15-819: Foundations of Quantitative Program Analysis

Lecture 16: AARA with Univariate Polynomial Potential

Jan Hoffmann

October 31, 2019

1 Introduction

Linear automatic amortized analysis works well because of three reasons: it is compositional, it computes precise bounds, and the type inference is based on efficient linear constraint solving. The main shortcoming of the analysis is its limitation to linear bounds.

In this lecture, we see how to overcome this shortcoming while preserving the beneficial features of the analysis system. We study an automatic amortized resource analysis that computes *univariate polynomial bounds* [HH10b, Hof11]. For simplicity, we return to our simple language with lists. The type system develop is mostly identical to the one for linear AARA. The only changes are in the definition of lists types and the type rules for introduction and elimination of lists. In particular, the structure of soundness proof does not change. Moreover, we are still able to reduce type inference to linear optimization.

2 Syntax and Dynamic Semantics

Like for linear AARA, we only introduce polynomial potential for lists. However, it can be extended to inductive types. Moreover, it is compatible with language features like recursive functions, sums, and products. The typing rules for these constructs do not have to be altered.

$$
\begin{array}{rcll}
e & ::= & \dots & \\
 & & \mathsf{nil} & [\,] \\
 & & \mathsf{cons}(x_1; x_2) & x_1 :: x_2 \\
 & & \mathrm{mat}_{\mathrm{L}}\{e_0; x_1, x_2.e_1\}(x) & \mathsf{case}\ x\ \{\mathsf{nil} \hookrightarrow e_0 \mid \mathsf{cons}\,(x_1, x_2) \hookrightarrow e_1\} \\
 & & \dots &
\end{array}
$$

Like before, we have list values.

$$
\begin{array}{rcl}
v & ::= & \dots \\
 & & [\,] \\
 & & v_1 :: v_2 \\
 & & \dots
\end{array}
$$

Similarly, the cost semantics defines the judgment $V \vdash e \Downarrow v \mid (q, q')$ using the same rules as in linear AARA.

3 Resource Annotations

In linear AARA, we annotated list types with a single non-negative rational number q that defines the potential function $q \cdot n$, where n is the length of the list. In this lecture, we use potential functions that are non-negative linear combinations of binomial coefficients $\binom{n}{k}$, where k is a natural number and n is length of the list.

In (univariate) polynomial AARA, a *resource annotation* for lists is a vector $\vec{q} = (q_1, \ldots, q_k) \in (\mathbb{Q}_{\geq 0})^k$ of non-negative rational numbers.

$$\begin{array}{rcl} \tau & ::= & \ldots \\ & & L^{\vec{q}}(\tau) \\ & & \ldots \end{array}$$

For two resource annotations $\vec{p} = (p_1, \ldots, p_k)$ and $\vec{q} = (q_1, \ldots, q_\ell)$ I write $\vec{p} \leq \vec{q}$ if $k \leq \ell$ and $p_i \leq q_i$ for all $1 \leq i \leq k$. If $\ell \geq k$ then we define $\vec{p} + \vec{q} = (p_1 + q_1, \ldots, p_k + q_k, q_{k+1}, \ldots, q_\ell)$.

One intuition for the resource annotations is as follows: The annotation $\vec{q}$ assigns the potential q_1 to every element of the data structures, the potential q_2 to every element of every proper suffix (sublist or subtree, respectively) of the data structure, q_3 to the elements of the suffixes of the suffixes, etc.

The Potential of Lists Let us now consider the construction or destruction of non-empty lists. For linear potential annotations we can simply assign potential to the tail by using the same annotations as on the original list. This would however lead to a substantial loss of potential in the polynomial case. For this reason, we use an additive shift operation to assign potential to sublists.

Let $\vec{q} = (q_1, \ldots, q_k)$ be a resource annotation. The *additive shift* of $\vec{q}$ is

$$\lhd(\vec{q}) = (q_1 + q_2, q_2 + q_3, \ldots, q_{k-1} + q_k, q_k) \,.$$

The definition of potential $\Phi(v : \tau)$ of a value v of type τ is extended as follows.

$$\begin{array}{lcl} \Phi([\,] : L^{\vec{q}}(\tau)) & = & 0 \\ \Phi(v_1 :: v_2 : L^{\vec{q}}(\tau)) & = & \Phi(v_1 : \tau) + q_1 + \Phi(v_2 : L^{\lhd \vec{q}}(\tau) \end{array}$$

As usual, we assume $\vec{q} = (q_1, \ldots, q_k)$ in the definition.

To understand the potential functions for lists, we first consider some simple examples. Let for instance $\ell = [v_1 \ldots, v_n] : L(\mathbf{1})$ be list of units. Then the following holds for all $q_1, q_2, q_3 \in \mathbb{Q}_{\geq 0}$.

$$\begin{array}{rcl} \Phi(\ell{:}L^{(q_1)}(\mathbf{1})) & = & q_1 \cdot n \\ \Phi(\ell{:}L^{(0,q_2)}(\mathbf{1})) & = & \displaystyle\sum_{i=1}^{n-1} q_2 \cdot i = q_2 \frac{n \cdot (n-1)}{2} \\ \Phi(\ell{:}L^{(0,0,q_3)}(\mathbf{1})) & = & \displaystyle\sum_{i=1}^{n-1} q_3 \frac{i \cdot (i-1)}{2} = q_3 \frac{n \cdot (n-1) \cdot (n-2)}{6} \end{array}$$

In fact, the potential of a list can always be written as a non-negative linear combination of binomial coefficients. This is proved by the following lemma. We define

$$\phi(n, \vec{q}) = \sum_{i=1}^{k} \binom{n}{i} q_i \,.$$

Lemma 1. *Let $\ell = [v_1 \ldots, v_n] : L(\tau)$ be a list of type τ and let $\vec{p} = (p_1, \ldots, p_k)$ be a resource annotation. Then*

$$\Phi(\ell{:}L^{\vec{p}}(\tau)) = \phi(n, \vec{p}) + \sum_{i=1}^{n} \Phi(v_i{:}\tau) \,.$$

Proof. We prove the statement by induction on n. If $n = 0$ then $\ell = [\,]$ and we have $\Phi(\ell{:}L^{\vec{p}}(\tau)) = 0 = \sum_{i=1}^{0} \Phi(v_i{:}\tau) + \phi(0, \vec{p})$.

Let $n > 0$. It then follows by induction that

$$\begin{array}{rcl} \Phi(\ell{:}L^{\vec{p}}(\tau)) & = & p_1 + \Phi(v_1{:}\tau) + \Phi([v_2, \ldots, v_n]{:}\, L^{\lhd(\vec{p})}(\tau)) \\ & = & p_1 + \displaystyle\sum_{i=1}^{n} \Phi(v_i{:}\tau) + \phi(n-1, \lhd(\vec{p})) \end{array}$$

But since

$$\binom{n-1}{i}+\binom{n-1}{i+1}=\binom{n}{i+1} \tag{1}$$

it follows that

$$\begin{aligned}
\phi(n-1,\lhd(\vec{p})) &= \sum_{i=1}^{k}\binom{n-1}{i}p_i+\sum_{i=1}^{k-1}\binom{n-1}{i}p_{i+1}\\
&= (n-1)p_1+\sum_{i=1}^{k-1}\left(\binom{n-1}{i+1}+\binom{n-1}{i}\right)p_{i+1}\\
&= (n-1)p_1+\sum_{i=1}^{k-1}\binom{n}{i+1}p_{i+1} \qquad \text{(by (1))}\\
&= \sum_{i=1}^{k}\binom{n}{i}p_i-p_1=\phi(n,\vec{p})-p_1
\end{aligned}$$

□

It is essential for the type system that ϕ is linear in the sense of the following lemma that follows directly from the definition of ϕ.

Lemma 2. *Let $n\in\mathbb{N}$, $\alpha\in\mathbb{Q}$ and let $\vec{p},\vec{q}$ be resource annotations. Then $\phi(n,\vec{p})+\phi(n,\vec{q})=\phi(n,\vec{p}+\vec{q})$ and $\alpha\cdot\phi(n,\vec{p})=\phi(n,\alpha\cdot\vec{p})$.*

The use of binomial coefficients rather than powers of variables has many advantages. In particular, the identity

$$\sum_{i=1,\ldots,k}q_i\binom{n+1}{i}=q_1+\sum_{i=1,\ldots,k-1}q_{i+1}\binom{n}{i}+\sum_{i=1,\ldots,k}q_i\binom{n}{i}$$

gives rise to a local typing rule for cons and pattern matching, which naturally allows the typing of both recursive calls and other calls to subordinate functions in branches of a pattern match.

It is a general pattern in functional programs to compute a task on a list recursively for the tail of the list and to use the result of the recursive call to compute the result of the function. In such a recursive function it is natural to assign a uniform potential to each sublist (depending on its length) that occurs in a recursive call. In other words: one wants to use the potential of the input list to assign a uniform potential to every suffix of the list. With this view, the list potential $\alpha=\phi(n,(p_1,p_2,\cdots,p_k))$ can be read as follows: a recursive function on a list ℓ of length n that has the potential α can use the potential $\phi(i,(p_2,\cdots,p_k)$ for the suffixes of ℓ of length $1\le i<n$ that occurs in the recursion. This intuition is proved by the following lemma.

Lemma 3. *Let $\vec{p}$=$(p_1,\ldots,p_k)$ be a resource annotation, let $n\in\mathbb{N}$ and define $\phi(n,())=0$. Then $\phi(n,(p_1,\ldots,p_k))=n\cdot p_1+\sum_{i=1}^{n-1}\phi(i,(p_2,\ldots,p_k))$.*

Proof. The proof uses the following well-known equation.

$$\sum_{i=1}^{n-1}\binom{i}{k}=\binom{n}{k+1}\text{ for each }k\in\mathbb{N} \tag{2}$$

Let now $k \geq 0$. Then

$$\begin{aligned}
\phi(n,(p_1,\ldots,p_{k+1})) &= \sum_{j=1}^{k+1} \binom{n}{j} p_j \\
&= n \cdot p_1 + \sum_{j=1}^{k} \binom{n}{j+1} p_{j+1} \\
&= n \cdot p_1 + \sum_{j=1}^{k} (\sum_{i=1}^{n-1} \binom{i}{j} p_{j+1})) && \text{(by (2))} \\
&= n \cdot p_1 + \sum_{i=1}^{n-1} (\sum_{j=1}^{k} \binom{i}{j} p_{j+1})) \\
&= n \cdot p_1 + \sum_{i=1}^{n-1} \phi(i,(p_2,\ldots,p_{k+1})) && \text{(by definition)}
\end{aligned}$$

□

Note that the binomial coefficients are a basis of the vector space of the polynomials. Here, however, we are only interested in non-negative linear combinations of binomial coefficients. These admit a natural characterization in terms of growth: for $f : \mathbb{N} \to \mathbb{N}$ define $(\Delta f)(n) = f(n+1) - f(n)$. Call f *hereditarily non-negative* if $\Delta^i f \geq 0$ for all $i \geq 0$. One can show that a polynomial p is hereditarily non-negative if and only if it can be written as a non-negative linear combination of binomial coefficients. To wit, the coefficient of $\binom{n}{i}$ in the representation of p is $(\Delta^i p)(0)$. Note that they include all non-negative linear combinations of the polynomials $(x^i)_{i\in\mathbb{N}}$.

4 Static Semantics

Like for linear potential functions, the static semantics defines a judgment

$$\Gamma; q \vdash e : A\,.$$

The rules of linear AARA remain unchanged except for the rules for the introduction and elimination of lists.

$$\frac{}{\cdot; 0 \vdash \mathrm{nil} : \langle L^{\vec{p}}(\tau), 0\rangle}\ (\text{U:NIL}) \qquad \frac{}{x_1 : \tau, x_2 : L^{\lhd(\vec{p})}(\tau); p_1 \vdash \mathrm{cons}(x_1; x_2) : \langle L^{\vec{p}}(\tau), 0\rangle}\ (\text{U:CONS})$$

$$\frac{\Gamma; q \vdash e_0 : B \qquad \Gamma, x_1 : \tau, x_2 : L^{\lhd(\vec{p})}(\tau); q + p_1 \vdash e_1 : B}{\Gamma, x : L^{\vec{p}}(\tau); q \vdash \mathrm{mat_L}\{e_0; x_1, x_2.e_1\}(x) : B}\ (\text{U:MATL})$$

The rule U:NIL requires that the constant potential 0 and an empty context. It is sound to attach any potential annotation $\vec{p}$ to the empty list since the resulting potential is always zero. So not potential is gained or lost.

The rule U:CONS reflects the fact that we have to cover the potential that is assigned to the new list of type $L^{\vec{p}}(\vec{\tau})$. We do so by requiring x_2 to have the type $L^{\lhd(\vec{p})}(\tau)$ and to have p_1 resource units available. It corresponds exactly to the recursive definition of the potential function Φ and ensures that potential is neither gained nor lost.

The rule U:MATL defines how to use the potential of a list to pay for resource consumption. It accounts for the fact that either e_1 or e_2 is evaluated. The cons case is inverse to the rule U:CONS and allows us to use the potential associated with a list. For one thing, p_1 resource units become available as constant potential. For another thing, the tail of the list is annotated with $\lhd(\vec{p})$ rather than $\vec{p}$, permitting for example a recursive call requiring annotation $\vec{p}$ and an additional use of the tail with annotation $(p_2,\ldots,p_k)$ (e.g., to cover the cost of a recursive call).

We still have the structural rules for subtyping and the sharing rule and need to extend the subtyping and sharing relations to the new potential annotations.

$$\frac{\tau \curlyvee (\tau_1, \tau_2) \qquad \Gamma, x_1 : \tau_1, x_2 : \tau_2; q \vdash e : B}{\Gamma, x : \tau; q \vdash \mathrm{share}(x; x_1, x_2.e) : B}\ (\text{U:SHARE}) \qquad \frac{\Gamma; q \vdash e : \langle \tau', q' \rangle \qquad \tau' <: \tau}{\Gamma; q \vdash e : \langle \tau, q' \rangle}\ (\text{U:SUB})$$

$$\frac{\Gamma, x : \tau; q \vdash e : B \qquad \tau' <: \tau}{\Gamma, x : \tau'; q \vdash e : B}\ (\text{U:SUP})$$

Subtyping The subtyping relation is extended with the following rule.

$$\frac{\tau <: \tau' \qquad q \leq p}{L^{\vec{p}}(\tau) <: L^{\vec{q}}(\tau')}\ (\text{SUB:LIST})$$

We can still show the following lemma.

Lemma 4. *If $v : \tau$ and $\tau <: \tau'$ then $\Phi(v : \tau') \leq \Phi(v : \tau)$.*

Sharing The sharing relation is extended with the following rule.

$$\frac{q = q_1 + q_2 \qquad \tau \curlyvee (\tau_1, \tau_2)}{\langle \tau, q \rangle \curlyvee (\langle \tau_1, q_1 \rangle, \langle \tau_2, q_2 \rangle)}\ (\text{SH:POT})$$

We still have the following lemma.

Lemma 5. *If $v : \tau$ and $\tau \curlyvee (\tau_1, \tau_2)$ then $\Phi(v : \tau) = \Phi(v : \tau_1) + \Phi(v : \tau_2)$.*

Example 1. *As example, let us consider a very expensive identity function. In the code, we represent the typing $x : L^{(q_1,q_2)}(\mathbf{1})$ by writing $x^{(q_1,q_2)}$.*

```
fun id1 l^(1,0) =
  match l^(1,0) with
  | [] → []
  | x::xs^(1,0) →
     let _ = tick 1.0 in
     let xs'^(0,0) = id1 xs^(1,0) in
     (x::xs'^(0,0))^(0,0)

fun id2 l^(0,1) =
  match l^(0,1) with
  | [] → []^(0,0)
  | x::xs^(1,1) →
     share xs^(1,1) as xs1^(1,0), xs2^(0,1) in
     let _ = id1 xs1^(1,0) in
     let xs' = id2 xs2^(0,1) in
     (x::xs'^(0,1))^(0,0)
```

We can derive the following typings.

$$\begin{array}{lcl} \mathit{id1} & : & \langle L^{(1,0)}(\mathbf{1}), 0\rangle \to \langle L^{(0,0)}(\mathbf{1}), 0\rangle \\ \mathit{id2} & : & \langle L^{(0,1)}(\mathbf{1}), 0\rangle \to \langle L^{(0,0)}(\mathbf{1}), 0\rangle \end{array}$$

5 Soundness

We can prove the same soundness theorem as for linear AARA. Given the extended lemmas for sharing and subtyping, the change in the proof is limited to the cases that involve the syntactic forms for lists.

Theorem 1 (Soundness of AARA)**.** *Let* $\Gamma; q \vdash e : A$ *and* $V : \Gamma$*. If* $V \vdash e \Downarrow v \mid (p, p')$ *for some* v *and* (p, p') *then* $\Phi(V : \Gamma) + q \geq p$ *and* $\Phi(V : \Gamma) + q - \Phi(v : A) \geq p - p'$*.*

The proof is by induction on the evaluation judgment and an inner induction on the type judgment. The inner induction is needed because of the structural rules.

6 Resource-Polymorphic Recursion

In AARA with polynomial potential, it is often necessary to type recursive calls with a type that has different resource annotations than the function type of the recursive functions. Consider for example again the function *id2*. Let us assume we would like to type the function

$$\mathit{fun\ f\ l = id2\ (id2\ l)}$$

The outer call to *id2* can be typed again with the following type.

$$\mathit{id2} \quad : \quad \langle L^{(0,1)}(\mathbf{1}), 0\rangle \to \langle L^{(0,0)}(\mathbf{1}), 0\rangle$$

For the inner call, we would like to derive the typing

$$\mathit{id2} \quad : \quad \langle L^{(0,2)}(\mathbf{1}), 0\rangle \to \langle L^{(0,1)}(\mathbf{1}), 0\rangle$$

which is sound in the sense of the soundness theorem: the initial potential $2\binom{n}{2}$ is sufficient to cover the cost $\binom{n}{2}$ and the potential $\binom{n}{2}$ of the result of the call. Below is a sketch a "derivation" of this typing. However, we cannot derive this typing with our type rules.

```
fun id2 l^(0,2) =
  match l^(0,2) with
  | [] → []^(0,1)
  | x::xs^(2,2) →
     share xs^(2,2) as xs1^(1,0), xs2^(1,1) in
     let _ = id1 xs1^(1,0) in
     let xs'^(1,1) = id2 xs2^(1,2) in
     (x::xs')^(1,0)
```

The problem is the recursive call, which requires the following typing

$$\mathit{id2} \quad : \quad \langle L^{(1,2)}(\mathbf{1}), 0\rangle \to \langle L^{(1,1)}(\mathbf{1}), 0\rangle$$

which is different from the typing that we want to justify but also intuitively sounds. If we want to derive this typing we need the following typing in the recursive call.

$$\mathit{id2} \quad : \quad \langle L^{(2,2)}(\mathbf{1}), 0\rangle \to \langle L^{(2,1)}(\mathbf{1}), 0\rangle$$

The need of passing on potential of degree at most $k-1$ to the output of a function with a resource consumption of degree k is quite common in typical functions. It is present in the derivation of time bounds for most non-tail-recursive functions that we considered, for example, quick sort and insertion sort. The classic (resource-monomorphic) inference approach of requiring the type of the recursive call to match its specification fails for these functions and it was a non-trivial problem to address it with an efficient solution.

Type Rules for Resource-Polymorphic Recursion In general, we would like to state that *id2* has the following set of types.

$$\mathit{id2} \quad : \quad \{\langle L^{(q,2)}(\mathbf{1}), 0\rangle \to \langle L^{(q,1)}(\mathbf{1}), 0\rangle \mid q \in \mathbb{Q}_{\geq 0}\}$$

We can do so with the following rules for function abstraction and application.

$$\frac{(\langle \tau, q\rangle \to B) \in \mathcal{T}}{x_1 : \mathcal{T}, x_2 : \tau; q \vdash \mathrm{app}(x_1; x_2) : B}\ (\text{P:APP})$$

$$\frac{A = \langle \tau, q\rangle \qquad \Gamma = |\Gamma| \qquad \forall (\langle \tau, q\rangle \to B) \in \mathcal{T} : \Gamma, f : \mathcal{T}, x : \tau; q \vdash e : B}{\Gamma; 0 \vdash \mathrm{fun}(f, x.e) : \langle \mathcal{T}, 0\rangle}\ (\text{P:FUN})$$

While the P:FUN does look slightly worrisome, the soundness proof goes through just fine without major changes. Type inference for polymorphic recursion is more problematic.

7 Type Inference

The basis of the type inference for the univariate polynomial system is type inference algorithm for the linear system. A further challenge for the inference of polynomial bounds is the need to deal with *resource-polymorphic recursion*, which is required to type most programs that are not tail recursive. It seems to be a hard problem to infer general resource-polymorphic types, even for the original linear system.

This is why we use a pragmatic approach to resource-polymorphic recursion that works well and efficiently in practice. It infers types for most functions that admit a type-derivation, including all useful programs that we implemented. Nevertheless, it is not complete with respect to the general resource-polymorphic typing rules. At the end of this section is a somewhat artificial function with a linear resource consumption that admits a resource-polymorphic typing that can neither be inferred by the algorithm we present here nor in the classic linear system.

Resource-Monomorphic Inference

Cost-Free Types

Resource-Polymorphic Inference

Incompleteness

References

[GSS92] Jean-Yves Girard, Andre Scedrov, and Philip Scott. Bounded Linear Logic. *Theoret. Comput. Sci.*, 97(1):1–66, 1992.

[HH10a] Jan Hoffmann and Martin Hofmann. Amortized Resource Analysis with Polymorphic Recursion and Partial Big-Step Operational Semantics. In *8th Asian Symposium on Programming Languages (APLAS'10)*, 2010.

[HH10b] Jan Hoffmann and Martin Hofmann. Amortized Resource Analysis with Polynomial Potential. In *19th European Symposium on Programming (ESOP'10)*, 2010.

[HJ03] Martin Hofmann and Steffen Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th ACM Symp. on Principles of Prog. Langs. (POPL'03)*, 2003.

[Hof11] Jan Hoffmann. *Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis*. PhD thesis, Ludwig-Maximilians-Universität München, 2011.