Amortized Resource Analysis

Martin Hofmann

Ludwig-Maximilians-Universität München

EWSCS Winter School 2011, Palmse, Estonia

Why resource analysis

- Computing under severe resource restrictions: mobile phones, wireless sensors, embedded systems, smart cards.
- Issues of trust related to resources: grid & cloud computing (local computing power, bandwidth), active networking, mobile agents.
- Forge-proof certification of resource needs.
- Static analysis of resource needs to guarantee survival of sandboxing.
- Terminating a program due to resource bound violation may not be feasible

Type systems

Verification of arbitrary programs is infeasible and undecidable in general.
Verification of simple properties of sensibly written programs can be quite easy.
Type systems highlight simple properties of sensibly written programs.
Examples:

- Java type system prevents uncaught exceptions, certain segmentation faults, etc.
- ML type system often prevents one from supplying arguments to a function in the wrong order etc. ("If it typechecks it works")
- Abstract types promote modularity by preventing breaches of abstraction

Approaches to resource analysis

- Testing and extrapolating (Unnikrishnan et al)
- Inserting counter variables, analysing their values (Gulwani, SPEED)
- Extraction of recurrences (Grobauer, Hermenegildo, COSTA)
- Abstract interpretation for concrete fixed WCET bounds (Wilhelm, Ferdinand, ABSINT)
- Amortized analysis: this work

These lectures

- Automatic type-based inference of resource bounds for programs with
  - recursive data structures (lists, trees)
  - recursive functions
  - composition of helper functions via intermediate data structures
- We do not improve the precise analysis of basic blocks or programs with a simple loop structure;
- Method is parametric in cost model, can interact with complementary approaches for basic blocks.

Examples

Heap-, stack, time, general resource bounds for

- Functional programs on inductive data: Insertion sort, quicksort (destructive and non-destructive), tree sort, Huffman
- OO-versions of these programs
- Larger programs: filters from signal processing, block booking text messages.
- Programs with polynomial resource usage: matrix operations, dynamic programming, e.g. longest common subsequence

Summary

- Resource analysis
- Amortized complexity according to Tarjan
- Inference of linear resource bounds according to [H-Jost '03].
- A glance on objects
- Polynomial resource annotations with binomial coefficients
- Multivariate polynomial resource annotations
- Conclusion & wrap up

Joint work with Klaus Aehlig, Jan Hoffmann, Steffen Jost, Dulma Rodriguez.

Amortized Cost

- Introduced by R. Tarjan in the 70s to facilitate cost analysis of algorithms that repeatedly access a data structure.
- assign (nonnegative) potential to states of the data structure
- define *amortized cost* of a single operation as actual cost + potential difference.
- sum of amortized costs + potential of initial data structure bounds actual cost of sequence of operations.
- Usually, one arranges things so that amortised cost of operations are constant for then no sizes of intermediate data structures need to be maintained.

Queue as two stacks

A FIFO queue Q can be implemented with two stacks S_{in} and S_{out}:

Put, Get

Put(Q, x)

    Push(S_{in}, x)

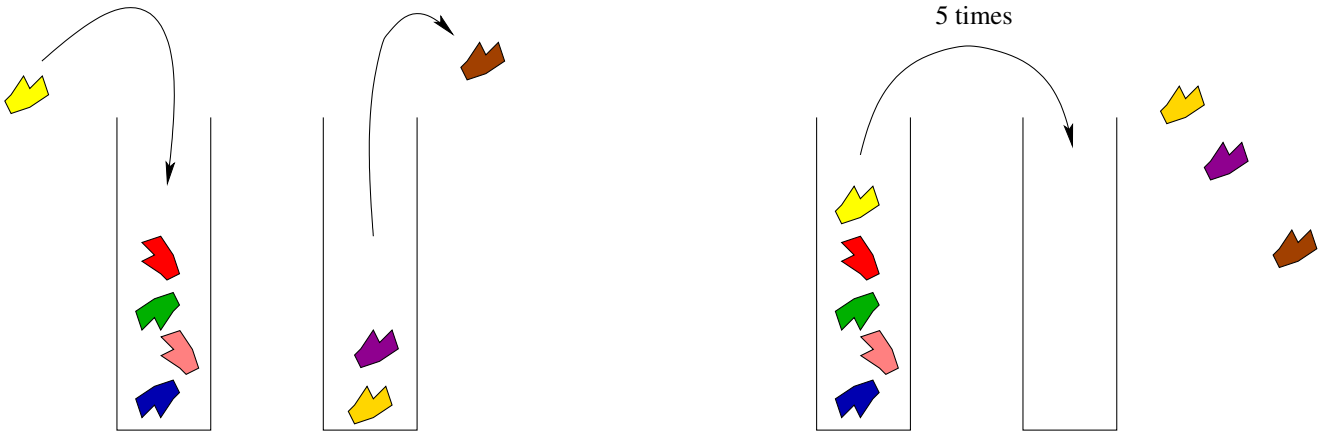
Get(Q)

    if Empty(S_{out})
      then while not Empty(S_{in}) **do**
                Push$(S_{\text{out}}, \text{Pop}(S_{\text{in}}))$
    return Pop(S_{out})

Cost of PUT and GET

How many stack operations do the queue operations need in the worst-case?

$\textsc{Put}(Q, x)$:	1
$\textsc{Get}(Q)$:	$2n + 1$



Actual cost

. . . of a sequence of m Put/Get-operations is $O(m)$ and not $O(m^2)$, because every element is moved at most 3 $\times$

Justification: assign potential 2 to each element of entry stack. Amortized cost of PUT: 3 (overcharge by 2)
Amortized cost of GET: 1 (pay expensive case from saved capital)

Automatic inference

- Our approach: work with potential that is an *unknown* nonnegative linear combination of nonnegative basis functions
- use type-based analysis to infer linear bounds on the coefficients.
- use standard linear programming package to solve these constraints.

H-Jost '03: linear potentials

- Refined type system for first-order functional programs with lists (and other inductive data),
- A refined type A determines a function assigning to each value of that type a nonnegative potential $\Phi_A(v)$.
  For example, for the refined list type $L^{(7)}(int)$ refining the type $L(int)$ of integer lists one has

$$\Phi_{L^{(7)}(int)}([3;4;5;6] = 28$$

  I.e. 7 units per list entry.
  More interestingly, for the refined type $A = L^{(7)}(L^{(3)}(int), L^{(2)}(int))$ one has

$$\Phi_A([(l_1,k_1);(l_2,k_2);\ldots;(l_j,k_j)]) = 7j + 3\sum_i |l_i| + 2\sum_i |k_i|$$

Cost models

Cost is defined by an instrumented operational semantics:

$$S, h \vdash^{k}_{k'} e \rightsquigarrow v, h'$$

means that given environment $S = x_1{=}v_1, \ldots, x_n{=}v_n$ and heap h then the evaluation of e results in value v and result heap h' and $k, k' \geq 0$ describe resource usage in the following way:

- Imagine a resource counter that counts free resources (freelist, egg timer, ...).
- If prior to execution the counter's value is $\geq k$ then the counter will not run dry (become negative) during execution and at the end of the execution its value will be $\geq k'$.
- Alternative reading: temporary resource (space) usage will be $\leq k$ (high watermark). Net resource usage upon termination will be $\leq k - k'$.
- For timelike resources we can always assume $k' = 0$; for space-like resources we can have $k = k' = 1$: net resource usage zero, high watermark 1 (allocation of 1 cell followed by deallocation before termination).

Typing judgement

The typing judgement takes the form

$$x_1{:}A_1, \ldots, x_n{:}A_n \vdash^{q}_{q'} e : B$$

and the typing rules are set up in such a way that if

$$x_1{=}v_1, \ldots, x_n{=}v_n, h \vdash e \rightsquigarrow v$$

then for each $k \geq \sum_i \Phi_{A_i}(v_i, h) + q$ there exists $k' \geq q' + \Phi_B(v, h')$ such that $x_1{=}v_1, \ldots, x_n{=}v_n, h \vdash^{k}_{k'} e \rightsquigarrow v$.

Thus, temporary resource usage $\leq \sum_i \Phi_{A_i}(v_i, h) + q$ and total resource usage $\leq \sum_i \Phi_{A_i}(v_i, h') + q - q' - \Phi_B(v, h')$.
Total resource usage may be negative (deallocation).

Typing rules

- Allocation must be paid for: $x{:}A, y{:}L^{(r)}(A) \vdash^{r+1}_{0} x :: y : L^{(r)}(A)$;
- Pattern matching frees potential: If $\Gamma, x{:}A, y{:}L^{(r)}(A) \vdash^{q+r}_{q'} e : C$ then $\Gamma, l{:}L^{(r)}(A) \vdash^{q}_{q'} \mathit{match}\ l\ \mathit{with}\ [] \to \cdots \mid x :: y \to e : C$
- Destructive pattern matching even allows one to reclaim resource: If $\Gamma, x{:}A, y{:}L^{(r)}(A) \vdash^{q+r+1}_{q'} e : C$ then $\Gamma, l{:}L^{(r)}(A) \vdash^{q}_{q'} \mathit{match}\ l\ \mathit{with}\ [] \to \cdots \mid x :: y@\_ \to e : C$
- Duplicating a variable requires splitting of potential: can use a variable $x : L^{(7)}(\mathit{int})$ twice: once with type $L^{(3)}(\mathit{int})$ and once with type $L^{(4)}(\mathit{int})$. But *not* twice with type $L^{(7)}(\mathit{int})$!
- Otherwise, typing rules are pretty standard.

Remark: instead of $+1$ one can charge other quantities. Charging for operations other than allocation $\rightsquigarrow$ general resource analysis.

Typing rules: more detail

Function symbols are declared with types like $A_1, \ldots, A_\ell \xrightarrow{q/q'} B$. We then expect the judgement

$$x_1{:}A_1, \ldots, x_l{:}A_l \vdash^{q}_{q'} e : B$$

to hold for the body e of `f`.
Slogan: Assume any typing for function symbols. Justify it for the body. Use the assumed typing judgement for recursive calls.

Some (expected) typing rules

$$\frac{\Gamma_1 \vdash^{q}_{q_0} e_1{:}A \qquad \Gamma_2, x{:}A \vdash^{q_0}_{q'} e_2{:}C}{\Gamma_1, \Gamma_2 \vdash^{q}_{q'} \texttt{let}\ \ x{=}e_1\ \texttt{in}\ \ e_2{:}C} \qquad \frac{q \geq \underline{1} + k + q'}{\Gamma, x_h{:}A, x_t{:}L^{(k)}(A), n \vdash^{q}_{q'} x_h :: x_t{:}L^{(k)}(A)}$$

$$\frac{\begin{array}{c}\texttt{f} : A_1, \ldots, A_p \xrightarrow{k/k'} C \\ q \geq k \qquad q - k + k' \geq q'\end{array}}{\Gamma, x_1{:}A_1, \ldots, x_p{:}A_p \vdash^{q}_{q'} f(x_1, \ldots, x_p){:}C}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{q}_{q'} e_1 : C \\ \Gamma, x_h{:}A, x_t{:}L^{(k)}(A), q+a+k \vdash^{q+a+k}_{e} {}_2 : C, q' \\ a = 1 \text{ if } \spadesuit \text{ is } \texttt{@\_} \text{ and } 0 \text{ otherwise.}\end{array}}{\Gamma, x{:}L^{(k)}(A) \vdash^{q}_{q'} \texttt{match } x \texttt{ with} [] \texttt{=>} e_1 \mid (x_h :: x_t)\spadesuit \texttt{=>} e_2 : C}$$

$$\frac{\curlyvee(A \mid A_1, A_2 \mid \quad)\Gamma, x{:}A_1, y{:}A_2 \vdash^{q}_{q'} e : C}{\Gamma, z{:}A \vdash^{q}_{q'} e[z/x, z/y] : C}$$

The sharing judgement $\curlyvee(A \mid A_1, A_2 \mid)$ asserts that a variable of type A can be used twice: once with type A_1 and then again with type A_2. We have e.g. $\curlyvee(L^{(5)}(bool) \mid L^{(2)}(bool), L^{(3)}(bool) \mid)$.

Example

$$\text{append}(l, ys) = \textbf{match } l \textbf{ with } \mid \text{nil} \rightarrow ys \mid (x :: xs) \rightarrow \textbf{let } l' = \text{append}(xs, ys) \textbf{ in } x :: l'$$

can be given the type

$$\textit{append: } (L^{(r+1)}(A), L^{(r)}(A)) \xrightarrow{0/0} L^{(r)}(A).$$

Likewise:

$$\text{attach}(x, l) = \textbf{match } l \textbf{ with } \mid \text{nil} \rightarrow \text{nil} \mid (y :: ys) \rightarrow (x,y)::(\text{attach } (x, ys))$$

can be given the type

$$\textit{attach: } (\textit{int}, L^{(2)}(\textit{int})) \xrightarrow{0/0} L^{(0)}(\textit{int}, \textit{int}).$$

Tree sort

```
type tree = !Leaf | Node of int * tree * tree

let insert x t = match t with
  Leaf => Node(x,Leaf,Leaf)
| Node(y,l,r)@_ => if x<=y then Node(y,insert x l,r)
                           else Node(y,l,insert x r)

let mk_tree l = match l with [] => Leaf
                | (y::t) =>insert y (mk_tree t)

let append u v = match u with [] => v | (y::t)@_ => y::append t v

let extract t = match t with Leaf => []
     | Node(x,l,r) => append (extract l) (x::extract r)
```

Scaffolding

```
let intlist_of_stringlist l = match l with
[] => []
| y::t => int_of_string y :: intlist_of_stringlist t

let print_list_aux l= match l with
 [] => ()
| y::t => let _ = print_string ", " in let _=print_int y in
print_list_aux t

let print_list l = let _ = print_string "[" in
match l with [] => print_string "]\n"
| y::t => let _= print_int y in let _=print_list_aux t in print_str
"]\n"

let start args = print_list (extract (mk_tree (intlist_of_stringlis
```

Analysing

With

```
camelot -a2 tree.cmlt > tree.lfd
```

we generate (readable) intermediate code in monomorphised `let` normal form:
With

```
lfd_infer -orhs 4 tree.lfd
```

we invoke the analysis.

tree.lfd

```
type  list_2 = Cons$_2(*1*) of string * list_2 | Nil$_2(*0*)
type  list_1 = Cons$_1(*1*) of int * list_1 | Nil$_1(*0*)
type  tree = Leaf(*0*) | Node(*1*) of int * tree* tree

val insert: (int ) -> (( tree) -> ( tree) )
val int_of_string: string -> int
...
let start args =
let ?t13 = intlist_of_stringlist (args)
in let ?t12 = mk_tree (?t13)
in let ?t11 = extract (?t12)
in print_list (?t11)
```

Output

```
append_1 : <0>, list_1(int,$,<0>|<0>) -> list_1(int,$,<0>|<0>) ->
               list_1(int,$,<0>|<0>), <0>;
extract        : <0>, tree(<0>|int,$,$,<1>) -> list_1(int,$,<0>|<0
insert         : <2>, int -> tree(<0>|int,$,$,<1>) -> tree(<0>|int
intlist_of_stringlist : <0>, list_2(string,$,<3>|<0>) ->
                                 list_1(int,$,<2>|<0>), <0>;
mk_tree        : <0>, list_1(int,$,<2>|<0>) -> tree(<0>|int,$,$,<1
print_int      : <0>, int -> unit, <0>;
print_list     : <0>, list_1(int,$,<0>|<0>) -> unit, <0>;
print_list_aux : <0>, list_1(int,$,<0>|<0>) -> unit, <0>;
print_string   : <0>, string -> unit, <0>;
start          : <0>, list_2(string,$,<3>|<0>) -> unit, <0>;
```

Fractional solutions

```
let test1 l = match l with
[] => [0] | y::t => test2 y t
and test2 y l = match l with
[] => [0] | z::t => test3 y z t
and test3 y z l = match l with
[] => [0] | w::t => y::z::test1 t
```

yields

```
test1: <1>, list_1(int,$,<0.67>|<0>) -> list_1(int,$,<0>|<0>), <0>;
test2: <1.67>, int, list_1(int,$,<0.67>|<0>)->list_1(int,$,<0>|<0>)
test3: <2.33>, int, int, list_1(int,$,<0.67>|<0>) ->
                    list_1(int,$,<0>|<0>), <0>;
```

Other examples include . . .

- sorting algorithms including quicksort,
- Binary search trees of various kinds,
- functional heaps,
- Huffman coding,
- Algorithms from signal processing (EmBounded project),
- small mobile phone applications (MOBIUS project)

Type soundness

- Our goal is to prove that if $\Gamma \vdash^{q}_{q'} e : A$ and $S, h \vdash e \rightsquigarrow v, h'$ then for each $k \geq \sum_i \Phi_{A_i}(v_i, h) + q$ there exists $k' \geq q' + \Phi_B(v, h')$ such that $x_1{=}v_1, \ldots, x_n{=}v_n, h \vdash^{k}_{k'} e \rightsquigarrow v$.
- In order to prove this we must additionally assume that S, h are in accordance with Γ, e.g. contain lists not trees, etc.
- We also have to introduce an extra additive parameter: $\forall t \forall k \geq t + \sum_i \ldots \exists k' \geq t + \ldots$
- We also have to prove that potentials outside the "footprint" of the computation of e are not affected. Do this by quantifying over extra context Δ.
- Finally, the soundness is proved by induction on $\Gamma \vdash e \rightsquigarrow v$ (can't use induction on typing derivations)

Inference of these annotations

We adopt the following approach due to Palsberg, Schwartzbach, Henglein (and many others):

- Construct a skeleton derivation containing variables wherever a numerical annotation is required.
- Collect all the inequality constraints arising as side-conditions in typing rules.
- Solve constraints using suitable algorithm: here simplex algorithm (e.g. `lp_solve`)

An aside on certification

- We can use typing derivations to generate unforgeable certificates of resource consumption
- Translate typing assertions into formalised program logic assertions of a specific format
- Have done this with respect to a formalisation of Java bytecode and a formalised (Coq, Isabelle) program logic.

Summary

- Types define linear potential by (intuitively) attaching credits to nodes of data structures.
- Typing rules formulated in such a way that potential is correctly passed around, in particular not spuriously created
- Careful treatment of aliasing
- Type inference by extracting constraints and linear programming
- Examples include list and tree processing textbook functions

A glance at objects

We consider a version of Java with imperative update and explicit deallocation in the style of C / C++. We assume a free-list from which cells are taken upon object creation and to which deallocated objects are returned for future use. Our goal is to infer an upper bound as a function of the input on the size of a free-list required to successfully evaluate a program phrase.

We require that "free" is used benignly in that dangling pointers are never accessed or compared against.

Small example

```
class List1 { /** @ann = 1 */ boolean nil; int hd; List1 tl }
```

Each list node carries a potential of 1, a list of length 7 has a potential of 7.
We use a typing judgement of the form

$$\Gamma \vdash^{\frac{m}{m'}} e : A$$

which means that if $E, h \vdash e \rightsquigarrow v, h'$ with unlimited freelist then a freelist whose size exceeds

$$m + \sum_{x:\mathrm{dom}(()\Gamma)} \Phi_h(E(x) : \Gamma(x))$$

will suffice for successful evaluation and the freelist size upon completion will exceed $m' + \Phi_{h'}(v : A)$.
Typing rule for `new`:

$$\frac{}{\emptyset \vdash^{\frac{1+p}{0}} \texttt{new A()} : A}$$

here p is the numerical annotation of class A, e.g., $p = 1$ in the example.

Reclaiming potential

Q Shouldn't you get the potential back only when you destroy an object?
A Reclaiming potential in this case would be sound but not sufficient, e.g., it would prevent us from writing a clone function that nondestructively copies a data structure and in doing so strips off its potential. See next example.
Q But what if you call a method more than once? Do you get a multiplication of potential.
A No you don't. Our sharing rules will ensure that the second time around the callee has a different type which carries less if any potential.
Q Why can you then not strip off potential without actually calling a method? Well, "`this`" is the only pointer of which you know it's not `null`. In a language with a separate category of non-null pointers or without `null` altogether this would be possible.

Example cont'd

Let us also form

```
class List0 { /** @ann = 0 */ boolean nil; int hd; List0 tl }
```

Now we can add to List1 a method:

```
List0,0 copy(0) {
  List0 result = new List0();
  if (this.nil) result.nil=true;
  else { result.nil=false;
         result.hd = this.hd;
         result.tl = this.tl.copy();
        }
  return result;}
```

This typing shows us that the heap consumption of copy is equal to the potential of the callee, i.e. one plus its length.

Motivation cont'd

For purely functional data structures this was already possible with the H-Jost type system.
We want to extend this to Java, that is Featherweight Java extended with field update (FJEU). No exceptions, threads, arrays, so far.
This should be done in such a way that the usual H-Jost typings for lists, trees, etc can be recovered when using the OO-encodings of these data structures.
In addition we want to be able to analyse more imperative programming patterns such as iterator or visitor.
Ideally, amortised analysis of imperative data structures, e.g. Fibonacci heaps should be subsumed.
Our analysis should allow arbitrary aliasing, even circular data.

Sketch of system

An annotation of a given FJEU program P will consist of a set of *views*. If C is a class of P then for each view r we get an annotated version C^r of that class.
Thus, there is an annotation function that maps classes C and views r to annotations $\Diamond(C^r) \in \mathbb{R}^+$ and if in addition a is a field of A then we get two views $\mathrm{A}^{\mathrm{get}}(C^r, a)$ and $\mathrm{A}^{\mathrm{set}}(C^r, a)$ specifying the view of the field a.
While the class type of the attribute specified in C is independent of access mode there will be a different (stronger) view for write access than for read access.
Finally, for each method m in C and view r there must be a (possibly empty) set of annotations of the form $r_1, \ldots, r_j \xrightarrow{p/q} r_0$.
Subtyping of annotated classes is covariant w.r.t. $\Diamond(\cdot)$, $\mathrm{A}^{\mathrm{get}}(\cdot\,,\cdot)$ and w.r.t. result types of methods. It is contravariant w.r.t. argument types of methods and $\mathrm{A}^{\mathrm{set}}(\cdot\,,\cdot)$.

Example: OO-Lists

```
abstract class List { abstract List copy(); }

class Nil extends List {
    List copy(){
        return this; } }

class Cons extends List {
    int car;
    List cdr;

    List copy() {
        Cons res = new Cons();
        res.car = this.car;
        res.cdr = this.cdr.copy();
        return res; } }
```

Two views for OO-Lists

```
view rich is
   List:0
      List<poor>,0 copy(0)
   Nil:0
   Cons:1
      List<get:rich,set:rich> cdr

view poor is
   List:0
   Nil:0
   Cons:0
      List<get:poor,set:poor> cdr
```

Of course, we must justify the announced typing of copy using the typing rules.

The potential

Unlike in previous work, the potential of a value is not defined inductively (and not coinductively either).
Instead, we define the potential as an infinite sum over all *access paths*.
An access path is a finite word over field names.
If $\vec{p}$ is an access path, h is a heap (mapping of locations to objects), v is a value (location or `null`.) and r is a view (obtained from the static typing of v) then

$$\varphi_h\big((v{:}r).\vec{p}\big) = \Diamond(D^s)$$

where D is the dynamic type of $v \rightsquigarrow \vec{p}$ and s is the view obtained by chaining r through the various dynamic types encountered along $\vec{p}$ using $\mathrm{A}^{\mathrm{get}}(\cdot\,,\cdot)$.
If $v \rightsquigarrow \vec{p}$ is undefined or null then $\varphi_h\big((v{:}r).\vec{p}\big) = 0$.

Potential cont'd

The potential of v w.r.t. h and r is now defined by

$$\Phi_h(v : r) = \sum_{\vec{p}} \varphi_h\big((v{:}r).\vec{p}\big)$$

The potential is always an infinite sum; in most cases the summand will vanish almost everywhere.
This can happen even with cyclic data structures.

Example: OO-Lists

If v points to a chain of three consecutive `Cons` objects followed by a `Nil` object then

$$\begin{aligned}
&\varphi_h\big((v{:}\texttt{rich}).\epsilon\big) = \Diamond\big(\texttt{Cons}^{\texttt{rich}}\big) = 1\\
&\varphi_h\big((v{:}\texttt{rich}).\texttt{cdr}\big) = 1\\
&\varphi_h\big((v{:}\texttt{rich}).\texttt{cdr}.\texttt{cdr}\big) = 1\\
&\varphi_h\big((v{:}\texttt{rich}).\texttt{cdr}.\texttt{cdr}.\texttt{cdr}\big) = \Diamond\big(\texttt{Nil}^{\texttt{rich}}\big) = 0\\
&\varphi_h\big((v{:}\texttt{rich}).\texttt{cdr}.\texttt{cdr}.\texttt{cdr}.\texttt{cdr}^*\big) = 0
\end{aligned}$$

Therefore $\Phi_h(v : \texttt{rich}) = 3$ but $\Phi_h(v : \texttt{poor}) = 0$.

Circular list

Let us add a third view:

```
view middle is
   List:0
   Nil:0
   Cons:1
      List<get:poor,set:poor> cdr
```

If v points to a circular list of `Cons` objects then

$$\Phi_h(v : \texttt{poor}) = 0$$
$$\Phi_h(v : \texttt{middle}) = 1$$
$$\Phi_h(v : \texttt{rich}) = \infty$$

Tail-recursive append

Let's augment the `List` class with methods

```
abstract void append_aux(List y, Cons res);
abstract List append(List y);
```

The idea being that `x.append_aux(y,res)` appends `y` to a fresh copy of `x` and writes the result into `res.cdr`.
The point is that this can be implemented tail recursively ("destination passing style").
`append` then temporarily allocates a `Cons` object to invoke `append_aux` with.

```
/* in List */
 abstract void append_aux(List y, Cons res);
 List append(List y) {
      Cons res = new Cons();
      this.append(y,res);List result = res.cdr;
      free(res);return result; }
/* in Nil */
 void append_aux(List y, Cons res) { res.cdr = y;}
/* in Cons */
 void append_aux(List y, Cons res) {
     Cons hd = new Cons();
     hd.car = this.car; res.cdr = hd;
     this.cdr.append_aux(y,hd); }
```

We can annotate this in the `rich` view as

```
void,0 append(List<poor>,Cons<poor>,0)
void,1 append(List<poor>,1)
```

Some typing rules

- Upon object creation (`new`) one must pay not only the actual cost (size of the object to be constructed) but also the annotation, e.g. 1 in the case of `Cons<rich>`.
- In the body of a method one gets access to the annotation of the callee, however it must be shared with possible uses of `this` in the method body.
- In a deallocation (`free`) one gets access to both the annotation and the actual size of the object (assumed equal to 1).
- To prevent multiple access to annotations via multiple method calls, we use a linear type system with an explicit sharing rule:

Sharing

$$\frac{\curlyvee(s\,|\,q_1, q_2) \qquad \Gamma, y{:}D^{q_1}, z{:}D^{q_2} \vdash^{n}_{n'} e : C^r}{\Gamma, x{:}D^s \vdash^{n}_{n'} e[x/y, x/z] : C^r}(\Diamond\textsc{Share})$$

Here $\curlyvee(\cdot\,|\,\cdot)$ is a coinductively defined relation between views and multisets of views. We have in particular $\curlyvee(\texttt{poor}\,|\{\texttt{poor}, \texttt{poor}, \dots\})$, but not $\curlyvee(\texttt{rich}\,|\{\texttt{rich}, \texttt{rich}\})$.
We do not even have $\curlyvee(\texttt{rich}\,|\{\texttt{rich}, \texttt{poor}\})$, but we do have $\curlyvee(\texttt{rich}\,|\{\texttt{rich}, \texttt{poorest}, \texttt{poorest}, \texttt{poorest}\})$.

```
view poorest is
   List:0
   Nil:0
   Cons:0
      List<get:poor,set:rich> cdr
```

Of course, $\texttt{rich} <: \texttt{poor} <: \texttt{poorest}$.

The update rule

$$\frac{\mathrm{A}^{\mathrm{set}}(C^r, a) = s \qquad C.a = D}{x{:}C^r, y{:}D^s \vdash^{0}_{0} x.a{=}y : C^r}(\Diamond\textsc{Update})$$

In order to update a field you must have enough resources available to feed all the different paths that might lead into this field.
Thus, if x has type `List<poorest>` then in

```
x.cdr = y;
```

one must have y :`List<rich>`. After all, this code could have been preceded by

```
x = z;
```

with "rich" `z`. Then after the assignment we would still expect z to be "`rich`" and fortunately it is!

The soundness theorem

The statement of the soundness theorem is similar to [H-Jost 2003].
If $\Gamma \vdash^{n}_{n'} e : C^r \quad \eta, h \vdash e \rightsquigarrow v, \tau \quad h \vDash \eta : (\Gamma, \Delta)$ then

$$\eta, h \vdash^{n + \Phi_h(\eta : \Gamma) + \Phi_h(\eta : \Delta)}_{n' + \Phi_\tau(v : r) + \Phi_\tau(\eta : \Delta)} e \rightsquigarrow v, \tau \tag{1}$$

$$\tau \vDash \eta[x_{res} \mapsto v] : (\Delta, x_{res} : C^r) \tag{2}$$

Here Δ is an arbitrary context representing other parts of the program that may share with the currently focused on heap portion.

Heap consistency

The judgement $h \vDash \eta : \Theta :$ subsumes ordinary heap soundness in the sense of Nipkow but also states that for each location ℓ there exists an proto-view r_ℓ such that $\Downarrow(r \,|\mathscr{L}_{h,\eta,\Gamma}(\ell))$ where $\mathscr{L}_{h,\eta,\Gamma}(\ell) = \{ \mathbb{|}(\eta_x{:}\langle\!\langle \Gamma_x \rangle\!\rangle).\vec{p} \mathbb{|}_h^{\mathrm{stat}} \mid x \in \Gamma, [\![\eta_x.\vec{p}]\!]_h = \ell \}$

Operational semantics

The judgement $\eta, h \vdash e \rightsquigarrow v, \tau$ asserts that e evaluates to v in stack η and heap h with result heap τ and that

no disposed location is ever returned by "new"

This is formalised by marking disposed locations as "invalid" rather than actually removing them from the heap.

In reality this does not happen, however, we claim that type systems such as alias types or lightweight program verification can be used to show that actual evaluation is equivalent to this idealised one.

The crux is: we must prevent that stale pointers accidentally become accesses to freshly allocated locations.

Main difficulty: the update rule $\texttt{this}.a := x$

We define $\mathit{meets}(h, v, p, \ell, a)$ as "there is a proper prefix q of p such that $[\![v.q]\!]_h = \ell$ and qa is a prefix of p."
We define $\mathcal{P}(\ell_1, \ell_2) = \{p \mid [\![\ell_1.p]\!] = \ell_2 \wedge \neg \mathit{meets}(h, \ell_1, p, \ell, a)\}$
Here ℓ, a are fixed.
Let $\ell = \eta(\texttt{this})$ and $v = \eta(x)$. Recall $h' = h[\ell.a \mapsto v]$.
The set of p such that $[\![\ell_1.p]\!]_{h'} = \ell_2$ equals

$$\mathcal{P}(\ell_1, \ell_2) \cup \mathcal{P}(\ell_1, \ell)(a\mathcal{P}(v, \ell))^* a\mathcal{P}(v, \ell_2)$$

This allows us to obtain knowledge about paths in h' from knowledge about h and thus proceed with the proof.

More examples

- Doubly-linked lists: even in the `rich` version the `back`-pointers are `poor` so that only the access paths of the form $\texttt{next}^*$ contribute.
- Iterators on doubly linked lists: as soon as you move the iterator backwards it changes view so no more potential can be extracted.

Planned examples: visitor, subject-observer, union-find.

Summary (objects)

- Refinement of class types into several views that carry annotations
- Views cut across the class hierarchy because one and the same object can have different class types
- Potentials are calculated on the basis of access paths rather than shapes of data structures
- Type inference based on constraint solving and view elimination underway but not finished yet (PhD Dulma Rodriguez)
- Type system should be useful for tasks other than resource analysis (cf. HMX by Thiemann et al)

Need for polynomial bounds

A type like

$$f\colon (L^{(7)}(A), L^{(2)}(A)) \xrightarrow{3/0} L^{(0)}(A).$$

yields a linear resource bound (resource usage $= 3 + 7m + 2n$ where $m =$ length of first argument and $n =$ length of second argument.
All resource bounds obtainable in the functional system are of that form.
Thus, functions with quadratic resource usage cannot be analysed.
Example:

```
pairs l = match l with | nil       → nil
                       | (x :: xs) → append(attach(x,xs),pairs xs)
```

The expression *pairs([1,2,3,4])* evaluates to the list *[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]* (quadratic number of allocations.)

Polynomial resource bounds

Annotate (list) types with a tuple of nonnegative numbers (not just one):

$\mathsf{attach}(\mathsf{x},\mathsf{l}) = \textbf{match } \mathsf{l} \textbf{ with } \mid \mathsf{nil} \rightarrow \mathsf{nil}$
$\quad \mid (\mathsf{y} :: \mathsf{ys}) \rightarrow \textbf{let } \mathsf{l'} = \mathsf{attach}(\mathsf{x},\mathsf{ys}) \textbf{ in } (\mathsf{x},\mathsf{y}) :: \mathsf{l'}$

$\mathsf{attach}\colon (int, L^{(p+2)}(int)) \xrightarrow{c/c} L^{(p)}(int)$
$\mathsf{l}\colon L^{(p+2)}(int) \quad \mathsf{ys}\colon L^{(p+2)}(int) \quad \mathsf{l'}\colon L^{(p)}(int)$

$\mathsf{pairs}(\mathsf{l}) = \textbf{match } \mathsf{l} \textbf{ with } \mid \mathsf{nil} \rightarrow \mathsf{nil}$
$\quad \mid (\mathsf{x} :: \mathsf{xs}) \rightarrow \textbf{let } \mathsf{nps} = \mathsf{attach}(\mathsf{x},\mathsf{xs}^1) \textbf{ in}$
$\qquad\qquad \textbf{let } \mathsf{rps} = \mathsf{pairs}(\mathsf{xs}^2) \textbf{ in } \mathsf{append}(\mathsf{nps},\mathsf{rps})$

$\mathsf{pairs}\colon L^{(0,p_2+3)}(int) \xrightarrow{c/c} L^{(p_2)}(int)$
$\mathsf{l}\colon L^{(0,p_2+3)}(int) \quad \mathsf{xs}^1\colon L^{(p_2+3)}(int) \quad \mathsf{rps}\colon L^{(p_2)}(int)$
$\mathsf{xs}\colon L^{(p_2+3,p_2+3)}(int) \quad \mathsf{xs}^2\colon L^{(0,p_2+3)}(int) \quad \mathsf{nps}\colon L^{(p_2+1)}(int)$

Meaning of the tuples

$$\Phi_{L^{(p_1,\ldots,p_k)}(A)}([v_1;\ldots;v_n]) = np_1 + \binom{n}{2}p_2 + \cdots + \binom{n}{k}p_k + \sum_i \Phi_A(v_i)$$

E.g.: $\Phi_{L^{(2,3)}(int)}([v_1;\ldots;v_n]) = 2n + 3\binom{n}{2}$

Shifting

Write $\varphi(\vec{p}, n) = \sum_{i=1}^{k} p_i \binom{n}{i}$ when $\vec{p} = (p_1, \ldots p_k)$.

- $\varphi(\vec{p}, 0) = 0$
- $\varphi(\vec{p}, n+1) = p_1 + \varphi(\vec{p}, n) + \varphi(\partial\vec{p})$

where $\partial(p_1, \ldots, p_k) = (p_2, \ldots, p_k)$.

Examples

- Computing list of all pairs (quadratic heap usage)
- Computing all subsets of size k (heap usage $O(n^k)$)
- Dyadic products (heap usage $O(n^2)$)
- Nested "fold" (heap usage $O(n^2)$)
- Longest common subsequence (requires size $O(n^2)$ dynamic programming table).

Nonnegative linear combinations of binomial coefficients

The polynomials arising as resource bounds are linear combinations of binomial coefficients with nonnegative coefficients.
For a function f define $\Delta(f) = f(n+1) - f(n)$. Call f hereditarily nonnegative if $\Delta^i f \geq 0$ for all i.
NB: f and Δf nonnegative are strictly necessary.

Theorem

A polynomial p is hereditarily nonnegative iff it is a linear combination of binomial coefficients with nonnegative coefficients.

Remark: the hereditarily nonnegative polynomials are the scalar multiples of the unary resource polynomials from Girard-Scedrov-Scott.

Multivariate polynomials

- So far: potential functions take essentially the form $\sum_i p_i(n_i)$ with p_i *univariate* polynomials and n_i size parameters, e.g., lengths of lists. Multiplicative bounds such as mn must be grossly overapproximated by $m^2 + n^2$.
- Since 2010 we can accommodate arbitrary multivariate polynomials.
- Simplest example:

$$\begin{aligned}\text{dyade } [x_1; \ldots; x_m]\ [y_1; \ldots; y_n] \ = \\ [[x_1y_1; x_1y_2; \ldots; x_1y_n]; \\ \ [x_2y_1; x_2y_2; \ldots; x_2y_n]; \\ \ \ldots \\ \ [x_my_1; x_my_2; \ldots; x_my_n]]\end{aligned}$$

Going multivariate involves a lot of bookkeeping

cf. Tensor calculus.
For each type A define set $\mathcal{P}(A)$ of basic polynomials and set $\mathcal{I}(A)$ to index the basic polynomials (and implicitly a bijection $\mathcal{I}(A) \simeq \mathcal{P}(A)$):

- $\mathcal{P}(int) = \{\lambda x.1\}$ and $\mathcal{I}(int) = \{\star\}$ and $p_\star(x) = 1$.
- $\mathcal{P}(A_1 \times A_2) = \{\lambda(x_1, x_2).p_1(x_1)p_2(x_2) \mid p_1 \in \mathcal{P}(A_1), p_2 \in \mathcal{P}(A_2)$ and $\mathcal{I}(A_1 \times A_2) = \mathcal{I}(A_1) \times \mathcal{I}(A_2)$ and $p_{(i_1,i_2)}(x_1, x_2) = p_{i_1}(x_1)p_{i_2}(x_2)$.
- $\mathcal{P}(L(A)) = \ldots$

The actual potential and resource bounding functions are nonnegative linear combinations of the $\mathcal{P}(A)$ thus we can use the $\mathcal{I}(A)$ as "unknowns" (or take λ_i instead).

Measuring lists

Each $p \in \mathcal{P}(L(A))$ is of the form

$$[a_1; \ldots; a_n] \mapsto \sum_{1 \leq j_1 < j_2 < \cdots < j_k \leq n} p_1(a_{j_1}) \cdot p_2(a_{j_2}) \cdot \cdots \cdot p_k(a_{j_k})$$

Examples where $A = int$:

- $k = 0$: $l \mapsto 1$
- $k = 1$: $l \mapsto n$ where $n = |l|$
- $k = 2$: $l \mapsto \sum_{1 \leq j < j' \leq n} 1 = \binom{n}{2}$
- arbitrary: $l \mapsto \binom{n}{k}$

Measuring lists of lists

An example of $p \in \mathcal{P}(L(L(int)))$:

$$[l_1; \ldots; l_n] \mapsto \sum_{1 \leq j_1 < j_2 \leq n} \binom{|l_{j_1}|}{3} \binom{|l_{j_2}|}{1}$$

Recall that $\mathcal{I}(L(A)) = L(\mathcal{I}(A))$ and

$$p_{[i_1;\ldots;i_k]}([a_1; \ldots; a_n]) = \sum_{1 \leq j_1 < \cdots < j_k \leq n} p_{i_1}(a_{j_1}) \cdot \cdots \cdot p_{i_k}(a_{j_k})$$

Typically, of course, $k = 2, 3$.

Interaction with composition

Suppose that $f : A \to B$ and $g : B \to C$ and let $K_f(a)$ be the cost of computing $f(a)$ and likewise K_g.
Suppose that $p \in \mathcal{P}(A), q \in \mathcal{P}(B), r \in \mathcal{P}(C)$ and

$$\forall a,\ p(a) \geq K_f(a) + q(f(a))$$
$$\forall b,\ q(b) \geq K_g(b) + r(g(b))$$

Then, if $K_{gf}(a) = 7 + K_f(a) + K_g(f(a))$ is the cost of computing $g(f(a))$:

$$\forall a, p(a) + 7 \geq K_{gf}(a) + r(g(f(a)))$$

I.e., we only have constant amortised cost 7.

Parallel composition

The passage from $f : A \to B$ to $f \times C : A \times C \to B \times C$ was trivial in the univariate and linear cases. Not so here:
If

$$\forall a,\ p^{(0)}(a) \geq K_f(a) + q^{(0)}(f(a))$$
$$\forall a,\ p^{(1)}(a) \geq q^{(1)}(f(a))$$
$$\forall a,\ p^{(2)}(a) \geq q^{(2)}(f(a))$$

then

$$\forall a\ c, p(a,c) \geq K_f(a) + q(a,c)$$

where $p(a,c) = \sum_{i=0}^{2} p^{(i)}(a) r^{(i)}(c)$ and $q(a,c) = \sum_{i=0}^{2} q^{(i)}(a) r^{(i)}(c)$ for arbitrary $r^{(i)} \in \mathcal{P}(C)$. And of course 2 is arbitrary here.

Duplication

For each basic polynomial $p \in \mathcal{P}(A \times A)$ the function $t(a) = p(a, a)$ is a nonnegative linear combination of basic polynomials in $\mathcal{P}(A)$.
For example: if $A = L(int)$ and $p(l, l') = |l|\binom{|l'|}{2}$ then

$$t(l) = |l|\binom{|l|}{2} = 3\binom{|l|}{3} + 2\binom{|l|}{2}$$

This allows us to give a typing rule for duplication (aka splitting, contraction).

Putting it together

- Devise typing rules with a priori concrete bounding functions
- Use previous two technical slides for contraction and "let" rules (combines serial and parallel composition)
- Approximate polymorphic recursion by degree-wise monomorphic recursion. To justify a degree $k+1$ typing of a recursive function we may invoke it with that very same degree $k+1$ *superposed* with an arbitrary degree k typing.

Experiments

Function	Computed Evaluation-Step Bound	Simplified Computed Bound	Act. Behav.	Run Time
isortlist: $L(L(int)) \to L(L(int))$	$\sum_{1 \le i < j \le n} 16m_i + 16\binom{n}{2} + 12n + 3$	$8n^2m + 8n^2 - 8nm + 4n + 3$	$O(n^2m)$	0.91 s
nub: $L(L(int)) \to L(L(int))$	$\sum_{1 \le i < j \le n} 12m_i + 18\binom{n}{2} + 12n + 3$	$6n^2m + 9n^2 - 6nm + 3n + 3$	$O(n^2m)$	0.97 s
transpose: $L(L(int)) \to L(L(int))$	$\sum_{1 \le i \le n} 32m_i + 2n + 13$	$32nm + 2n + 13$	$O(nm)$	0.04 s
mmult: $(L(L(int)))^2 \to L(L(int))$	$(\sum_{1 \le i \le x} y_i)(32 + 28n) + 14n + 2x + 21$	$28xyn + 32xy + 2x + 14n + 21$	$O(nxy)$	1.96 s
dyade: $(L(int), L(int)) \to L(L(int))$	$10nx + 14n + 3$	$10nx + 14n + 3$	$O(nx)$	0.03 s
lcs: $(L(int), L(int)) \to int$	$39nx + 6x + 21n + 19$	$39nx + 6x + 21n + 19$	$O(nx)$	0.36 s
subtrees: $T(int) \to L(T(int))$	$8\binom{n}{2} + 23n + 3$	$4n^2 + 19n + 3$	$O(n^2)$	0.06 s
eratos: $L(int) \to L(int)$	$16\binom{n}{2} + 12n + 3$	$8n^2 + 4n + 3$	$O(n^2)$	0.02 s

Try out at raml.tcs.ifi.lmu.de:

```
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
```

future and other work

- Inference for objects (ongoing with Dulma Rodriguez)
- Higher-order functions (Thesis S Jost)
- Garbage collection
- C-programs
- Concurrency
- Other bounding functions (in part. max, log)

PhD in Munich?

- The Munich Graduate School on Program- and Model Analysis (PUMA) has some open PhD positions starting soon.
- Topics in: static analysis (Seidl), model checking (Esparza, Rybalchenko), theorem proving (Nipkow), type systems (MH) (not exclusively)
- Apply anytime at puma.in.tum.de