs);

append : (L(int),L(int)) -> L(int)

append(l,ys) = match l with | nil -> ys
                | (x::xs) -> x::append(xs,ys);
```

The simplified computed evaluation-step bound for sortAll is $12nm^2 + 14nm + 14n + 3$ where n is the length of the outer list and m is the maximal length of the inner lists.

Now consider the composed function splitAndSort again and assume we would like to derive a bound for the function using the simplified bounds for sortAll and split. This would lead to a cubic bound for splitAndSort rather than a tight quadratic bound. The

reason is that—in the evaluation of splitAndSort(ℓ)—both n and m can only be bounded by $|\ell|$ the bound $12nm^2 + 14nm + 14n + 3$ for sortAll.

In contrast, the use of the multivariate resource polynomials enables the inference of a quadratic bound for splitAndSort. For one thing, the actual bound $\sum_{1 \leq i \leq n} \left(24\binom{m_i}{2} + 26m_i \right) + 14n + 3$ for sortAll incorporates the individual lengths m_i of the inner lists. For another thing, the type annotation for the function split passes potential from the function's argument to the inner lists of the result without losses.

As a result, the prototype computes the asymptotically tight, quadratic bound $21n^2 + 37n + 9$ for the function splitAndSort. The constant factors are, however, not tight. The reason is that the worst-case behavior of the function split emerges if all values in the input have different keys but the worst case of sortAll emerges if all values in the input have the same key. The analysis cannot infer that the worst-case behaviors are mutually exclusive but assumes that they can occur for the same input.

8.4. Breadth-First Traversal with Matrix Multiplication

A classic example that motivates amortized analysis is a functional queue. A *queue* is a first-in-first-out data structure with the operations enqueue and dequeue. The operation enqueue(a) adds a new element a to the queue. The operation dequeue() removes the oldest element from the queue. A queue is often implemented with two lists L_{in} and L_{out} that function as stacks. To enqueue a new element in the queue, one simply attaches it to the beginning of L_{in}. To dequeue an element from the queue, one detaches the first element from L_{out}. If L_{out} is empty then one transfers the elements from L_{in} to L_{out}, thereby reversing the elements' order.

Later in this example we shall store trees of matrices (lists of lists of integers) in our queue. So the two lists of queue have type $L(T(L(L(\text{int}))))$ in the following RAML implementation.

```
dequeue :  (L(T(L(L(int)))),L(T(L(L(int)))))
        -> (L(T(L(L(int)))),(L(T(L(L(int)))),L(T(L(L(int)))))))

dequeue (outq,inq) = match outq with
    | nil -> match reverse inq with | nil -> ([],([],[]))
            | t::ts -> ([t],(ts,[]))
    | t::ts -> ([t],(ts,inq));

enqueue :  (T(L(L(int))),(L(T(L(L(int)))),L(T(L(L(int))))))
        -> (L(T(L(L(int)))),L(T(L(L(int)))))

enqueue (t,queue) = let (outq,inq) = queue in (outq,t::inq);

appendreverse : (L(T(L(L(int)))),L(T(L(L(int))))) -> L(T(L(L(int))))

appendreverse (toreverse,sofar) = match toreverse with
    | nil -> sofar
    | (a::as) -> appendreverse(as,a::sofar);

reverse: L(T(L(L(int)))) -> L(T(L(L(int))))

reverse xs = appendreverse(xs,[]);
```

The prototype implementation infers precise linear bounds for the aforesaid functions. The evaluation-step bound for reverse is for instance $8n + 7$ where n is the length of the input list.

The point of this example is nevertheless the use of a queue in a breadth-first traversal of a binary tree. Suppose we are given a binary tree of matrices and we want to

multiply the matrices in breadth-first order. The matrices are represented as lists of lists of integers and can have different dimensions. However, we assume that the dimensions fit if the matrices are multiplied in breadth-first order. Before we implement the actual breadth-first traversal, we first implement matrix multiplication as follows. We use accumulation to avoid transposing matrices before the multiplication.

```
matrixMult : (L(L(int)),L(L(int))) -> L(L(int))

matrixMult (m1,m2) = match m1 with | [] -> []
      | (l::ls) -> (computeLine(l,m2,[])) :: matrixMult(ls,m2);

computeLine : (L(int),L(L(int)),L(int)) -> L(int)

computeLine (line,m,acc) = match line with | [] -> acc
      | (x::xs) -> match m with [] -> []
                   | (l::ls) -> computeLine(xs,ls,lineMult(x,l,acc));

lineMult : (int,L(int),L(int)) -> L(int)

lineMult (n,l1,l2) = match l1 with | [] -> []
      | (x::xs) -> match l2 with | [] -> x*n::lineMult(n,xs,[])
                   | (y::ys) -> x*n + y :: lineMult(n,xs,ys);
```

The computed evaluation-step bound for matrixMult is $15mkn + 16nm + 15n + 3$ if the first matrix is of dimension $n \times m$ and the second matrix is of dimension $m \times k$.[10]

Eventually, we implement the breadth-first traversal with matrix multiplication as follows.

```
bftMult : (T(L(L(int))),L(L(int))) -> L(L(int))

bftMult (t,acc) = bftMult'(([t],[]),acc);

bftMult' : ((L(T(L(L(int)))),L(T(L(L(int))))),L(L(int))) -> L(L(int))

bftMult'(queue,acc) =
  let (elem,queue) = dequeue queue in
  match elem with | nil -> acc
    | t::_ -> match t with | leaf -> bftMult'(queue,acc)
              | node(y,t1,t2) -> let queue' = enqueue(t2,enqueue(t1,queue)) in
                                 bftMult'(queue',matrixMult(acc,y));
```

If parametrized with the evaluation-step metric, the prototype produces the following output for bftMult.

```
bftMult: (T(L(L(int))),L(L(int))) --> L(L(int))
Positive annotations of the argument
(0,0)     --> 51.0      (1,1)     --> 15.0
(0,[1])   --> 2.0       ([1],1)   --> 14.0
(1,0)     --> 104.0     ([[1]],1) --> 15.0
The number of evaluation steps consumed by bftMult is at most:
        2.0*y*z + 15.0*y*n*m*x + 14.0*y*n*m + 15.0*y*n + 104.0*n + 51.0
where
   n is the size of the first component of the input
   m is the length of the nodes of the first component of the input
   x is the length of the elems. of the nodes of the first comp. of the input
```

---

[10]In fact, the bound that is presented to a user is at bit more general because the analysis cannot assume that the dimensions of the matrices fit.

```
    y is the length of the second component of the input
    z is the length of the elements of the second component of the input
```

The analysis thus derives a nontrivial asymptotically tight bound on the number of evaluation steps of bftMult. The analysis of the whole program takes about 30 seconds on a usual desktop computer and the prototype generates 947650 constraints. It is unclear how such a bound can be computed without the use of amortized analysis.

We compared the computed evaluation-step bound with the measured runtime of bftMult for balanced binary trees with quadratic matrices. Figure 9 shows the result of this experiment where x denotes the number of nodes in the tree and $y \times y$ is the dimension of the matrices. The constant factors in the bound almost match the optimal ones.

8.5. Nontermination and Reuse of Resources

Note that there is no syntactic restriction on the functions that can by analyzed by automatic amortized resource analysis. If a function does not consume resources then even nontermination is unproblematic.

Consider the function omega defined as follows.

```
omega : L(int) -> L(int)

omega (x) = omega (x)
```

The function does not terminate but neither does it consume any heap cells. For the heap-space metric and the type annotation omega: $(L(\text{int}), (q_0, q_1)) \rightarrow (L(\text{int}), (q'_0, q'_1))$, the constraint system that is generated by the prototype implementation poses no restrictions on the values of the resource annotations. Consequently, our prototype infers that no heap space is used by omega.

Assume that a cell of an integer list occupies two heap cells. The function append then consumes two heap cells per list element in the first input list. Since the prototype can infer the typing omega:$(L(\text{int}), (0, 0)) \rightarrow (L(\text{int}), (0, 2))$ it can also infer that the expression let l' = omega l in append(l',l') needs zero heap cells.

For an example of a nonterminating function that consumes and restitutes resources, consider the following function fibs that successively stores pairs of Fibonacci numbers on the heap.

```
fibs : L(int) -> unit

fibs l = matchD l with | nil -> ()
           | n::ls -> matchD ls with | nil -> ()
                      | m::_ -> fibs (m::(n+m)::nil);

main = fibs (0::1::nil)
```

The destructive pattern matching matchD deallocates the matched node of the list ℓ and frees 2 memory cells.[11] As a result, the function fibs stores the Fibonacci numbers in the heap space that is occupied by the input list ℓ without requiring additional space. The prototype implementation infers the following types for the program.

$$\text{fibs} \; : \; (L(\text{int}), (0, 0)) \rightarrow (\text{unit}, 0)$$
$$\text{main} \; : \; (\text{unit}, 4) \rightarrow (\text{unit}, 0)$$

The type of fibs states that the function does not need any heap space and the type of main states that the main expressions require four heap cells. These cells are used to create the initial list [0,1].

---

[11]See Hofmann and Jost [2003] for details on destructive pattern matches.

Finally, the function facList, which is defined next, is potentially nonterminating but has a worst-case heap-space consumption that is linear in the input.

```
fac : int -> int

fac n = if n == 0 then 1 else n*fac(n-1);

facList : L(int) -> L(int)

facList l = match l with | nil -> nil
                  | x::xs -> (fac x) :: (facList xs);
```

The function fac computes the factorial $n!$ for a nonnegative input $n \geq 0$ and diverges if $n < 0$. The function call facList(ℓ) applies fac to every element of the integer list ℓ. Consequently, facList(ℓ) terminates if and only if all elements of ℓ are nonnegative. However, the worst-case heap-space consumption of the function is $2|\ell|$ (if we again assume that a list element occupies two heap cells). The prototype computes this heap-space bound since it infers the typing facList : $(L(\text{int}), (0, 2)) \rightarrow (L(\text{int}), (0, 0))$.

9. THEORETICAL AND PRACTICAL LIMITATIONS

We only study the problem of deriving worst-case resource bounds rather than determining average-case resource behavior or lower bounds. Our analysis also does not establish any guaranties on the quality of the bounds.

Moreover, it is undecidable whether the resource usage of a RAML program is bounded. As a consequence, every method that automatically computes resource bounds has limitations.

9.1. Resource Polynomials

The bounds that we compute are resource polynomials. That means that we cannot express nonpolynomial bounds like $n \cdot \log n$ or 2^n. It also means that we cannot express every polynomial bound.

Recall from Section 4 that $\mathcal{R}_n = \mathcal{R}(L(\text{int}), \dots, L(\text{int})) = \{\sum_{i=1}^{m} q_i \prod_{j=1}^{n} \binom{x_j}{k_{ij}} \mid q_i \in \mathbb{Q}_0^+, m \in \mathbb{N}, k_{ij} \in \mathbb{N}\}$ of resource polynomials for n-tuples of integer lists is the set linear combinations of products of binomial coefficients. The variables x_i represent the lengths of the ith input list. The polynomials in $\mathcal{R}_n$ are nonnegative and closed under discrete differentiation Δ_i in the variable x_i for every i. Since it is the largest set of polynomials that enjoys these properties [Hoffmann and Hofmann 2010b], it includes polynomials of the form $\sum q_i \cdot \prod_{j=1}^{n} x_j^{k_{ij}}$ for $q_i \geq 0$. This shows that for every given polynomial $p(\vec{x})$ in variables x_j there is a resource polynomial $r(\vec{x})$ of the degree of p such that $p(\vec{x}) \leq r(\vec{x})$ for all x. One can obtain r, for example, by removing all the negative terms from p but often there exist resource polynomials r that are tighter upper bounds.

On the other hand, there exist monotonically nondecreasing polynomials that are not resource polynomials. An example is $F(x) = x^3 - 6x^2 + 12x$. Then $\Delta(F)(x) = 3x^2 - 15x + 19$ and $\Delta(\Delta(F))(x) = 6x - 18$. So the second discrete derivative is negative at $x = 1$ and F is not an element of $\mathcal{R}_1$.

Of course, it is possible to implement a program P_F whose worst-case behavior is exactly described by F. Since F is monotonically nondecreasing, the worst-case resource consumption of P_F for inputs of size x is greater than the worst-case resource consumption for inputs of size $x - 1$. This is a property that one would possibly expect from a typical program. However, the difference $\Delta(F)(x)$ between the worst-case behavior of P_F for inputs of size x and inputs of size $x - 1$ is not monotonically nondecreasing. Right now, we are not aware of a simple, well-known function or algorithm that has this property.

In general, resource polynomials are functions that can only be described by infinitely many variables or as a family of polynomials. Consider, for example, the resource polynomial $p_{[2,1]}([\ell_1, \ldots, \ell_n]) = \sum_{1 \leq i < j \leq n} \binom{|\ell_i|}{2} \cdot |\ell_j|$ that maps lists of lists of integers to nonnegative rational numbers. It can be seen as the family $(p_n)_{n \in \mathbb{N}}$ where $p_n(x_1, \ldots, x_n) = \sum_{1 \leq i < j \leq n} \binom{x_i}{2} \cdot x_j$.

More interestingly, every resource polynomial $p \in \mathcal{R}(L(L(\text{int})))$ for lists of lists of integers with $p([[], \ldots, []]) = 0$ (i.e., p is given by $(q_i)_{i \in I(L(L(\text{int})))}$ such that $q_n = 0$ for $n \in \mathbb{N}$ and thus p does not assign potential to the outer list) can be written as quasisymmetric function. A quasisymmetric function [Stanley 2001] is a formal power series of bounded degree in variables $x_1, x_2, \ldots$ with coefficients in $\mathbb{Q}$ such that the coefficient of the monomial $x_1^{d_1}, \ldots, x_n^{d_n}$ is equal to the coefficient of the monomial $x_{i_1}^{d_1}, \ldots, x_{i_n}^{d_n}$ for any strictly increasing sequence of positive integers $i_1 < \cdots < i_n$ and any positive integer sequence of exponents $d_1, \ldots, d_n$. For instance, we have $p_{[2,1]} = \sum_{i<j} \frac{1}{2} x_i^2 x_j - \frac{1}{2} x_i x_j$.

Clearly, resource polynomials cannot describe every formal power series. Apart from power series and quasisymmetric functions, we are not aware of any concepts of polynomials in infinitely many variables that we could rely on to discuss the limitations of our resource polynomials.

It seems to be possible to establish an analogous property for $\mathcal{R}(L(L(\text{int})))$ as for $\mathcal{R}_n$ in the sense that $\mathcal{R}(L(L(\text{int})))$ contains the largest set of nonnegative quasisymmetric functions that is closed under discrete derivation. It seems also possible to give a full characterization of the resource polynomials in terms of generalized quasisymmetric functions. However, such a characterization is beyond the scope of this article.

9.2. Type System

Our type system is a compromise between expressiveness and effectiveness. To enable automatic type inference and type checking, we restrict the entities that appear as sizes in the resource polynomials to be sizes of data structures. As a result, the derived bounds cannot depend on any user-defined measures such as the size difference between two values. For one thing, such measures would be useful to design a semiautomatic analysis that could be used interactively to develop resource bounds for more complicated programs.

For another thing, the limitation to sizes is problematic for numeric programs where recursive calls are often guarded by boolean expressions like $n > m$ and $i \leq j$. The resource cost of the program would then depend on the differences $j - i$ and $n - m$ between the initial values of $n, m, i,$ and j. In general, it is possible to derive bounds on programs whose resource consumption depends on natural numbers by interpreting the numbers as unit lists in our type system. It would then be possible to derive a bound for a function like the factorial function where recursive calls are made on smaller numbers only (fac(n) = ... n∗fac(n−1) ...). However, it is not possible to derive a bound for a function that is recursively called with larger arguments (f(n,m) = ... f(n+1,m) ...). Another problem that we do not address here is the computation of bounds for programs whose resource consumption depends on integers. The main challenges there are the treatment of overflows and negative numbers.

There are cases in which the uniform treatment of data structures in the type system inevitably leads to loose bounds. The problem is, for instance, that the types $L(L(\text{int}))$ and $(L(\text{int}), L(L(\text{int})))$ are not isomorphic in our type system, that is, $L(L(\text{int})) \not\cong (L(\text{int}), L(L(\text{int})))$. The reason is that the pair type allows for more resource annotations than the list type. Consider, for example, a function f of type $(L(\text{int}), L(L(\text{int}))) \rightarrow$ unit whose resource consumption is given by the resource polynomial $p_{(1,[1])}(\ell, [\ell_1, \ldots, \ell_n]) = |\ell| \cdot \sum_{1 \leq i \leq n} |\ell_i|$. To bound the resource consumption

of a function g(ls)=f(head(ls),tail(ls)) we have to use the resource polynomial $p_{[1,1]}([\ell_1, \ldots, \ell_n]) = \cdot \sum_{1 \leq i < j \leq n} |\ell_i| \cdot |\ell_j|$. This is nonetheless a crude upper bound since we do not need the terms $|\ell_i| \cdot |\ell_j|$ for $i > 1$.

9.3. Implementation

As mentioned in Section 8, our main focus in the implementation was correctness rather than performance. As a result, the analysis takes several minutes for lager programs if the given maximal degree is larger than five.

There are many ways to speed-up the generation of constraints which takes currently almost as long the LP solving. More importantly, the generation of constraints for all potential annotation of a fixed maximal degree for the whole program as well as a separate constraint set for every call site of a function are very coarse heuristics. To enable the analysis of very large programs, it would be necessary to develop a smarter heuristic to select the possible annotations for different parts of the program.

10. RELATED WORK

Automatic computation of resource bounds for program has been the topic of extensive study. In this section, we compare our work with related research on automatic resource analysis and on verification of resource bounds.

Classically, automatic resource analysis is based on recurrence relations. We discuss this long line of work in Section 10.1. Most closely related to this article is the previous work on automatic amortized analysis, which we describe in Section 10.2.

Other important techniques for resource analysis are based on sized types, or on abstract interpretation and invariant generation. We discuss this research in Sections 10.3 and 10.4, respectively. Further related work is discussed in Section 10.5.

10.1. Recurrence Relations

The use of recurrence relations (or recurrences) in automatic resource analysis has been pioneered by Wegbreit [1975]. The proposed analysis is performed in two steps: first extract recurrences from the program, then compute closed expressions for the recurrences. Wegbreit has implemented his analysis in the METRIC system for analyzing LISP programs but notices that it "can only handle simple programs" [Wegbreit 1975]. The most complicated examples that he provides are a reverse function for lists and a union function for sets represented by lists.

Webreit's method has dominated automatic resource analysis for many years. Benzinger [2001] notices the following.

> Automated complexity analysis is a perennial yet surprisingly disregarded aspect of static program analysis. The seminal contribution to this area was Wegbreit's METRIC system, which even today still represents the state-of-the-art in many aspects.

Ramshaw [1979] and Hickey and Cohen [1988] address the derivation of recurrences for average-case analysis. Flajolet et al. [1991] describe a theory of exact analysis in terms of generating functions for average-case analysis. A fragment of this theory has been implemented in an automatic average-case analyses of algorithms for "decomposable" combinatorial structures. Possible applications of Flajolet's method to worst-case analysis have not been explored.

The ACE system of Le Métayer [1988] analyses FP programs in two phases. A recursive function is first transformed into a recursive function that bounds the complexity of the original function. This function is then transformed into a nonrecursive one,

using predefined patterns. The ACE system can only derive asymptotic bounds rather than constant factors as is done in RAML.

Rosendahl [1989] has implemented an automatic resource analysis for first-order LISP programs. The analysis first converts programs into a step-counting version which is then converted into a time bound function via abstract interpretation of the step-counting version. The reported results are similar to Wegbreit's results and programs with nested data structures and typical compositional programs cannot be handled.

Benzinger [2001, 2004] has applied Wegbreit's method in an automatic complexity analysis for higher-order Nuprl terms. He uses Mathematica to solve the generated recurrence equations. Grobauer [2001] has reported an interesting mechanism to automatically derive cost recurrences from DML programs using dependent types. The computation of closed forms for the recurrences is, however, not discussed.

Recurrence relations have also been proposed to automatically derive resource bounds for logic programs [Debray and Lin 1993].

In the COSTA project, both the derivation and the solution of recurrences are studied. Albert et al. [2007] have introduced a method for automatically inferring recurrence relations from Java bytecode. They rely on abstract interpretation to generate size relations between program variables at different program points. A recent advancement is the derivation of bounds that take garbage collection into account [Albert et al. 2010].

The COSTA team states that existing computer algebra systems are in most cases not capable of handling recurrences that originate from resource analysis [Albert et al. 2008]. As a result, a series of papers [Albert et al. 2008, 2011a, 2011b] studies the derivation of closed forms for so-called cost relations, that is, recurrences that are produced by automatic resource analysis. They use partial evaluation and apply static analysis techniques such as abstract interpretation to obtain loop invariants and ranking functions. Another work [Albert et al. 2009a] studies the computation of asymptotic bounds for recurrences.

While the COSTA system can compute bounds that contain integers, the amortized method is favorable in the presence of (nested) data structures and function composition.

10.2. Automatic Amortized Analysis

The concept of automatic amortized resource analysis has been introduced in Hofmann and Jost [2003]. The potential method is used there to analyze the heap-space consumption of first-order functional programs, establishing the idea of attaching potential to data structures, the use of type systems to proof bounds, and the inference of type annotations using linear programming. Other than here, the analysis system uses linear potential annotations and thus derives linear resource bounds only.

The subsequent work on amortized analysis for functional programs successively broadened the range of this analysis method while the limitation to linear bounds remained. Jost et al. [2009] have extended automatic amortized analysis to generic resource metrics and user-defined inductive data structures, Campbell [2009] has developed an amortized resource analysis that computes bounds on the stack space of functional programs, and Jost et al. [2010] have extended linear amortized resource analysis to polymorphic and higher-order programs.

Automatic amortized resource analysis was successfully applied to object-oriented programs [Hofmann and Jost 2006; Hofmann and Rodriguez 2009] requiring refined potential annotations to deal with object-oriented language features such as inheritance, casts, and imperative updates. Atkey [2010] integrated linear amortized analysis into a program logic for Java-like bytecode using bunched implications and separation logic.

All the previous works on amortized analysis only describe systems that are restricted to linear bounds. In Hoffmann and Hofmann [2010b, 2010a], we have

described the first automatic amortized analysis that can compute superlinear bounds. In this work, we extend this system from univariate bounds (such as $n^2 + m^2$) to multivariate bounds (such as nm). This article is the full version of an earlier conference article [Hoffmann et al. 2011].

10.3. Sized Types

A *sized type* is a type that contains size bounds for the inhabiting values. The size information is usually attached to inductive datatypes via natural numbers. The difference to the potential annotations of amortized analysis is that sized types bound sizes of data while potential annotations define a potential as a function of the data size.

Sized types have been introduced by Hughes et al. [1996] in the context of functional reactive programming to prove that stream manipulating functions are productive or in other words, that the computation of each stream element terminates.

Hughes and Pareto [1999] have studied the use of sized types to derive space bounds for a functional language with region-based memory management. The type system features both resource and size annotations to express bounds but the annotations have to be provided by the programmer.

Type inference for sized types has first been studied by Chin and Khoo [2001]. They employ an approximation algorithm for the transitive closure of Presburger constraints to infer size relations for recursive functions. The algorithm only computes *linear* relations and does not scale well for nested data structures.

Vasconcelos [2008] studies sized types to infer upper bounds on the resource usage of higher-order functional programs. He employs abstract interpretation techniques for automatically inferring linear approximations of the sizes of data structures and the resource usage of recursive functions. Other than RAML, this system can only compute linear bounds.

10.4. Abstract Interpretation

Abstract interpretation is a well-established framework for static program analysis. There are several works that employ abstract interpretation to compute symbolic complexity bounds. Unfortunately, none of the described prototype implementations is publicly available. Hence, we can compare our analysis only to the results that are reported in the respective papers.

Worst-Case Execution Time (WCET) analysis is a large research area that traditionally computes time bounds for "a restricted form of programming, which guarantees that programs always terminate and recursion is not allowed or explicitly bounded as are the iteration counts of loops" [Wilhelm et al. 2008]. The time bounds are computed for specific hardware architectures and are very precise because the analysis takes into account low-level features like hardware caches and instruction pipelines.

In contrast to traditional WCET analysis, *parametric* WCET analysis uses abstract interpretation to compute symbolic clock-cycle bounds for specific hardware. Lisper [2003] has proposed the use of a flow analysis with a polyhedral abstraction and symbolically bounds the points in polyhedrals. This method has recently been implemented [Bygde et al. 2009]. It can only handle integer programs without dynamic allocation and recursion.

Altmeyer et al. [2008, 2011] have reported a similar approach. They propose a parametric loop analysis that consists of four phases: identifying loop counters, deriving loop invariants, evaluation of loop exits, and finally constructing loop bounds. The analysis operates directly on executables and can also handle recursion. However, a user has to provide a parameters that bound the recursion (or loop iterations) that traverses a data structure. In contrast, our analysis is fully automatic.

Cook et al. [2009] describe a method for computing symbolic heap bounds for C programs that are used in hardware synthesis. They use a numerical heap abstraction and rely on nonlinear constraint solving. While it is in principle possible to derive nonlinear bounds with this technique, the paper only describes examples with linear bounds.

A successful method to estimate time bounds for C++ procedures with loops and recursion has recently been developed by Gulwani et al. [2008, 2009b] in the SPEED project. They annotate programs with counters and use automatic invariant discovery between their values using off-the-shelf program analysis tools which are based on abstract interpretation. An alternative approach that leads to impressive experimental results is to use "a database of known loop iteration lemmas" instead of the counter instrumentation [Gulwani et al. 2009a].

Another recent innovation for nonrecursive programs is the combination of disjunctive invariant generation via abstract interpretation with proof rules that employ SMT-solvers [Gulwani and Zuleger 2010].

In contrast to our method, these techniques cannot fully automatically analyze iterations over data structures. Instead, the user needs to define numerical "quantitative functions". This seems to be less modular for nested data structures where the user needs to specify an "owner predicate" for inner data structures. It is also unclear if quantitative functions can represent complex mixed bounds such as $\sum_{1 \leq i < j \leq n}(10m_i + 2m_j) + 16\binom{n}{2} + 12n + 3$ which RAML computes for isortlist. Moreover, our method infers tight bounds for functions such as insertion sort that admit a worst-case time usage of the form $\sum_{1 \leq i \leq n} i$. In contrast, Gulwani et al. [2009b] indicate that a nested loop on $1 \leq i \leq n$ and $1 \leq j \leq i$ is overapproximated with the bound n^2.

A methodological difference to techniques based on abstract interpretation is that we infer (using linear programming) an abstract potential function which indirectly yields a resource-bounding function. The potential-based approach may be favorable in the presence of compositions and data scattered over different locations (partitions in quick sort). Additionally, there seem to be no experiments that relate the derived bounds to the actual worst-case behavior and there is no publically available implementation.

As any type system, our approach is naturally compositional and lends itself to the smooth integration of components whose implementation is not available. Moreover, type derivations can be seen as certificates and can be automatically translated into formalized proofs in program logic [Beringer et al. 2004]. On the other hand, our method does not model the interaction of integer arithmetic with resource usage.

10.5. Other Work

There are techniques [Braberman et al. 2008; Clauss et al. 2009] that can compute the memory requirements of object-oriented programs with region-based garbage collection. These systems infer invariants and use external tools that count the number of integer points in the corresponding polytopes to obtain bounds. The described technique can handle loops but not recursive or composed functions.

Taha et al. [2003] describe a two-stage programming language in which the first stage can arbitrarily allocate memory and the second stage—that uses LFPL [Hofmann 2000]—can allocate no memory. The work explores, however, no method to derive a memory bound for the first stage.

Other related works use type systems to validate resource bounds. Crary and Weirich [2000] have presented a (monomorphic) type system capable of specifying and certifying resource consumption. Danielsson [2008] has provided a library, based on dependent types and manual cost annotations, that can be used for complexity analyses of purely functional data structures and algorithms. In contrast, our focus is on the inference of bounds.

Chin et al. [2008] use a Presburger solver to obtain *linear* memory bounds for low-level programs. In contrast, the analysis system we present can compute polynomial bounds.

Polynomial resource bounds have also been studied by Shkaravska et al. [2007] who address the derivation of polynomial size bounds for functions whose exact growth rate is polynomial. Besides this strong restriction, the efficiency of inference remains unclear.

11. CONCLUSION AND DIRECTIONS FOR FUTURE WORK

We have introduced a quantitative amortized analysis for first-order functions with multiple arguments. For the first time, we have been able to fully automatically derive complex multivariate resource bounds for recursive functions on nested inductive data structures such as lists and trees. Our experiments have shown that the analysis is sufficiently efficient for the functions we have tested, and that the resulting bounds are not only asymptotically tight but are also surprisingly precise in terms of constant factors.

The system we have developed will be the basis of various future projects. A challenging unsolved problem we are interested in is the computation of precise heap-space bounds in the presence of automatic memory management.

We have first ideas for extending the type system to derive bounds that contain not only polynomial but also involve logarithmic and exponential functions. The extension of linear amortized analysis to polymorphic and higher-order programs [Jost et al. 2010] seems to be compatible with our system and it would be interesting to integrate it. Finally, we plan to investigate to what extent our multivariate amortized analysis can be used for programs with cyclic data structures (following Hofmann and Jost [2006], Hofmann and Rodriguez [2009], and Atkey [2010]) and recursion (including loops) on integers. For the latter it might be beneficial to merge the amortized method with successful existing techniques on abstract interpretation [Gulwani et al. 2009b; Albert et al. 2009b].

Another very interesting and rewarding piece of future work would be an adaptation of our method to imperative languages without built-in inductive types such as C. One could try to employ pattern-based discovery of inductive data structures as is done, for example, in separation logic.

APPENDIX (SOUNDNESS PROOF)

In the following we prove Theorem 6.7 which states the soundness of the type system. Let $\mathcal{H} \vDash \mathcal{V}{:}\Gamma$ and $\Sigma; \Gamma; Q \vdash e{:}(B, Q')$. We prove the following two statements.

(1) If $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (p, p')$ then $p \leq \Phi_{\mathcal{V}, \mathcal{H}}(\Gamma; Q)$ and $p - p' \leq \Phi_{\mathcal{V}, \mathcal{H}}(\Gamma; Q) - \Phi_{\mathcal{H}'}(v{:}(B, Q'))$.
(2) If $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \mid p$ then $p \leq \Phi_{\mathcal{V}, \mathcal{H}}(\Gamma; Q)$.

Lemma A.1 is used to show the soundness of the rule T:LET and states that the potential of a context is invariant during the evaluation. This is a consequence of allocated heap cells being immutable with the language features that we describe in this article.

LEMMA A.1. *Let $\mathcal{H} \vDash \mathcal{V}{:}\Gamma$, $\Sigma; \Gamma; Q \vdash e : (B, Q')$ and $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (p, p')$. Then it is true that $\Phi_{\mathcal{V}, \mathcal{H}}(\Gamma; Q) = \Phi_{\mathcal{V}, \mathcal{H}'}(\Gamma; Q)$.*

PROOF. The lemma is a direct consequence of the definition of the potential Φ and the fact that $\mathcal{H}'(\ell) = \mathcal{H}(\ell)$ for all $\ell \in \mathrm{dom}(\mathcal{H})$ which is proved in Proposition 3.2. $\square$

Proof of Part 1

We prove $p \leq \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q)$ and $p - p' \leq \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q) - \Phi_{\mathcal{H}'}(v{:}(B, Q'))$ by induction on the derivations of $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (p, p')$ and $\Sigma; \Gamma; Q \vdash e : (B, Q')$, where the induction on the evaluation judgment takes priority.

(T:SHARE). Suppose that the derivation of $\Sigma; \Gamma; Q \vdash e : (B, Q')$ ends with an application of the rule T:SHARE. Then $\Gamma = \Gamma', z{:}A$. It follows from the premise that

$$\Gamma', x{:}A, y{:}A; P \vdash e' : (B, Q') \tag{5}$$

for a type annotation P with $Q = \curlyvee(P)$ and an expression e' with $e'[z/x, z/y] = e$. Since $\mathcal{H} \vDash \mathcal{V} : \Gamma', z{:}A$ and $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (p, p')$ it follows that $\mathcal{H} \vDash \mathcal{V}_{xy} : \Gamma', x{:}A, y{:}A$ and

$$\mathcal{V}_{xy}, \mathcal{H} \vdash e' \rightsquigarrow v, \mathcal{H}' \mid (p, p'), \tag{6}$$

where $\mathcal{V}_{xy} = \mathcal{V} \cup \{x \mapsto \mathcal{V}(x), y \mapsto \mathcal{V}(z)\}$. Thus we can apply the induction hypothesis to (5) and (6) to derive

$$p \leq \Phi_{\mathcal{V}_{xy},\mathcal{H}}(\Gamma', x{:}A, y{:}A; P) \tag{7}$$

and

$$p - p' \leq \Phi_{\mathcal{V}_{xy},\mathcal{H}}(\Gamma', x{:}A, y{:}A; P) - (\Phi_{\mathcal{H}'}(v{:}(B, Q'))) . \tag{8}$$

From the definition of the sharing annotation $\curlyvee(Q)$ (compare Lemma 6.6) it follows that

$$\Phi_{\mathcal{V}_{xy},\mathcal{H}}(\Gamma', x{:}A, y{:}A; P) = \Phi_{\mathcal{V},\mathcal{H}}(\Gamma', z{:}A; Q). \tag{9}$$

The claim follows from (7), (8), and (9).

(T:AUGMENT). If the derivation of $\Sigma; \Gamma; Q \vdash e : (B, Q')$ ends with an application of the rule T:AUGMENT then we have $\Sigma; \Gamma'; Q \vdash e : (B, Q')$ for a context Γ' with $\Gamma', x{:}A = \Gamma$. From the assumption $\mathcal{H} \vDash \mathcal{V} : \Gamma', x{:}A$ it follows that $\mathcal{H} \vDash \mathcal{V} : \Gamma'$. Thus we can apply the induction hypothesis to the premise $\Gamma'; \pi_{\bar{0}}^{\Gamma'}(Q) \vdash e : (B, Q')$ of T:AUGMENT. We derive

$$p \leq \Phi_{\mathcal{V},\mathcal{H}}\big(\Gamma'; \pi_{\bar{0}}^{\Gamma'}(Q)\big) \tag{10}$$

and

$$p - p' \leq \Phi_{\mathcal{V},\mathcal{H}}\big(\Gamma'; \pi_{\bar{0}}^{\Gamma'}(Q)\big) - (\Phi_{\mathcal{H}'}(v{:}(B, Q'))) . \tag{11}$$

Assume that $\mathcal{H} \vDash \mathcal{V}(x) \mapsto a : A$. From Proposition 6.3 it follows that $\Phi_{\mathcal{V},\mathcal{H}}(\Gamma'; \pi_{\bar{0}}^{\Gamma'}(Q)) = \Phi_{\mathcal{V},\mathcal{H}}(\Gamma'; \pi_{\bar{0}}^{\Gamma}(Q)) \cdot p_{\bar{0}}(a) \leq \sum_{j \in I(A)} \Phi_{\mathcal{V},\mathcal{H}}(\Gamma'; \pi_j^{\Gamma'}(Q)) \cdot p_j(a) = \Phi_{\mathcal{V},\mathcal{H}}(\Gamma', x{:}A; Q)$. Hence we have

$$\Phi_{\mathcal{V},\mathcal{H}}\big(\Gamma'; \pi_{\bar{0}}^{\Gamma'}(Q)\big) \leq \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q) \tag{12}$$

and the claim follows from (10), (11), and (12).

(T:WEAKEN). Assume the derivation of the typing judgment ends with an application of the typing rule T:WEAKEN. Then we have $\Gamma; P \vdash e : (B, P')$, $Q \geq P$, and $Q' \leq P'$. We can conclude by induction that

$$p \leq \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; P) \qquad \text{and} \qquad p - p' \leq \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; P) - \Phi_{\mathcal{H}'}(v{:}(B, P')) . \tag{13}$$

From the definition of $\leq$ for type annotations it follows immediately that

$$\Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q) \geq \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; P) \qquad \text{and} \qquad \Phi_{\mathcal{H}'}(v{:}(B, P')) \leq \Phi_{\mathcal{H}'}(v{:}(B, Q')) . \tag{14}$$

The claim follows then from (13) and (14).

(T:OFFSET). The case T:OFFSET is similar to the case T:WEAKEN.

(T:VAR). Assume that e is a variable x that has been evaluated with the rule E:VAR. Then it is true that $\mathcal{H} = \mathcal{H}'$. The type judgment $\Sigma; \Gamma; Q \vdash x{:}(B, Q')$ has been derived by a single application of the rule T:VAR. Thus we have $\Gamma = x{:}B$,

$$\Phi_{\mathcal{V},\mathcal{H}}(x{:}B; Q) - \Phi_{\mathcal{V},\mathcal{H}'}(x{:}(B, Q')) = K^{\text{var}} \tag{15}$$

and in particular $\Phi_{\mathcal{V},\mathcal{H}}(x{:}B; Q) \geq K^{\text{var}}$.

Assume first that $K^{\text{var}} \geq 0$. Then it follows by definition that $p = K^{\text{var}}$, $p' = 0$ and thus $p - p' = K^{\text{var}}$. The claim follows from (15). Assume now that $K^{\text{var}} < 0$. Then it follows by definition that $p = 0$, $p' = -K^{\text{var}}$. We have again that $p - p' = K^{\text{var}}$ and the claim follows from (15). (Remember that we have the implicit side condition that $\Phi_{\mathcal{V},\mathcal{H}}(x{:}B; Q) \geq 0$.)

(T:CONST*). Similar to the case (T:VAR).

(T:OPINT). Assume that the type derivation ends with an application of the rule T:OPINT. Then e has the form $x_1\ op\ x_2$ and the evaluation consists of a single application of the rule E:BINOP. From the rule T:OPINT it follows that $\Gamma = x_1{:}\text{int}, x_2{:}\text{int}$ and $\Phi_{\mathcal{V},\mathcal{H}}(x_1{:}\text{int}, x_2{:}\text{int}; Q) - \Phi_{\mathcal{V},\mathcal{H}'}(v : (\text{int}, Q')) = q_{(0,0)} - q_0' = K^{op}$.

If $K^{op} \geq 0$ then $p = K^{op}$ and $p' = 0$. Thus $p = K^{op} \leq q_{(0,0)} = \Phi_{\mathcal{V},\mathcal{H}}(x_1{:}\text{int}, x_2{:}\text{int}; Q)$ and $p - p' = K^{op} = \Phi_{\mathcal{V},\mathcal{H}}(x_1{:}\text{int}, x_2{:}\text{int}; Q) - (\Phi_{\mathcal{V},\mathcal{H}'}(v{:}(\text{int}, Q')))$.

If $K^{op} < 0$ then $p = 0$ and $p' = -K^{op}$. Thus $p \leq q = \Phi_{\mathcal{V},\mathcal{H}}(x_1{:}\text{int}, x_2{:}\text{int}; Q)$ and $p - p' = K^{op} = \Phi_{\mathcal{V},\mathcal{H}}(x_1{:}\text{int}, x_2{:}\text{int}; Q) - (\Phi_{\mathcal{V},\mathcal{H}'}(v : (\text{int}, Q')))$.

(T:OPBOOL). The case in which the type derivation ends with an application of T:OPBOOL is similar to the case (T:OPINT).

(T:LET). If the type derivation ends with an application of T:LET then e is a let expression of the from let $x = e_1$ in e_2 that has eventually been evaluated with the rule E:LET. Then it follows that $\mathcal{V}, \mathcal{H} \vdash e_1 \rightsquigarrow v_1, \mathcal{H}_1 \mid (r, r')$ and $\mathcal{V}', \mathcal{H}_1 \vdash e_2 \rightsquigarrow v_2, \mathcal{H}_2 \mid (t, t')$ for $\mathcal{V}' = \mathcal{V}[x \mapsto v_1]$ and r, r', t, t' with

$$(p, p') = K_1^{\text{let}} \cdot (r, r') \cdot K_2^{\text{let}} \cdot (t, t') \cdot K_3^{\text{let}}. \tag{16}$$

The derivation of the type judgment for e ends with an application of L:LET. Hence $\Gamma = \Gamma_1, \Gamma_2$, $\Sigma; \Gamma_1; P \vdash e_1 : (A, P')$, $\Sigma; \Gamma_2, x{:}A; R \vdash e_2 : (B, R')$ and

$$P + K_1^{\text{let}} = \pi_0^{\Gamma_1}(Q) \tag{17}$$

$$P' = \pi_0^{x:A}(R) + K_2^{\text{let}} \tag{18}$$

$$R' = Q' + K_3^{\text{let}}. \tag{19}$$

Furthermore we have for every $\vec{0} \neq j \in I(\Gamma_2)$: $\quad \Gamma_1; P_j \vdash^{\underline{\text{cf}}} e_1 : (A, P_j')$,

$$P_j = \pi_j^{\Gamma_1}(Q) \tag{20}$$

$$P_j' = \pi_j^{x:A}(R). \tag{21}$$

Since $\mathcal{H} \vDash \mathcal{V} : \Gamma$ we have also $\mathcal{H} \vDash \mathcal{V} : \Gamma_1$ and can thus apply the induction hypothesis for the evaluation judgment of e_1 to derive

$$r \leq \Phi_{\mathcal{V},\mathcal{H}}(\Gamma_1; P) \tag{22}$$

$$r - r' \leq \Phi_{\mathcal{V},\mathcal{H}}(\Gamma_1; P) - \Phi_{\mathcal{H}_1}(v_1{:}(A, P')). \tag{23}$$

From Theorem 3.4 it follows that $\mathcal{H}_2 \vDash \mathcal{V}' : \Gamma_2, x{:}A$ and thus again by induction

$$t \leq \Phi_{\mathcal{V}',\mathcal{H}_1}(\Gamma_2, x{:}A; R) \tag{24}$$

$$t - t' \leq \Phi_{\mathcal{V}',\mathcal{H}_1}(\Gamma_2, x{:}A; R) - \Phi_{\mathcal{H}_2}(v_2{:}(B, R')). \tag{25}$$

Furthermore we apply the induction hypothesis to the evaluation judgment for e_1 with the cost-free metric. Then we have $r = r' = 0$ and therefore for every $\vec{0} \neq j \in I(\Gamma_2)$

$$\Phi_{\mathcal{V},\mathcal{H}}(\Gamma_1; P_{\bar{j}}) \geq \Phi_{\mathcal{H}_1}(v_1{:}(A, P'_{\bar{j}})) \,. \tag{26}$$

Let $\Gamma_1 = x_1, \ldots, x_n$, $\Gamma_2 = y_1, \ldots, y_m$, $\mathcal{H} \vDash \mathcal{V}(x_j) \mapsto a_{x_j} : \Gamma(x_j)$, and $\mathcal{H} \vDash \mathcal{V}(y_j) \mapsto b_{y_j} : \Gamma(y_j)$. Define

$$\phi_P = \Phi_{\mathcal{V},\mathcal{H}}(\Gamma_1; P) + \sum_{\vec{0} \neq \vec{j} \in I_k(\Gamma_2)} \Phi_{\mathcal{V},\mathcal{H}}(\Gamma_1; P_{\bar{j}}) \cdot \prod_{k=1}^{m} p_{j_k}(b_{x_k})$$

$$\phi_{P'} = \Phi_{\mathcal{H}_1}(v_1{:}(A, P')) + \sum_{\vec{0} \neq \vec{j} \in I_k(\Gamma_2)} \Phi_{\mathcal{H}_1}(v_1{:}(A, P'_{\bar{j}})) \cdot \prod_{k=1}^{m} p_{j_k}(b_{x_k}).$$

We argue that

$$\Phi_{\mathcal{V},\mathcal{H}}(\Gamma_1, \Gamma_2; Q) \overset{\text{Prop. 6.3}}{=} \sum_{\vec{j} \in I_k(\Gamma_2)} \Phi_{\mathcal{V},\mathcal{H}}\big(\Gamma_1; \pi_{\bar{j}}^{\Gamma_1}(Q)\big) \prod_{k=1}^{m} p_{j_k}(b_{x_k})$$

$$\overset{(17,20)}{=} \Phi_{\mathcal{V},\mathcal{H}}(\Gamma_1; P) + K_1^{\text{let}} + \sum_{\vec{0} \neq \vec{j} \in I_k(\Gamma_2)} \Phi_{\mathcal{V},\mathcal{H}}(\Gamma_1; P_{\bar{j}}) \cdot \prod_{k=1}^{m} p_{j_k}(b_{x_k})$$

$$= \phi_P + K_1^{\text{let}}. \tag{27}$$

Similarly, we use Proposition 6.3, (18), and (21) to see that

$$\phi_{P'} = \Phi_{\mathcal{V},\mathcal{H}_1}(\Gamma_2, x{:}A; R) + K_2^{\text{let}}. \tag{28}$$

Additionally we have

$$r - r' \overset{(23)}{\leq} \Phi_{\mathcal{V},\mathcal{H}}(\Gamma_1; P) - \Phi_{\mathcal{H}_1}(v_1{:}(A, P'))$$

$$\overset{(26)}{\leq} \Phi_{\mathcal{V},\mathcal{H}}(\Gamma_1; P) - \Phi_{\mathcal{H}_1}(v_1{:}(A, P')) + \sum_{\vec{0} \neq \vec{j} \in I_k(\Gamma_2)} \Phi_{\mathcal{V},\mathcal{H}}(\Gamma_1; P_{\bar{j}}) \cdot \prod_{k=1}^{m} p_{j_k}(b_{x_k})$$

$$- \sum_{\vec{0} \neq \vec{j} \in I_k(\Gamma_2)} \Phi_{\mathcal{H}_1}(v_1{:}(A, P'_{\bar{j}})) \cdot \prod_{k=1}^{m} p_{j_k}(b_{x_k})$$

$$= \phi_P - \phi_{P'}. \tag{29}$$

Now let

$$(u, u') = K_1^{\text{let}} \cdot (\phi_P, \phi_{P'}) \cdot K_2^{\text{let}} \cdot (\Phi_{\mathcal{V}',\mathcal{H}_1}(\Gamma_2, x{:}A; R), \Phi_{\mathcal{H}_2}(v_2{:}(B, R'))) \cdot K_3^{\text{let}}.$$

Then it follows that

$$(u, u') \overset{(19,28)}{=} K_1^{\text{let}} \cdot \big(\phi_P, \phi_{P'} - K_2^{\text{let}}\big) \cdot (\Phi_{\mathcal{V}',\mathcal{H}_1}(\Gamma_2, x{:}A; R), \Phi_{\mathcal{H}_2}\big(v_2{:}(B, R')) - K_3^{\text{let}}\big)$$

$$\overset{(28)}{=} K_1^{\text{let}} \cdot (\phi_P, v')$$

for some $v' \in \mathbb{Q}_0^+$. Now we can conclude that

$$u \leq \max(0, \phi_P + K_1^{\text{let}}) \overset{(27)}{\leq} \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q).$$

Finally, it follows with Proposition 3.1 applied to (22), (29), (24), (25), and (16) that $u \geq p$.

For the second part of the statement we apply Proposition 3.1 to (16) and derive the following.

$$
\begin{aligned}
p - p' &= r - r' + t - t' + K_1^{\text{let}} + K_2^{\text{let}} + K_3^{\text{let}} \\
&\overset{(25,29)}{\leq} \phi_P - \phi_{P'} + \Phi_{\mathcal{V}',\mathcal{H}_1}(\Gamma_2, x{:}A; R) - \Phi_{\mathcal{H}_2}(v_2{:}(B, R')) + K_1^{\text{let}} + K_2^{\text{let}} + K_3^{\text{let}} \\
&\overset{(28)}{=} \phi_P - \Phi_{\mathcal{H}_2}(v_2{:}(B, R')) + K_1^{\text{let}} + K_3^{\text{let}} \\
&\overset{(27)}{=} \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q) - \Phi_{\mathcal{H}_2}(v_2{:}(B, R')) + K_3^{\text{let}} \\
&\overset{(19)}{\leq} \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q) - \Phi_{\mathcal{H}_2}(v_2{:}(B, Q'))
\end{aligned}
$$

(T:App).   Assume that e is a function application of the form $f(x)$. The evaluation of e then ends with an application of the rule E:App. Thus we have $\mathcal{V}(x) = v'$ and $[y_f \mapsto v'], \mathcal{H} \vdash e_f \rightsquigarrow v, \mathcal{H}' \mid (r, r')$ for some r, r' with

$$
(p, p') = K_1^{\text{app}} \cdot (r, r') \cdot K_2^{\text{app}}. \tag{30}
$$

The derivation of the type judgment for e ends with an application of T:Fun. Therefore it is true that $\Gamma = x{:}A; Q$, $(A, P) \rightarrow (B, P') \in \Sigma(f)$, and

$$
P + K_1^{\text{app}} = Q \qquad \text{and} \qquad P' = Q' + K_2^{\text{app}}. \tag{31}
$$

In order to apply the induction hypothesis to the evaluation of the function body e_f we recall from the definition of a well-formed program that $(A, P) \rightarrow (B, P') \in \Sigma(f)$ implies that $\Sigma; y_f{:}A; P \vdash e_f{:}P'$. Since $\mathcal{H} \vDash \mathcal{V} : x{:}A$ and $\mathcal{V}(x) = v'$ it follows $\mathcal{H} \vDash [y_f \mapsto v'] : y_f{:}A$. We obtain by induction that

$$
r \leq \Phi_{[y_f \mapsto v'], \mathcal{H}}(y_f{:}A; P) \tag{32}
$$

$$
r - r' \leq \Phi_{[y_f \mapsto v'], \mathcal{H}}(y_f{:}A; P) - \Phi_{\mathcal{H}'}(v{:}(B, P')). \tag{33}
$$

Now define

$$
(u, u') = K_1^{\text{app}} \cdot (\Phi_{[y_f \mapsto v'], \mathcal{H}}(y_f{:}A; P), \Phi_{\mathcal{H}'}(v{:}(B, P'))) \cdot K_2^{\text{app}}. \tag{34}
$$

From (31) it follows that $\Phi_{\mathcal{H}'}(v{:}(B, P')) \geq K_2^{\text{app}}$ and thus $u = \max(0, K_1^{\text{app}} + \Phi_{[y_f \mapsto v'], \mathcal{H}}(y_f{:}A; P))$. We apply Proposition 3.1 to (30), (32), (33), (34) and obtain $p \leq u$. If $u = 0$ then $p = 0 \leq \Phi_{\mathcal{V},\mathcal{H}}(x{:}A; Q)$. Otherwise we have $u = \Phi_{[y_f \mapsto v'], \mathcal{H}}(y_f{:}A; P) + K_1^{\text{app}}$. Furthermore it follows from (31) that $\Phi_{[y_f \mapsto v'], \mathcal{H}}(y_f{:}A; P) + K_1^{\text{app}} = \Phi_{\mathcal{V},\mathcal{H}}(x{:}A; Q)$ and therefore $p \leq \Phi_{\mathcal{V},\mathcal{H}}(x{:}A; Q)$.

For the second part for the statement observe that

$$
\begin{aligned}
p - p' &= r - r' + K_1^{\text{app}} + K_2^{\text{app}} \\
&\overset{(33)}{\leq} \Phi_{[y_f \mapsto v'], \mathcal{H}}(y_f{:}A; P) - \Phi_{\mathcal{H}'}(v{:}(B, P')) + K_1^{\text{app}} + K_2^{\text{app}} \\
&\overset{(31)}{=} \Phi_{\mathcal{V},\mathcal{H}}(x{:}A; Q) - \Phi_{\mathcal{H}'}(v{:}(B, Q')).
\end{aligned}
$$

(T:Nil).   If the type derivation ends with an application of T:Nil then we have $e = \text{nil}$, $\Gamma = \emptyset$, $B = L(A)$ for some A, and $q_0 = q_0' + K^{\text{nil}}$. The corresponding evaluation rule E:Nil has been applied to derive the evaluation judgment and hence $v = \text{Null}$.

If $K^{\text{nil}} \geq 0$ then $p = K^{\text{nil}}$ and $p' = 0$. Thus $p = K^{\text{nil}} \leq q_0 = \Phi_{\mathcal{V},\mathcal{H}}(\emptyset, Q)$. Furthermore it follows from the definition of Φ that $\Phi_{\mathcal{V},\mathcal{H}'}(\text{Null}{:}(L(A), Q')) = q_0'$. Thus $p - p' = K^{\text{nil}} = \Phi_{\mathcal{V},\mathcal{H}}(\emptyset; Q) - \Phi_{\mathcal{V},\mathcal{H}'}(\text{Null}{:}(L(A), Q'))$. If $K^{\text{nil}} < 0$ then $p = 0$ and $p' = -K^{\text{nil}}$. Then clearly $p \leq \Phi_{\mathcal{V},\mathcal{H}}(\emptyset, Q)$ and again $p - p' = K^{\text{nil}}$.

(T:CONS).   If the type derivation ends with an application of the rule T:CONS then
e has the form $\text{cons}(x_h, x_t)$ and it has been evaluated with the rule E:CONS. It follows
by definition that $\mathcal{V}, \mathcal{H} \vdash \text{cons}(x_h, x_t) \rightsquigarrow \ell, \mathcal{H}[\ell \mapsto v'] \mid K^{\text{cons}}, x_h, x_t \in \text{dom}(\mathcal{V}), v' = (\mathcal{V}(x_h), \mathcal{V}(x_t))$, and $\ell \notin \text{dom}(\mathcal{H})$. Thus

$$p = K^{\text{cons}} \text{ and } p' = 0 \qquad \text{or (if } K^{\text{cons}} < 0) \qquad p = 0 \text{ and } p' = -K^{\text{cons}}.$$

Furthermore $B = L(A)$ and the type judgment $\Sigma; x_h{:}A, x_t{:}L(A); Q \vdash \text{cons}(x_h, x_t) : (L(A), Q')$ has been derived by a single application of the rule T:CONS; thus

$$Q = \vartriangleleft_L(Q') + K^{\text{cons}}. \tag{35}$$

If $p = 0$ then $p \leq \Phi_{\mathcal{V}, \mathcal{H}}(\Gamma; Q)$ follows because potential is always nonnegative. Otherwise
we have $p = K^{\text{cons}} \leq \Phi_{\mathcal{V}, \mathcal{H}}(\Gamma; Q)$ from (35).
   From Lemma 6.4 it follows that $\Phi_{\mathcal{V}, \mathcal{H}}(x_h{:}A, x_t{:}L(A); \vartriangleleft_L(Q')) = \Phi_{\mathcal{V}, \mathcal{H}'}(\ell{:}(L(A), Q'))$ and
therefrom with (35) $\Phi_{\mathcal{V}, \mathcal{H}}(x_h{:}A, x_t{:}L(A); Q) - \Phi_{\mathcal{V}, \mathcal{H}[\ell \mapsto v']}(\ell{:}(L(A), Q')) = K^{\text{cons}} = p - p'$.

(T:MATL).   Assume that the type derivation of e ends with an application of the rule
T:MATL. Then e is a pattern match of the form match x with $\mid$ nil $\rightarrow e_1 \mid \text{cons}(x_h, x_t) \rightarrow$
e_2 whose evaluation ends with an application of the rule E:MATCONS or E:MATNIL.
Assume first that the derivation of the evaluation judgment ends with an application
of E:MATCONS.
   Then $\mathcal{V}(x) = \ell, \mathcal{H}(\ell) = (v_h, v_t)$, and $\mathcal{V}', \mathcal{H} \vdash e_2 \rightsquigarrow v, \mathcal{H}' \mid (r, r')$ for $\mathcal{V}' = \mathcal{V}[x_h \mapsto v_h, x_t \mapsto v_t]$ and some r, r' with

$$(p, p') = K_1^{\text{matC}} \cdot (r, r') \cdot K_2^{\text{matC}}. \tag{36}$$

Since the derivation of $\Sigma; \Gamma; Q \vdash e : (B, Q)$ ends with an application of T:MATL, we have
$\Gamma = \Gamma', x{:}L(A), \quad \Sigma; \Gamma', x_h{:}A, x_t{:}L(A); P \vdash e_2 : (B, P')$,

$$P + K_1^{\text{matC}} = \vartriangleleft_L(Q) \qquad \text{and} \qquad P' = Q' + K_2^{\text{matC}}. \tag{37}$$

It follows from Lemma 6.4 that

$$\Phi_{\mathcal{V}, \mathcal{H}}(\Gamma; Q) = \Phi_{\mathcal{V}', \mathcal{H}}(\Gamma', x_h{:}A, x_t{:}L(A); \vartriangleleft_L(Q)) . \tag{38}$$

Since $\mathcal{H} \vDash \mathcal{V}' : \Gamma', x_h{:}A, x_t{:}L(A)$ we can apply the induction hypothesis to $\mathcal{V}', \mathcal{H} \vdash e_2 \rightsquigarrow$
$v, \mathcal{H}' \mid (r, r')$ and obtain

$$r \ \leq \ \Phi_{\mathcal{V}', \mathcal{H}}(\Gamma', x_h{:}A, x_t{:}L(A); P) \tag{39}$$
$$r - r' \ \leq \ \Phi_{\mathcal{V}', \mathcal{H}}(\Gamma', x_h{:}A, x_t{:}L(A); P) - \Phi_{\mathcal{H}'}(v{:}(B, P')). \tag{40}$$

We define

$$(u, u') = K_1^{\text{matC}} \cdot (\Phi_{\mathcal{V}', \mathcal{H}}(\Gamma', x_h{:}A, x_t{:}L(A); P), \Phi_{\mathcal{H}'}(v{:}(B, P'))) \cdot K_2^{\text{matC}} . \tag{41}$$

Per definition and from (37) it follows that $\Phi_{\mathcal{H}'}(v{:}(B, P')) \geq K_2^{\text{matC}}$ and thus $u = \max(0, \Phi_{\mathcal{V}', \mathcal{H}}(\Gamma', x_h{:}A, x_t{:}L(A); P) + K_1^{\text{matC}})$. From Proposition 3.1 applied to (39), (40),
(41), and (36) we derive $u \geq p$. If $\Phi_{\mathcal{V}', \mathcal{H}}(\Gamma', x_h{:}A, x_t{:}L(A); P) + K_1^{\text{matC}} \leq 0$ then $u = p = 0$
and $\Phi_{\mathcal{V}, \mathcal{H}}(\Gamma; Q) \geq p$ trivially holds. If $\Phi_{\mathcal{V}', \mathcal{H}}(\Gamma', x_h{:}A, x_t{:}L(A); P) + K_1^{\text{matC}} > 0$ then it
follows from (37) and (38) that

$$\Phi_{\mathcal{V}, \mathcal{H}}(\Gamma; Q) = \Phi_{\mathcal{V}', \mathcal{H}}(\Gamma', x_h{:}A, x_t{:}L(A); P) + K_1^{\text{matC}} = u \geq p .$$

Finally, we apply Proposition 3.1 to (36) to see that

$$
\begin{aligned}
p - p' \;&=\; r - r' + K_1^{\mathrm{matC}} + K_2^{\mathrm{matC}} \\
&\overset{(40)}{\leq}\; \Phi_{\mathcal{V}',\mathcal{H}}(\Gamma', x_h{:}A, x_t{:}L(A); P) - \Phi_{\mathcal{H}'}(v{:}(B, P')) + K_1^{\mathrm{matC}} + K_2^{\mathrm{matC}} \\
&\overset{(37)}{=}\; \Phi_{\mathcal{V}',\mathcal{H}}(\Gamma', x_h{:}A, x_t{:}L(A); \lhd_L(Q))) - \Phi_{\mathcal{H}'}(v{:}(B, Q')) \\
&\overset{(38)}{=}\; \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q) - \Phi_{\mathcal{H}'}(v{:}(B, Q')).
\end{aligned}
$$

Assume now that the derivation of the evaluation judgment ends with an application of E:MatNil. Then $\mathcal{V}(x) = \mathrm{Null}$, and $\mathcal{V}, \mathcal{H} \vdash e_1 \rightsquigarrow v, \mathcal{H}' \mid (r, r')$ for some r, r' with

$$
(p, p') = K_1^{\mathrm{matN}} \cdot (r, r') \cdot K_2^{\mathrm{matN}}. \tag{42}
$$

Since the derivation of $\Sigma; \Gamma; q \vdash e : (B, Q')$ ends with an application of T:MatL, we have $\Sigma; \Gamma; R \vdash e_1 : (B, R')$,

$$
R + K_1^{\mathrm{matN}} = \pi_0^\Gamma(Q) \qquad \text{and} \qquad R' = Q' + K_2^{\mathrm{matN}}. \tag{43}
$$

From Proposition 6.3 it follows that

$$
\Phi_{\mathcal{V},\mathcal{H}}(\Gamma; R) + K_1^{\mathrm{matN}} \leq \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q). \tag{44}
$$

Because $\mathcal{H} \vDash \mathcal{V} : \Gamma$ we can apply the induction hypothesis to $\mathcal{V}, \mathcal{H} \vdash e_1 \rightsquigarrow v, \mathcal{H}' \mid (r, r')$ and obtain

$$
r \;\leq\; \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; R) \tag{45}
$$
$$
r - r' \;\leq\; \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; R) - \Phi_{\mathcal{H}'}(v{:}(B, R')). \tag{46}
$$

Now let

$$
(u, u') = K_1^{\mathrm{matN}} \cdot (\Phi_{\mathcal{V},\mathcal{H}}(\Gamma; R), \Phi_{\mathcal{H}'}(v{:}(B, R'))) \cdot K_2^{\mathrm{matN}}. \tag{47}
$$

Per definition and from (43) it follows that $u = \max(0, \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; R) + K_1^{\mathrm{matN}})$. From Proposition 3.1 applied to (45), (46), (47), and (42) we derive $u \geq p$. If $\Phi_{\mathcal{V},\mathcal{H}}(\Gamma; R) + K_1^{\mathrm{matN}} \leq 0$ then $u = p = 0$ and $\Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q) \geq p$ trivially holds. If $\Phi_{\mathcal{V},\mathcal{H}}(\Gamma; R) + K_1^{\mathrm{matN}} > 0$ then it follows from (44) that

$$
\Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q) \geq \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; R) + K_1^{\mathrm{matN}} = u \geq p.
$$

Finally, we apply Proposition 3.1 to (42) to see that

$$
\begin{aligned}
p - p' \;&=\; r - r' + K_1^{\mathrm{matN}} + K_2^{\mathrm{matN}} \\
&\overset{(46)}{\leq}\; \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; R) - \Phi_{\mathcal{H}'}(v{:}(B, R')) + K_1^{\mathrm{matN}} + K_2^{\mathrm{matN}} \\
&\overset{(44)}{\leq}\; \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q) - \big(\Phi_{\mathcal{H}'}(v{:}(B, R)) - K_2^{\mathrm{matN}}\big) \\
&\overset{(43)}{=}\; \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q) - \Phi_{\mathcal{H}'}(v{:}(B, Q')).
\end{aligned}
$$

(T:Leaf). This case is nearly identical to the case (T:Nil).

(T:Node). If the type derivation ends with an application of the rule T:Node then e has the form $\mathrm{node}(x_0, x_1, x_2)$ and it has been evaluated with the rule E:Node. It follows by definition that $\mathcal{V}, \mathcal{H} \vdash \mathrm{node}(x_0, x_1, x_2) \rightsquigarrow \ell, \mathcal{H}[\ell \mapsto v'] \mid K^{\mathrm{node}}$, $v' = (\mathcal{V}(x_0), \mathcal{V}(x_1), \mathcal{V}(x_2))$, and $\ell \notin \mathrm{dom}(\mathcal{H})$. Thus

$$
p = K^{\mathrm{node}} \text{ and } p' = 0 \qquad \text{or (if } K^{\mathrm{node}} < 0\text{)} \qquad p = 0 \text{ and } p' = -K^{\mathrm{node}}.
$$

Furthermore we have $B = T(A)$ and the type judgment $x_0{:}A, x_1{:}T(A), x_2{:}T(A); Q \vdash$ node$(x_0, x_1, x_2) : (T(A), Q')$ has been derived by a single application of the rule T:NODE; thus

$$Q = \lhd_T(Q') + K^{\text{node}}. \tag{48}$$

If $p = 0$ then clearly $p \leq \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q)$. Otherwise we have $p = K^{\text{node}} \leq \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q)$ from (48).

By applying Lemma 6.5 we derive that $\Phi_{\mathcal{V},\mathcal{H}}(x_1{:}A, x_2{:}T(A), x_3{:}T(A); \lhd_T(Q')) = \Phi_{\mathcal{V},\mathcal{H}'}(\ell{:}(T(A), Q'))$ and therefrom with (48)

$$\Phi_{\mathcal{V},\mathcal{H}}(x_1{:}A, x_2{:}T(A), x_3{:}T(A); \lhd_T(Q')) - \Phi_{\mathcal{V},\mathcal{H}[\ell \mapsto v']}(\ell{:}(T(A), Q')) = K^{\text{node}} = p - p' .$$

(T:MATT).   Assume that the type derivation of e ends with an application of the rule T:MATT. Then e is a pattern match of the form match x with $\mid$ leaf $\rightarrow$ $e_1 \mid$ node$(x_0, x_1, x_2) \rightarrow e_2$ whose evaluation ends with an application of the rule E:MATNODE or E:MATLEAF. The case E:MATLEAF is similar to the case E:MATNIL. So assume that the derivation of the evaluation judgment ends with an application of E:MATNODE. Then $\mathcal{V}(x) = \ell$, $\mathcal{H}(\ell) = (v_0, v_1, v_2)$, and $\mathcal{V}', \mathcal{H} \vdash e_2 \rightsquigarrow v, \mathcal{H}' \mid (r, r')$ for $\mathcal{V}' = \mathcal{V}[x_0 \mapsto v_0, x_1 \mapsto v_1, x_2 \mapsto v_2]$ and some r, r' with

$$(p, p') = K_1^{\text{matTL}} \cdot (r, r') \cdot K_2^{\text{matTL}}. \tag{49}$$

Since the derivation of $\Sigma; \Gamma; Q \vdash e : (B, Q)$ ends with an application of T:MATT, we have $\Gamma = \Gamma', x{:}T(A)$, $\Sigma; \Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A); P \vdash e_2 : (B, P')$,

$$P + K_1^{\text{matTN}} = \lhd_L(Q) \qquad \text{and} \qquad P' = Q' + K_2^{\text{matTN}} . \tag{50}$$

It follows from Lemma 6.5 that

$$\Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q) = \Phi_{\mathcal{V}',\mathcal{H}}(\Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A); \lhd_T(Q)) . \tag{51}$$

Since $\mathcal{H} \models \mathcal{V}' : \Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A)$ we can apply the induction hypothesis to $\mathcal{V}', \mathcal{H} \vdash e_2 \rightsquigarrow v, \mathcal{H}' \mid (r, r')$ and obtain

$$r \leq \Phi_{\mathcal{V}',\mathcal{H}}(\Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A); P) \tag{52}$$

$$r - r' \leq \Phi_{\mathcal{V}',\mathcal{H}}(\Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A); P) - \Phi_{\mathcal{H}'}(v{:}(B, P')). \tag{53}$$

We define

$$(u, u') = K_1^{\text{matTN}} \cdot (\Phi_{\mathcal{V}',\mathcal{H}}(\Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A); P), \Phi_{\mathcal{H}'}(v{:}(B, P'))) \cdot K_2^{\text{matTN}} . \tag{54}$$

Per definition and from (50) it follows that $\Phi_{\mathcal{H}'}(v{:}(B, P')) \geq K_2^{\text{matTN}}$ and thus $u = \max(0, \Phi_{\mathcal{V}',\mathcal{H}}(\Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A); P) + K_1^{\text{matTN}})$.

From Proposition 3.1 applied to (52), (53), (54), and (49) we derive $u \geq p$. If $\Phi_{\mathcal{V}',\mathcal{H}}(\Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A); P) + K_1^{\text{matTN}} \leq 0$ then $u = p = 0$ and $\Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q) \geq p$ trivially holds. If $\Phi_{\mathcal{V}',\mathcal{H}}(\Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A); P) + K_1^{\text{matTN}} > 0$ then it follows from (50) and (51) that

$$\Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q) = \Phi_{\mathcal{V}',\mathcal{H}}(\Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A); P) + K_1^{\text{matTN}} = u \geq p .$$

Finally, we apply Proposition 3.1 to (49) to see that

$$\begin{aligned}
p - p' &= r - r' + K_1^{\text{matTN}} + K_2^{\text{matTN}} \\
&\overset{(53)}{\leq} \Phi_{\mathcal{V}',\mathcal{H}}(\Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A); P) - \Phi_{\mathcal{H}'}(v{:}(B, P')) + K_1^{\text{matTN}} + K_2^{\text{matTN}} \\
&\overset{(50)}{=} \Phi_{\mathcal{V}',\mathcal{H}}(\Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A); \lhd_L(Q))) - \Phi_{\mathcal{H}'}(v{:}(B, Q')) \\
&\overset{(51)}{=} \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q) - \Phi_{\mathcal{H}'}(v{:}(B, Q')).
\end{aligned}$$

(T:Pair).   This case is similar to the case in which the type derivation ends with an application of the rule T:Cons.

(T:MatP).   This case is proved like the case T:MatL.

(T:Cond).   This case is similar to (but also simpler than) the case T:MatL.

Proof of Part 2

The proof of part 2 is similar but simpler than the proof of part 1. However, it uses part 1 in the case of the rule P:Let2. Like in the proof of part 1, we prove $p \leq \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q)$ by induction on the derivations of $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow | p$ and $\Sigma; \Gamma; Q \vdash e : (B, Q')$, where the induction on the partial evaluation judgment takes priority.

We only present a few cases to show that the proof is similar to the poof of part 1.

(T:Var).   Assume that e is a variable x and the type judgment $\Sigma; Q \vdash x : (B, Q')$ has been derived by a single application of the rule T:Var. Thus we have $\Gamma = x{:}B$,

$$\Phi_{\mathcal{V},\mathcal{H}}(x{:}B; Q) - \Phi_{\mathcal{V},\mathcal{H}'}(x{:}(B, Q')) = K^{\mathrm{var}}$$

and in particular $\Phi_{\mathcal{V},\mathcal{H}}(x{:}B; Q) \geq K^{\mathrm{var}}$.

Furthermore e has been evaluated with a single application of the rule P:Var and it follows by definition that $p = \max(K^{\mathrm{var}}, 0)$. (Remember that $\mathcal{V}, \mathcal{H} \vdash x \rightsquigarrow | K^{\mathrm{var}}$ is an abbreviation for $\mathcal{V}, \mathcal{H} \vdash x \rightsquigarrow | \max(K^{\mathrm{var}}, 0)$ in P:Var.)

Assume first that $K^{\mathrm{var}} \geq 0$. Then we have $p = K^{\mathrm{var}} \leq \Phi_{\mathcal{V},\mathcal{H}}(x{:}B; Q)$. Assume now that $K^{\mathrm{var}} < 0$. Then it follows by definition that $p = 0$ and $p \leq \Phi_{\mathcal{V},\mathcal{H}}(x{:}B; Q)$ trivially holds.

(T:MatL).   Assume that the type derivation of e ends with an application of the rule T:MatL. Then e is a pattern match of the form match x with $| \, \mathrm{nil} \rightarrow e_1 \, | \, \mathrm{cons}(x_h, x_t) \rightarrow e_2$ whose evaluation ends with an application of the rule P:MatCons or P:MatNil. Assume first that the derivation of the evaluation judgment ends with an application of P:MatCons.

Then $\mathcal{V}(x) = l$, $\mathcal{H}(l) = (v_h, v_t)$, and $\mathcal{V}', \mathcal{H} \vdash e_2 \rightsquigarrow | r$ for $\mathcal{V}' = \mathcal{V}[x_h \mapsto v_h, x_t \mapsto v_t]$ and some r with

$$p = \max\left(K_1^{\mathrm{matC}} + r, 0\right). \tag{55}$$

Since the derivation of $\Sigma; \Gamma; Q \vdash e : (B, Q)$ ends with an application of T:MatL, we have $\Gamma = \Gamma', x{:}L(A)$, $\Sigma; \Gamma', x_h{:}A, x_t{:}L(A); P \vdash e_2 : (B, P')$,

$$P + K_1^{\mathrm{matC}} = \lhd_L(Q). \tag{56}$$

It follows from Lemma 6.4 that

$$\Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q) = \Phi_{\mathcal{V}',\mathcal{H}}(\Gamma', x_h{:}A, x_t{:}L(A); \lhd_L(Q)) . \tag{57}$$

Since $\mathcal{H} \vDash \mathcal{V}' : \Gamma', x_h{:}A, x_t{:}L^t(A)$ we can apply the induction hypothesis to $\mathcal{V}', \mathcal{H} \vdash e_2 \rightsquigarrow | r$ and obtain

$$r \leq \Phi_{\mathcal{V}',\mathcal{H}}(\Gamma', x_h{:}A, x_t{:}L(A); P). \tag{58}$$

If $p = 0$ then the claim follows immediately. Thus assume that $p = K_1^{\mathrm{matC}} + r$. Then it follows that

$$
\begin{aligned}
p = K_1^{\mathrm{matC}} + r &\overset{(58)}{\leq} K_1^{\mathrm{matC}} + \Phi_{\mathcal{V}',\mathcal{H}}(\Gamma', x_h{:}A, x_t{:}L(A); P) \\
&\overset{(56)}{\leq} K_1^{\mathrm{matC}} + \Phi_{\mathcal{V}',\mathcal{H}}(\Gamma', x_h{:}A, x_t{:}L(A); \lhd_L(Q)) \\
&\overset{(57)}{\leq} \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q).
\end{aligned}
$$

Assume now that the derivation of the evaluation judgment ends with an application of P:MatNil. Then $\mathcal{V}, \mathcal{H} \vdash e_1 \rightsquigarrow |\, r$ for a r with

$$p = \max \left(K_1^{\mathrm{matN}} + r, 0 \right).$$

Since the derivation of $\Sigma; \Gamma; q \vdash e : (B, Q')$ ends with an application of T:MatL, we have $\Sigma; \Gamma; R \vdash e_1 : (B, R')$,

$$R + K_1^{\mathrm{matN}} = \pi_0^{\Gamma}(Q).$$

From Proposition 6.3 it follows that

$$\Phi_{\mathcal{V}, \mathcal{H}}(\Gamma; R) + K_1^{\mathrm{matN}} \leq \Phi_{\mathcal{V}, \mathcal{H}}(\Gamma; Q). \tag{59}$$

Since $\mathcal{H} \vDash \mathcal{V} : \Gamma'$ we can apply the induction hypothesis to $\mathcal{V}, \mathcal{H} \vdash e_1 \rightsquigarrow |\, r$ and obtain

$$r \ \leq \ \Phi_{\mathcal{V}, \mathcal{H}}(\Gamma; R). \tag{60}$$

If $p = 0$ then the claim follows immediately. So assume that $p = K_1^{\mathrm{matN}} + r$. Then it follows from (59) and (60) that

$$p = K_1^{\mathrm{matN}} + r \leq K_1^{\mathrm{matN}} + \Phi_{\mathcal{V}, \mathcal{H}}(\Gamma; R) \leq \Phi_{\mathcal{V}, \mathcal{H}}(\Gamma; Q) \,.$$

(T:Let). If the type derivation ends with an application of T:Let then e is a let expression of the from let $x = e_1$ in e_2 that has eventually been evaluated with the rule P:Let1 or with the rule P:Let2.

The case P:Let1 is similar to the case P:MatCons. So assume that the evaluation judgment ends with an application of the rule P:Let2. Then it follows that $\mathcal{V}, \mathcal{H} \vdash e_1 \rightsquigarrow v_1, \mathcal{H}_1 \,|\, (r, r')$ and $\mathcal{V}', \mathcal{H}_1 \vdash e_2 \rightsquigarrow |\, t$ for $\mathcal{V}' = \mathcal{V}[x \mapsto v_1]$ and r, r', t with

$$(p, p') = K_1^{\mathrm{let}} \cdot (r, r') \cdot K_2^{\mathrm{let}} \cdot (t, 0). \tag{61}$$

The derivation of the type judgment for e ends with an application of L:Let. Hence $\Gamma = \Gamma_1, \Gamma_2, \quad \Sigma; \Gamma_1; P \vdash e_1 : (A, P'), \ \Sigma; \Gamma_2, x{:}A; R \vdash e_2 : (B, R')$ and

$$P + K_1^{\mathrm{let}} \ = \ \pi_0^{\Gamma_1}(Q) \tag{62}$$

$$P' \ = \ \pi_0^{x:A}(R) + K_2^{\mathrm{let}}. \tag{63}$$

Furthermore we have for every $\vec{0} \neq j \in I(\Gamma_2)$: $\quad \Gamma_1; P_j \vdash^{\underline{\mathrm{cf}}} e_1 : (A, P'_j),$

$$P_j = \pi_j^{\Gamma_1}(Q) \tag{64}$$

$$P'_j = \pi_j^{x:A}(R). \tag{65}$$

Since $\mathcal{H} \vDash \mathcal{V} : \Gamma$ we have also $\mathcal{H} \vDash \mathcal{V} : \Gamma_1$ and can thus apply part 1 of the soundness theorem to the evaluation judgment of e_1 and derive

$$r \ \leq \ \Phi_{\mathcal{V}, \mathcal{H}}(\Gamma_1; P) \tag{66}$$

$$r - r' \ \leq \ \Phi_{\mathcal{V}, \mathcal{H}}(\Gamma_1; P) - \Phi_{\mathcal{H}_1}(v_1{:}(A, P')) \tag{67}$$

Theorem 3.4 it follows that $\mathcal{H}_2 \vDash \mathcal{V}' : \Gamma_2, x{:}A$. Thus we can apply the induction hypothesis of part 2 to the partial evaluation judgment for e_2 and obtain

$$t \ \leq \ \Phi_{\mathcal{V}', \mathcal{H}_1}(\Gamma_2, x{:}A; R). \tag{68}$$

Furthermore we apply part 1 of the theorem to the evaluation judgment for e_1 with the cost-free metric. Then we have $r = r' = 0$ and therefore for every $\vec{0} \neq j \in I(\Gamma_2)$

$$\Phi_{\mathcal{V}, \mathcal{H}}(\Gamma_1; P_j) \geq \Phi_{\mathcal{H}_1}(v_1{:}(A, P'_j)) \,. \tag{69}$$

Let $\Gamma_1 = x_1, \ldots, x_n$, $\Gamma_2 = y_1, \ldots, y_m$, $\mathcal{H} \vDash \mathcal{V}(x_j) \mapsto a_{x_j} : \Gamma(x_j)$, and $\mathcal{H} \vDash \mathcal{V}(y_j) \mapsto b_{y_j} : \Gamma(y_j)$. Define

$$\phi_P = \Phi_{\mathcal{V},\mathcal{H}}(\Gamma_1; P) + \sum_{\vec{0} \neq \vec{j} \in I_k(\Gamma_2)} \Phi_{\mathcal{V},\mathcal{H}}(\Gamma_1; P_{\vec{j}}) \cdot \prod_{k=1}^{m} p_{j_k}(b_{x_k})$$

$$\phi_{P'} = \Phi_{\mathcal{H}_1}(v_1{:}(A, P')) + \sum_{\vec{0} \neq \vec{j} \in I_k(\Gamma_2)} \Phi_{\mathcal{H}_1}(v_1{:}(A, P'_{\vec{j}})) \cdot \prod_{k=1}^{m} p_{j_k}(b_{x_k}).$$

We argue that

$$\Phi_{\mathcal{V},\mathcal{H}}(\Gamma_1, \Gamma_2; Q) \overset{\text{Prop. 6.3}}{=} \sum_{\vec{j} \in I_k(\Gamma_2)} \Phi_{\mathcal{V},\mathcal{H}}\big(\Gamma_1; \pi_{\vec{j}}^{\Gamma_1}(Q)\big) \prod_{k=1}^{m} p_{j_k}(b_{x_k})$$

$$\overset{(62,64)}{=} \Phi_{\mathcal{V},\mathcal{H}}(\Gamma_1; P) + K_1^{\text{let}} + \sum_{\vec{0} \neq \vec{j} \in I_k(\Gamma_2)} \Phi_{\mathcal{V},\mathcal{H}}(\Gamma_1; P_{\vec{j}}) \cdot \prod_{k=1}^{m} p_{j_k}(b_{x_k})$$

$$= \phi_P + K_1^{\text{let}}. \tag{70}$$

Similarly, we use Proposition 6.3, (63), and (65) to see that

$$\phi_{P'} = \Phi_{\mathcal{V}',\mathcal{H}_1}(\Gamma_2, x{:}A; R) + K_2^{\text{let}}. \tag{71}$$

Additionally we have

$$r - r' \overset{(67)}{\leq} \Phi_{\mathcal{V},\mathcal{H}}(\Gamma_1; P) - \Phi_{\mathcal{H}_1}(v_1{:}(A, P'))$$

$$\overset{(69)}{\leq} \Phi_{\mathcal{V},\mathcal{H}}(\Gamma_1; P) - \Phi_{\mathcal{H}_1}(v_1{:}(A, P')) + \sum_{\vec{0} \neq \vec{j} \in I_k(\Gamma_2)} \Phi_{\mathcal{V},\mathcal{H}}(\Gamma_1; P_{\vec{j}}) \cdot \prod_{k=1}^{m} p_{j_k}(b_{x_k})$$

$$- \sum_{\vec{0} \neq \vec{j} \in I_k(\Gamma_2)} \Phi_{\mathcal{H}_1}(v_1{:}(A, P'_{\vec{j}})) \cdot \prod_{k=1}^{m} p_{j_k}(b_{x_k})$$

$$= \phi_P - \phi_{P'}. \tag{72}$$

Now let

$$(u, u') = K_1^{\text{let}} \cdot (\phi_P, \phi_{P'}) \cdot K_2^{\text{let}} \cdot (\Phi_{\mathcal{V}',\mathcal{H}_1}(\Gamma_2, x{:}A; R), 0).$$

Then it follows that

$$(u, u') \overset{(71)}{=} K_1^{\text{let}} \cdot \big(\phi_P, \phi_{P'} - K_2^{\text{let}}\big) \cdot (\Phi_{\mathcal{V}',\mathcal{H}_1}(\Gamma_2, x{:}A; R), 0)$$

$$\overset{(71)}{=} K_1^{\text{let}} \cdot (\phi_P, 0).$$

Now we conclude that $u \leq \max(0, \phi_P + K_1^{\text{let}}) \overset{(70)}{\leq} \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q)$. Finally, it follows with Proposition 3.1 applied to (66), (72), (68), and (61) that $u \geq p$.

ACKNOWLEDGMENTS

REFERENCES

ALBERT, E., ALONSO, D., ARENAS, P., GENAIM, S., AND PUEBLA, G. 2009a. Asymptotic resource usage bounds. In *Proceedings of the 7th Asian Symposium Programming Languages and Systems (APLAS'09)*. Springer, 294–310.

ALBERT, E., ARENAS, P., GENAIM, S., GÓMEZ-ZAMALLOA, M., PUEBLA, G., RAMÍREZ, D., ROMÁN, G., AND ZANARDINI, D. 2009b. Termination and cost analysis with COSTA and its user interfaces. *Electr. Notes Theor. Comput. Sci. 258,* 1, 109–121.

ALBERT, E., ARENAS, P., GENAIM, S., AND PUEBLA, G. 2008. Automatic inference of upper bounds for recurrence relations in cost analysis. In *Proceedings of the 15th International Symposium on Static Analysis (SAS'08)*. Springer, 221–237.

ALBERT, E., ARENAS, P., GENAIM, S., AND PUEBLA, G. 2011a. Closed-form upper bounds in static cost analysis. *J. Autom. Reason. 46,* 2, 161–203.

ALBERT, E., ARENAS, P., GENAIM, S., PUEBLA, G., AND ZANARDINI, D. 2007. Cost analysis of java bytecode. In *Proceedings of the 16th European Symposium on Programming Languages and Systems (ESOP'07)*. Springer, 157–172.

ALBERT, E., GENAIM, S., AND GÓMEZ-ZAMALLOA, M. 2010. Parametric inference of memory requirements for garbage collected languages. In *Proceedings of the 9th International Symposium on Memory Management (ISMM'10)*. ACM, New York, 121–130.

ALBERT, E., GENAIM, S., AND MASUD, A. N. 2011b. More precise yet widely applicable cost analysis. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*. Springer, 38–53.

ALTHAUS, E., ALTMEYER, S., AND NAUJOKS, R. 2011. Precise and efficient parametric path analysis. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'11)*. ACM, New York, 141–150.

ALTMEYER, S., HUMBERT, C., LISPER, B., AND WILHELM, R. 2008. Parametric timing analysis for complex architectures. In *Proceedings of the 4th IEEE Internationl Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'08)*. IEEE, 367–376.

ATKEY, R. 2010. Amortised resource analysis with separation logic. In *Proceedings of the 19th European Symposium on Programming Languages and Systems (ESOP'10)*. Springer, 85–103.

BENZINGER, R. 2001. Automated complexity analysis of nuprl extracted programs. *J. Funct. Program. 11,* 1, 3–31.

BENZINGER, R. 2004. Automated higher-order complexity analysis. *Theor. Comput. Sci. 318,* 1-2, 79–103.

BERINGER, L., HOFMANN, M., MOMIGLIANO, A., AND SHKARAVSKA, O. 2004. Automatic certification of heap consumption. In *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'04)*. Springer, 347–362.

BRABERMAN, V. A., FERNÁNDEZ, F. J., GARBERVETSKY, D., AND YOVINE, S. 2008. Parametric prediction of heap memory requirements. In *Proceedings of the 7th International Symposium on Memory Management (ISMM'08)*. ACM, New York, 141–150.

BYGDE, S., ERMEDAHL, A., AND LISPER, B. 2009. An efficient algorithm for parametric WCET calculation. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'09)*. IEEE, Los Alamitos, CA, 13–21.

CAMPBELL, B. 2009. Amortised memory analysis using the depth of data structures. In *Proceedings of the 18th European Symposium on Programming Languages and Systems (ESOP'09)*. Springer, 190–204.

CHIN, W.-N. AND KHOO, S.-C. 2001. Calculating sized types. *High.-Ord. Symb. Comp. 14,* 2-3, 261–300.

CHIN, W.-N., NGUYEN, H. H., POPEEA, C., AND QIN, S. 2008. Analysing memory resource bounds for low-level programs. In *Proceedings of the 7th International Symposium on Memory Management (ISMM'08)*. ACM, New York, 151–160.

CLAUSS, P., FERNÁNDEZ, F. J., GARBERVETSKY, D., AND VERDOOLAEGE, S. 2009. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. *IEEE Trans. VLSI Syst. 17,* 8, 983–996.

COOK, B., GUPTA, A., MAGILL, S., RYBALCHENKO, A., SIMSA, J., SINGH, S., AND VAFEIADIS, V. 2009. Finding heap-bounds for hardware synthesis. In *Proceedings of the 9th International Conference on Formal Methods in Computer-Aided Design (FMCAD'09)*. IEEE, Los Alamitos, CA, 205–212.

COUSOT, P. AND COUSOT, R. 1992. Inductive definitions, semantics and abstract interpretations. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*. ACM, New York, 83–94.

CRARY, K. AND WEIRICH, S. 2000. Resource bound certification. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages (POPL'00)*. ACM, New York, 184–198.

DANIELSSON, N. A. 2008. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proceedings of the 35th ACM Symposium on Principles of Programming Languages (POPL'08)*. ACM, New York, 133–144.

DEBRAY, S. K. AND LIN, N.-W. 1993. Cost analysis of logic programs. *ACM Trans. Program. Lang. Syst. 15*, 5, 826–875.

FLAJOLET, P., SALVY, B., AND ZIMMERMANN, P. 1991. Automatic average-case analysis of algorithms. *Theoret. Comput. Sci. 79*, 1, 37–109.

GROBAUER, B. 2001. Cost recurrences for DML programs. In *Proceedings of the 6th International Conference on Functional Programming (ICFP'01)*. ACM, New York, 253–264.

GULAVANI, B. S. AND GULWANI, S. 2008. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *Proceedings of the 20th International Conference Computer Aided Verification (CAV'08)*. Springer, 370–384.

GULWANI, S., JAIN, S., AND KOSKINEN, E. 2009a. Control-flow refinement and progress invariants for bound analysis. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'09)*. ACM, New York, 375–385.

GULWANI, S., MEHRA, K. K., AND CHILIMBI, T. M. 2009b. SPEED: Precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL'09)*. ACM, New York, 127–139.

GULWANI, S. AND ZULEGER, F. 2010. The reachability-bound problem. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'10)*. ACM, New York, 292–304.

HAMMOND, K. AND MICHAELSON, G. 2003. Hume: A domain-specific language for real-time embedded systems. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE'03)*. Springer, 37–56.

HICKEY, T. J. AND COHEN, J. 1988. Automating program analysis. *J. ACM 35*, 1, 185–220.

HOFFMANN, J. 2011. Types with potential: Polynomial resource bounds via automatic amortized analysis. Ph.D. thesis, Ludwig-Maximilians-Universiät, München, Germany.

HOFFMANN, J., AEHLIG, K., AND HOFMANN, M. 2011. Multivariate amortized resource analysis. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL'11)*. ACM, New York, 357–370.

HOFFMANN, J. AND HOFMANN, M. 2010a. Amortized resource analysis with polymorphic recursion and partial big-step operational semantics. In *Proceedings of the 8th Asian Symposium on Programming Languages and Systems (APLAS'10)*. Springer, 172–187.

HOFFMANN, J. AND HOFMANN, M. 2010b. Amortized resource analysis with polynomial potential. In *Proceedings of the 19th European Symposium on Programming Languages and Systems (ESOP'10)*. ACM, New York, 287–306.

HOFMANN, M. 2000. A type system for bounded space and functional in-place update–extended abstract. In *Proceedings of the 9th European Symposium on Programming Languages and Systems (ESOP'00)*. Springer, 165–179.

HOFMANN, M. AND JOST, S. 2003. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM Symposium on Principles of Programming Languages (POPL'03)*. ACM, New York, 185–197.

HOFMANN, M. AND JOST, S. 2006. Type-based amortised heap-space analysis. In *Proceedings of the 15th European Symposium on Programming Languages and Systems (ESOP'06)*. Springer, 22–37.

HOFMANN, M. AND RODRIGUEZ, D. 2009. Efficient type-checking for amortised heap-space analysis. In *Proceedings of the 18th Conference on Computer Science Logic (CSL'09)*. Springer, 317–331.

HUGHES, J. AND PARETO, L. 1999. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *Proceedings of the 4th International Conference on Functional Programming (ICFP'99)*. ACM, New York, 70–81.

HUGHES, J., PARETO, L., AND SABRY, A. 1996. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages (POPL'96)*. ACM, New York, 410–423.

JOST, S., HAMMOND, K., LOIDL, H.-W., AND HOFMANN, M. 2010. Static determination of quantitative resource usage for higher-order programs. In *Proceedings of the 37th ACM Symposium on Principles of Programming Languages (POPL'10)*. ACM, New York, 223–236.

JOST, S., LOIDL, H.-W., HAMMOND, K., SCAIFE, N., AND HOFMANN, M. 2009. Carbon credits for resource-bounded computations using amortised analysis. In *Proceedings of the 16th International Symposium on Formal Methods (FM'09)*. Springer, 354–369.

LEROY, X. 2006. Coinductive big-step operational semantics. In *Proceedings of the 15th European Symposium on Programming Languages and Systems (ESOP'06)*. Springer, 54–68.

LISPER, B. 2003. Fully automatic, parametric worst-case execution time analysis. In *Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis (WCET'03)*. 99–102.

MÉTAYER, D. L. 1988. ACE: An automatic complexity evaluator. *ACM Trans. Program. Lang. Syst. 10,* 2, 248–266.

PIERCE, B. C. 2004. *Advanced Topics in Types and Programming Languages*. The MIT Press, Cambridge, MA.

RAMSHAW, L. H. 1979. Formalizing the analysis of algorithms. Ph.D. thesis, Stanford University, Stanford, CA, USA.

ROSENDAHL, M. 1989. Automatic complexity analysis. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*. ACM, New York, 144–156.

SHKARAVSKA, O., VAN KESTEREN, R., AND VAN EEKELEN, M. C. 2007. Polynomial size analysis of first-order functions. In *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications (TLCA'07)*. Springer, 351–365.

STANLEY, R. P. 2001. *Enumerative Combinatorics* Vol. 2. Cambridge University Press, New York, USA.

TAHA, W., ELLNER, S., AND XI, H. 2003. Generating heap-bounded programs in a functional setting. In *Proceedings of the 3rd International Conference on Embedded Software (EMSOFT'03)*. Springer, 340–355.

TARJAN, R. E. 1985. Amortized computational complexity. *SIAM J. Algebr. Discr. Methods 6,* 2, 306–318.

VASCONCELOS, P. 2008. Space cost analysis using sized types. Ph.D. thesis, School of Computer Science, University of St Andrews, St Andrews, UK.

WEGBREIT, B. 1975. Mechanical program analysis. *Commun. ACM 18,* 9, 528–539.

WILHELM, R., ENGBLOM, J., ERMEDAHL, A., HOLSTI, N., THESING, S., WHALLEY, D. B., BERNAT, G., FERDINAND, C., HECKMANN, R., MITRA, T., MUELLER, F., PUAUT, I., PUSCHNER, P. P., STASCHULAT, J., AND STENSTRÖM, P. 2008. The worst-case execution-time problem – Overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst. 7,* 3, 36:1–36:53.