Foundational aspects of size analysis

Tutorial

M. van Eekelen[1] O. Shkaravska[2]

[1]Institute for Computing and Information Sciences, Radboud University Nijmegen
Faculty of Computer Science, Open University the Netherlands, Heerlen

[2]Institute for Computing and Information Sciences, Radboud University Nijmegen

SICSA International Summer School
on Advances in Programming Languages, Edinburgh 09

Outline

1. Motivation
   - What is “size analysis“?
   - Why do we need size analysis?
2. Size analysis: an overview
   - What is a size dependency?
   - Two problems of analysis: checking and inference
   - Related work
3. Size analysis in AHA project
   - Amortised heap analysis and sizes
   - Size analysis of 1-st order function definitions
   - Beyond shapely programs

Outline

Analyse dependency of the size of an output on the sizes of input

Example: *copy* : $\mathsf{L}(\alpha) \rightarrow \mathsf{L}(\alpha)$

We start with a simple example: an ML-style program that creates a fresh copy of a list:

$copy : \mathsf{L}(\alpha) \rightarrow \mathsf{L}(\alpha)$, e.g. it maps $[1, 2]$ onto $[1, 2]$.

$$copy(l) = \text{match } l \text{ with } \mid \text{nil} \Rightarrow \text{nil}$$
$$\mid \text{cons}(hd, tl) \Rightarrow \text{cons}(hd, copy(tl))$$

Size dependency of *copy*

- Informally: an output has the same length as its input.
- Formally: it maps a list of length n onto a list of length n,
- Very formal: *copy* is of type $\mathsf{L}_n(\alpha) \rightarrow \mathsf{L}_n(\alpha)$

Analyse dependency of the size of an output on the sizes of input

Example: $copy : \mathsf{L}(\alpha) \to \mathsf{L}(\alpha)$

We start with a simple example: an ML-style program that creates a fresh copy of a list:

$copy : \mathsf{L}(\alpha) \to \mathsf{L}(\alpha)$, e.g. it maps $[1,2]$ onto $[1,2]$.

$$copy(l) = \text{match } l \text{ with } \mid \text{nil} \Rightarrow \text{nil}$$
$$\mid \text{cons}(hd, tl) \Rightarrow \text{cons}(hd, copy(tl))$$

Size dependency of *copy*

- Informally: an output has the same length as its input.
- Formally: it maps a list of length n onto a list of length n,
- Very formal: *copy* is of type $\mathsf{L}_n(\alpha) \to \mathsf{L}_n(\alpha)$

Analyse dependency of the size of an output on the sizes of input

Example: $copy : \mathsf{L}(\alpha) \to \mathsf{L}(\alpha)$

We start with a simple example: an ML-style program that creates a fresh copy of a list:

$copy : \mathsf{L}(\alpha) \to \mathsf{L}(\alpha)$, e.g. it maps $[1, 2]$ onto $[1, 2]$.

$$copy(l) = \text{match } l \text{ with } \mid \text{nil} \Rightarrow \text{nil}$$
$$\mid \text{cons}(hd, tl) \Rightarrow \text{cons}(hd, copy(tl))$$

Size dependency of *copy*

- Informally: an output has the same length as its input.
- Formally: it maps a list of length n onto a list of length n,
- Very formal: *copy* is of type $\mathsf{L}_n(\alpha) \to \mathsf{L}_n(\alpha)$

Analyse dependency of the size of an output on the sizes of input

Example: $copy : \mathsf{L}(\alpha) \to \mathsf{L}(\alpha)$

We start with a simple example: an ML-style program that creates a fresh copy of a list:

$copy : \mathsf{L}(\alpha) \to \mathsf{L}(\alpha)$, e.g. it maps $[1,2]$ onto $[1,2]$.

$$copy(l) = \text{match } l \text{ with } \mid \text{nil} \Rightarrow \text{nil}$$
$$\mid \text{cons}(hd, tl) \Rightarrow \text{cons}(hd, copy(tl))$$

Size dependency of *copy*

- Informally: an output has the same length as its input.
- Formally: it maps a list of length n onto a list of length n,
- Very formal: *copy* is of type $\mathsf{L}_n(\alpha) \to \mathsf{L}_n(\alpha)$

Analyse dependency of the size of an output on the sizes of input

Our formalism: annotated types

Size analysis

studies dependencies of the size of an output on the sizes of the corresponding inputs.

Our formalism for size analysis is *annotated type systems*

- *copy* : $L_n(\alpha) \rightarrow L_n(\alpha)$
- *append* : $L_n(\alpha) \times L_m(\alpha) \rightarrow L_{n+m}(\alpha)$
- *insert* : $Int \times L_n(Int) \rightarrow L_{\{n+i\}_{0 \leq i \leq 1}}(Int)$
- etc. ...

Analyse dependency of the size of an output on the sizes of input

Our formalism: annotated types

Size analysis

studies dependencies of the size of an output on the sizes of the corresponding inputs.

Our formalism for size analysis is *annotated type systems*

- $copy : \mathsf{L}_n(\alpha) \rightarrow \mathsf{L}_n(\alpha)$
- $append : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \rightarrow \mathsf{L}_{n+m}(\alpha)$
- $insert : Int \times \mathsf{L}_n(Int) \rightarrow \mathsf{L}_{\{n+i\}_{0 \leq i \leq 1}}(Int)$
- etc. ...

Analyse dependency of the size of an output on the sizes of input

Our formalism: annotated types

Size analysis

studies dependencies of the size of an output on the sizes of the corresponding inputs.

Our formalism for size analysis is *annotated type systems*

- *copy* : $\mathsf{L}_n(\alpha) \rightarrow \mathsf{L}_n(\alpha)$
- *append* : $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \rightarrow \mathsf{L}_{n+m}(\alpha)$
- *insert* : $Int \times \mathsf{L}_n(Int) \rightarrow \mathsf{L}_{\{n+i\}_{0\leq i\leq 1}}(Int)$
- etc. ...

Analyse dependency of the size of an output on the sizes of input

Our formalism: annotated types

Size analysis

studies dependencies of the size of an output on the sizes of the corresponding inputs.

Our formalism for size analysis is *annotated type systems*

- *copy* : $\mathsf{L}_{n}(\alpha) \rightarrow \mathsf{L}_{n}(\alpha)$
- *append* : $\mathsf{L}_{n}(\alpha) \times \mathsf{L}_{m}(\alpha) \rightarrow \mathsf{L}_{n+m}(\alpha)$
- *insert* : $\mathit{Int} \times \mathsf{L}_{n}(\mathit{Int}) \rightarrow \mathsf{L}_{\{n+i\}_{0 \leq i \leq 1}}(\mathit{Int})$
- etc. ...

Outline

1. Motivation
   - What is “size analysis“?
   - Why do we need size analysis?
2. Size analysis: an overview
   - What is a size dependency?
   - Two problems of analysis: checking and inference
   - Related work
3. Size analysis in AHA project
   - Amortised heap analysis and sizes
   - Size analysis of 1-st order function definitions
   - Beyond shapely programs

Size analysis for resource management

Memory resources: heap, stack

Knowing sizes of the structures involved in a computation is necessary:

- in safety and security critical applications: to prevent abrupt termination due to the lack of memory, because output and intermediate structures are too large,
- to optimise memory management, e.g. by allocation in advance junks of a heap.

Size analysis for resource management

Memory resources: heap, stack

Knowing sizes of the structures involved in a computation is necessary:

- in safety and security critical applications: to prevent abrupt termination due to the lack of memory, because output and intermediate structures are too large,
- to optimise memory management, e.g. by allocation in advance junks of a heap.

Size analysis for resource management

Memory resources: heap, stack

Knowing sizes of the structures involved in a computation is necessary:

- in safety and security critical applications: to prevent abrupt termination due to the lack of memory, because output and intermediate structures are too large,
- to optimise memory management, e.g. by allocation in advance junks of a heap.

Size analysis for resource management

Time resources

Knowing sizes helps to predict computation time:

- practice: in the simplest case, the bigger an input is the longer a program runs ...
- theory: termination analysis.

Outline

Size dependency formally: a function, a multivalued function, a relation

- Size dependency may be a numerical function of the sizes of arguments: *append* : $\mathrm{L}_n(\alpha) \times \mathrm{L}_m(\alpha) \rightarrow \mathrm{L}_{f_{append}(n,m)}(\alpha)$, where $f_{append}(n, m) = n + m$,
- It may be a *multivalued* numerical function of the sizes of arguments: *insert* : $Int \times \mathrm{L}_n(\alpha) \rightarrow \mathrm{L}_{f_{insert}(n)}(\alpha)$, where $f_{insert}(n) = \{n, n+1\}$,
- Size dependency may be a relation: *split* : $\mathrm{L}_n(\alpha) \rightarrow \mathrm{L}_{n_1}(\alpha) \times \mathrm{L}_{n_2}(\alpha)$, where $n_1 + n_2 = n$,
- Size dependency may be a function of program arguments directly (dep. types): $makelist(x) : Int \rightarrow \mathrm{L}_{f_{makelist}(x)}(Int)$, where $f_{makelist}(x) = x$,
- etc. ...

Size dependency formally: a function, a multivalued function, a relation

- Size dependency may be a numerical function of the sizes of arguments: $append : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \rightarrow \mathsf{L}_{f_{append}(n,m)}(\alpha)$, where $f_{append}(n, m) = n + m$,
- It may be a *multivalued* numerical function of the sizes of arguments: $insert : Int \times \mathsf{L}_n(\alpha) \rightarrow \mathsf{L}_{f_{insert}(n)}(\alpha)$, where $f_{insert}(n) = \{n, n+1\}$,
- Size dependency may be a relation: $split : \mathsf{L}_n(\alpha) \rightarrow \mathsf{L}_{n_1}(\alpha) \times \mathsf{L}_{n_2}(\alpha)$, where $n_1 + n_2 = n$,
- Size dependency may be a function of program arguments directly (dep. types): $makelist(x) : Int \rightarrow \mathsf{L}_{f_{makelist}(x)}(Int)$, where $f_{makelist}(x) = x$,
- etc. ...

Size dependency formally: a function, a multivalued function, a relation

- Size dependency may be a numerical function of the sizes of arguments: *append* : $L_n(\alpha) \times L_m(\alpha) \to L_{f_{append}(n,m)}(\alpha)$, where $f_{append}(n, m) = n + m$,
- It may be a *multivalued* numerical function of the sizes of arguments: *insert* : $Int \times L_n(\alpha) \to L_{f_{insert}(n)}(\alpha)$, where $f_{insert}(n) = \{n, n+1\}$,
- Size dependency may be a relation: *split* : $L_n(\alpha) \to L_{n_1}(\alpha) \times L_{n_2}(\alpha)$, where $n_1 + n_2 = n$,
- Size dependency may be a function of program arguments directly (dep. types): $makelist(x) : Int \to L_{f_{makelist}(x)}(Int)$, where $f_{makelist}(x) = x$,
- etc. ...

Size dependency formally: a function, a multivalued function, a relation

- Size dependency may be a numerical function of the sizes of arguments: *append* : $L_n(\alpha) \times L_m(\alpha) \rightarrow L_{f_{append}(n,m)}(\alpha)$, where $f_{append}(n, m) = n + m$,
- It may be a *multivalued* numerical function of the sizes of arguments: *insert* : $Int \times L_n(\alpha) \rightarrow L_{f_{insert}(n)}(\alpha)$, where $f_{insert}(n) = \{n, n + 1\}$,
- Size dependency may be a relation: *split* : $L_n(\alpha) \rightarrow L_{n_1}(\alpha) \times L_{n_2}(\alpha)$, where $n_1 + n_2 = n$,
- Size dependency may be a function of program arguments directly (dep. types): $makelist(x) : Int \rightarrow L_{f_{makelist}(x)}(Int)$, where $f_{makelist}(x) = x$,
- etc. ...

Size dependency formally: a function, a multivalued function, a relation

- Size dependency may be a numerical function of the sizes of arguments: *append* $: \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{f_{append}(n,m)}(\alpha)$, where $f_{append}(n, m) = n + m$,
- It may be a *multivalued* numerical function of the sizes of arguments: *insert* $: Int \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{f_{insert}(n)}(\alpha)$, where $f_{insert}(n) = \{n, n+1\}$,
- Size dependency may be a relation: *split* $:$ $\mathsf{L}_n(\alpha) \to \mathsf{L}_{n_1}(\alpha) \times \mathsf{L}_{n_2}(\alpha)$, where $n_1 + n_2 = n$,
- Size dependency may be a function of program arguments directly (dep. types): $makelist(x) : Int \to \mathsf{L}_{f_{makelist}(x)}(Int)$, where $f_{makelist}(x) = x$,
- etc. ...

Correct size dependencies

Formally, the *correctness* of a size dependency is defined w.r.t. the way, we formalise the notion of a size and a size dependency.

A 1-variable multivalued size function f is a correct dependency

if and only if for all inputs of size n the size of the corresponding output is in the set $f(n)$.

For instance, for *insert* we have the following:

- $f_{insert}(n) = n$ is not correct, since it gives a wrong output size, when an element is inserted,
- $f_{insert}(n) = \{n, n+1\}$ is correct,
- $f_{insert}(n) = \{n, n+1, n+2\}$ is correct as well, although it is not that precise, as the previous function.

Correct size dependencies

Formally, the *correctness* of a size dependency is defined w.r.t. the way, we formalise the notion of a size and a size dependency.

A 1-variable multivalued size function f is a correct dependency

if and only if for all inputs of size n the size of the corresponding output is in the set $f(n)$.

For instance, for *insert* we have the following:

- $f_{insert}(n) = n$ is not correct, since it gives a wrong output size, when an element is inserted,
- $f_{insert}(n) = \{n, n+1\}$ is correct,
- $f_{insert}(n) = \{n, n+1, n+2\}$ is correct as well, although it is not that precise, as the previous function.

Correct size dependencies

Formally, the *correctness* of a size dependency is defined w.r.t. the way, we formalise the notion of a size and a size dependency.

A 1-variable multivalued size function f is a correct dependency

if and only if for all inputs of size n the size of the corresponding output is in the set $f(n)$.

For instance, for *insert* we have the following:

- $f_{insert}(n) = n$ is not correct, since it gives a wrong output size, when an element is inserted,
- $f_{insert}(n) = \{n, n+1\}$ is correct,
- $f_{insert}(n) = \{n, n+1, n+2\}$ is correct as well, although it is not that precise, as the previous function.

Correct size dependencies

Formally, the *correctness* of a size dependency is defined w.r.t. the way, we formalise the notion of a size and a size dependency.

A 1-variable multivalued size function f is a correct dependency

if and only if for all inputs of size n the size of the corresponding output is in the set $f(n)$.

For instance, for *insert* we have the following:

- $f_{insert}(n) = n$ is not correct, since it gives a wrong output size, when an element is inserted,
- $f_{insert}(n) = \{n, n+1\}$ is correct,
- $f_{insert}(n) = \{n, n+1, n+2\}$ is correct as well, although it is not that precise, as the previous function.

Correct size dependencies

Formally, the *correctness* of a size dependency is defined w.r.t. the way, we formalise the notion of a size and a size dependency.

A 1-variable multivalued size function f is a correct dependency

if and only if for all inputs of size n the size of the corresponding output is in the set $f(n)$.

For instance, for *insert* we have the following:

- $f_{insert}(n) = n$ is not correct, since it gives a wrong output size, when an element is inserted,
- $f_{insert}(n) = \{n, n+1\}$ is correct,
- $f_{insert}(n) = \{n, n+1, n+2\}$ is correct as well, although it is not that precise, as the previous function.

Outline

Checking of size dependencies

A *checking procedure*:

- *Input*: a program, and its size dependency
- *Output*: "yes" if the dependency is *correct*, "no" otherwise.

E.g. a sound checker gives the answer

- "yes" for the type $Int \times \mathrm{L}_n(Int) \rightarrow \mathrm{L}_{\{n+i\}_{0 \leq i \leq 1}}(Int)$ for *insert*,
- "no" for the type $Int \times \mathrm{L}_n(Int) \rightarrow \mathrm{L}_n(Int)$ for *insert*.

As a rule, checking is reduced to checking arithmetic predicates in (here, first-order).

Checking of size dependencies

A *checking procedure*:

- *Input*: a program, and its size dependency
- *Output*: "yes" if the dependency is *correct*, "no" otherwise.

E.g. a sound checker gives the answer

- "yes" for the type $Int \times L_n(Int) \rightarrow L_{\{n+i\}_{0 \le i \le 1}}(Int)$ for *insert*,
- "no" for the type $Int \times L_n(Int) \rightarrow L_n(Int)$ for *insert*.

As a rule, checking is reduced to checking arithmetic predicates in (here, first-order).

Checking of size dependencies

A *checking procedure*:

- *Input*: a program,
  and its size dependency
- *Output*: "yes" if the dependency is *correct*,
  "no" otherwise.

E.g. a sound checker gives the answer

- "yes" for the type $Int \times L_n(Int) \rightarrow L_{\{n+i\}_{0 \le i \le 1}}(Int)$ for *insert*,
- "no" for the type $Int \times L_n(Int) \rightarrow L_n(Int)$ for *insert*.

As a rule, checking is reduced to checking arithmetic predicates in (here, first-order).

Checking of size dependencies

A *checking procedure*:

- *Input*: a program,
  and its size dependency
- *Output*: “yes” if the dependency is *correct*,
  “no” otherwise.

E.g. a sound checker gives the answer

- “yes” for the type $Int \times \mathsf{L}_n(Int) \rightarrow \mathsf{L}_{\{n+i\}_{0 \leq i \leq 1}}(Int)$ for *insert*,
- “no” for the type $Int \times \mathsf{L}_n(Int) \rightarrow \mathsf{L}_n(Int)$ for *insert*.

As a rule, checking is reduced to checking arithmetic predicates in (here, first-order).

Checking of size dependencies

A *checking procedure*:

- *Input*: a program,
  and its size dependency
- *Output*: "yes" if the dependency is *correct*,
  "no" otherwise.

E.g. a sound checker gives the answer

- "yes" for the type $Int \times L_n(Int) \rightarrow L_{\{n+i\}_{0 \le i \le 1}}(Int)$ for *insert*,
- "no" for the type $Int \times L_n(Int) \rightarrow L_n(Int)$ for *insert*.

As a rule, checking is reduced to checking arithmetic predicates in (here, first-order).

Inference of size dependencies

An *inference procedure*:

- *Input*: a program,
  (in our case: "plus" its underlying type)
- *Output*: a correct size dependency for the program.

E.g. there are size inference procedures that generate the annotations in the typing
$insert : Int \times \mathsf{L}_n(Int) \rightarrow \mathsf{L}_{\{n+i\}_{0 \leq i \leq 1}}(Int)$ (in this or equivalent forms).

As a rule, inference amounts to solving recurrences [2].

Inference of size dependencies

An *inference procedure*:

- *Input*: a program,
  (in our case: "plus" its underlying type)
- *Output*: a correct size dependency for the program.

E.g. there are size inference procedures that generate the annotations in the typing $insert : Int \times \mathsf{L}_n(Int) \rightarrow \mathsf{L}_{\{n+i\}_{0 \leq i \leq 1}}(Int)$ (in this or equivalent forms).

As a rule, inference amounts to solving recurrences [2].

Inference of size dependencies

An *inference procedure*:

- *Input*: a program,
  (in our case: “plus” its underlying type)
- *Output*: a correct size dependency for the program.

E.g. there are size inference procedures that generate the annotations in the typing
$insert : Int \times \mathsf{L}_n(Int) \to \mathsf{L}_{\{n+i\}_{0 \le i \le 1}}(Int)$ (in this or equivalent forms).

As a rule, inference amounts to solving recurrences [2].

Inference of size dependencies

An *inference procedure*:

- *Input*: a program,
  (in our case: "plus" its underlying type)
- *Output*: a correct size dependency for the program.

E.g. there are size inference procedures that generate the annotations in the typing
$insert : Int \times L_n(Int) \to L_{\{n+i\}_{0 \leq i \leq 1}}(Int)$ (in this or equivalent forms).

As a rule, inference amounts to solving recurrences [2].

Outline

Sized Types

Lars Pareto designed a type system for *linear size analysis*

and termination proofs.

$insert : Int \times \mathsf{L}_{i \leq n}(Int) \rightarrow \mathsf{L}_{i \leq n+1}(Int)$.

The typing $f : \mathsf{L}_{i \leq n}(\alpha) \rightarrow \mathsf{L}_{i \leq f_f(n)}(\alpha)$ means that a list of a length at most n is mapped onto a list of a length at most $f_f(n)$.

See [9]

Andreas Abel: linear size analysis over orders

for termination proofs.

$copy\_stream : \mathsf{L}_{\omega}(\alpha) \rightarrow \mathsf{L}_{\omega}(\alpha)$

See [1]

Sized Types

Lars Pareto designed a type system for *linear size analysis*

and termination proofs.

$insert : Int \times \mathsf{L}_{i \leq n}(Int) \rightarrow \mathsf{L}_{i \leq n+1}(Int)$.

The typing $f : \mathsf{L}_{i \leq n}(\alpha) \rightarrow \mathsf{L}_{i \leq f_f(n)}(\alpha)$ means that a list of a length at most n is mapped onto a list of a length at most $f_f(n)$.

See [9]

Andreas Abel: linear size analysis over orders

for termination proofs.

$copy\_stream : \mathsf{L}_{\omega}(\alpha) \rightarrow \mathsf{L}_{\omega}(\alpha)$

See [1]

Sized Types

Lars Pareto designed a type system for *linear size analysis*

and termination proofs.
$insert : Int \times \mathsf{L}_{i \leq n}(Int) \rightarrow \mathsf{L}_{i \leq n+1}(Int)$.

The typing $f : \mathsf{L}_{i \leq n}(\alpha) \rightarrow \mathsf{L}_{i \leq f_f(n)}(\alpha)$ means that a list of a length at most n is mapped onto a list of a length at most $f_f(n)$.

See [9]

Andreas Abel: linear size analysis over orders

for termination proofs.
$copy\_stream : \mathsf{L}_{\omega}(\alpha) \rightarrow \mathsf{L}_{\omega}(\alpha)$
See [1]

Sized Types

Lars Pareto designed a type system for *linear size analysis*

and termination proofs.
$insert : Int \times \mathsf{L}_{i \leq n}(Int) \to \mathsf{L}_{i \leq n+1}(Int)$.

The typing $f : \mathsf{L}_{i \leq n}(\alpha) \to \mathsf{L}_{i \leq f_f(n)}(\alpha)$ means that a list of a length at most n is mapped onto a list of a length at most $f_f(n)$.

See [9]

Andreas Abel: linear size analysis over orders

for termination proofs.
copy_stream : $\mathsf{L}_{\omega}(\alpha) \to \mathsf{L}_{\omega}(\alpha)$
See [1]

Polynomial quasi-interpretations

This approach is developed by J.-Y. Marion, J.-Y. Moyen, G. Bonfante, R. Amadio, for *monotonic* size bounds.

A program f is interpreted as a nondecreasing (piece-wise) polynomial:

- $insert(l)$ is interpreted as $(\!|insert|\!)(X) = X + 1$,
- $sqdiff : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{(n-m)^2}(\alpha)$ cannot be interpreted
- still covers lots of interesting programs,
- inference is decidable in reals, implemented in integers (as far as we know) [5],
- inference is decidable in integers for a subclass: $(\max, +)$-quasi-interpretations by Amadio [3].

Polynomial quasi-interpretations

This approach is developed by J.-Y. Marion, J.-Y. Moyen, G. Bonfante, R. Amadio, for *monotonic* size bounds.

A program *f* is interpreted as a nondecreasing (piece-wise) polynomial:

- *insert*(l) is interpreted as $(\!|insert|\!)(X) = X + 1$,
- *sqdiff* : $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{(n-m)^2}(\alpha)$ cannot be interpreted
- still covers lots of interesting programs,
- inference is decidable in reals, implemented in integers (as far as we know) [5],
- inference is decidable in integers for a subclass: (max, $+$)-quasi-interpretations by Amadio [3].

Polynomial quasi-interpretations

This approach is developed by J.-Y. Marion, J.-Y. Moyen, G. Bonfante, R. Amadio, for *monotonic* size bounds.

A program *f* is interpreted as a nondecreasing (piece-wise) polynomial:

- *insert*(*l*) is interpreted as $(\!|insert|\!)(X) = X + 1$,
- *sqdiff* : $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{(n-m)^2}(\alpha)$ cannot be interpreted
- still covers lots of interesting programs,
- inference is decidable in reals, implemented in integers (as far as we know) [5],
- inference is decidable in integers for a subclass: (max, $+$)-quasi-interpretations by Amadio [3].

Polynomial quasi-interpretations

This approach is developed by J.-Y. Marion, J.-Y. Moyen, G. Bonfante, R. Amadio, for *monotonic* size bounds.

A program *f* is interpreted as a nondecreasing (piece-wise) polynomial:

- $insert(l)$ is interpreted as $(\!| insert |\!)(X) = X + 1$,
- $sqdiff : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{(n-m)^2}(\alpha)$ cannot be interpreted
- still covers lots of interesting programs,
- inference is decidable in reals, implemented in integers (as far as we know) [5],
- inference is decidable in integers for a subclass: (max, $+$)-quasi-interpretations by Amadio [3].

Polynomial quasi-interpretations

This approach is developed by J.-Y. Marion, J.-Y. Moyen, G. Bonfante, R. Amadio, for *monotonic* size bounds.

A program *f* is interpreted as a nondecreasing (piece-wise) polynomial:

- *insert*(*l*) is interpreted as $(\!|insert|\!)(X) = X + 1$,
- *sqdiff* : $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{(n-m)^2}(\alpha)$ cannot be interpreted
- still covers lots of interesting programs,
- inference is decidable in reals, implemented in integers (as far as we know) [5],
- inference is decidable in integers for a subclass: (max, $+$)-quasi-interpretations by Amadio [3].

Polynomial quasi-interpretations

This approach is developed by J.-Y. Marion, J.-Y. Moyen, G. Bonfante, R. Amadio, for *monotonic* size bounds.

A program *f* is interpreted as a nondecreasing (piece-wise) polynomial:

- *insert*(l) is interpreted as $(\!| insert |\!)(X) = X + 1$,
- *sqdiff* : $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{(n-m)^2}(\alpha)$ cannot be interpreted
- still covers lots of interesting programs,
- inference is decidable in reals, implemented in integers (as far as we know) [5],
- inference is decidable in integers for a subclass: (max, $+$)-quasi-interpretations by Amadio [3].

Resource Analysis

- Resource analysis for "Hume" (G. Michaelson, Ph. Trinder, K. Hammond, H.-W. Loidl, et all) is one of the topic of this school.
- Linear heap space analysis of M. Hofmann and S. Jost, [7], will be discussed soon in this tutorial.
- The same authors are developing amortised analysis for object-oriented languages, [8].
- Resource recurrences generation and solving by German Puebla's group, [2].
- *Implicit Computational Complexity*, started by Girard's affine type systems and developed by S. Ronchi della Rocca [6], Patrick Baillot [4] and their colleagues in Italy and France.

Resource Analysis

- Resource analysis for "Hume" (G. Michaelson, Ph. Trinder, K. Hammond, H.-W. Loidl, et all) is one of the topic of this school.
- Linear heap space analysis of M. Hofmann and S. Jost, [7], will be discussed soon in this tutorial.
- The same authors are developing amortised analysis for object-oriented languages, [8].
- Resource recurrences generation and solving by German Puebla's group, [2].
- *Implicit Computational Complexity*, started by Girard's affine type systems and developed by S. Ronchi della Rocca [6], Patrick Baillot [4] and their colleagues in Italy and France.

Resource Analysis

- Resource analysis for “Hume” (G. Michaelson, Ph. Trinder, K. Hammond, H.-W. Loidl, et all) is one of the topic of this school.
- Linear heap space analysis of M. Hofmann and S. Jost, [7], will be discussed soon in this tutorial.
- The same authors are developing amortised analysis for object-oriented languages, [8].
- Resource recurrences generation and solving by German Puebla’s group, [2].
- *Implicit Computational Complexity*, started by Girard’s affine type systems and developed by S. Ronchi della Rocca [6], Patrick Baillot [4] and their colleagues in Italy and France.

Resource Analysis

- Resource analysis for "Hume" (G. Michaelson, Ph. Trinder, K. Hammond, H.-W. Loidl, et all) is one of the topic of this school.
- Linear heap space analysis of M. Hofmann and S. Jost, [7], will be discussed soon in this tutorial.
- The same authors are developing amortised analysis for object-oriented languages, [8].
- Resource recurrences generation and solving by German Puebla's group, [2].
- *Implicit Computational Complexity*, started by Girard's affine type systems and developed by S. Ronchi della Rocca [6], Patrick Baillot [4] and their colleagues in Italy and France.

Resource Analysis

- Resource analysis for “Hume” (G. Michaelson, Ph. Trinder, K. Hammond, H.-W. Loidl, et all) is one of the topic of this school.
- Linear heap space analysis of M. Hofmann and S. Jost, [7], will be discussed soon in this tutorial.
- The same authors are developing amortised analysis for object-oriented languages, [8].
- Resource recurrences generation and solving by German Puebla’s group, [2].
- *Implicit Computational Complexity*, started by Girard’s affine type systems and developed by S. Ronchi della Rocca [6], Patrick Baillot [4] and their colleagues in Italy and France.

Outline

Heap consumption and sizes

Do not mix size analysis and heap consumption analysis despite they are very related. Examples:

- *copy_silly* : $L_n(\alpha) \to L_n(\alpha)$ creates some dummy structures during the computations that are never used. So, it consumes more than *n* heap units (*cons-cells*). But often intermediate structures are indeed necessary.
- in-place programs consume less, like e.g. in-place *reverse* : $L_n(\alpha) \to L_n(\alpha)$, that consumes 0 heap units.
- our *copy* : $L_n(\alpha) \to L_n(\alpha)$ consumes exactly *n* heap units.
- in practice we deal with compositions of these sorts of programs, that makes heap consumption analysis a challenging task.

What we can say in general: heap consumption often depends on the sizes of structures involved in computations.

Heap consumption and sizes

Do not mix size analysis and heap consumption analysis despite they are very related. Examples:

- *copy_silly* : $L_n(\alpha) \rightarrow L_n(\alpha)$ creates some dummy structures during the computations that are never used. So, it consumes more than n heap units (*cons-cells*). But often intermediate structures are indeed necessary.
- in-place programs consume less, like e.g. in-place *reverse* : $L_n(\alpha) \rightarrow L_n(\alpha)$, that consumes 0 heap units.
- our *copy* : $L_n(\alpha) \rightarrow L_n(\alpha)$ consumes exactly n heap units.
- in practice we deal with compositions of these sorts of programs, that makes heap consumption analysis a challenging task.

What we can say in general: heap consumption often depends on the sizes of structures involved in computations.

Heap consumption and sizes

Do not mix size analysis and heap consumption analysis despite they are very related. Examples:

- *copy_silly* : $\mathsf{L}_n(\alpha) \rightarrow \mathsf{L}_n(\alpha)$ creates some dummy structures during the computations that are never used. So, it consumes more than *n* heap units (*cons-cells*). But often intermediate structures are indeed necessary.
- in-place programs consume less, like e.g. in-place *reverse* : $\mathsf{L}_n(\alpha) \rightarrow \mathsf{L}_n(\alpha)$, that consumes 0 heap units.
- our *copy* : $\mathsf{L}_n(\alpha) \rightarrow \mathsf{L}_n(\alpha)$ consumes exactly *n* heap units.
- in practice we deal with compositions of these sorts of programs, that makes heap consumption analysis a challenging task.

What we can say in general: heap consumption often depends on the sizes of structures involved in computations.

Heap consumption and sizes

Do not mix size analysis and heap consumption analysis despite they are very related. Examples:

- *copy_silly* : $\mathsf{L}_n(\alpha) \rightarrow \mathsf{L}_n(\alpha)$ creates some dummy structures during the computations that are never used. So, it consumes more than n heap units (*cons-cells*). But often intermediate structures are indeed necessary.
- in-place programs consume less, like e.g. in-place *reverse* : $\mathsf{L}_n(\alpha) \rightarrow \mathsf{L}_n(\alpha)$, that consumes 0 heap units.
- our *copy* : $\mathsf{L}_n(\alpha) \rightarrow \mathsf{L}_n(\alpha)$ consumes exactly n heap units.
- in practice we deal with compositions of these sorts of programs, that makes heap consumption analysis a challenging task.

What we can say in general: heap consumption often depends on the sizes of structures involved in computations.

Heap consumption and sizes

Do not mix size analysis and heap consumption analysis despite they are very related. Examples:

- *copy_silly* : $\mathsf{L}_n(\alpha) \rightarrow \mathsf{L}_n(\alpha)$ creates some dummy structures during the computations that are never used. So, it consumes more than *n* heap units (*cons-cells*). But often intermediate structures are indeed necessary.
- in-place programs consume less, like e.g. in-place *reverse* : $\mathsf{L}_n(\alpha) \rightarrow \mathsf{L}_n(\alpha)$, that consumes 0 heap units.
- our *copy* : $\mathsf{L}_n(\alpha) \rightarrow \mathsf{L}_n(\alpha)$ consumes exactly *n* heap units.
- in practice we deal with compositions of these sorts of programs, that makes heap consumption analysis a challenging task.

What we can say in general: heap consumption often depends on the sizes of structures involved in computations.

Heap consumption and sizes

Do not mix size analysis and heap consumption analysis despite they are very related. Examples:

- *copy_silly* : $\mathsf{L}_n(\alpha) \rightarrow \mathsf{L}_n(\alpha)$ creates some dummy structures during the computations that are never used. So, it consumes more than *n* heap units (*cons-cells*). But often intermediate structures are indeed necessary.
- in-place programs consume less, like e.g. in-place *reverse* : $\mathsf{L}_n(\alpha) \rightarrow \mathsf{L}_n(\alpha)$, that consumes 0 heap units.
- our *copy* : $\mathsf{L}_n(\alpha) \rightarrow \mathsf{L}_n(\alpha)$ consumes exactly *n* heap units.
- in practice we deal with compositions of these sorts of programs, that makes heap consumption analysis a challenging task.

What we can say in general: heap consumption often depends on the sizes of structures involved in computations.

Basis of AHA: linear amortised heap analysis of Hofmann and Jost

Martin Hofmann and Steffen Jost noticed that if heap consumption depends on the sizes of input structures linearly, we do not need to know the sizes of structures to compute a LINEAR upper bound on heap consumption!

They used amortisation to obtain linear bounds.

Amortisation in resource analysis

means, in particular, that you distribute consumed resource across the input structure. E.g., informally: to compute *copy* there must be at least 1 heap cell available per each input constructor cell. Formally, using Hofmann-Jost heap-aware type system: *copy* : $\mathsf{L}(\alpha,\ 1) \rightarrow \mathsf{L}(\alpha,\ 0)$.

Basis of AHA: linear amortised heap analysis of Hofmann and Jost

Martin Hofmann and Steffen Jost noticed that if heap consumption depends on the sizes of input structures linearly, we do not need to know the sizes of structures to compute a LINEAR upper bound on heap consumption!

They used amortisation to obtain linear bounds.

Amortisation in resource analysis

means, in particular, that you distribute consumed resource across the input structure. E.g., informally: to compute *copy* there must be at least 1 heap cell available per each input constructor cell. Formally, using Hofmann-Jost heap-aware type system: *copy* : $\mathsf{L}(\alpha,\ 1) \rightarrow \mathsf{L}(\alpha,\ 0)$.

Basis of AHA: linear amortised heap analysis of Hofmann and Jost

Martin Hofmann and Steffen Jost noticed that if heap consumption depends on the sizes of input structures linearly, we do not need to know the sizes of structures to compute a LINEAR upper bound on heap consumption!

They used amortisation to obtain linear bounds.

Amortisation in resource analysis

means, in particular, that you distribute consumed resource across the input structure. E.g., informally: to compute *copy* there must be at least 1 heap cell available per each input constructor cell. Formally, using Hofmann-Jost heap-aware type system: *copy* : $\mathrm{L}(\alpha,\ 1) \rightarrow \mathrm{L}(\alpha,\ 0)$.

Basis of AHA: linear amortised heap analysis of Hofmann and Jost

Martin Hofmann and Steffen Jost noticed that if heap consumption depends on the sizes of input structures linearly, we do not need to know the sizes of structures to compute a LINEAR upper bound on heap consumption!

They used amortisation to obtain linear bounds.

Amortisation in resource analysis

means, in particular, that you distribute consumed resource across the input structure. E.g., informally: to compute *copy* there must be at least 1 heap cell available per each input constructor cell. Formally, using Hofmann-Jost heap-aware type system: *copy* : $\mathrm{L}(\alpha,\ 1) \rightarrow \mathrm{L}(\alpha,\ 0)$.

Basis of AHA: linear amortised heap analysis of Hofmann and Jost

Martin Hofmann and Steffen Jost noticed that if heap consumption depends on the sizes of input structures linearly, we do not need to know the sizes of structures to compute a LINEAR upper bound on heap consumption!

They used amortisation to obtain linear bounds.

Amortisation in resource analysis

means, in particular, that you distribute consumed resource across the input structure. E.g., informally: to compute *copy* there must be at least 1 heap cell available per each input constructor cell. Formally, using Hofmann-Jost heap-aware type system: *copy* : $\mathsf{L}(\alpha,\ 1) \rightarrow \mathsf{L}(\alpha,\ 0)$.

Basis of AHA: linear amortised heap analysis of Hofmann and Jost

$f : \mathsf{L}(\alpha,\ k),\ k_0 \rightarrow \mathsf{L}(\alpha,\ k'),\ k_0'$

- To begin computation we need at least k heap units per each constructor cell and k_0 heap cells on top of it,
- that is, heap consumption is at least $kn + k_0$ heap units, where n is the length of an input list.
- After the computation, per each output constructor cell we will have at least k' free heap units, and we have k_0' free heap units on top of all the output list,
- i.e., after the computation there will be at least $k'n' + k_0'$ free heap units free.

Basis of AHA: linear amortised heap analysis of Hofmann and Jost

$f : \mathsf{L}(\alpha,\ k),\ k_0 \to \mathsf{L}(\alpha,\ k'),\ k'_0$

- To begin computation we need at least k heap units per each constructor cell and k_0 heap cells on top of it,
- that is, heap consumption is at least $kn + k_0$ heap units, where n is the length of an input list.
- After the computation, per each output constructor cell we will have at least k' free heap units, and we have k'_0 free heap units on top of all the output list,
- i.e., after the computation there will be at least $k'n' + k'_0$ free heap units free.

Basis of AHA: linear amortised heap analysis of Hofmann and Jost

$f : \mathsf{L}(\alpha,\ k),\ k_0 \rightarrow \mathsf{L}(\alpha,\ k'),\ k'_0$

- To begin computation we need at least k heap units per each constructor cell and k_0 heap cells on top of it,
- that is, heap consumption is at least $kn + k_0$ heap units, where n is the length of an input list.
- After the computation, per each output constructor cell we will have at least k' free heap units, and we have k'_0 free heap units on top of all the output list,
- i.e., after the computation there will be at least $k'n' + k'_0$ free heap units free.

Basis of AHA: linear amortised heap analysis of Hofmann and Jost

$f : \mathsf{L}(\alpha,\ k),\ k_0 \rightarrow \mathsf{L}(\alpha,\ k'),\ k_0'$

- To begin computation we need at least k heap units per each constructor cell and k_0 heap cells on top of it,
- that is, heap consumption is at least $kn + k_0$ heap units, where n is the length of an input list.
- After the computation, per each output constructor cell we will have at least k' free heap units, and we have k_0' free heap units on top of all the output list,
- i.e., after the computation there will be at least $k'n' + k_0'$ free heap units free.

Basis of AHA: linear amortised heap analysis of Hofmann and Jost

$f : \mathsf{L}(\alpha,\ k),\ k_0 \rightarrow \mathsf{L}(\alpha,\ k'),\ k'_0$

- Heap consumption is at least $kn + k_0$ heap units, where n is the length of an input list.
- After the computation there will be at least $k'n' + k'_0$ free heap units, where n' is the length of the output list.

Inference of linear heap consumption bounds amounts to computing the coefficients k, k_0, k', k'_0 in this type system. Hofmann and Jost reduce it to *solving linear programming task*, [7].
They do not need to know sizes to do that.

Our project AHA: towards nonlinear heap bounds

Our project abbreviation stays for Amotrised Heap Analysis.

Our initial project aim: to extend Hofmann and Jost method to non-linear heap bounds.

Sizes are necessary to obtain non-linear heap bounds

Making an initial table of n rows and m columns consumes nm heap units:

$$\begin{array}{l} init\_table(l_1, l_2) : \mathrm{L}_n(\alpha, m) \times \mathrm{L}_m(\alpha, 0), 0 \to \mathrm{L}_n(\mathrm{L}_m(\alpha)) \\ \text{match } l \text{ with } \mid \text{nil} \Rightarrow \text{nil} \\ \qquad\qquad\quad \mid \text{cons}(hd, tl) \Rightarrow \text{cons}(l_2, init\_table(tl, l_2)) \end{array}$$

Our project AHA: towards nonlinear heap bounds

Our project abbreviation stays for Amotrised Heap Analysis.

Our initial project aim: to extend Hofmann and Jost method to non-linear heap bounds.

Sizes are necessary to obtain non-linear heap bounds

Making an initial table of n rows and m columns consumes nm heap units:

$$init\_table(l_1, l_2) : \mathrm{L}_n(\alpha, m) \times \mathrm{L}_m(\alpha, 0), 0 \to \mathrm{L}_n(\mathrm{L}_m(\alpha))$$

match l with | nil $\Rightarrow$ nil
| cons$(hd, tl) \Rightarrow$ cons$(l_2, init_table(tl, l_2))$

Our project AHA: towards nonlinear heap bounds

Our project abbreviation stays for Amotrised Heap Analysis.

Our initial project aim: to extend Hofmann and Jost method to non-linear heap bounds.

Sizes are necessary to obtain non-linear heap bounds

Making an initial table of n rows and m columns consumes nm heap units:

$$\begin{array}{l} init\_table(l_1, l_2) : \mathsf{L}_n(\alpha,\ m) \times \mathsf{L}_m(\alpha, 0), 0 \to \mathsf{L}_n(\mathsf{L}_m(\alpha)) \\ \text{match } l \text{ with } \mid \text{nil} \Rightarrow \text{nil} \\ \qquad\qquad\qquad \mid \text{cons}(hd, tl) \Rightarrow \text{cons}(l_2, init\_table(tl, l_2)) \end{array}$$

Outline

ML-like 1-st-order language over lists

$$
\begin{array}{llll}
\mathit{Basic} & b & ::= & c \mid x \,\text{binop}\, y \mid \text{nil} \mid \text{cons}(z, l) \mid f(z_1, \ldots, z_n) \\
\mathit{Expr} & e & ::= & b \\
 & & & \mid \text{let } z = b \text{ in } e_1 \\
 & & & \mid \text{if } x \text{ then } e_1 \text{ else } e_2 \\
 & & & \mid \text{match } l \text{ with } \mid \text{nil} \Rightarrow e_1 \\
 & & & \qquad\qquad\qquad \mid \text{cons}(z, l') \Rightarrow e_2 \\
 & & & \mid \text{letfun } f(z_1, \ldots, z_n) = e_1 \text{ in } e_2
\end{array}
$$

where c ranges over integer constants, z, x, y, l denote zero-order program variables (x and y range over integer variables, l possibly decorated with sub- ans superscripts, ranges over lists and z ranges over program variables when their types are not relevant).

We consider now only “shapely” programs

To give an idea of our approach we consider here only shapely function definitions: the size of an output is exactly the polynomial function of the size of the corresponding input [12].

Example – desugared *copy* : $\mathsf{L}_n(\alpha) \rightarrow \mathsf{L}_n(\alpha)$

$$copy(l) = \text{match } l \text{ with } \mid \text{nil} \Rightarrow \text{nil}$$
$$\mid \text{cons}(hd, tl) \Rightarrow \text{let } l' = copy(tl) \text{ in cons}(hd, l')$$

Another example – *append* : $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \rightarrow \mathsf{L}_{n+m}(\alpha)$

$$append(l_1, l_2) = \text{match } l_1 \text{ with } \mid \text{nil} \Rightarrow l_2$$
$$\mid \text{cons}(hd, tl) \Rightarrow \text{let } l' = append(tl, l_2) \text{ in cons}(hd, l')$$

We consider now only “shapely” programs

To give an idea of our approach we consider here only shapely function definitions: the size of an output is exactly the polynomial function of the size of the corresponding input [12].

Example – desugared *copy* : $L_n(\alpha) \rightarrow L_n(\alpha)$

$$\begin{array}{ll} copy(l) = \text{match } l \text{ with} & \mid \text{nil} \Rightarrow \text{nil} \\ & \mid \text{cons}(hd, tl) \Rightarrow \text{let } l' = copy(tl) \\ & \qquad\qquad\qquad\quad \text{in cons}(hd, l') \end{array}$$

Another example – *append* : $L_n(\alpha) \times L_m(\alpha) \rightarrow L_{n+m}(\alpha)$

$$\begin{array}{ll} append(l_1, l_2) = \text{match } l_1 \text{ with} & \mid \text{nil} \Rightarrow l_2 \\ & \mid \text{cons}(hd, tl) \Rightarrow \text{let } l' = append(tl, l_2) \\ & \qquad\qquad\qquad\quad \text{in cons}(hd, l') \end{array}$$

We consider now only "shapely" programs

To give an idea of our approach we consider here only shapely function definitions: the size of an output is exactly the polynomial function of the size of the corresponding input [12].

Example – desugared $copy : \mathsf{L}_n(\alpha) \to \mathsf{L}_n(\alpha)$

$$copy(l) = \text{match } l \text{ with } \mid \text{nil} \Rightarrow \text{nil} \\ \mid \text{cons}(hd, tl) \Rightarrow \text{let } l' = copy(tl) \text{ in cons}(hd, l')$$

Another example – $append : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n+m}(\alpha)$

$$append(l_1, l_2) = \text{match } l_1 \text{ with } \mid \text{nil} \Rightarrow l_2 \\ \mid \text{cons}(hd, tl) \Rightarrow \text{let } l' = append(tl, l_2) \text{ in cons}(hd, l')$$

We consider now only “shapely” programs

A non-shapely program: $insert : Int \times \mathsf{L}_{n}(\alpha) \ \rightarrow \mathsf{L}_{\{n+i\}_{0 \leq i \leq 1}}(\alpha)$

$$
\begin{array}{l}
insert(x, l') = \\
\text{match } l \text{ with } \mid \text{nil} \Rightarrow \text{cons}(z, \text{nil}) \\
\qquad\qquad\quad \mid \text{cons}(hd, tl) \Rightarrow \text{if } x = hd \ \text{ then } l \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{else let } l' = insert(x, tl) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{in cons}(hd, l')
\end{array}
$$

Such functions definitions are considered in [10] and [11]. We give a bit more detail in the exercise sheet as well.

We consider now only “shapely” programs

A non-shapely program: $insert : Int \times \mathrm{L}_{n}(\alpha) \rightarrow \mathrm{L}_{\{n+i\}_{0 \leq i \leq 1}}(\alpha)$

$$
\begin{array}{l}
insert(x, l') = \\
\text{match } l \text{ with } \mid \text{nil} \Rightarrow \text{cons}(z, \text{nil}) \\
\qquad\qquad\qquad \mid \text{cons}(hd, tl) \Rightarrow \text{if } x = hd \;\; \text{then } l \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{else let } l' = insert(x, tl) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{in cons}(hd, l')
\end{array}
$$

Such functions definitions are considered in [10] and [11]. We give a bit more detail in the exercise sheet as well.

Type System

E.g. $f : \mathsf{L}_{n}(\alpha) \times \mathsf{L}_{m_1}(\mathsf{L}_{m_2}(\alpha)) \to \mathsf{L}_{p(n,m_1,m_2)}(\alpha)$

means that

- if f has two inputs that are a list of length n and a list of length m_1 of lists of length m_2,
- then the output will be a list of length precisely $p(n, m_1, m_2)$.

Type System

Another example: $cprod : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{nm}(\mathsf{L}_2(\alpha))$

$$
\begin{array}{l}
cprod(l_1, l_2) = \\
\text{match } l_1 \text{ with } \mid \text{nil} \Rightarrow \text{nil} \\
\qquad\qquad\qquad \mid \text{cons}(hd, tl) \Rightarrow \text{let } l' = pairs(hd, l_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{in let } l'' = cprod(tl, l_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{in } append(l', l'') \\
\text{where (sugared) } pairs(z, l) : \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}_n(\mathsf{L}_2(\alpha)) = \\
\text{match } l \text{ with } \mid \text{nil} \Rightarrow \text{nil} \\
\qquad\qquad\qquad \mid \text{cons}(hd, tl) \Rightarrow \text{cons}([z, hd], pairs(z, tl))
\end{array}
$$

E.g. it sends $[1, 2, 1]$ and $[3, 4]$ to
$[\,[1, 3], [1, 4], [2, 3], [2, 4], [1, 3], [1, 4]\,]$

Typing rules

CONS-rule

$$\frac{D \vdash p = p' + 1}{D;\ \Gamma,\ hd : \tau,\ tl : \mathrm{L}_{p'}(\tau)\ \vdash_{\Sigma} \mathrm{cons}(hd, tl) : \mathrm{L}_{p}(\tau)}\ \textsc{Cons}$$

MATCH-rule

$$\frac{\begin{array}{c} p = 0,\ D;\ \Gamma,\ l : \mathrm{L}_p(\tau')\ \vdash_{\Sigma} e_{\mathrm{nil}} : \tau \\ hd, tl \notin dom(\Gamma) \\ D;\ \Gamma, hd : \tau',\ l : \mathrm{L}_p(\tau'),\ tl : \mathrm{L}_{p-1\tau'}(\,) \vdash_{\Sigma} e_{\mathrm{cons}} : \tau \end{array}}{D;\ \Gamma,\ l : \mathrm{L}_p(\tau')\ \vdash_{\Sigma} \mathrm{match}\ l\ \mathrm{with}\ \begin{array}{l} \mid \mathrm{nil} \Rightarrow e_{\mathrm{nil}} \\ \mid \mathrm{cons}(hd, tl) \Rightarrow e_{\mathrm{cons}} \end{array} : \tau}\ \textsc{Match}$$

Typing rules

CONS-rule

$$\frac{D \vdash p = p' + 1}{D;\ \Gamma,\ hd : \tau,\ tl : \mathsf{L}_{p'}(\tau)\ \vdash_\Sigma \mathsf{cons}(hd, tl) : \mathsf{L}_p(\tau)}\ \textsc{Cons}$$

MATCH-rule

$$\frac{\begin{array}{c} p = 0,\ D;\ \Gamma,\ l : \mathsf{L}_p(\tau')\ \vdash_\Sigma e_{\mathsf{nil}} : \tau \\ \\ hd, tl \notin dom(\Gamma) \\ D;\ \Gamma, hd : \tau',\ l : \mathsf{L}_p(\tau'),\ tl : \mathsf{L}_{p-1 \tau'}(\,)\ \vdash_\Sigma e_{\mathsf{cons}} : \tau \end{array}}{D;\ \Gamma,\ l : \mathsf{L}_p(\tau')\ \vdash_\Sigma \mathsf{match}\ l\ \mathsf{with}\ \begin{array}{l} \mid \mathsf{nil} \Rightarrow e_{\mathsf{nil}} \\ \mid \mathsf{cons}(hd, tl) \Rightarrow e_{\mathsf{cons}} \end{array} : \tau}\ \textsc{Match}$$

Typing rules

LET-rule

$$\frac{z \notin dom(\Gamma) \qquad D;\ \Gamma \ \vdash_{\Sigma} e_1 : \tau_z \qquad D;\ \Gamma,\ z : \tau_z \ \vdash_{\Sigma} e_2 : \tau}{D;\ \Gamma \ \vdash_{\Sigma} \text{let } z = e_1 \text{ in } e_2 : \tau} \ \text{LET}$$

Typing rules

FUNNAPP-rule

$$\frac{\begin{array}{c}\Sigma(f) = \tau_1^\circ \times \ldots \times \tau_n^\circ \rightarrow \tau_{k+1} \\ \tau'_{k+1} = \sigma(\tau_{k+1}) \qquad D \vdash C \\ \langle \sigma, C\rangle = \Theta(\tau_1^\circ \times \cdots \times \tau_k^\circ, {\tau_1}' \times \cdots \times {\tau_k}')\end{array}}{D;\ \Gamma, z_1 : {\tau_1}', \ldots, z_k : {\tau_k}' \vdash_\Sigma f(z_1, \ldots, z_k) : {\tau_{k+1}}'}\ \textsc{FunApp}$$

where σ is a substitution of formal parameters for the actual ones (more in [12] and the exercise sheet).

Type checking amounts to verifying 1st order predicates

Given a program, backward style application of typing rules gives eventually proof obligations, that are first-order conditional equations of polynomials:

$cprod : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \rightarrow \mathsf{L}_{nm}(\mathsf{L}_2(\alpha))$

- Nil-branch gives $n = 0 \vdash nm = 0$
- Cons-branch gives (for the outer list) $\vdash nm = n + (n-1)m$

that are true, as we can see. (More details on this example are given in [12], and more details on "how to" are in th exercise sheet).

Type checking amounts to verifying 1st order predicates

Given a program, backward style application of typing rules gives eventually proof obligations, that are first-order conditional equations of polynomials:

$cprod : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \rightarrow \mathsf{L}_{nm}(\mathsf{L}_2(\alpha))$

- Nil-branch gives $n = 0 \vdash nm = 0$
- Cons-branch gives (for the outer list) $\vdash nm = n + (n-1)m$

that are true, as we can see. (More details on this example are given in [12], and more details on "how to" are in th exercise sheet).

Type checking amounts to verifying 1st order predicates

Given a program, backward style application of typing rules gives eventually proof obligations, that are first-order conditional equations of polynomials:

$cprod : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{nm}(\mathsf{L}_2(\alpha))$

- Nil-branch gives $n = 0 \vdash nm = 0$
- Cons-branch gives (for the outer list) $\vdash nm = n + (n-1)m$

that are true, as we can see. (More details on this example are given in [12], and more details on "how to" are in th exercise sheet).

Checking undecidable in integers in general: reduced to 10th Hilbert problem

Check if $f : \mathrm{L}_{n_1}(Int) \times \ldots \times \mathrm{L}_{n_k}(Int) \to \mathrm{L}_1(\alpha)$ for

$$f(x_1, \ldots, x_2) = \text{let } l = f_0(x_1, \ldots, x_k) \text{ in match } l \text{ with } \mid \text{nil} \Rightarrow \text{nil} \mid \text{cons}(hd, tl) \Rightarrow \text{cons}(l, \text{nil})$$

Decidability is reduced to the satisfiability of the predicate $p_{f_0}(n_1, \ldots, n_k) = 0 \vdash 1 = 0$.
It may be that the l.h.s. $p_{f_0}(n_1, \ldots, n_k) = 0$ never holds, so a checker should answer "yes". But then such a checker should have been able to answer the question if an arbitrary polynomial has natural roots or not. This is undecidable in general.

Checking undecidable in integers in general: reduced to 10th Hilbert problem

Check if $f : \mathrm{L}_{n_1}(Int) \times \ldots \times \mathrm{L}_{n_k}(Int) \rightarrow \mathrm{L}_1(\alpha)$ for

$$
\begin{array}{l}
f(x_1, \ldots, x_2) = \text{let } l = f_0(x_1, \ldots, x_k) \\
\qquad \text{in match } l \text{ with } \mid \text{nil} \Rightarrow \text{nil} \\
\qquad \qquad \mid \text{cons}(hd, tl) \Rightarrow \text{cons}(l, \text{nil})
\end{array}
$$

Decidability is reduced to the satisfiability of the predicate $p_{f_0}(n_1, \ldots, n_k) = 0 \vdash 1 = 0$.

It may be that the l.h.s. $p_{f_0}(n_1, \ldots, n_k) = 0$ never holds, so a checker should answer "yes". But then such a checker should have been able to answer the question if an arbitrary polynomial has natural roots or not. This is undecidable in general.

Checking undecidable in integers in general: reduced to 10th Hilbert problem

Check if $f : \mathsf{L}_{n_1}(Int) \times \ldots \times \mathsf{L}_{n_k}(Int) \rightarrow \mathsf{L}_1(\alpha)$ for

$$f(x_1, \ldots, x_2) = \text{let } l = f_0(x_1, \ldots, x_k) \text{ in match } l \text{ with } \mid \text{nil} \Rightarrow \text{nil} \mid \text{cons}(hd, tl) \Rightarrow \text{cons}(l, \text{nil})$$

Decidability is reduced to the satisfiability of the predicate $p_{f_0}(n_1, \ldots, n_k) = 0 \vdash 1 = 0$.
It may be that the l.h.s. $p_{f_0}(n_1, \ldots, n_k) = 0$ never holds, so a checker should answer "yes". But then such a checker should have been able to answer the question if an arbitrary polynomial has natural roots or not. This is undecidable in general.

Condition of decidability

We want to avoid solving complex Diophantine equations like $p(n_1, \ldots, n_k) = 0$.

The simplest way is to consider only programs where *pattern matching is done only on program parameters or their tails*.

Then l.h.s. conditions D in proof obligations will be conjunctions of very simple equations of the form $n - c = 0$. They are trivially solved $n = c$ and substituted to the r.h.s.

Lots of functions definitions (e.g. all primitive recursive functions) may be written so, that they satisfy this condition,which we call informally “no-let-before-match”.

However, there are milder conditions that proved decidability of type checking in this type system ...

Condition of decidability

We want to avoid solving complex Diophantine equations like $p(n_1, \ldots, n_k) = 0$.

The simplest way is to consider only programs where *pattern matching is done only on program parameters or their tails*.

Then l.h.s. conditions D in proof obligations will be conjunctions of very simple equations of the form $n - c = 0$. They are trivially solved $n = c$ and substituted to the r.h.s.

Lots of functions definitions (e.g. all primitive recursive functions) may be written so, that they satisfy this condition,which we call informally “no-let-before-match”.

However, there are milder conditions that proved decidability of type checking in this type system ...

Condition of decidability

We want to avoid solving complex Diophantine equations like $p(n_1, \ldots, n_k) = 0$.

The simplest way is to consider only programs where *pattern matching is done only on program parameters or their tails*.

Then l.h.s. conditions D in proof obligations will be conjunctions of very simple equations of the form $n - c = 0$. They are trivially solved $n = c$ and substituted to the r.h.s.

Lots of functions definitions (e.g. all primitive recursive functions) may be written so, that they satisfy this condition,which we call informally “no-let-before-match”.

However, there are milder conditions that proved decidability of type checking in this type system ...

Condition of decidability

We want to avoid solving complex Diophantine equations like $p(n_1, \ldots, n_k) = 0$.

The simplest way is to consider only programs where *pattern matching is done only on program parameters or their tails*.

Then l.h.s. conditions D in proof obligations will be conjunctions of very simple equations of the form $n - c = 0$. They are trivially solved $n = c$ and substituted to the r.h.s.

Lots of functions definitions (e.g. all primitive recursive functions) may be written so, that they satisfy this condition,which we call informally “no-let-before-match”.

However, there are milder conditions that proved decidability of type checking in this type system ...

Condition of decidability

We want to avoid solving complex Diophantine equations like $p(n_1, \ldots, n_k) = 0$.

The simplest way is to consider only programs where *pattern matching is done only on program parameters or their tails*.

Then l.h.s. conditions D in proof obligations will be conjunctions of very simple equations of the form $n - c = 0$. They are trivially solved $n = c$ and substituted to the r.h.s.

Lots of functions definitions (e.g. all primitive recursive functions) may be written so, that they satisfy this condition,which we call informally “no-let-before-match”.

However, there are milder conditions that proved decidability of type checking in this type system ...

Condition of decidability

We want to avoid solving complex Diophantine equations like $p(n_1, \ldots, n_k) = 0$.
The simplest way is to consider only programs where *pattern matching is done only on program parameters or their tails*.
Then l.h.s. conditions D in proof obligations will be conjunctions of very simple equations of the form $n - c = 0$. They are trivially solved $n = c$ and substituted to the r.h.s.
Lots of functions definitions (e.g. all primitive recursive functions) may be written so, that they satisfy this condition,which we call informally “no-let-before-match”.
However, there are milder conditions that proved decidability of type checking in this type system ...

Inference is semidecidable (for "no-let-before-match" programs)

The idea: fit a polynomial (by finite number of points) and check.

Fitting a polynomial

A polynomial is defined by a finite number of points on its graph, that define a system of linear equations w.r.t. its coefficinets. E.g. a linear function $p(n) = an + b$ is defined by any two different points, $(n_1, p(n_1))$ and $(n_2, p(n_2))$:

$$\begin{cases} n_1 * a + b = p(n_1) \\ n_2 * a + b = p(n_2) \end{cases}$$

The similar holds for any other polynomial of a finite degree d and a finite number of variables s : you must know as many points on the graph as many coefficients the polynomial has, i.e. $\binom{d+s}{d}$.

Inference is semidecidable (for "no-let-before-match" programs)

The idea: fit a polynomial (by finite number of points) and check.

Fitting a polynomial

A polynomial is defined by a finite number of points on its graph, that define a system of linear equations w.r.t. its coefficinets. E.g. a linear function $p(n) = an + b$ is defined by any two different points, $(n_1, p(n_1))$ and $(n_2, p(n_2))$:

$$\begin{cases} n_1 * a + b = p(n_1) \\ n_2 * a + b = p(n_2) \end{cases}$$

The similar holds for any other polynomial of a finite degree d and a finite number of variables s : you must know as many points on the graph as many coefficients the polynomial has, i.e. $\binom{d+s}{d}$.

Inference is semidecidable (for "no-let-before-match" programs)

The idea: fit a polynomial (by finite number of points) and check.

Fitting a polynomial

A polynomial is defined by a finite number of points on its graph, that define a system of linear equations w.r.t. its coefficinets. E.g. a linear function $p(n) = an + b$ is defined by any two different points, $(n_1, p(n_1))$ and $(n_2, p(n_2))$:

$$\begin{cases} n_1 * a + b = p(n_1) \\ n_2 * a + b = p(n_2) \end{cases}$$

The similar holds for any other polynomial of a finite degree d and a finite number of variables s : you must know as many points on the graph as many coefficients the polynomial has, i.e. $\binom{d+s}{d}$.

Inference: how it works, by example *cprod*

Let us want to reconstruct polynomials $f_{cprod\ 1}(n, m)$ and $f_{cprod\ 2}(n, m)$ in the typing

$$cprod : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \rightarrow \mathsf{L}_{f_{cprod\ 1}(n,m)}(\mathsf{L}_{f_{cprod\ 2}(n,m)}(\alpha))$$

- First, we must help our inference procedure and tell it a possible (maximal) degree of the size functions $f_{cprod\ 1}$ and $f_{cprod\ 2}$. Let's for simplicity $d = 2$ for both.

Inference: how it works, by example *cprod*

Let us want to reconstruct polynomials $f_{cprod\ 1}(n,m)$ and $f_{cprod\ 2}(n,m)$ in the typing

$$cprod : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{f_{cprod\ 1}(n,m)}(\mathsf{L}_{f_{cprod\ 2}(n,m)}(\alpha))$$

- First, we must help our inference procedure and tell it a possible (maximal) degree of the size functions $f_{cprod\ 1}$ and $f_{cprod\ 2}$. Let's for simplicity $d = 2$ for both.

Inference: how it works, by example *cprod*

- A polynomial of $s = 2$ variables of degree $d = 2$ as $\binom{d+s}{d} = \binom{4}{2} = 6$ coefficients:
  $p(n, m) = a_{20}n^2 + a_{02}m^2 + a_{11}nm + a_{10}n + a_{01}m + a_{00}$
- Our task now is to find these coefficients by constructing and solving the linear system for them:

$$\left.\begin{array}{ll}
a_{20}n_1^2 + a_{11}n_1m_1 + a_{02}m_1^2 + a_{10}n_1 + a_{01}m_1 + a_{00} & = f_1 \\
a_{20}n_2^2 + a_{11}n_2m_2 + a_{02}m_2^2 + a_{10}n_2 + a_{01}m_2 + a_{00} & = f_2 \\
a_{20}n_3^2 + a_{11}n_3m_3 + a_{02}m_3^2 + a_{10}n_3 + a_{01}m_3 + a_{00} & = f_3 \\
a_{20}n_4^2 + a_{11}n_4m_4 + a_{02}m_4^2 + a_{10}n_4 + a_{01}m_4 + a_{00} & = f_4 \\
a_{20}n_5^2 + a_{11}n_5m_5 + a_{02}m_5^2 + a_{10}n_5 + a_{01}m_5 + a_{00} & = f_5 \\
a_{20}n_6^2 + a_{11}n_6m_6 + a_{02}m_6^2 + a_{10}n_6 + a_{01}m_6 + a_{00} & = f_6
\end{array}\right\}$$

  where $f_i = f_{cprod}(n_i, m_i)$, with $1 \leq i \leq 6$.

Inference: how it works, by example *cprod*

- A polynomial of $s = 2$ variables of degree $d = 2$ as $\binom{d+s}{d} = \binom{4}{2} = 6$ coefficients:
  $p(n, m) = a_{20}n^2 + a_{02}m^2 + a_{11}nm + a_{10}n + a_{01}m + a_{00}$
- Our task now is to find these coefficients by constructing and solving the linear system for them:

$$\left.\begin{array}{ll}
a_{20}n_1^2 + a_{11}n_1m_1 + a_{02}m_1^2 + a_{10}n_1 + a_{01}m_1 + a_{00} & = f_1 \\
a_{20}n_2^2 + a_{11}n_2m_2 + a_{02}m_2^2 + a_{10}n_2 + a_{01}m_2 + a_{00} & = f_2 \\
a_{20}n_3^2 + a_{11}n_3m_3 + a_{02}m_3^2 + a_{10}n_3 + a_{01}m_3 + a_{00} & = f_3 \\
a_{20}n_4^2 + a_{11}n_4m_4 + a_{02}m_4^2 + a_{10}n_4 + a_{01}m_4 + a_{00} & = f_4 \\
a_{20}n_5^2 + a_{11}n_5m_5 + a_{02}m_5^2 + a_{10}n_5 + a_{01}m_5 + a_{00} & = f_5 \\
a_{20}n_6^2 + a_{11}n_6m_6 + a_{02}m_6^2 + a_{10}n_6 + a_{01}m_6 + a_{00} & = f_6
\end{array}\right\}$$

  where $f_i = f_{cprod}(n_i, m_i)$, with $1 \leq i \leq 6$.

Inference: how it works, by example *cprod*

- Now, we must chose test points (n_i, m_i) in such a way that the system above has a unique solution. This solution is exactly the collections of the coefficients for $p(n, m)$. From interpolation theory it is known that it is sufficient, that points satisfy NCA-configuration, in our case – on the plane. The full definition and references are given in [12].

Inference: how it works, by example *cprod*

- Now, we must chose test points (n_i, m_i) in such a way that the system above has a unique solution. This solution is exactly the collections of the coefficients for $p(n, m)$. From interpolation theory it is known that it is sufficient, that points satisfy NCA-configuration, in our case – on the plane. The full definition and references are given in [12].

Inference: how it works, by example *cprod*

- Here we describe its partial case: if you need to pick up $\binom{d+2}{d}$ points, on the plane that satisfy NCA-configuration, you choose $d+1$ parallel lines, and pick up $d+1$ points on the first line, d points on the second line, ... and 1 point on the last line.

  In our example we choose these lines to be parallel to the $y = 0$ axis, so the lines are $y = 1, 2, 3$ and points are
  $(n_1, m_1) = (1, 1)$ $(n_2, m_2) = (2, 1)$ $(n_3, m_3) = (3, 1)$
  $(n_4, m_4) = (1, 2)$ $(n_5, m_5) = (2, 2)$
  $(n_6, m_6) = (1, 3)$

Inference: how it works, by example *cprod*

- Here we describe its partial case: if you need to pick up $\binom{d+2}{d}$ points, on the plane that satisfy NCA-configuration, you choose $d+1$ parallel lines, and pick up $d+1$ points on the first line, d points on the second line, ... and 1 point on the last line.

  In our example we choose these lines to be parallel to the $y = 0$ axis, so the lines are $y = 1, 2, 3$ and points are
  $(n_1, m_1) = (1, 1)$ $(n_2, m_2) = (2, 1)$ $(n_3, m_3) = (3, 1)$
  $(n_4, m_4) = (1, 2)$ $(n_5, m_5) = (2, 2)$
  $(n_6, m_6) = (1, 3)$

Inference: how it works, by example $f_{cprod}(n, m)$

- Now, we construct a collection of 6 input pairs of lists, that have the sizes (n_i, m_i) and run *cprod* on these data:

	(n, m)	Input lists	*cprod*	$f_{cprod\,1}$
$i = 1$	$(1, 1)$	$[1],\ [1]$	$[\,[1, 1]\,]$	1
$i = 2$	$(2, 1)$	$[1, 2],\ [1]$	$[\,[1, 1], [2, 1]\,]$	2
$i = 3$	$(3, 1)$	$[1, 2, 3],\ [1]$	$[\,[1, 1], [2, 1], [3, 1]\,]$	3
$i = 4$	$(1, 2)$	$[1],\ [1, 2]$	$[\,[1, 1], [1, 2]\,]$	2
$i = 5$	$(2, 2)$	$[1, 2],\ [1, 2]$	$[\,[1, 1], [2, 1], [1, 2], [2, 2]\,]$	4
$i = 6$	$(1, 3)$	$[1],\ [1, 2, 3]$	$[\,[1, 1], [1, 2], [1, 3]\,]$	3

Inference: how it works, by example *cprod*

- Now, construct the linear system form the data above:

$$\left.\begin{array}{ll} a_{20}+a_{11}+a_{02}+a_{10}+a_{01}+a_{00} & =1 \\ 4a_{20}+2a_{11}+a_{02}+2a_{10}+a_{01}+a_{00} & =2 \\ 9a_{20}+3a_{11}+a_{02}+3a_{10}+a_{01}+a_{00} & =3 \\ a_{20}+2a_{11}+4a_{02}+a_{10}+2a_{01}+a_{00} & =2 \\ 4a_{20}+4a_{11}+4a_{02}+2a_{10}+2a_{01}+a_{00} & =4 \\ a_{20}+3a_{11}+9a_{02}+a_{10}+3a_{01}+a_{00} & =3 \end{array}\right\}$$

- Solving this system gives that $a_{11}=1$ and the rest of the coefficients are zero. Thus, $f_{cprod\ 1}(n,m)=nm$.

Inference: how it works, by example *cprod*

- Now, construct the linear system form the data above:

$$\left.\begin{array}{ll} a_{20}+a_{11}+a_{02}+a_{10}+a_{01}+a_{00} & =1 \\ 4a_{20}+2a_{11}+a_{02}+2a_{10}+a_{01}+a_{00} & =2 \\ 9a_{20}+3a_{11}+a_{02}+3a_{10}+a_{01}+a_{00} & =3 \\ a_{20}+2a_{11}+4a_{02}+a_{10}+2a_{01}+a_{00} & =2 \\ 4a_{20}+4a_{11}+4a_{02}+2a_{10}+2a_{01}+a_{00} & =4 \\ a_{20}+3a_{11}+9a_{02}+a_{10}+3a_{01}+a_{00} & =3 \end{array}\right\}$$

- Solving this system gives that $a_{11}=1$ and the rest of the coefficients are zero. Thus, $f_{cprod\ 1}(n,m)=nm$.

Inference: how it works, by example *cprod*

- Similarly we obtain that $f_{cprod\ 2}(n, m) = 2$. Note, that if we choose the test data in such a way that the length of the outer list of the output is $f_{cprod\ 1}(n'_i, m'_i) = 0$, then the length of the inner list is undefined: $\mathsf{L}_0(\mathsf{L}_{???}(\alpha))$.
- E.g. it may happen if one of the input lists is empty: $n'_1 = 0, m'_1 = 1$ and on the inputs [], [1] the program *cprod* produces [], on which $f_{cprod\ 2}$ is undefined.
- In this case we run a program a bit more times (on other data), so that we can fully define the "inner" polynomial. It is treated in details in [12].

Inference: how it works, by example *cprod*

- Similarly we obtain that $f_{cprod\ 2}(n, m) = 2$. Note, that if we choose the test data in such a way that the length of the outer list of the output is $f_{cprod\ 1}(n'_i, m'_i) = 0$, then the length of the inner list is undefined: $\mathsf{L}_0(\mathsf{L}_{???}(\alpha))$.
- E.g. it may happen if one of the input lists is empty: $n'_1 = 0, m'_1 = 1$ and on the inputs $[\,],\ [1]$ the program *cprod* produces $[\,]$, on which $f_{cprod\ 2}$ is undefined.
- In this case we run a program a bit more times (on other data), so that we can fully define the "inner" polynomial. It is treated in details in [12].

Inference: how it works, by example *cprod*

- Similarly we obtain that $f_{cprod\ 2}(n, m) = 2$. Note, that if we choose the test data in such a way that the length of the outer list of the output is $f_{cprod\ 1}(n'_i, m'_i) = 0$, then the length of the inner list is undefined: $\mathsf{L}_0(\mathsf{L}_{???}(\alpha))$.
- E.g. it may happen if one of the input lists is empty: $n'_1 = 0, m'_1 = 1$ and on the inputs $[\,],\ [1]$ the program *cprod* produces $[\,]$, on which $f_{cprod\ 2}$ is undefined.
- In this case we run a program a bit more times (on other data), so that we can fully define the "inner" polynomial. It is treated in details in [12].

Inference: how it works, by example *cprod*

The last step is to check the obtained polynomials by sending the annotated typing to a type checker. E.g. our typing *cprod* : $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \rightarrow \mathsf{L}_{nm}(\mathsf{L}_2(\alpha))$ is accepted.

If the typing is not accepted, then it may be due to the following reasons:

- the proposed degree *d* is lower than the degree of the actual size function – then you can repeat the procedure with a higher degree,
- you have chosen bad set of size variables for input types – then you may change the assignment of size variables to annotated input types and repeat the procedure,
- the program under consideration is not shapely (either not a precise dependency, or no polynomial bounds at all) – then the inference procedure does not terminate.

Inference: how it works, by example *cprod*

The last step is to check the obtained polynomials by sending the annotated typing to a type checker. E.g. our typing *cprod* : $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \rightarrow \mathsf{L}_{nm}(\mathsf{L}_2(\alpha))$ is accepted.
If the typing is not accepted, then it may be due to the following reasons:

- the proposed degree *d* is lower than the degree of the actual size function – then you can repeat the procedure with a higher degree,
- you have chosen bad set of size variables for input types – then you may change the assignment of size variables to annotated input types and repeat the procedure,
- the program under consideration is not shapely (either not a precise dependency, or no polynomial bounds at all) – then the inference procedure does not terminate.

Inference: how it works, by example *cprod*

The last step is to check the obtained polynomials by sending the annotated typing to a type checker. E.g. our typing *cprod* : $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \rightarrow \mathsf{L}_{nm}(\mathsf{L}_2(\alpha))$ is accepted.
If the typing is not accepted, then it may be due to the following reasons:

- the proposed degree *d* is lower than the degree of the actual size function – then you can repeat the procedure with a higher degree,
- you have chosen bad set of size variables for input types – then you may change the assignment of size variables to annotated input types and repeat the procedure,
- the program under consideration is not shapely (either not a precise dependency, or no polynomial bounds at all) – then the inference procedure does not terminate.

Inference: how it works, by example *cprod*

The last step is to check the obtained polynomials by sending the annotated typing to a type checker. E.g. our typing $cprod : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \rightarrow \mathsf{L}_{nm}(\mathsf{L}_2(\alpha))$ is accepted.
If the typing is not accepted, then it may be due to the following reasons:

- the proposed degree d is lower than the degree of the actual size function – then you can repeat the procedure with a higher degree,
- you have chosen bad set of size variables for input types – then you may change the assignment of size variables to annotated input types and repeat the procedure,
- the program under consideration is not shapely (either not a precise dependency, or no polynomial bounds at all) – then the inference procedure does not terminate.

Outline

Programs with lower and upper polynomial bounds: *insert*

Consider polymorphic version of *insert*:

$insert : (\alpha \times \alpha \rightarrow Bool) \times \alpha \times \mathsf{L}_n(\alpha) \rightarrow \mathsf{L}_{\{n+i\}_{0 \leq i \leq 1}}(\alpha)$

$$
\begin{array}{l}
insert(g, z, l) = \\
\text{match } l \text{ with } \mid \text{nil} \Rightarrow \text{cons}(z, \text{nil}) \\
\qquad\qquad\quad \mid \text{cons}(hd, tl) \Rightarrow \ \text{if } g(z, hd) \ \text{ then } l \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{else} \\
\qquad\qquad\qquad\qquad\qquad\quad \text{let } l' = insert(z, tl) \\
\qquad\qquad\qquad\qquad\qquad\quad \text{in cons}(hd, l')
\end{array}
$$

For instance, with *g* being equality of two integers,

- on 2, $[1, 2, 3]$ it returns $[1, 2, 3]$,
- and on 2, $[3, 4, 5]$ it returns $[3, 4, 5, 2]$.

Programs with lower and upper polynomial bounds: *insert*

Consider polymorphic version of *insert*:

$insert : (\alpha \times \alpha \rightarrow Bool) \times \alpha \times \mathsf{L}_n(\alpha) \rightarrow \mathsf{L}_{\{n+i\}_{0 \leq i \leq 1}}(\alpha)$

$insert(g, z, l) =$
match l with | nil $\Rightarrow$ cons(z, nil)
| cons$(hd, tl) \Rightarrow$ if $g(z, hd)$ then l
else
let $l' = insert(z, tl)$
in cons(hd, l')

For instance, with g being equality of two integers,

- on 2, $[1, 2, 3]$ it returns $[1, 2, 3]$,
- and on 2, $[3, 4, 5]$ it returns $[3, 4, 5, 2]$.

Programs with lower and upper polynomial bounds: *insert*, IFL'08 paper, [10]

The first improvement from IFL'08 paper, [10]: in many cases (including shapely programs) while studying size dependencies it is convenient to reduce an original program under consideration to its size abstraction, that is to collection of (recursive) rewriting rules for its size functions.

Example: rewriting rules for *insert*

nil-branch $\quad n = 0 \vdash \quad f_{insert}(n) \rightarrow 1$

cons-branch $\quad n \geq 1 \vdash \quad f_{insert}(n) \rightarrow \; n \mid 1 + f_{insert}(n-1)$

where $\mid$ denotes two options in computing f_{insert} corresponding to the *true*- and *false*-branches of the if-expression, resp.

Programs with lower and upper polynomial bounds: *insert*, IFL'08 paper, [10]

The first improvement from IFL'08 paper, [10]: in many cases (including shapely programs) while studying size dependencies it is convenient to reduce an original program under consideration to its size abstraction, that is to collection of (recursive) rewriting rules for its size functions.

Example: rewriting rules for *insert*

nil-branch $n = 0 \vdash \quad f_{insert}(n) \to 1$

cons-branch $n \geq 1 \vdash \quad f_{insert}(n) \to \quad n \mid 1 + f_{insert}(n-1)$

where $\mid$ denotes two options in computing f_{insert} corresponding to the *true*- and *false*-branches of the if-expression, resp.

Programs with lower and upper polynomial bounds: *insert*, IFL'08 paper, [10]

The first improvement from IFL'08 paper, [10]: in many cases (including shapely programs) while studying size dependencies it is convenient to reduce an original program under consideration to its size abstraction, that is to collection of (recursive) rewriting rules for its size functions.

Example: rewriting rules for *insert*

nil-branch $n = 0 \vdash \quad f_{insert}(n) \to 1$

cons-branch $n \geq 1 \vdash \quad f_{insert}(n) \to \; n \mid 1 + f_{insert}(n-1)$

where $\mid$ denotes two options in computing f_{insert} corresponding to the *true*- and *false*-branches of the if-expression, resp.

Programs with lower and upper polynomial bounds: *insert*, IFL'08 paper, [10]

We want to compute a lower $f_{insert\ \min}(n)$ and an upper $f_{insert\ \max}(n)$ bounds for *insert*.

- Assume that bounds depend on the size variable n, and, the degree $d = 1$ for both.
- A polynomial of one variable of the degree 1 (linear) is given by two coefficients. Thus

$$f_{insert\ \min}(n) = a_{\min\ 1} n + a_{\min\ 0}$$
$$f_{insert\ \max}(n) = a_{\max\ 1} n + a_{\max\ 0}$$

are defined by two nodes (1-dimensional points).

- Let these nodes are $n = 1, 2$.

Programs with lower and upper polynomial bounds: *insert*, IFL'08 paper, [10]

We want to compute a lower $f_{insert\ \min}(n)$ and an upper $f_{insert\ \max}(n)$ bounds for *insert*.

- Assume that bounds depend on the size variable n, and, the degree $d = 1$ for both.
- A polynomial of one variable of the degree 1 (linear) is given by two coefficients. Thus

$$f_{insert\ \min}(n) = a_{\min\ 1} n + a_{\min\ 0}$$
$$f_{insert\ \max}(n) = a_{\max\ 1} n + a_{\max\ 0}$$

are defined by two nodes (1-dimensional points).

- Let these nodes are $n = 1, 2$.

Programs with lower and upper polynomial bounds: *insert*, IFL'08 paper, [10]

We want to compute a lower $f_{insert\ \mathrm{min}}(n)$ and an upper $f_{insert\ \mathrm{max}}(n)$ bounds for *insert*.

- Assume that bounds depend on the size variable n, and, the degree $d = 1$ for both.
- A polynomial of one variable of the degree 1 (linear) is given by two coefficients. Thus

$$f_{insert\ \mathrm{min}}(n) = a_{\mathrm{min}\ 1} n + a_{\mathrm{min}\ 0}$$
$$f_{insert\ \mathrm{max}}(n) = a_{\mathrm{max}\ 1} n + a_{\mathrm{max}\ 0}$$

  are defined by two nodes (1-dimensional points).
- Let these nodes are $n = 1, 2$.

Programs with lower and upper polynomial bounds: *insert*

- Differently to the initial version of our method we do not need to generate test data, if we have a rewriting system for a size function. We compute the values of a size function directly on concrete sizes.

  Thus, in our example we compute

  $$f_{insert}(1) \to 1 \mid 1 + f_{insert}(0) = \{1, 1+1\} = \{1, 2\}$$
  $$f_{insert}(2) \to 2 \mid 1 + f_{insert}(1) = \{2, 1+1, 1+2\} = \{2, 3\}$$

- Now, we see that

$$f_{insert\,\min}(1) = 1 \quad f_{insert\,\min}(2) = 2$$
$$f_{insert\,\max}(1) = 2 \quad f_{insert\,\max}(2) = 3$$

Programs with lower and upper polynomial bounds: *insert*

- Differently to the initial version of our method we do not need to generate test data, if we have a rewriting system for a size function. We compute the values of a size function directly on concrete sizes.
  Thus, in our example we compute
  $f_{insert}(1) \rightarrow 1 \mid 1 + f_{insert}(0) = \{1, 1+1\} = \{1, 2\}$
  $f_{insert}(2) \rightarrow 2 \mid 1 + f_{insert}(1) = \{2, 1+1, 1+2\} = \{2, 3\}$
- Now, we see that

$$f_{insert\,\min}(1) = 1 \quad f_{insert\,\min}(2) = 2$$
$$f_{insert\,\max}(1) = 2 \quad f_{insert\,\max}(2) = 3$$

Programs with lower and upper polynomial bounds: *insert*

- Differently to the initial version of our method we do not need to generate test data, if we have a rewriting system for a size function. We compute the values of a size function directly on concrete sizes.
  Thus, in our example we compute
  $f_{insert}(1) \rightarrow 1 \mid 1 + f_{insert}(0) = \{1, 1+1\} = \{1, 2\}$
  $f_{insert}(2) \rightarrow 2 \mid 1 + f_{insert}(1) = \{2, 1+1, 1+2\} = \{2, 3\}$
- Now, we see that

$$\begin{array}{ll} f_{insert\,\min}(1) = 1 & f_{insert\,\min}(2) = 2 \\ f_{insert\,\max}(1) = 2 & f_{insert\,\max}(2) = 3 \end{array}$$

Programs with lower and upper polynomial bounds: *insert*

- So we have two systems of linear equations, for the lower and upper bounds respectively:

$$\left.\begin{array}{ll} a_{\min\ 1} + a_{\min\ 0} & = f_{insert\,\min}(1) = 1 \\ 2a_{\min\ 1} + a_{\min\ 0} & = f_{insert\,\min}(2) = 2 \end{array}\right\}$$

$$\left.\begin{array}{ll} a_{\max\ 1} + a_{\max\ 0} & = f_{insert\,\max}(1) = 2 \\ 2a_{\max\ 1} + a_{\max\ 0} & = f_{insert\,\max}(2) = 3 \end{array}\right\}$$

- Solving these systems gives $a_{\min\ 1} = 1$, $a_{\min\ 0} = 0$ and $a_{\max\ 1} = 1$, $a_{\max\ 0} = 1$. That is,

$$f_{insert\,\min}(n) = n$$
$$f_{insert\,\max}(n) = n + 1$$

Programs with lower and upper polynomial bounds: *insert*

- So we have two systems of linear equations, for the lower and upper bounds respectively:

$$\left.\begin{aligned} a_{\min\ 1} + a_{\min\ 0} &= f_{insert\,\min}(1) = 1 \\ 2a_{\min\ 1} + a_{\min\ 0} &= f_{insert\,\min}(2) = 2 \end{aligned}\right\}$$

$$\left.\begin{aligned} a_{\max\ 1} + a_{\max\ 0} &= f_{insert\,\max}(1) = 2 \\ 2a_{\max\ 1} + a_{\max\ 0} &= f_{insert\,\max}(2) = 3 \end{aligned}\right\}$$

- Solving these systems gives $a_{\min\ 1} = 1$, $a_{\min\ 0} = 0$ and $a_{\max\ 1} = 1$, $a_{\max\ 0} = 1$. That is,

$$\begin{aligned} f_{insert\,\min}(n) &= n \\ f_{insert\,\max}(n) &= n + 1 \end{aligned}$$

Programs with lower and upper polynomial bounds: *insert*

- So we have two systems of linear equations, for the lower and upper bounds respectively:

$$\left.\begin{aligned} a_{\min\ 1} + a_{\min\ 0} &= f_{insert\,\min}(1) = 1 \\ 2a_{\min\ 1} + a_{\min\ 0} &= f_{insert\,\min}(2) = 2 \end{aligned}\right\}$$

$$\left.\begin{aligned} a_{\max\ 1} + a_{\max\ 0} &= f_{insert\,\max}(1) = 2 \\ 2a_{\max\ 1} + a_{\max\ 0} &= f_{insert\,\max}(2) = 3 \end{aligned}\right\}$$

- Solving these systems gives $a_{\min\ 1} = 1$, $a_{\min\ 0} = 0$ and $a_{\max\ 1} = 1$, $a_{\max\ 0} = 1$. That is,

$$f_{insert\,\min}(n) = n$$
$$f_{insert\,\max}(n) = n + 1$$

Programs with lower and upper polynomial bounds: *insert*

- So we have two systems of linear equations, for the lower and upper bounds respectively:

$$\left.\begin{array}{ll} a_{\min\ 1} + a_{\min\ 0} & = f_{insert\,\min}(1) = 1 \\ 2a_{\min\ 1} + a_{\min\ 0} & = f_{insert\,\min}(2) = 2 \end{array}\right\}$$

$$\left.\begin{array}{ll} a_{\max\ 1} + a_{\max\ 0} & = f_{insert\,\max}(1) = 2 \\ 2a_{\max\ 1} + a_{\max\ 0} & = f_{insert\,\max}(2) = 3 \end{array}\right\}$$

- Solving these systems gives $a_{\min\ 1} = 1$, $a_{\min\ 0} = 0$ and $a_{\max\ 1} = 1$, $a_{\max\ 0} = 1$. That is,

$$f_{insert\,\min}(n) = n$$
$$f_{insert\,\max}(n) = n + 1$$

Programs with lower and upper polynomial bounds: *insert*

- So, $f_{insert}(n) \subseteq \{f_{insert\,\min}(n) + i\}_{0 \leq i \leq f_{insert\,\max}(n) - f_{insert\,\min}(n)}$
  $\subseteq \{n + i\}_{0 \leq i \leq 1}$
- Now we have to check, if indeed,
  $insert : (\alpha \times \alpha \to Bool) \times \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{n+i\}_{0 \leq i \leq 1}}(\alpha)$.
  The checking extends the checking procedure for shapely functions. Here, the output annotation should contain *all* the values of the size function in any branch of the computations: ...

Programs with lower and upper polynomial bounds: *insert*

- So, $f_{insert}(n) \subseteq \{f_{insert\,\min}(n) + i\}_{0 \leq i \leq f_{insert\,\max}(n) - f_{insert\,\min}(n)}$
  $\subseteq \{n + i\}_{0 \leq i \leq 1}$
- Now we have to check, if indeed,
  $insert : (\alpha \times \alpha \rightarrow Bool) \times \alpha \times L_n(\alpha) \rightarrow L_{\{n+i\}_{0 \leq i \leq 1}}(\alpha)$.
  The checking extends the checking procedure for shapely functions. Here, the output annotation should contain *all* the values of the size function in any branch of the computations: ...

Programs with lower and upper polynomial bounds: *insert*

- So, $f_{insert}(n) \subseteq \{f_{insert\,\min}(n) + i\}_{0 \leq i \leq f_{insert\,\max}(n) - f_{insert\,\min}(n)}$
  $\subseteq \{n + i\}_{0 \leq i \leq 1}$
- Now we have to check, if indeed,
  $insert : (\alpha \times \alpha \rightarrow Bool) \times \alpha \times \mathrm{L}_n(\alpha) \rightarrow \mathrm{L}_{\{n+i\}_{0 \leq i \leq 1}}(\alpha)$.
  The checking extends the checking procedure for shapely functions. Here, the output annotation should contain *all* the values of the size function in any branch of the computations: ...

Programs with lower and upper polynomial bounds: *insert*

- ...
  - the nil-branch we have the following inclusion: $n = 0 \vdash \{n + i\}_{0 \leq i \leq 1} \supseteq \{1\}$,
  - for the true-branch in the cons-branch we have $n \geq 1 \vdash \{n + i\}_{0 \leq i \leq 1} \supseteq \{n\}$,
  - for the false-branch $n \geq 1 \vdash \{n + i\}_{0 \leq i \leq 1} \supseteq \{1\} + \{(n - 1) + i'\}_{0 \leq i' \leq 1}$

  where $+$ is lifted to sets and defined as pairwise addition of the sets' elements.

Programs with lower and upper polynomial bounds: *insert*

- ...
  - the nil-branch we have the following inclusion: $n = 0 \vdash \{n + i\}_{0 \leq i \leq 1} \supseteq \{1\}$,
  - for the true-branch in the cons-branch we have $n \geq 1 \vdash \{n + i\}_{0 \leq i \leq 1} \supseteq \{n\}$,
  - for the false-branch $n \geq 1 \vdash \{n + i\}_{0 \leq i \leq 1} \supseteq \{1\} + \{(n - 1) + i'\}_{0 \leq i' \leq 1}$

  where $+$ is lifted to sets and defined as pairwise addition of the sets' elements.

Programs with lower and upper polynomial bounds: *insert*

- ...
  - the nil-branch we have the following inclusion: $n = 0 \vdash \{n + i\}_{0 \leq i \leq 1} \supseteq \{1\}$,
  - for the true-branch in the cons-branch we have $n \geq 1 \vdash \{n + i\}_{0 \leq i \leq 1} \supseteq \{n\}$,
  - for the false-branch $n \geq 1 \vdash \{n + i\}_{0 \leq i \leq 1} \supseteq \{1\} + \{(n-1) + i'\}_{0 \leq i' \leq 1}$

  where $+$ is lifted to sets and defined as pairwise addition of the sets' elements.

Programs with lower and upper polynomial bounds: *insert*

- The inclusions above are turned into the first-order predicates by unfolding the definition of a set inclusion :
  - $n = 0 \Rightarrow \exists\, i.\, 0 \leq i \leq 1 \wedge n + i = 1,$
  - $n \geq 1 \Rightarrow \exists\, i.\, 0 \leq i \leq 1 \wedge n + i = n,$
  - $\forall\, n\, i'.\, 0 \leq i' \leq 1 \wedge n \geq 1 \Rightarrow \exists\, i.\, 0 \leq i \leq 1 \wedge n + i = 1 + (n - 1) + i'$

  It is easy to check that i may be instantiated as $i = 1$, $i = 0$ and $i = i'$ for each of the branches respectively.

  In general for checking one have to instantiate existential quantifiers in the first-order arithmetics. This is, in general, undecidable in integers (but still, decidable e.g. for linear size functions). It is decidable in reals, however real arithmetics has some disadvantages which we do not discuss here.

Programs with lower and upper polynomial bounds: *insert*

- The inclusions above are turned into the first-order predicates by unfolding the definition of a set inclusion :
  - $n = 0 \Rightarrow \exists\, i.\, 0 \leq i \leq 1 \wedge n + i = 1,$
  - $n \geq 1 \Rightarrow \exists\, i.\, 0 \leq i \leq 1 \wedge n + i = n,$
  - $\forall\, n\, i'.\, 0 \leq i' \leq 1 \wedge n \geq 1 \Rightarrow \exists\, i.\, 0 \leq i \leq 1 \wedge n + i = 1 + (n - 1) + i'$

  It is easy to check that i may be instantiated as $i = 1$, $i = 0$ and $i = i'$ for each of the branches respectively.

  In general for checking one have to instantiate existential quantifiers in the first-order arithmetics. This is, in general, undecidable in integers (but still, decidable e.g. for linear size functions). It is decidable in reals, however real arithmetics has some disadvantages which we do not discuss here.

Programs with lower and upper polynomial bounds: *insert*

- The inclusions above are turned into the first-order predicates by unfolding the definition of a set inclusion :
  - $n = 0 \Rightarrow \exists\, i.\, 0 \leq i \leq 1 \wedge n + i = 1$,
  - $n \geq 1 \Rightarrow \exists\, i.\, 0 \leq i \leq 1 \wedge n + i = n$,
  - $\forall\, n\, i'.\, 0 \leq i' \leq 1 \wedge n \geq 1 \Rightarrow \exists\, i.\, 0 \leq i \leq 1 \wedge n + i = 1 + (n - 1) + i'$

  It is easy to check that i may be instantiated as $i = 1$, $i = 0$ and $i = i'$ for each of the branches respectively.
  In general for checking one have to instantiate existential quantifiers in the first-order arithmetics. This is, in general, undecidable in integers (but still, decidable e.g. for linear size functions). It is decidable in reals, however real arithmetics has some disadvantages which we do not discuss here.

Programs with lower and upper polynomial bounds: *insert*

- The inclusions above are turned into the first-order predicates by unfolding the definition of a set inclusion :
  - $n = 0 \Rightarrow \exists\, i.\, 0 \leq i \leq 1 \wedge n + i = 1$,
  - $n \geq 1 \Rightarrow \exists\, i.\, 0 \leq i \leq 1 \wedge n + i = n$,
  - $\forall\, n\, i'.\, 0 \leq i' \leq 1 \wedge n \geq 1 \Rightarrow \exists\, i.\, 0 \leq i \leq 1 \wedge n + i = 1 + (n - 1) + i'$

  It is easy to check that i may be instantiated as $i = 1$, $i = 0$ and $i = i'$ for each of the branches respectively.
  In general for checking one have to instantiate existential quantifiers in the first-order arithmetics. This is, in general, undecidable in integers (but still, decidable e.g. for linear size functions). It is decidable in reals, however real arithmetics has some disadvantages which we do not discuss here.

Programs with polynomial bounds over nested lists (submitted TFP'09)

In types like $\mathsf{L}(\mathsf{L}(\alpha))$ the internal lists may be of different length: e.g. $[\,[1,2],\ [3,4,5],\ [\,]\,]$.

Formally, this fact may be expressed by introduction of length functions $\lambda\, k.M(k)$ that express the lengths of internal lists, where

- $M(0)$ is the length of the head list,
- $M(1)$ is the length of the element following the head,
- ... etc.

In the example above $M(0) = 2$, $M(1) = 3$, $M(2) = 0$ and $M(k)$ is arbitrary for $k \geq 3$.

Programs with polynomial bounds over nested lists (submitted TFP'09)

In types like $\mathsf{L}(\mathsf{L}(\alpha))$ the internal lists may be of different length: e.g. $[\,[1,2],\ [3,4,5],\ [\,]\,]$.

Formally, this fact may be expressed by introduction of length functions $\lambda\, k.M(k)$ that express the lengths of internal lists, where

- $M(0)$ is the length of the head list,
- $M(1)$ is the length of the element following the head,
- ... etc.

In the example above $M(0) = 2$, $M(1) = 3$, $M(2) = 0$ and $M(k)$ is arbitrary for $k \geq 3$.

Programs with polynomial bounds over nested lists (submitted TFP'09)

In types like $\mathsf{L}(\mathsf{L}(\alpha))$ the internal lists may be of different length: e.g. $[\,[1,2],\ [3,4,5],\ [\,]\,]$.

Formally, this fact may be expressed by introduction of length functions $\lambda\, k.M(k)$ that express the lengths of internal lists, where

- $M(0)$ is the length of the head list,
- $M(1)$ is the length of the element following the head,
- ... etc.

In the example above $M(0) = 2$, $M(1) = 3$, $M(2) = 0$ and $M(k)$ is arbitrary for $k \geq 3$.

Programs with polynomial bounds over nested lists

It leads to higher-order size functions, since size variables may represent functions.

Example: $conc : \mathsf{L}_n(\mathsf{L}_M(\alpha)) \rightarrow \mathsf{L}_{f_{conc}(n,M)}(\alpha)$

Given a list of lists it returns the concatenation of its elements:

$$
\begin{array}{l}
conc(l) = \\
\text{match } l \text{ with } \mid \text{nil} \Rightarrow \text{nil} \\
\qquad\qquad\quad \mid \text{cons}(hd, tl) \Rightarrow \text{let } l' = conc(tl) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{in } append(hd, l')
\end{array}
$$

For instance, on our list $[\,[1, 2],\ [3, 4, 5],\ [\,]\,]$ it returns $[\,1, 2, 3, 4, 5\,]$.

Programs with polynomial bounds over nested lists

It leads to higher-order size functions, since size variables may represent functions.

Example: $conc : \mathrm{L}_n(\mathrm{L}_M(\alpha)) \to \mathrm{L}_{f_{conc}(n,M)}(\alpha)$

Given a list of lists it returns the concatenation of its elements:

$$
\begin{array}{l}
conc(l) = \\
\text{match } l \text{ with } \mid \text{nil} \Rightarrow \text{nil} \\
\qquad\qquad\qquad \mid \text{cons}(hd, tl) \Rightarrow \text{let } l' = conc(tl) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{in } append(hd, l')
\end{array}
$$

For instance, on our list $[\,[1,2],\ [3,4,5],\ [\,]\,]$ it returns $[\,1,2,3,4,5\,]$.

Programs with polynomial bounds over nested lists

We start with generating the collection of the *rewriting rules computing the size function* $f_{conc}(n, M)$.

- *conc* is called recursively on the tail of the list argument. So, we need to express the length function of the tail, M', via the length function M of the whole list: $M'(k) = M(k+1)$.
  E.g., for our list $[\,[1,2],\ [3,4,5],\ [\,]\,]$ we have $M'(0) = 3$, $M'(1) = 0$ and $M(k)$ is arbitrary for $k \geq 2$.
  We will denote the left shift of M via M_{+1}, so $M' = M_{+1}$.

Programs with polynomial bounds over nested lists

We start with generating the collection of the *rewriting rules computing the size function* $f_{conc}(n, M)$.

- *conc* is called recursively on the tail of the list argument. So, we need to express the length function of the tail, M', via the length function M of the whole list:
  $M'(k) = M(k+1)$.
  E.g., for our list $[[1,2],\ [3,4,5],\ [\,]]$ we have $M'(0) = 3$, $M'(1) = 0$ and $M(k)$ is arbitrary for $k \geq 2$.
  We will denote the left shift of M via M_{+1}, so $M' = M_{+1}$.

Programs with polynomial bounds over nested lists

We start with generating the collection of the *rewriting rules computing the size function* $f_{conc}(n, M)$.

- *conc* is called recursively on the tail of the list argument. So, we need to express the length function of the tail, M', via the length function M of the whole list:
  $M'(k) = M(k + 1)$.
  E.g., for our list $[\,[1, 2],\ [3, 4, 5],\ [\,]\,]$ we have $M'(0) = 3$, $M'(1) = 0$ and $M(k)$ is arbitrary for $k \geq 2$.
  We will denote the left shift of M via M_{+1}, so $M' = M_{+1}$.

Programs with polynomial bounds over nested lists

We start with generating the collection of the *rewriting rules computing the size function* $f_{conc}(n, M)$.

- *conc* is called recursively on the tail of the list argument. So, we need to express the length function of the tail, M', via the length function M of the whole list: $M'(k) = M(k+1)$. E.g., for our list $[\,[1,2],\ [3,4,5],\ [\,]\,]$ we have $M'(0) = 3$, $M'(1) = 0$ and $M(k)$ is arbitrary for $k \geq 2$. We will denote the left shift of M via M_{+1}, so $M' = M_{+1}$.

Programs with polynomial bounds over nested lists

- We parse the body of *conc* and obtain the following rewriting system for its size function:

$$n = 0 \quad \vdash f_{conc}(n, M) \to 0$$
$$n \geq 1 \quad \vdash f_{conc}(n, M) \to M(0) + f_{conc}(n-1, M_{+1})$$

- As in the case of *insert*, the rewriting system is not our end result in size analysis, but is just a tool to compute closed, i.e. recursion free, forms of lower and upper bounds on the size function of *conc*.

Programs with polynomial bounds over nested lists

- We parse the body of *conc* and obtain the following rewriting system for its size function:

$$n = 0 \quad \vdash f_{conc}(n, M) \rightarrow 0$$
$$n \geq 1 \quad \vdash f_{conc}(n, M) \rightarrow M(0) + f_{conc}(n - 1, M_{+1})$$

- As in the case of *insert*, the rewriting system is not our end result in size analysis, but is just a tool to compute closed, i.e. recursion free, forms of lower and upper bounds on the size function of *conc*.

Programs with polynomial bounds over nested lists

- What to do with the higher-order parameter M? We introduce a fresh usual size variable for it, m, meaning that for all $k \geq 0$ we have $0 \leq M(k) \leq m$.
- We want to obtain a typing of the following form

$$conc : \mathsf{L}_n(\mathsf{L}_{\{i\}_{0 \leq i \leq m}}(\alpha)) \rightarrow$$
$$\mathsf{L}_{\{f_{conc\ \min}(n,m)+i\}_{0 \leq i \leq f_{conc\ \max}(n,m) - f_{conc\ \min}(n,m)}}(\alpha)$$

Programs with polynomial bounds over nested lists

- What to do with the higher-order parameter M? We introduce a fresh usual size variable for it, m, meaning that for all $k \geq 0$ we have $0 \leq M(k) \leq m$.
- We want to obtain a typing of the following form
  $conc : \mathrm{L}_n(\mathrm{L}_{\{i\}_{0 \leq i \leq m}}(\alpha)) \rightarrow$
  $\mathrm{L}_{\{f_{conc\ \min}(n,m)+i\}_{0 \leq i \leq f_{conc\ \max}(n,m) - f_{conc\ \min}(n,m)}}(\alpha)$

Programs with polynomial bounds over nested lists

- We assume the degree of lower and upper bounds: let it be $d = 2$.
- A polynomial of degree two of two variables is defined by six coefficients, so we need to know the values of $f_{conc\ \min}$ and $f_{conc\ \max}$ in six 2-dimensional points, satisfying NCA-configuration. Then we will have to solve the linear systems for the coefficients of $f_{conc\ \min}$ and $f_{conc\ \max}$. (The systems will have the form as for *cprod*.)
- We choose the nodes as for *cprod*

$$(n_1, m_1) = (1, 1) \quad (n_2, m_2) = (2, 1) \quad (n_3, m_3) = (3, 1)$$
$$(n_4, m_4) = (1, 2) \quad (n_5, m_5) = (2, 2)$$
$$(n_6, m_6) = (1, 3)$$

Programs with polynomial bounds over nested lists

- We assume the degree of lower and upper bounds: let it be $d = 2$.
- A polynomial of degree two of two variables is defined by six coefficients, so we need to know the values of $f_{conc\ \mathrm{min}}$ and $f_{conc\,\mathrm{max}}$ in six 2-dimensional points, satisfying NCA-configuration. Then we will have to solve the linear systems for the coefficients of $f_{conc\ \mathrm{min}}$ and $f_{conc\ \mathrm{max}}$. (The systems will have the form as for *cprod*.)
- We choose the nodes as for *cprod*

$$(n_1, m_1) = (1, 1) \quad (n_2, m_2) = (2, 1) \quad (n_3, m_3) = (3, 1)$$
$$(n_4, m_4) = (1, 2) \quad (n_5, m_5) = (2, 2)$$
$$(n_6, m_6) = (1, 3)$$

Programs with polynomial bounds over nested lists

- We assume the degree of lower and upper bounds: let it be $d = 2$.
- A polynomial of degree two of two variables is defined by six coefficients, so we need to know the values of $f_{conc\ \mathrm{min}}$ and $f_{conc\,\mathrm{max}}$ in six 2-dimensional points, satisfying NCA-configuration. Then we will have to solve the linear systems for the coefficients of $f_{conc\ \mathrm{min}}$ and $f_{conc\ \mathrm{max}}$. (The systems will have the form as for *cprod*.)
- We choose the nodes as for *cprod*

$$(n_1, m_1) = (1, 1) \quad (n_2, m_2) = (2, 1) \quad (n_3, m_3) = (3, 1)$$
$$(n_4, m_4) = (1, 2) \quad (n_5, m_5) = (2, 2)$$
$$(n_6, m_6) = (1, 3)$$

Programs with polynomial bounds over nested lists

- Now we need to compute $f_{conc\,\max}$ (resp. $f_{conc\ \min}$) in these nodes. We transform the rewriting system for the higher-order function f_{conc} into a rewriting system for the function f'_{conc} over numerical sets (and numbers):

$$\begin{array}{lll} n = 0 & \vdash f'_{conc}(n, \{i\}_{0 \leq i \leq m}) \to \{0\} & \\ n \geq & \vdash f'_{conc}(n, \{i\}_{0 \leq i \leq m}) \to \{i\}_{0 \leq i \leq m} + & \\ & \qquad f'_{conc}(n-1, \{i\}_{0 \leq i \leq m}) & \end{array}$$

  The second argument in the recursive call is the same as the second argument of the function f'_{conc}: this is because the elements of the tail have the same length bounds as the elements of the list. It easy to see that $f'_{conc}(n, m) \supseteq f_{conc}(n, M)$ if $M(k) \leq m$ for all k.

Programs with polynomial bounds over nested lists

- Now we need to compute $f_{conc\,\max}$ (resp. $f_{conc\ \min}$) in these nodes. We transform the rewriting system for the higher-order function f_{conc} into a rewriting system for the function f'_{conc} over numerical sets (and numbers):
  $$n = 0 \quad \vdash f'_{conc}(n, \{i\}_{0 \leq i \leq m}) \to \{0\}$$
  $$n \geq \quad \vdash f'_{conc}(n, \{i\}_{0 \leq i \leq m}) \to \{i\}_{0 \leq i \leq m} + f'_{conc}(n-1, \{i\}_{0 \leq i \leq m})$$
  The second argument in the recursive call is the same as the second argument of the function f'_{conc}: this is because the elements of the tail have the same length bounds as the elements of the list. It easy to see that $f'_{conc}(n, m) \supseteq f_{conc}(n, M)$ if $M(k) \leq m$ for all k.

Programs with polynomial bounds over nested lists

- Now we need to compute $f_{conc\,\max}$ (resp. $f_{conc\ \min}$) in these nodes. We transform the rewriting system for the higher-order function f_{conc} into a rewriting system for the function f'_{conc} over numerical sets (and numbers):

  $$n = 0 \quad \vdash f'_{conc}(n, \{i\}_{0\leq i\leq m}) \to \{0\}$$
  $$n \geq \quad \vdash f'_{conc}(n, \{i\}_{0\leq i\leq m}) \to \{i\}_{0\leq i\leq m} + f'_{conc}(n-1, \{i\}_{0\leq i\leq m})$$

  The second argument in the recursive call is the same as the second argument of the function f'_{conc}: this is because the elements of the tail have the same length bounds as the elements of the list. It easy to see that $f'_{conc}(n, m) \supseteq f_{conc}(n, M)$ if $M(k) \leq m$ for all k.

Programs with polynomial bounds over nested lists

- Now we can compute all possible values of f'_{conc} in the given nodes using the new rewriting system:

$$
\begin{array}{ll}
f'_{conc}(1, \{0,1\}) & \to \{0,1\} + f'_{conc}(0, \{0,1\}) \to \{0,1\} \\
f'_{conc}(2, \{0,1\}) & \to \{0,1\} + f'_{conc}(1, \{0,1\}) \to \{0,1,2\} \\
f'_{conc}(3, \{0,1\}) & \to \{0,1\} + f'_{conc}(2, \{0,1\}) \to \{0,1,2,3\} \\
f'_{conc}(1, \{0,1,2\}) & \to \{0,1,2\} + f'_{conc}(0, \{0,1,2\}) \to \{0,1,2\} \\
f'_{conc}(2, \{0,1,2\}) & \to \{0,1,2\} + f'_{conc}(1, \{0,1,2\}) \to \{0,1,2,3,4\} \\
f'_{conc}(1, \{0,1,2,3\}) & \to \{0,1,2,3\} + f'_{conc}(0, \{0,1,2,3\}) \to \{0,1,2,3\}
\end{array}
$$

Programs with polynomial bounds over nested lists

- Now, pick up the maximal elements of each set. They constitute the r.h.s of the linear system for the coefficients of $f_{conc\ \mathrm{max}}$:

$$
\begin{array}{l}
a_{\mathrm{max}\ 20} + a_{\mathrm{max}\ 11} + a_{\mathrm{max}\ 02} + a_{\mathrm{max}\ 10} + a_{\mathrm{max}\ 01} + a_{\mathrm{max}\ 00} = 1 \\
4a_{\mathrm{max}\ 20} + 2a_{\mathrm{max}\ 11} + a_{\mathrm{max}\ 02} + 2a_{\mathrm{max}\ 10} + a_{\mathrm{max}\ 01} + a_{\mathrm{max}\ 00} = 2 \\
9a_{\mathrm{max}\ 20} + 3a_{\mathrm{max}\ 11} + a_{\mathrm{max}\ 02} + 3a_{\mathrm{max}\ 10} + a_{\mathrm{max}\ 01} + a_{\mathrm{max}\ 00} = 3 \\
a_{\mathrm{max}\ 20} + 2a_{\mathrm{max}\ 11} + 4a_{\mathrm{max}\ 02} + a_{\mathrm{max}\ 10} + 2a_{\mathrm{max}\ 01} + a_{\mathrm{max}\ 00} = 2 \\
4a_{\mathrm{max}\ 20} + 4a_{\mathrm{max}\ 11} + 4a_{\mathrm{max}\ 02} + 2a_{\mathrm{max}\ 10} + 2a_{\mathrm{max}\ 01} + a_{\mathrm{max}\ 00} = 4 \\
a_{\mathrm{max}\ 20} + 3a_{\mathrm{max}\ 11} + 9a_{\mathrm{max}\ 02} + a_{\mathrm{max}\ 10} + 3a_{\mathrm{max}\ 01} + a_{\mathrm{max}\ 00} = 3
\end{array}
$$

- Solving this system gives that $a_{\mathrm{max}\ 11} = 1$ and the rest of the coefficients are zero. Thus, $f_{conc\ \mathrm{max}}(n, m) = nm$.
- Similarly, $f_{conc\ \mathrm{min}}(n, m) = 0$.

Programs with polynomial bounds over nested lists

- Now, pick up the maximal elements of each set. They constitute the r.h.s of the linear system for the coefficients of $f_{conc\,\max}$:

$$
\begin{array}{l}
a_{\max\ 20} + a_{\max\ 11} + a_{\max\ 02} + a_{\max\ 10} + a_{\max\ 01} + a_{\max\ 00} = 1 \\
4a_{\max\ 20} + 2a_{\max\ 11} + a_{\max\ 02} + 2a_{\max\ 10} + a_{\max\ 01} + a_{\max\ 00} = 2 \\
9a_{\max\ 20} + 3a_{\max\ 11} + a_{\max\ 02} + 3a_{\max\ 10} + a_{\max\ 01} + a_{\max\ 00} = 3 \\
a_{\max\ 20} + 2a_{\max\ 11} + 4a_{\max\ 02} + a_{\max\ 10} + 2a_{\max\ 01} + a_{\max\ 00} = 2 \\
4a_{\max\ 20} + 4a_{\max\ 11} + 4a_{\max\ 02} + 2a_{\max\ 10} + 2a_{\max\ 01} + a_{\max\ 00} = 4 \\
a_{\max\ 20} + 3a_{\max\ 11} + 9a_{\max\ 02} + a_{\max\ 10} + 3a_{\max\ 01} + a_{\max\ 00} = 3
\end{array}
$$

- Solving this system gives that $a_{\max\ 11} = 1$ and the rest of the coefficients are zero. Thus, $f_{conc\ \max}(n, m) = nm$.
- Similarly, $f_{conc\ \min}(n, m) = 0$.

Programs with polynomial bounds over nested lists

- Now, pick up the maximal elements of each set. They constitute the r.h.s of the linear system for the coefficients of $f_{conc\ \mathrm{max}}$:

$$
\begin{array}{l}
a_{\mathrm{max}\ 20} + a_{\mathrm{max}\ 11} + a_{\mathrm{max}\ 02} + a_{\mathrm{max}\ 10} + a_{\mathrm{max}\ 01} + a_{\mathrm{max}\ 00} = 1 \\
4a_{\mathrm{max}\ 20} + 2a_{\mathrm{max}\ 11} + a_{\mathrm{max}\ 02} + 2a_{\mathrm{max}\ 10} + a_{\mathrm{max}\ 01} + a_{\mathrm{max}\ 00} = 2 \\
9a_{\mathrm{max}\ 20} + 3a_{\mathrm{max}\ 11} + a_{\mathrm{max}\ 02} + 3a_{\mathrm{max}\ 10} + a_{\mathrm{max}\ 01} + a_{\mathrm{max}\ 00} = 3 \\
a_{\mathrm{max}\ 20} + 2a_{\mathrm{max}\ 11} + 4a_{\mathrm{max}\ 02} + a_{\mathrm{max}\ 10} + 2a_{\mathrm{max}\ 01} + a_{\mathrm{max}\ 00} = 2 \\
4a_{\mathrm{max}\ 20} + 4a_{\mathrm{max}\ 11} + 4a_{\mathrm{max}\ 02} + 2a_{\mathrm{max}\ 10} + 2a_{\mathrm{max}\ 01} + a_{\mathrm{max}\ 00} = 4 \\
a_{\mathrm{max}\ 20} + 3a_{\mathrm{max}\ 11} + 9a_{\mathrm{max}\ 02} + a_{\mathrm{max}\ 10} + 3a_{\mathrm{max}\ 01} + a_{\mathrm{max}\ 00} = 3
\end{array}
$$

- Solving this system gives that $a_{\mathrm{max}\ 11} = 1$ and the rest of the coefficients are zero. Thus, $f_{conc\ \mathrm{max}}(n, m) = nm$.
- Similarly, $f_{conc\ \mathrm{min}}(n, m) = 0$.

Programs with polynomial bounds over nested lists

- The length of the output on an input of the type $\mathsf{L}_n(\mathsf{L}_M(\alpha))$ should be in the set

  $\{f_{conc\ \min}(n, m) + i\}_{0 \leq i \leq f_{conc\ \max}(n,m) - f_{conc\ \min}(n,m)} = \{i\}_{0 \leq i \leq nm}$
- To check if the computed bounds $f_{conc\ \max}(n, m) = nm$ and $f_{conc\ \min}(n, m) = 0$ are indeed correct, we need to check the following typing:

  $conc : \mathsf{L}_n(\mathsf{L}_{\{i\}_{0 \leq i \leq m}}(\alpha)) \to \mathsf{L}_{\{i\}_{0 \leq i \leq nm}}(\alpha)$

Programs with polynomial bounds over nested lists

- The length of the output on an input of the type $\mathsf{L}_n(\mathsf{L}_M(\alpha))$ should be in the set

  $\{f_{conc\ \min}(n, m) + i\}_{0 \le i \le f_{conc\ \max}(n,m) - f_{conc\ \min}(n,m)} = \{i\}_{0 \le i \le nm}$

- To check if the computed bounds $f_{conc\ \max}(n, m) = nm$ and $f_{conc\ \min}(n, m) = 0$ are indeed correct, we need to check the following typing:

  $conc : \mathsf{L}_n(\mathsf{L}_{\{i\}_{0 \le i \le m}}(\alpha)) \to \mathsf{L}_{\{i\}_{0 \le i \le nm}}(\alpha)$

Programs with polynomial bounds over nested lists

- Following the computation scheme for f'_{conc}, defined by its rewriting rules, we conclude that the following inclusions must hold
  $n = 0 \vdash \{i\}_{0 \leq i \leq nm} \supseteq \{0\}$
  $n \geq 1 \vdash \{i\}_{0 \leq i \leq nm} \supseteq \{i\}_{0 \leq i \leq m} + \{i\}_{0 \leq i \leq (n-1)m}$
- Unfolding the definition of set inclusions we obtain the following first-order entailments:
  $\forall\, n\, m \geq 0.\, n = 0 \Rightarrow \exists\, i. 0 \leq i \leq nm \wedge i = 0$
  $\forall\, n\, m\, i'\, i'' \geq 0. n \geq 1 \wedge i' \leq m \wedge i'' \leq (n-1)m \Rightarrow$
  $\exists\, i.\, 0 \leq i \leq nm \wedge i = i' + i''$
  These entailments hold.

Programs with polynomial bounds over nested lists

- Following the computation scheme for f'_{conc}, defined by its rewriting rules, we conclude that the following inclusions must hold
  $n = 0 \vdash \{i\}_{0 \leq i \leq nm} \supseteq \{0\}$
  $n \geq 1 \vdash \{i\}_{0 \leq i \leq nm} \supseteq \{i\}_{0 \leq i \leq m} + \{i\}_{0 \leq i \leq (n-1)m}$
- Unfolding the definition of set inclusions we obtain the following first-order entailments:
  $$\forall\, n\, m \geq 0.\, n = 0 \Rightarrow \exists\, i.0 \leq i \leq nm \wedge i = 0$$
  $$\forall\, n\, m\, i'\, i'' \geq 0.n \geq 1 \wedge i' \leq m \wedge i'' \leq (n-1)m \Rightarrow \exists\, i.\, 0 \leq i \leq nm \wedge i = i' + i''$$
  These entailments hold.

Programs with polynomial bounds over nested lists

Conclusion:
we have derived and proven the correctness of the lower $f_{conc\ \min}(n, m) = 0$ and the upper $f_{conc\ \max}(n, m) = nm$ polynomial size bounds for *conc*, given the internal lists of an input do not contain more than m elements.

Summary

- We can infer and check bounds for shapely programs, [12]. We know the syntactic condition sufficient for checking to be decidable in integers.
- We have extended this technique for algebraic data types [13].
- We have adopted the inference procedure for programs with lower and upper polynomial bounds, over matrix-like structures [10].
- We have been extending the method for programs over arbitrary nested lists, [11].

Summary

- We can infer and check bounds for shapely programs, [12]. We know the syntactic condition sufficient for checking to be decidable in integers.
- We have extended this technique for algebraic data types [13].
- We have adopted the inference procedure for programs with lower and upper polynomial bounds, over matrix-like structures [10].
- We have been extending the method for programs over arbitrary nested lists, [11].

Summary

- We can infer and check bounds for shapely programs, [12]. We know the syntactic condition sufficient for checking to be decidable in integers.
- We have extended this technique for algebraic data types [13].
- We have adopted the inference procedure for programs with lower and upper polynomial bounds, over matrix-like structures [10].
- We have been extending the method for programs over arbitrary nested lists, [11].

Summary

- We can infer and check bounds for shapely programs, [12]. We know the syntactic condition sufficient for checking to be decidable in integers.
- We have extended this technique for algebraic data types [13].
- We have adopted the inference procedure for programs with lower and upper polynomial bounds, over matrix-like structures [10].
- We have been extending the method for programs over arbitrary nested lists, [11].

Future work

- We have been working on what we call a stopping criterion: analyse a size recurrence and find d such that if the recurrence has a polynomial solution, then it is of a degree at most d.
- Extend the method to lower and upper polynomial bounds for programs over algebraic data types.
- Study lazy languages: how to measure closures?
- Transfer results to an object-oriented setting.
- Study higher-order programs of types like $(a_n \to b_{f(n)}) \times c_m \to d_{F(f,m)}$

Future work

- We have been working on what we call a stopping criterion: analyse a size recurrence and find d such that if the recurrence has a polynomial solution, then it is of a degree at most d.
- Extend the method to lower and upper polynomial bounds for programs over algebraic data types.
- Study lazy languages: how to measure closures?
- Transfer results to an object-oriented setting.
- Study higher-order programs of types like $(a_n \to b_{f(n)}) \times c_m \to d_{F(f,m)}$

Future work

- We have been working on what we call a stopping criterion: analyse a size recurrence and find d such that if the recurrence has a polynomial solution, then it is of a degree at most d.
- Extend the method to lower and upper polynomial bounds for programs over algebraic data types.
- Study lazy languages: how to measure closures?
- Transfer results to an object-oriented setting.
- Study higher-order programs of types like $(a_n \rightarrow b_{f(n)}) \times c_m \rightarrow d_{F(f,m)}$

Future work

- We have been working on what we call a stopping criterion: analyse a size recurrence and find d such that if the recurrence has a polynomial solution, then it is of a degree at most d.
- Extend the method to lower and upper polynomial bounds for programs over algebraic data types.
- Study lazy languages: how to measure closures?
- Transfer results to an object-oriented setting.
- Study higher-order programs of types like $(a_n \to b_{f(n)}) \times c_m \to d_{F(f,m)}$

Future work

- We have been working on what we call a stopping criterion: analyse a size recurrence and find d such that if the recurrence has a polynomial solution, then it is of a degree at most d.
- Extend the method to lower and upper polynomial bounds for programs over algebraic data types.
- Study lazy languages: how to measure closures?
- Transfer results to an object-oriented setting.
- Study higher-order programs of types like $(a_n \rightarrow b_{f(n)}) \times c_m \rightarrow d_{F(f,m)}$

References I

[1] Andreas Abel.
A Polymorphic Lambda-Calculus with Sized Higher-Order Types.
PhD thesis, Ludwig-Maximilians-Universität München, 2006.

[2] Elvira Albert, Puri Arenas, Samir Genaim, and German Puebla.
Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis.
In *Static Analysis, 15-th International Symposium*, volume 5079 of *LNCS*, pages 221–237, 2008.

References II

[3] Roberto M. Amadio.
Synthesis of max-plus quasi-interpretations.
Fundamenta Informaticae, 65(1-2):29–60, 2004.

[4] Vincent Atassi, Patrick Baillot, and Kazushige Terui.
Verification of Ptime Reducibility for System F Terms: Type Inference in Dual Light Affine Logic.
Logical Methods in Computer Science, 3(4), 2007.

[5] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen.
Quasi-interpretations, a way to control resources.
Theoretical Computer Science, 2009, to appear.

References III

[6] Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca.
A logical account of PSPACE.
In 35^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL 2008, San Francisco, January 10-12, 2008, Proceedings*, pages 121–131, 2008.

[7] Martin Hofmann and Steffen Jost.
Static prediction of heap space usage for first-order functional programs.
SIGPLAN Not., 38(1):185–197, 2003.

References IV

[8] Martin Hofmann and Steffen Jost.
Type-Based Amortised Heap-Space Analysis (for an Object-Oriented Language).
In Peter Sestoft, editor, *Proceedings of the* 15th *European Symposium on Programming (ESOP), Programming Languages and Systems*, volume 3924 of *LNCS*, pages 22–37. Springer, 2006.

[9] L. Pareto.
Sized Types.
Chalmers University of Technology, 1998.
Dissertation for the Licentiate Degree in Computing Science.

References V

[10] Olha Shkaravska, Marko van Eekelen, and Alejandro Tamalet.
Collected Size Semantics for Functional Programs.
In S.-B. Scholz, editor, *Implementation and Application of Functional Languages: 20th International Workshop, IFL 2008, Hertfordshire, UK, 2008. Revised Papers*, LNCS. Springer-Verlag, 2008.
to appear.

References VI

[11] Olha Shkaravska, Marko van Eekelen, and Alejandro Tamalet.
Collected size semantics for functional programs over polymorphic nested lists.
In Zoltan Horvath and Victoria Szok, editors, *Trends in Functional Programming 2009*. Etovos Lorand University of Budapest, 2009.

[12] Olha Shkaravska, Marko C. J. D. van Eekelen, and Ron van Kesteren.
Polynomial size analysis of first-order shapely functions.
Logical Methods in Computer Science, 5, issue 2, paper 10:1–35, 2009.
Special Issue with Selected Papers from TLCA 2007.

References VII

[13] Alejandro Tamalet, Olha Shkaravska, and Marko van Eekelen.
Size Analysis of Algebraic Data Types.
In Peter Achten, Pieter Koopman, and Marco Morazán, editors, *Trends in Functional Programming Volume 9 (TFP'08)*. Intellect Publishers, 2009.