15-819: Resource Aware Programming Languages

Lecture 12-13: Linear Automatic Amortized Resource Analysis

Jan Hoffmann

October 16, 2020

1 Introduction

After the detour to implicit computational complexity, we now turn to resource bound analysis. Given a program, we want to describe its resource usage—as defined by a cost semantics— as a function of its inputs. So the goal is to get more fine-grained information than to just learn that the function runs in polynomial time.

In this lecture, we study *linear automatic amortized resource analysis (AARA)*, which has been introduced by Hofmann and Jost in 2003 [HJ03]. Linear AARA is a type-based resource analysis system. That means that bounds are implicitly defined by function types and type derivations are certificates for the soundness of the bounds. The word linear in the title does not refer to a linear type system (in fact, AARA is based on an affine type system) but to the kind of bounds we can derive: Bounds are linear functions $q_0 + \sum_{1 \le i \le q} q_i n_i$ (roughly) of the sizes $\vec{n}$ of the arguments of a function. In the following lectures, we will also discuss polynomial AARA, which derives polynomial bounds. The word *automatic* refers to the fact that bounds can be derived automatically, using type inference.

Linear AARA uses almost all concepts that we discussed in the course so far: amortized analysis, cost semantics, an affine type system, implicit computational complexity (diamonds), and constraint-based type inference.

2 Resource Bounds with LFPL

Linear AARA is inspired by LFPL. Assume we would like to implement a function *double* in LFPL that takes a list and returns a list that is twice as long as the input. We are not able to implement this function with type

$$double : L(\text{bool}) \multimap L(\text{bool})$$

since the function would be size increasing. However, we can implement the function with the following type. Note that this is only possible since bool is a heap-free type and we can thus duplicate Booleans.

$$double : L(\Diamond \otimes \text{bool}) \multimap L(\text{bool})$$

The function *double* can be implemented as follows.

```
$\lambda(\ell : L(\Diamond \otimes \text{bool}))$
    iter $\ell$ {nil $\hookrightarrow$ nil
        | cons$(d, x, \_)$ with $y \hookrightarrow$ letp$\langle d', x' \rangle = x$ in cons$(d, x', $cons$(d', x', y))$}
```

We add two elements x' to the result list in each iteration and thus need two diamonds d and d' to "pay" for the use of cons. The computation is non-size increasing because the size of an input $\ell : L(\Diamond \otimes \text{bool})$ is $2|\ell| = |double(\ell)|$ where $|\cdot|$ is the length of a list.

The first observation we make is that the diamonds in the input—or,equivalently, the size of the input as defined in the previous lecture—are an upper bound on the number of cons operations that are performed during the evaluation. So the type derivation of double shows that we do not perform more than $2|\ell|$ cons operations during an evaluation of $\mathit{double}(\ell)$.

The second observation we make is that we can change LFPL to derive worst-case bounds not only on the number of cons operations but on arbitrary resources. To achieve this, we separate diamonds from lists and add a syntactic form tick that represents a cost of 1 when it is executed. However, we require that we need to have a diamond available to pay for the cost of tick. To simplify the use of ticks we also build in sequential composition and add the syntactic form $\mathrm{tick}(e_1;e_2)$, which corresponds to evaluating a tick of cost 1 and then evaluating e_2. The expression e_1 evaluates to the diamond that we need to cover the cost. So we have the following type rules.

$$\frac{\Gamma_1 \vdash e_1 : \Diamond \qquad \Gamma_2 \vdash e_2 : \tau}{\Gamma_1, \Gamma_2 \vdash \mathrm{tick}(e_1;e_2) : \tau}\ (\textsc{Tick}) \qquad\qquad \frac{\Gamma_1 \vdash e_1 : \tau \qquad \Gamma_2 \vdash e_2 : L(\tau)}{\Gamma_1, \Gamma_2 \vdash \mathrm{cons}(e_1;e_2) : L(\tau)}\ (\textsc{Cons})$$

$$\frac{\Gamma_1 \vdash e : L(\tau') \qquad \Gamma_2 \vdash e_1 : \tau \qquad x:\tau', y:\tau \vdash e_2 : \tau}{\Gamma_1, \Gamma_2 \vdash \mathrm{iter}_{\mathrm{L}}\{e_0; x, y.e_1\}(e) : \tau}\ (\textsc{Iter})$$

Let us now derive a bound on the number of constructors we use in an inefficient identity function for lists. In the function, we traverse the list and rebuild the same list step by step. So $\mathit{id}(\ell)$ performs $|\ell|+1$ constructor calls: 1 call of nil and $|\ell|$ calls of cons. We can express this bound in our modified LFPL as follows.

$$\begin{array}{lll} \mathit{id} & : & L(\Diamond \otimes \mathit{bool}) \multimap \Diamond \multimap L(\Diamond) \\ \mathit{id} & \equiv & \lambda(\ell : L(\Diamond \otimes \mathrm{bool}))\ \lambda(d : \Diamond) \\ & & \quad \mathsf{iter}\ \ell\ \{\mathsf{nil} \hookrightarrow \mathsf{tick}\,(d, \mathsf{nil}) \\ & & \qquad\quad \mid \mathsf{cons}\,(x,\_)\ \mathsf{with}\ y \hookrightarrow \mathsf{letp}\,\langle d', x'\rangle = x\ \mathsf{in}\ \mathsf{tick}\,(d', \mathsf{cons}\,(x', y)))\} \end{array}$$

This is basically the idea of linear AARA. However, the are a few restrictions that make LFPL inconvenient to use. LFPL uses an affine type discipline which prevents us from using some variables multiple times in a program. For example, we cannot for every type τ implement a function like

$$\lambda(\ell : L(\tau))\ \langle \mathit{id}(\ell), \mathit{id}(\ell)\rangle\ .$$

Moreover, there is a lot of book-keeping in the code and we have to micro-manage which diamond to use to pay for a particular tick. To alleviate these restrictions, we remove diamonds from the expressions and only keep track of them in the types.

3 Linear AARA with Lists

Syntax We start the definition of AARA with a simple language with units and lists. We later add sums, products, and recursive types. Expressions are defined as follows.

$$\begin{array}{lll} e & ::= & x & x \\ & & \mathrm{triv} & \langle\rangle \\ & & \mathrm{app}(x_1;x_2) & x_1(x_2) \\ & & \mathrm{fun}(f,x.e) & \mathsf{fun}\ f\,x = e \\ & & \mathrm{nil} & [\,] \\ & & \mathrm{cons}(x_1;x_2) & x_1 :: x_2 \\ & & \mathrm{mat}_{\mathrm{L}}\{e_0;x_1,x_2.e_1\}(x) & \mathsf{case}\ x\ \{\mathsf{nil} \hookrightarrow e_0 \mid \mathsf{cons}\,(x_1,x_2) \hookrightarrow e_1\} \\ & & \mathrm{tick}\{q\} & \mathsf{tick}\ q \\ & & \mathrm{let}(e_1;x.e_2) & \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\ & & \mathrm{share}(x;x_1,x_2.e) & \mathsf{share}\ x\ \mathsf{as}\ x_1,x_2\ \mathsf{in}\ e \end{array}$$

The elimination form for lists is standard pattern matching as opposed to iteration as in LFPL. Instead, we add general recursive functions. So the only recursively defined expressions are function abstractions. This is not a limitation of AARA but it simplifies the presentation and technical development. We also have a syntactic from share for explicit contraction that we discuss later.

Expressions are given in *share-let-normal form*, which means that subexpressions in syntactic forms can only be variables whenever this does not expressivity. For example, applications have the form $\mathrm{app}(x_1; x_2)$ where x_1 and x_2 are variables. Moreover, function abstractions and the constructor nil are not annotated with types. As a result, we forgo unique typing of expressions, which is a property that is not desirable in the context of AARA.

We also have explicit tick expressions $\mathrm{tick}\{q\}$ that define resource usage $q \in \mathbb{Q}$ as before. If $q < 0$ then resources become available. As we have seen on Assignment 2, it is possible to translate (arbitrary) expressions with resource metrics into equivalent expressions in let-normal form with ticks.

We define the following syntactic abbreviations. We write

$$\begin{array}{lll l} \mathsf{let}\ f\,x = e\ \mathsf{in}\ e' & \text{for} & \mathsf{let}\ f = \mathsf{fun}\ f\,x = e\ \mathsf{in}\ e' & \\ \lambda(x)e & \text{for} & \mathrm{fun}\ f\,x = e & \text{where } f \text{ is fresh} \end{array}$$

Types and Potential Annotated types are defined by the following (mutually recursive) grammar.

$$\begin{array}{llll} \tau & ::= & \mathrm{unit} & \mathbf{1} \\ & & \mathrm{arr}(A;B) & A \to B \\ & & L(A) & L^q(\tau) \\ A,B & ::= & \mathrm{pot}(\tau;q) & \langle \tau, q \rangle \end{array}$$

We require $q \in \mathbb{Q}_{\geq 0}$ in $\langle \tau, q \rangle$ and call q a potential annotation. For a list type $L(\langle \tau, q \rangle)$ we often write $L^q(\tau)$.

One way to build an intuition about potential annotation is to think of them as the diamonds of LFPL. So we have $L(\langle \tau, 1 \rangle)$ instead of $L(\tau \otimes \Diamond)$ but lists of type $L(\langle \tau, 1 \rangle)$ have elements of type τ.

Resource bounds are implicitly given by types in the form of *potential*, which maps values to non-negative rational numbers. To define potential, we first sketch the definition of values. We leave the definition of functions open until we define the dynamic semantics. Here, we do not need details since the potential of a function is zero.

$$\begin{array}{lll} [\![\mathbf{1}]\!] & = & \{\langle\rangle\} \\ [\![L(A)]\!] & = & \{[v_1,\ldots,v_n] \mid n \in \mathbb{N}, v_i \in [\![A]\!]\} \\ [\![\langle \tau, q \rangle]\!] & = & [\![\tau]\!] \\ [\![A \to B]\!] & = & \text{(defined later)} \end{array}$$

We now define the potential $\Phi(v : \tau)$ of a value v under type τ. We have $\Phi(\cdot : \tau) : [\![\tau]\!] \to \mathbb{Q}_{\geq 0}$.

$$\begin{array}{lll} \Phi(\langle\rangle : \mathbf{1}) & = & 0 \\ \Phi([\,] : L(A)) & = & 0 \\ \Phi(v_1 :: v_2 : L(A)) & = & \Phi(v_1 : A) + \Phi(v_2 : L(A) \\ \Phi(v : \langle \tau, q \rangle) & = & \Phi(v : \tau) + q \\ \Phi(v : A \to B) & = & 0 \end{array}$$

So the only values that are carrying potential are lists and we have

$$\Phi([v_1,\ldots,v_n] : L^q(\tau)) = q \cdot n + \sum_{1 \leq i \leq n} \Phi(v_i : \tau)$$

Example 1. *Consider again the inefficient identity function.*

```
fun id l =
  match l with
  | [] → []
  | x::xs → let _ = tick 2 in
            let ys = id xs in
            x::ys
```

The evaluation cost of id ℓ is then $2|\ell|$. We can reflect this bound by assigning the following type to id.

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

The potential of $\Phi(\ell : L^2(\mathbf{1})) + 0 = 2|\ell|$ of the argument is identical to the cost of the evaluation.

Interestingly, we can justify a similar type for an arbitrary element type τ.

$$id : \langle L^2(\tau), 0\rangle \to \langle L^0(\tau), 0\rangle$$

The potential $\Phi(\ell : L^2(\tau))$ now includes the potential that is assigned by τ to the elements of the list. The same potential appears in the result type of the function where it is assigned to the elements of the result list. This potential can be used to pay the cost of another function that is applied to the resulting list. This is sound because every element in the argument of id appears exactly once in the output.

Similarly, we can assign more potential to the input list and pass it through to the result. For example, we can assign the following type.

$$id : \langle L^4(\tau), 5\rangle \to \langle L^2(\tau), 5\rangle$$

This type is sound because the length of the result of id is bounded by the length of its argument. It is useful to type the first call to id in the following program.

```
fun id2 x =
 let y = id x in
 let _ = tick 5 in
 id y
```

Using the previously defined type of id we can justify the following typing.

$$id2 : \langle L^4(\tau), 5\rangle \to \langle L^0(\tau), 0\rangle$$

Static Semantics The typing judgment $\Gamma; q \vdash e : A$ is defined by the rules in Figure 1. The intuitive meaning of the judgment is that the potential given by Γ and q is sufficient to cover the evaluation cost of e and the potential defined by A. We will formalize this intuition when we discuss soundness.

The rules are formulated in a linear style with additional structural rules that relax the judgment to treat potential in an affine way. In the rule L:APP, we require that we have the exact potential annotations ($x_2 : \tau$ and q) that are required by the argument. The resulting potential is given by the result type B.

In the rule L:CONS, we have to provide potential p to account for the potential of the new list element. Conversely, the potential of the head x_1 of the list $x : L^p(\tau)$ becomes available in the cons branch of the pattern match in the rule L:MATL. As a result, we have constant potential $p + q$ available when typing e_1.

In the rule L:FUN for (recursive) function abstraction, we require that the potential of the variables captured in the context Γ is zero. We write $|\Gamma|$ for the context Γ in which every potential annotation q is replaced by 0. This is formally defined below. The reason for this requirement is that we allow functions to be used an arbitrary number of times (see the later definition of sharing). If Γ would carry potential then we could use this potential multiple times to account for cost, which is not sound. Since functions do not carry potential, we do not have to restrict the type of the recursively defined function f in a similar way. An alternative would be to remove the premise $\Gamma = |\Gamma|$ and to treat functions in an affine way. Of course, we can also design a type system with both, affine and unrestricted functions.

The formal definition of $|\cdot|$ for annotated types follows.

$$\begin{array}{lcl}
|\text{unit}| & = & \text{unit} \\
|L(A)| & = & L(|A|) \\
|A \to B| & = & A \to B \\
|\langle \tau, q\rangle| & = & \langle |\tau|, 0\rangle
\end{array}$$

$$\boxed{\Gamma; q \vdash e : A \qquad \text{“expression } e \text{ has annotated type } A \text{ under context } \Gamma \text{ and potential } q\text{”}}$$

Syntax directed rules:

$$\frac{}{x : \tau; 0 \vdash x : \langle \tau, 0 \rangle}\ (\text{L:Var}) \qquad \frac{}{\cdot; 0 \vdash \mathrm{triv} : \langle \mathrm{unit}, 0 \rangle}\ (\text{L:Unit})$$

$$\frac{A = \langle \tau, q \rangle}{x_1 : A \to B, x_2 : \tau; q \vdash \mathrm{app}(x_1; x_2) : B}\ (\text{L:App})$$

$$\frac{A = \langle \tau, q \rangle \qquad \Gamma = |\Gamma| \qquad \Gamma, f : A \to B, x : \tau; q \vdash e : B}{\Gamma; 0 \vdash \mathrm{fun}(f, x.e) : \langle A \to B, 0 \rangle}\ (\text{L:Fun}) \qquad \frac{}{\cdot; 0 \vdash \mathrm{nil} : \langle L^p(\tau), 0 \rangle}\ (\text{L:Nil})$$

$$\frac{}{x_1 : \tau, x_2 : L^p(\tau); p \vdash \mathrm{cons}(x_1; x_2) : \langle L^p(\tau), 0 \rangle}\ (\text{L:Cons})$$

$$\frac{\Gamma; q \vdash e_0 : B \qquad \Gamma, x_1 : \tau, x_2 : L^p(\tau); q + p \vdash e_1 : B}{\Gamma, x : L^p(\tau); q \vdash \mathrm{mat}_{\mathrm{L}}\{e_0; x_1, x_2.e_1\}(x) : B}\ (\text{L:MatL}) \qquad \frac{q \geq 0}{\cdot; q \vdash \mathrm{tick}\{q\} : \langle \mathrm{unit}, 0 \rangle}\ (\text{L:Tick1})$$

$$\frac{q < 0}{\cdot; 0 \vdash \mathrm{tick}\{-q\} : \langle \mathrm{unit}, q \rangle}\ (\text{L:Tick2}) \qquad \frac{\Gamma_1; q \vdash e_1 : \langle \tau, p \rangle \qquad \Gamma_2, x : \tau; p \vdash e_2 : B}{\Gamma_1, \Gamma_2; q \vdash \mathrm{let}(e_1; x.e_2) : B}\ (\text{L:Let})$$

$$\frac{\tau \curlyvee (\tau_1, \tau_2) \qquad \Gamma, x_1 : \tau_1, x_2 : \tau_2; q \vdash e : B}{\Gamma, x : \tau; q \vdash \mathrm{share}(x; x_1, x_2.e) : B}\ (\text{L:Share})$$

Structural rules:

$$\frac{\Gamma; q \vdash e : \langle \tau', q' \rangle \qquad \tau' <: \tau}{\Gamma; q \vdash e : \langle \tau, q' \rangle}\ (\text{L:Sub}) \qquad \frac{\Gamma, x : \tau; q \vdash e : B \qquad \tau' <: \tau}{\Gamma, x : \tau'; q \vdash e : B}\ (\text{L:Sup})$$

$$\frac{\Gamma; q \vdash e : B}{\Gamma, x : \tau; q \vdash e : B}\ (\text{L:Weak}) \qquad \frac{\Gamma; p \vdash e : \langle \tau, p' \rangle \qquad q \geq p \qquad q - q' \geq p - p'}{\Gamma; q \vdash e : \langle \tau, q' \rangle}\ (\text{L:Relax})$$

Figure 1: Static semantics of linear AARA.

For arrow type, we do not have to recursively eliminate potential since the potential of a function is already 0 for every arrow type.

The definition is lifted point-wise to annotated contexts Γ.

$$\begin{array}{rcl} |\cdot| & = & \cdot \\ |\Gamma, x : \tau| & = & |\Gamma|, x : |\tau| \end{array}$$

The weakening rule L:Weak is standard. However, there is another form of weakening: The rule L:Relax, states that, given a judgment $\Gamma; p \vdash e : \langle \tau, p' \rangle$, we can also have more potential q in the context and give up some of the potential p'. Additionally, the rule also covers the case in which we pass through additional potential $c \geq 0$ yielding the judgment $\Gamma; p + c \vdash e : \langle \tau, p' + x \rangle$.

Example 2. *Below is the definition of an append function for lists.*

```
fun append l1 = fun _ l2 =
  match l1 with
    | nil → l2
    | x::xs → let _ = tick 1 in
              let y = append xs l2 in
```

```
        x::y
```

We would like to derive a type like

$$\langle L^1(\text{unit}), 0\rangle \to \langle L^0(\text{unit}), 0\rangle \to \langle L^0(\text{unit}), 0\rangle$$

to append. However, the second function abstraction fun _ l2 = ... *would have to be typed in the context* $\Gamma = l1 : L^1(\text{unit})$ *and not* $l1 : L^0(\text{unit})$ *as require by the rule* L:FUN.

Subtyping The subtyping rules T:SUB and T:SUP enable us to relax the potential requirements for potential in data structures in the same way as T:RELAX does for constant potential. The subtyping relation for types is defined by the following rules.

$$\frac{}{\text{unit} <: \text{unit}}\ (\text{SUB:UNIT}) \qquad \frac{A_2 <: A_1 \qquad B_1 <: B_2}{A_1 \to B_1 <: A_2 \to B_2}\ (\text{SUB:ARR})$$

$$\frac{q_1 \geq q_2 \qquad \tau_1 <: \tau_2}{\langle \tau_1, q_1\rangle <: \langle \tau_2, q_2\rangle}\ (\text{SUB:POT}) \qquad \frac{A <: B}{L(A) <: L(B)}\ (\text{SUB:LIST})$$

Lemma 1. *If* $v : \tau$ *and* $\tau <: \tau'$ *then* $\Phi(v : \tau') \leq \Phi(v : \tau)$.

Example 3. *For example we have*

$$\begin{array}{lcl} L^{10}(L^4(\mathbf{1})) & <: & L^8(L^3(\mathbf{1})) \\ \langle L^4(\mathbf{1}), 2\rangle \to \langle L^2(\mathbf{1}), 4\rangle & <: & \langle L^5(\mathbf{1}), 4\rangle \to \langle L^0(\mathbf{1}), 0\rangle \\ \langle L^2(\mathbf{1}), 2\rangle \to \langle L^0(\mathbf{1}), 0\rangle & \not<: & \langle L^4(\mathbf{1}), 4\rangle \to \langle L^2(\mathbf{1}), 2\rangle \end{array}$$

Sharing The sharing relation $\tau \curlyvee (\tau_1, \tau_2)$ specifies how potential in a type τ can be split between two types τ_1 and τ_2. Note that function types can be shared freely.

$$\frac{}{\text{unit} \curlyvee (\text{unit}, \text{unit})}\ (\text{SH:UNIT}) \qquad \frac{}{A \to B \curlyvee (A \to B, A \to B)}\ (\text{SH:ARR})$$

$$\frac{A \curlyvee (A_1, A_2)}{L(A) \curlyvee (L(A_1), L(A_2))}\ (\text{SH:LIST}) \qquad \frac{q = q_1 + q_2 \qquad \tau \curlyvee (\tau_1, \tau_2)}{\langle \tau, q\rangle \curlyvee (\langle \tau_1, q_1\rangle, \langle \tau_2, q_2\rangle)}\ (\text{SH:POT})$$

We can prove the following lemma.

Lemma 2. *If* $v : \tau$ *and* $\tau \curlyvee (\tau_1, \tau_2)$ *then* $\Phi(v : \tau) = \Phi(v : \tau_1) + \Phi(v : \tau_2)$.

Example 4. *For example, we have*

$$\begin{array}{l} L^{10}(L^4(\mathbf{1})) \curlyvee (L^5(L^4(\mathbf{1})), L^5(L^0(\mathbf{1}))) \\ L^{10}(L^4(\mathbf{1})) \curlyvee (L^6(L^2(\mathbf{1})), L^4(L^2(\mathbf{1}))) \\ \langle L^4(\mathbf{1}), 2\rangle \to \langle L^2(\mathbf{1}), 4\rangle \curlyvee (\langle L^4(\mathbf{1}), 2\rangle \to \langle L^2(\mathbf{1}), 4\rangle, \langle L^4(\mathbf{1}), 2\rangle \to \langle L^2(\mathbf{1}), 4\rangle) \end{array}$$

Example 5. *Consider the following function* double*, which doubles the length of a list by copying each element.*

```
fun double l =
  match l with
    | nil → nil
    | x::xs → let y = double xs in
             share x as x1,x2 in
             let y' = x1::y in
             x2::y'
```

Note that double does not contain any ticks and has thus cost 0 *for all inputs. However, it is still interesting to study the function to see how potential from the argument can be assigned to the result. In a larger program, this potential could then be used to pay for the cost of a later function call that takes the result of* double *as an argument. Using the typing rule for sharing, we can derive the following judgments.*

$$\begin{array}{lcl} \textit{double} & : & \langle L^2(\text{unit}), 0\rangle \to \langle L^1(\text{unit}), 0\rangle \\ \textit{double} & : & \langle L^2(L^6(\text{unit})), 0\rangle \to \langle L^1(L^3(\textit{unit})), 1\rangle \end{array}$$

The first typing is sound because $|\text{double}(\ell)| \le 2|\ell|$. *The second typing is sound because every list element of the argument appears (at most) twice in the result list.*

Example 6. *Functions can be freely shared. As a result, we can give the following type to the map function for lists.*

$$\textit{map} : (A \to B) \to L(A) \to L(B)$$

The key for deriving the typing is that we can use the higher-order argument twice: once to apply it to the head of the list and once in the recursive call. Not that this would not be possible if we would treat functions in an affine way like in LFPL.

```
fun map f = fun _ l =
  match l with
    | nil → nil
    | x::xs → share f as f1,f2 in
               let x' = f1 x in
               let xs' = map f2 xs in
               x'::xs'
```

4 Dynamic Semantics

To define the resource usage of programs, we define a cost semantics that is based on an evaluation dynamics. Since our expressions are in share-let-normal form, we use an evaluation environment

$$V : \textit{Var} \to \textit{Val}$$

that maps variables to values. To track the resource usage, we use the and a cost dynamics with *resource effects* and the resource monoid that we have introduced in the cost dynamics lecture. The evaluation judgment has the form

$$V \vdash e \Downarrow v \mid (q, q')\,.$$

The intuitive meaning is that expression e evaluates to value v with high-water mark cost q and net cost $q - q'$.

Resource Monoid As a reminder, the pairs (q, q') in the evaluation judgments are elements of monoid $(\mathbb{Q}_{\ge 0} \times \mathbb{Q}_{\ge 0}, \cdot)$. The neutral element is $(0,0)$ and the operation $(q, q') \cdot (p, p')$ is defined as follows.

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

If resources are never returned (as with time) then we only have elements of the form $(q, 0)$ and $(q, 0) \cdot (p, 0)$ is just $(q + p, 0)$. We identify a rational number $q \in \mathbb{Q}$ with an element of $\mathcal{Q}$ as follows: $q \ge 0$ denotes $(q, 0)$ and $q < 0$ denotes $(0, -q)$.

$\boxed{v : \tau \qquad \text{“value } v \text{ is of type } \tau\text{”}}$

$$\frac{}{\langle\rangle : \text{unit}}\ (\text{V:UNIT}) \qquad \frac{}{[\,] : L(A)}\ (\text{V:NIL}) \qquad \frac{v_1 : A \qquad v_2 : L(A)}{v_1 :: v_2 : L(A)}\ (\text{V:CONS})$$

$$\frac{v : \tau}{v : \langle \tau, q\rangle}\ (\text{V:ANNO}) \qquad \frac{A = \langle \tau, q\rangle \qquad V : \Gamma \qquad |\Gamma|, f : A \to B, x : \tau; q \vdash e : B}{\text{clo}(V; f, x.e) : A \to B}\ (\text{V:FUN})$$

$$\frac{}{V : \cdot}\ (\text{V:ENV-1}) \qquad \frac{V : \Gamma \qquad V(x) : \tau}{V : \Gamma, x : \tau}\ (\text{V:ENV-2})$$

Figure 2: Typing rules for values.

Values Previously, we defined values but left the definition of functions open. We now define function values and use a syntactic approach that fits well with our evaluation dynamics and the soundness proof. Continuing our denotational approach and defining a set $[\![A \to B]\!]$ of mathematical function would introduce complications that arise from the combination of higher-order functions and divergence. Instead introduce a type judgment $v : \tau$ for values and define

$$[\![\tau]\!] = \{v \mid v : \tau\}\,.$$

Values are defined by the following grammar. Function values are *function closures* that consist of an environment and a (recursive) lambda abstraction. The definition is mutually recursive with the definition of environments.

$$\begin{array}{rcl} v & ::= & \langle\rangle \\ & & [\,] \\ & & v_1 :: v_2 \\ & & \text{clo}(V; f, x.e) \\ \\ V & ::= & \cdot \\ & & V, x \mapsto v \end{array}$$

In the value typing $v : \tau$, which is defined in Figure 2, we ignore the potential annotations. The most interesting rule is the rule for V:FUN for function closures. It matches the type rule L:FUN for expressions and the context Γ is existentially quantified. The motivation for the rule is that we require exactly the conditions that are needed to prove the soundness of the analysis. This works because the preconditions are true for closure that are created in the evaluation of a well-typed program.

The value typing is defined recursively with the typing $V : \Gamma$ of environments. It states that all variables x that are assigned a type in Γ must be mapped to a value of type $\Gamma(x)$ in V, that is, $V(x) : \Gamma(x)$.

Evaluation Rules Figure 3 defines the evaluation judgment $V \vdash e \Downarrow v \mid (q, q')$. Like before, (q, q') is an element of the resource monoid that we use to keep track of the high-water mark (q) and the remaining resources (q'). The main difference to the evaluation dynamics that we discussed earlier is the evaluation environment V. With an evaluation environment, we do not use substitution but instead keep variable-value bindings in V. The notion

$$V, x \mapsto v$$

is interpreted differently as for contexts. The meaning is that the binding $x \mapsto v$ overwrites existing bindings $x \mapsto v'$ that might already exist in V.

$$\boxed{V \vdash e \Downarrow v \mid (q,q') \qquad \text{“in environment } V\text{, expression } e \text{ valuates to value } v \text{ with cost } (q,q')\text{”}}$$

$$\frac{}{V \vdash x \Downarrow V(x) \mid 0}\ (\text{E:Var}) \qquad \frac{}{V \vdash \mathrm{triv} \Downarrow \langle\rangle \mid 0}\ (\text{E:Unit})$$

$$\frac{}{V \vdash \mathrm{fun}(f,x.e) \Downarrow \mathrm{clo}(V;f,x.e) \mid 0}\ (\text{E:Fun})$$

$$\frac{V(x_1) = \mathrm{clo}(V';f,x.e) \qquad V(x_2) = v_2 \qquad V', f \mapsto \mathrm{clo}(V';f,x.e), x \mapsto v_2 \vdash e \Downarrow v \mid (q,q')}{V \vdash \mathrm{app}(x_1;x_2) \Downarrow v \mid (q,q')}\ (\text{E:App})$$

$$\frac{}{V \vdash \mathrm{nil} \Downarrow [\,] \mid 0}\ (\text{E:Nil}) \qquad \frac{V(x_1) = v_1 \qquad V(x_1) = v_2}{V \vdash \mathrm{cons}(x_1;x_2) \Downarrow v_1 :: v_2 \mid 0}\ (\text{E:Cons})$$

$$\frac{V(x) = [\,] \qquad V \vdash e_0 \Downarrow v \mid (q,q')}{V \vdash \mathrm{mat_L}\{e_0;x_1,x_2.e_1\}(x) \Downarrow v \mid (q,q')}\ (\text{E:MatL-1})$$

$$\frac{V(x) = v_1 :: v_2 \qquad V, x_1 \mapsto v_1, x_2 \mapsto v_2 \vdash e_1 \Downarrow (v) \mid (q,q')}{V \vdash \mathrm{mat_L}\{e_0;x_1,x_2.e_1\}(x) \Downarrow v \mid (q,q')}\ (\text{E:MatL-2})$$

$$\frac{}{V \vdash \mathrm{tick}\{q\} \Downarrow \langle\rangle \mid q}\ (\text{E:Tick}) \qquad \frac{V \vdash e_1 \Downarrow v_1 \mid (q,q') \qquad V, x \mapsto v_1 \vdash e_1 \Downarrow v \mid (p,p')}{V \vdash \mathrm{let}(e_1;x.e_2) \Downarrow v \mid (q,q') \cdot (p,p')}\ (\text{E:Let})$$

$$\frac{V(x) = v \qquad V, x_1 \mapsto v, x_2 \mapsto v \vdash e \Downarrow v' \mid (q,q')}{V \vdash \mathrm{share}(x;x_1,x_2.e) \Downarrow v' \mid (q,q')}\ (\text{E:Share})$$

Figure 3: Cost semantics with resource effects.

A consequence of using an evaluation environment is that we (in general) evaluate open expressions (i.e., expressions with free variables). So we have an evaluation rule E:Var for variables that looks up the value of a variable in the environment V. In the evaluation of a well-typed, closed expression $e : A$, the V always contains a value for x (see Theorem 1 below).

The most interesting rules are the rules E:Fun and E:App for (recursive) function abstraction and function application. In the rule E:Fun, we have to store the current environment together with the function to record the current values of the free variables in the function body. This is usually take care of by substitution. Here, since variables might later be overwritten, we have to store V in alongside with the function definition in a so-called *function closure*. The self-reference inherent in the function abstraction is not resolved when creating a closure (we just store V and do not create a binding that maps f to some value) but during function application.

In the rule E:App, x_1 maps to a function closure $\mathrm{clo}(V';f,x.e)$. We evaluate the function body e under the stored environment V' with added bindings for the argument x and the recursive reference f. We get the value for x by from the value of the concrete argument x_2. The value for f is simply the closure $\mathrm{clo}(V';f,x.e)$ again, which introduces the self reference.

The type soundness theorem is given below. It makes a statement about terminating evaluations only. Using partial evaluations, we can state and prove a stronger version of the theorem that distinguishes between diverging and failing evaluations.

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

We can prove the theorem by induction on the evaluation judgment $V \vdash e \Downarrow v \mid (q,q')$.

$$\boxed{V \vdash e \Downarrow \circ \mid q \qquad \text{“in environment } V\text{, expression } e \text{ uses } q \text{ at some point during the evaluation”}}$$

$$\frac{}{V \vdash e \Downarrow \circ \mid 0}\ (\text{P:Stop})$$

$$\frac{V(x_1) = \mathrm{clo}(V'; f, x.e) \qquad V(x_2) = v_2 \qquad V', f \mapsto \mathrm{clo}(V'; f, x.e), x \mapsto v_2 \vdash e \Downarrow \circ \mid q}{V \vdash \mathrm{app}(x_1; x_2) \Downarrow v \Downarrow \circ \mid q}\ (\text{P:App})$$

$$\frac{V(x) = [\,] \qquad V \vdash e_0 \Downarrow \circ \mid q}{V \vdash \mathrm{mat}_{\mathrm{L}}\{e_0; x_1, x_2.e_1\}(x) \Downarrow \circ \mid q}\ (\text{P:MatL-1})$$

$$\frac{V(x) = v_1 :: v_2 \qquad V, x_1 \mapsto v_1, x_2 \mapsto v_2 \vdash e_1 \Downarrow \circ \mid q}{V \vdash \mathrm{mat}_{\mathrm{L}}\{e_0; x_1, x_2.e_1\}(x) \Downarrow \circ \mid q}\ (\text{P:MatL-2})$$

$$\frac{V \vdash e_1 \Downarrow \circ \mid q \qquad V, x \mapsto v_1 \vdash e_1 \Downarrow v \mid (p, p')}{V \vdash \mathrm{let}(e_1; x.e_2) \Downarrow \circ \mid q}\ (\text{P:Let-1})$$

$$\frac{V \vdash e_1 \Downarrow v_1 \mid (q_1, q_1') \qquad V, x \mapsto v_1 \vdash e_1 \Downarrow \circ \mid q_2 \qquad (p, p') = (q_1, q_1') \cdot q_2}{V \vdash \mathrm{let}(e_1; x.e_2) \Downarrow \circ \mid p}\ (\text{P:Let-2})$$

$$\frac{V(x) = v \qquad V, x_1 \mapsto v, x_2 \mapsto v \vdash e \Downarrow \circ \mid q}{V \vdash \mathrm{share}(x; x_1, x_2.e) \Downarrow \circ \mid q}\ (\text{P:Share})$$

Figure 4: Partial cost semantics with resource effects.

5 Soundness

The soundness theorem makes our intuition about the type system precise. Before we can state the theorem we need to extend the definition of potential to contexts and environments. Let $V : \Gamma$. We define

$$\Phi(V : \Gamma) = \sum_{x \in \mathrm{dom}(\Gamma)} \Phi(V(x) : \Gamma(x))$$

Theorem 2 (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* $\Phi(V : \Gamma) + q - \Phi(v : A) \geq p - p'$*.*

If we would only have the syntax directed rules then we could prove the theorem by rule induction on the evaluation judgment. However, the structural rules and give us only a weak inversion lemma for the type judgments. This is why we have to also do an inner induction on the type judgment.

6 Partial Evaluation

A shortcoming of Theorem 2 is that it only makes a statement about terminating evaluations. For one thing, diverging evaluations can also have interesting (i.e., finite) resource behaviors if resource can become available. For another thing, we would like to be able to show that resource bounds on resources like time imply termination.

Fortunately, we can use the idea of partial evaluations to extend the Theorem 2 to diverging computations. Like in the lecture on cost semantics, we define the judgment

$$V \vdash e \Downarrow \circ \mid q$$

which states that e uses q resources at some point during its evaluation. The rules of the judgment are defined in Figure 4.

Theorem 3 (Soundness of AARA)**.** *Let* $\Gamma; q \vdash e : A$ *and* $V : \Gamma$. *If* $V \vdash e \Downarrow \circ \mid p$ *for some* p *then* $\Phi(V:\Gamma) + q \geq p$.

The proof of Theorem 3 is similar to the proof of Theorem 2 but uses Theorem 2 in the case of the rule P:LET-2. The following corollary is an immediate consequence of Theorem 3.

Corollary 1. *If* $\Gamma; q \vdash e : A$ *and* $V : \Gamma$ *then* $\Phi(V:\Gamma) + q \geq \max\{p \mid V \vdash e \Downarrow \circ \mid p\}$.

Termination We would like to prove a theorem for programs that add a tick to every function application. Instead of defining such programs we remove ticks from the language and change the cost semantics to account cost 1 for applications and cost 0 for all other syntactic forms. Since we do not have negative cost, we only need to keep track of one number. We write $V \vdash^s e \Downarrow v \mid n$ and $V \vdash^s e \Downarrow \circ \mid n$ for the resulting judgments. The rules for function applications change as follows.

$$\frac{V(x_1) = \mathrm{clo}(V'; f, x.e) \qquad V(x_2) = v_2 \qquad V', f \mapsto \mathrm{clo}(V'; f, x.e), x \mapsto v_2 \vdash^s e \Downarrow v \mid n}{V \vdash^s \mathrm{app}(x_1; x_2) \Downarrow v \mid 1 + n} \text{ (E:APP')}$$

$$\frac{V(x_1) = \mathrm{clo}(V'; f, x.e) \qquad V(x_2) = v_2 \qquad V', f \mapsto \mathrm{clo}(V'; f, x.e), x \mapsto v_2 \vdash^s e \Downarrow \circ \mid n}{V \vdash^s \mathrm{app}(x_1; x_2) \Downarrow v \Downarrow \circ \mid 1 + n} \text{ (P:APP')}$$

The other rules remain basically unchanged.

The key property that we need is stated by Lemma 3, which can be seen as a progress theorem.

Lemma 3. *Let* $\Gamma; q \vdash e : A$ *and* $V : \Gamma$. *Then either* $V \vdash^s e \Downarrow v \mid m$ *for some* v *and* $m \in \mathbb{N}$, *or* $V \vdash^s e \Downarrow \circ \mid n$ *for every* $n \in \mathbb{N}$.

Similarly, we modify the type system by changing the rule L:APP as follows.

$$\frac{A = \langle \tau, q \rangle}{x_1 : A \to B, x_2 : \tau; q + 1 \vdash^s \mathrm{app}(x_1; x_2) : B} \text{ (L:APP')}$$

Like in the general case, we can then prove the following theorem.

Theorem 4. *Let* $\Gamma; q \vdash^s e : A$ *and* $V : \Gamma$. *If* $V \vdash^s e \Downarrow \circ \mid n$ *for some* n *then* $\Phi(V:\Gamma) + q \geq n$.

From Theorem 5 and Lemma 3 we can derive the desired termination proof.

Theorem 5. *If* $\Gamma; q \vdash^s e : A$ *and* $V : \Gamma$ *then* $V \vdash^s e \Downarrow v \mid n$ *for some* v *and* p *with* $\Phi(V:\Gamma) + q \geq n$.

7 Mutually Recursive Functions

AARA has been introduced [HJ03] for a first-order language (i.e., a language without higher-order functions) with mutually recursive functions. Extending linear AARA as introduced here to mutually recursive function does not introduce any challenges. We just illustrate this by extending our language with a construct for defining two mutually recursive functions. Since we do not have pairs, we combine the (recursive) function definition with a let binding.

$$\begin{array}{rcl} e & ::= & \dots \\ & & \mathsf{letrec}\ f_1(x_1) = e_1\ \mathsf{and}\ f_2(x_2) = e_2\ \mathsf{in}\ e \end{array}$$

The static semantics of this construct can be defined as follows.

$$\frac{\begin{array}{c} \Gamma' = \Gamma, f_1 : A_1 \to B_1, f_2 : A_2 \to B_2 \qquad A_1 = \langle \tau_1, q_1 \rangle \qquad A_2 = \langle \tau_2, q_2 \rangle \\ \Gamma = |\Gamma| \qquad \Gamma', x : \tau_1; q_1 \vdash e_1 : B_1 \qquad \Gamma', x : \tau_2; q_2 \vdash e_2 : B_2 \qquad \Gamma', q \vdash e : \langle \tau, q' \rangle \end{array}}{\Gamma; q \vdash \mathsf{letrec}\ f_1(x_1) = e_1\ \mathsf{and}\ f_2(x_2) = e_2\ \mathsf{in}\ e : \langle \tau, q' \rangle} \text{ (L:FUN)}$$

References

[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.