# Complexity Analysis by Polymorphic Sized Type Inference and Constraint Solving.

**Martin Avanzini**[1]
(Joint work with Ugo Dal Lago[2])

[1]University of Innsbruck [2]Università di Bologna & INRIA, Sophia Antipolis

*ELICA meeting*

# Motivation

- worst-case **time-complexity** analysis of **functional programs**
- analysis should be **intensionally strong**, **precise**, amendable to **automation** and **modular**

# Example

$$\texttt{f} :: \mathsf{List\ Int} \to \mathsf{List\ Int} \to \mathsf{List}\ (\mathsf{Int} \times \mathsf{Int})$$

$$\texttt{f}\ \mathit{ms}\ \mathit{ns} = \texttt{filter}\ (/=)\ (\texttt{product}\ \mathit{ms}\ \mathit{ns})$$

$$\texttt{product} :: \forall\alpha\beta.\ \mathsf{List}\ \alpha \to \mathsf{List}\ \beta \to \mathsf{List}\ (\alpha \times \beta)$$

$$\texttt{product}\ \mathit{ms}\ \mathit{ns} = \texttt{foldr}\ (\lambda\ m\ ps.\ \texttt{foldr}\ (\lambda\ n.\ \texttt{Cons}\ (n, m))\ ps\ ns)\ \texttt{Nil}\ \mathit{ms}$$

$$\begin{aligned}
&\texttt{filter} :: \forall\alpha.\ (\alpha \to \mathsf{Bool}) \to \mathsf{List}\ \alpha \to \mathsf{List}\ \alpha \\
&\texttt{filter}\ p\ \texttt{Nil} &&= \texttt{Nil} \\
&\texttt{filter}\ p\ (\texttt{Cons}\ x\ xs) &&= \mathbf{if}\ p\ x \\
& && \quad \mathbf{then}\ \texttt{Cons}\ x\ (\texttt{filter}\ p\ xs) \\
& && \quad \mathbf{else}\ \texttt{filter}\ p\ xs
\end{aligned}$$

$$\begin{aligned}
&\texttt{foldr} :: \forall\alpha\beta.\ (\alpha \to \beta \to \beta) \to \beta \to \mathsf{List}\ \alpha \to \beta \\
&\texttt{foldr}\ f\ b\ \texttt{Nil} &&= b \\
&\texttt{foldr}\ f\ b\ (\texttt{Cons}\ x\ xs) &&= f\ x\ (\texttt{foldr}\ f\ b\ xs)
\end{aligned}$$

# Example

$\texttt{f} :: \mathsf{List\ Int} \to \mathsf{List\ Int} \to \mathsf{List}\ (\mathsf{Int} \times \mathsf{Int})$

$\texttt{f}\ ms\ ns = \texttt{filter}\ (/=)\ (\texttt{product}\ ms\ ns)$

$\texttt{product} :: \forall \alpha\beta.\ \mathsf{List}\ \alpha \to \mathsf{List}\ \beta \to \mathsf{List}\ (\alpha \times \beta)$

$\texttt{product}\ ms\ ns = \texttt{foldr}\ (\lambda\ m\ ps.\ \texttt{foldr}\ (\lambda\ n.\ \texttt{Cons}\ (n, m))\ ps\ ns)\ \texttt{Nil}\ ms$

$\texttt{filter} :: \forall \alpha.\ (\alpha \to \mathsf{Bool}) \to \mathsf{List}\ \alpha \to \mathsf{List}\ \alpha$

$$\begin{aligned}
&\texttt{filter}\ p\ \texttt{Nil} &&= \texttt{Nil} \\
&\texttt{filter}\ p\ (\texttt{Cons}\ x\ xs) &&= \textbf{if}\ p\ x \\
& &&\quad \textbf{then}\ \texttt{Cons}\ x\ (\texttt{filter}\ p\ xs) \\
& &&\quad \textbf{else}\ \texttt{filter}\ p\ xs
\end{aligned}$$

$\texttt{foldr} :: \forall \alpha\beta.\ (\alpha \to \beta \to \beta) \to \beta \to \mathsf{List}\ \alpha \to \beta$

$$\begin{aligned}
&\texttt{foldr}\ f\ b\ \texttt{Nil} &&= b \\
&\texttt{foldr}\ f\ b\ (\texttt{Cons}\ x\ xs) &&= f\ x\ (\texttt{foldr}\ f\ b\ xs)
\end{aligned}$$

# Higher-order combinators are hard to analyse...

$$\texttt{foldr}\ (\circ)\ b\ [e_1, e_2, \ldots, e_n] = e_1 \circ (e_2 \circ (\ldots (e_n \circ b) \ldots))$$

complexity depends very much on how $(\circ)$ uses its arguments

# Higher-order combinators are hard to analyse...

$$\texttt{foldr}\ (\circ)\ b\ [e_1, e_2, \ldots, e_n] = e_1 \circ (e_2 \circ (\ldots (e_n \circ b) \ldots))$$

complexity depends very much on how $(\circ)$ uses its arguments

1. `foldr append Nil` $[e_1, e_2, \ldots, e_n]$
   $\Rightarrow$ complexity $O(n \cdot m)$, where $m$ binds length of $e_i$'s

# Higher-order combinators are hard to analyse...

$$\texttt{foldr}\ (\circ)\ b\ [e_1, e_2, \ldots, e_n] = e_1 \circ (e_2 \circ (\ldots (e_n \circ b) \ldots))$$

complexity depends very much on how $(\circ)$ uses its arguments

1. $\texttt{foldr append Nil}\ [e_1, e_2, \ldots, e_n]$
   $\Rightarrow$ complexity $O(n \cdot m)$, where $m$ binds length of $e_i$'s
2. $\texttt{foldr (flip append) Nil}\ [e_1, e_2, \ldots, e_n]$
   $\Rightarrow$ complexity $O(\sum_{i=0}^{n-1} i \cdot m) = O(n^2 \cdot m)$

# Higher-order combinators are hard to analyse...

$$\texttt{foldr}\ (\circ)\ b\ [e_1, e_2, \ldots, e_n] = e_1 \circ (e_2 \circ (\ldots(e_n \circ b)\ldots))$$

complexity depends very much on how $(\circ)$ uses its arguments

1. $\texttt{foldr append Nil}\ [e_1, e_2, \ldots, e_n]$
   $\Rightarrow$ complexity $O(n \cdot m)$, where $m$ binds length of $e_i$'s
2. $\texttt{foldr (flip append) Nil}\ [e_1, e_2, \ldots, e_n]$
   $\Rightarrow$ complexity $O(\sum_{i=0}^{n-1} i \cdot m) = O(n^2 \cdot m)$

size analysis crucial step in runtime analysis

# Higher-order combinators are hard to analyse...

$$\texttt{foldr}\ (\circ)\ b\ [e_1, e_2, \dots, e_n] = e_1 \circ (e_2 \circ (\dots (e_n \circ b) \dots))$$

complexity depends very much on how $(\circ)$ uses its arguments

1. $\texttt{foldr append Nil}\ [e_1, e_2, \dots, e_n]$
   $\Rightarrow$ complexity $O(n \cdot m)$, where $m$ binds length of $e_i$'s
2. $\texttt{foldr (flip append) Nil}\ [e_1, e_2, \dots, e_n]$
   $\Rightarrow$ complexity $O(\sum_{i=0}^{n-1} i \cdot m) = O(n^2 \cdot m)$
3. $\texttt{foldr}\ (\lambda e.\texttt{Cons}\ (\texttt{append}\ xs\ e))\ \texttt{Nil}\ [e_1, e_2, \dots, e_n]$
   $\Rightarrow$ complexity $O(k \cdot n)$ where $k$ is the length of $xs$.

size analysis crucial step in runtime analysis

# Higher-order combinators are hard to analyse...

$$\texttt{foldr}\ (\circ)\ b\ [e_1, e_2, \ldots, e_n] = e_1 \circ (e_2 \circ (\ldots (e_n \circ b) \ldots))$$

complexity depends very much on how $(\circ)$ uses its arguments

1. $\texttt{foldr append Nil}\ [e_1, e_2, \ldots, e_n]$
   $\Rightarrow$ complexity $O(n \cdot m)$, where $m$ binds length of $e_i$'s
2. $\texttt{foldr (flip append) Nil}\ [e_1, e_2, \ldots, e_n]$
   $\Rightarrow$ complexity $O(\sum_{i=0}^{n-1} i \cdot m) = O(n^2 \cdot m)$
3. $\texttt{foldr}\ (\lambda e.\texttt{Cons}\ (\texttt{append}\ xs\ e))\ \texttt{Nil}\ [e_1, e_2, \ldots, e_n]$
   $\Rightarrow$ complexity $O(k \cdot n)$ where $k$ is the length of $xs$.

size analysis crucial step in runtime analysis

complexity depends not only on arguments, but also on the environment

# Sized-types

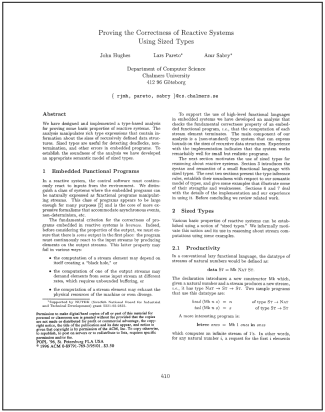
1. annotate datatypes with sizes

   $\mathsf{List}_1\ \alpha, \mathsf{List}_2\ \alpha, \mathsf{List}_3\ \alpha, \ldots$

2. extended type system that enables reasoning about sizes

   $\mathtt{append} :: \forall ij.\ \mathsf{List}_i\ \alpha \to \mathsf{List}_j\ \alpha \to \mathsf{List}_{i+j}\ \alpha$

3. inference generates set of constraints, solved by external tool (e.g. SMT solver)



Proving the Correctness of Reactive Systems Using Sized Types

John Hughes Lars Pareto* Amr Sabry*

Department of Computer Science
Chalmers University
412 96 Göteborg

{ rjmh, pareto, sabry }@cs.chalmers.se

**Abstract**

We have designed and implemented a type-based analysis for proving some basic properties of reactive systems. The analysis manipulates rich type expressions that contain information about the sizes of recursively defined data structures. Sized types are useful for detecting deadlocks, nontermination, and other errors in embedded programs. To establish the soundness of the analysis we have developed an appropriate semantic model of sized types.

**1 Embedded Functional Programs**

In a reactive system, the control software must continuously react to inputs from the environment. We distinguish a class of systems where the embedded programs can be naturally expressed as functional programs manipulating streams. This class of programs appears to be large enough for many purposes [2] and is the core of more expressive formalisms that accommodate asynchronous events, non-determinism, etc.

The fundamental criterion for the correctness of programs embedded in reactive systems is *liveness*. Indeed, before considering the properties of the output, we must ensure that there is *some* output in the first place: the program must continuously react to the input streams by producing elements on the output streams. This latter property may fail in various ways:

- the computation of a stream element may depend on itself creating a "black hole," or
- the computation of one of the output streams may demand elements from some input stream at different rates, which requires unbounded buffering, or
- the computation of a stream element may exhaust the physical resources of the machine or even diverge.

*Supported by NUTEK (Swedish National Board for Industrial and Technical Development) grant 5321-93-2833.

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee.
POPL '96, St. Petersburg FLA USA
© 1996 ACM 0-89791-769-3/95/01..$3.50

To support the use of high-level functional languages in embedded systems we have developed an analysis that checks the fundamental correctness property of an embedded functional program, *i.e.*, that the computation of each stream element terminates. The main component of our analysis is a (non-standard) type system that can express bounds on the sizes of recursive data structures. Experience with the implementation indicates that the system works remarkably well for small but realistic programs.

The next section motivates the use of sized types for reasoning about reactive systems. Section 3 introduces the syntax and semantics of a small functional language with sized types. The next two sections present the type inference rules, establish their soundness with respect to our semantic model of types, and give some examples that illustrate some of their strengths and weaknesses. Sections 6 and 7 deal with the details of the implementation and our experience in using it. Before concluding we review related work.

**2 Sized Types**

Various basic properties of reactive systems can be established using a notion of "sized types." We informally motivate this notion and its use in reasoning about stream computations using some examples.

**2.1 Productivity**

In a conventional lazy functional language, the datatype of streams of natural numbers would be defined as:

**data** ST = Mk NAT ST.

The declaration introduces a new constructor Mk which, given a natural number and a stream produces a new stream, *i.e.*, it has type NAT → ST → ST. Two sample programs that use this datatype are:

*head* (Mk *n s*) = *n* of type ST → NAT
*tail* (Mk *n s*) = *s* of type ST → ST

A more interesting program is:

**letrec** *ones* = Mk 1 *ones* **in** *ones*

which computes an infinite stream of 1's. In other words, for any natural number *i*, a request for the first *i* elements

410

# From sized-types to time complexity

**idea:** instrument program to compute complexity

$$\langle \tau \to \rho \rangle \quad \Rightarrow \quad \langle \tau \rangle \to \mathsf{N} \to (\langle \rho \rangle \times \mathsf{N});$$

# From sized-types to time complexity

idea: instrument program to compute complexity

$$\langle \tau \to \rho \rangle \quad \Rightarrow \quad \langle \tau \rangle \to \mathsf{N} \to (\langle \rho \rangle \times \mathsf{N});$$

$$\begin{aligned}
&\texttt{foldr}_3 :: \forall \alpha\beta.\ \langle \alpha \to \beta \to \beta \rangle \to \langle \beta \rangle \to \langle \mathsf{List}\ \alpha \rangle \to \mathsf{N} \to (\langle \beta \rangle \times \mathsf{N}) \\
&\texttt{foldr}_3\ f\ b\ \texttt{Nil}\qquad\qquad t = (b, \texttt{Succ}\ t) \\
&\texttt{foldr}_3\ f\ b\ (\texttt{Cons}\ x\ xs)\ t = \mathbf{let}\ (e_1, t_1) = \texttt{foldr}_3\ f\ b\ xs\ t \\
&\qquad\qquad\qquad\qquad\qquad \mathbf{in\ let}\ (e_2, t_2) = f\ x\ t_1 \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in\ let}\ (e_3, t_3) = e_2\ e_1\ t_2 \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ (e_3, \texttt{Succ}\ t_3)
\end{aligned}$$

$$\begin{aligned}
&\texttt{foldr}_1 :: \forall \alpha\beta.\ \langle \alpha \to \beta \to \beta \rangle \to \mathsf{N} \to (\langle \beta \to \mathsf{List}\ \alpha \to \beta \rangle \times \mathsf{N}) \\
&\texttt{foldr}_1\ f\ t = (\texttt{foldr}_2\ f, t)
\end{aligned}$$

$$\begin{aligned}
&\texttt{foldr}_2 :: \forall \alpha\beta.\ \langle \alpha \to \beta \to \beta \rangle \to \langle \beta \rangle \to \mathsf{N} \to (\langle \mathsf{List}\ \alpha \to \beta \rangle \times \mathsf{N}) \\
&\texttt{foldr}_2\ f\ b\ t = (\texttt{foldr}_3\ f\ b, t)
\end{aligned}$$

# Practical sized-type analysis (i)

- consider reversal of lists:

$$\begin{aligned}&\texttt{rev} :: \forall\alpha.\ \mathsf{List}\ \alpha \to \mathsf{List}\ \alpha \to \mathsf{List}\ \alpha\\ &\texttt{rev}\ \texttt{Nil}\qquad\qquad ys = ys\\ &\texttt{rev}\ (\texttt{Cons}\ x\ xs)\ ys = \texttt{rev}\ xs\ (\texttt{Cons}\ x\ ys)\end{aligned}$$

- usual let-polymorphism requires that recursive call is typed under monotype, …
- in sized-type setting, types of e.g. second argument changes from $\mathsf{List}_i\ \alpha$ to $\mathsf{List}_{i+1}\ \alpha$

# Practical sized-type analysis (i)

- consider reversal of lists:

$$\begin{aligned}
&\texttt{rev} :: \forall\alpha.\ \mathsf{List}\ \alpha \to \mathsf{List}\ \alpha \to \mathsf{List}\ \alpha\\
&\texttt{rev}\ \texttt{Nil}\qquad\qquad\quad ys = ys\\
&\texttt{rev}\ (\texttt{Cons}\ x\ xs)\ ys = \texttt{rev}\ xs\ (\texttt{Cons}\ x\ ys)
\end{aligned}$$

- usual let-polymorphism requires that recursive call is typed under monotype, ...
- in sized-type setting, types of e.g. second argument changes from $\mathsf{List}_i\ \alpha$ to $\mathsf{List}_{i+1}\ \alpha$

**extension ①:** type recursive calls with type polymorphic in size indices

# Practical sized-type analysis (ii)

- consider higher-order combinator `twice`:

$$\texttt{twice} :: \forall\alpha.\ (\alpha \to \alpha) \to \alpha \to \alpha$$
$$\texttt{twice}\ f\ x = f\ (f\ x)$$

- term `twice Succ`, where `Succ` $:: \forall i.\mathsf{N}_i \to \mathsf{N}_{i+1}$, cannot be typed

# Practical sized-type analysis (ii)

- consider higher-order combinator `twice`:

$$\texttt{twice} :: \forall\alpha.\ (\alpha \to \alpha) \to \alpha \to \alpha$$
$$\texttt{twice}\ f\ x = f\ (f\ x)$$

- term `twice Succ`, where `Succ` $:: \forall i.\mathsf{N}_i \to \mathsf{N}_{i+1}$, cannot be typed
- even when specializing $\alpha$ to $\mathsf{N}$, type in prenex form not enough

$$\texttt{twice} :: \forall ij.\ (\mathsf{N}_j \to \mathsf{N}_{j+1}) \to \mathsf{N}_i \to \mathsf{N}_{i+2}$$

# Practical sized-type analysis (ii)

- consider higher-order combinator `twice`:

$$\texttt{twice} :: \forall\alpha.\ (\alpha \to \alpha) \to \alpha \to \alpha$$
$$\texttt{twice}\ f\ x = f\ (f\ x)$$

- term `twice Succ`, where `Succ` $:: \forall i.\mathsf{N}_i \to \mathsf{N}_{i+1}$, cannot be typed
- even when specializing $\alpha$ to $\mathsf{N}$, type in prenex form not enough

$$\texttt{twice} :: \forall ij.\ (\mathsf{N}_j \to \mathsf{N}_{j+1}) \to \mathsf{N}_i \to \mathsf{N}_{i+2}$$

- concerns all function that use functional argument more than once, in particular recursive higher-order functions

# Practical sized-type analysis (ii)

- consider higher-order combinator `twice`:

$$\texttt{twice} :: \forall\alpha.\ (\alpha \to \alpha) \to \alpha \to \alpha$$
$$\texttt{twice}\ f\ x = f\ (f\ x)$$

- term `twice Succ`, where `Succ` $:: \forall i.\mathsf{N}_i \to \mathsf{N}_{i+1}$, cannot be typed
- even when specializing $\alpha$ to $\mathsf{N}$, type in prenex form not enough

$$\texttt{twice} :: \forall ij.\ (\mathsf{N}_j \to \mathsf{N}_{j+1}) \to \mathsf{N}_i \to \mathsf{N}_{i+2}$$

- concerns all function that use functional argument more than once, in particular recursive higher-order functions

**extension ②:** arbitrary-rank index polymorphic

$$\texttt{twice} :: \forall i.\ (\forall j.\mathsf{N}_j \to \mathsf{N}_{j+1}) \to \mathsf{N}_i \to \mathsf{N}_{i+2}$$
$$\texttt{foldr} :: \forall klm.(\forall ij.\mathsf{N}_i \to \mathsf{L}_j \to \mathsf{L}_{j+k}) \to \mathsf{L}_l \to \mathsf{L}_m \to \mathsf{L}_{k\cdot m+l}$$

# Sized-types revisited

## *Computational model*

| | | |
|---|---|---|
| (simple types) | $\tau, \rho ::= \mathtt{B}$ | *base type* |
| | $\mid \tau \times \rho$ | *pair type* |
| | $\mid \tau \to \rho$ | *function type* |
| (expressions) | $s, t ::= x^\tau$ | *variable* |
| | $\mid \mathtt{f}^\tau$ | *function* |
| | $\mid \mathtt{C}^\tau$ | *constructor* |
| | $\mid (s^{\tau \to \rho}\ t^\tau)^\rho$ | *application* |
| | $\mid (s^{\tau_1}, t^{\tau_2})^{\tau_1 \times \tau_2}$ | *pair* |
| | $\mid (\text{let } s^{\tau_1 \times \tau_2} \text{ be } (x^{\tau_1}, y^{\tau_2}) \text{ in } t^\rho)^\rho$ | *pair destructor* |
| (patterns) | $p ::= x^\tau \mid (\mathtt{C}^{\tau_1 \to \cdots \to \tau_n \to \mathtt{B}}\ p_1^{\tau_1} \cdots p_n^{\tau_n})^{\mathtt{B}}$ | |
| (equations) | $e ::= (\mathtt{f}\ p_1 \cdots p_n)^\tau = s^\tau$ | |

- **program** P is set of non-overlapping, left-linear equations

# Sized-types revisited

## *computational model*

- call-by-value reduction semantics

# Sized-types revisited

## *computational model*

- call-by-value reduction semantics
- plain simple types, e.g. NatList instead of List N, for simplicity
  - extension to polymorphic setting straight forward

# Sized-types revisited

## *computational model*

- call-by-value reduction semantics
- plain simple types, e.g. NatList instead of List N, for simplicity
  - extension to polymorphic setting straight forward
- no conditionals, case-expressions, $\lambda$-abstractions ...
  - does not improve expressiveness of our language
  - again straight forward to incorporate

# Sized-types revisited

## *ingredients*

| | | |
|---|---|---|
| (type) | $\tau, \rho ::= \mathtt{B}_a$ | *indexed base type* |
| | $\mid \tau \times \rho$ | *pair type* |
| | $\mid \sigma \to \tau$ | *function type* |
| (schema) | $\sigma ::= \mathtt{B}_a \mid \forall \vec{i}.\ \sigma \to \tau$ | |

# Sized-types revisited

## *ingredients*

| | | |
|---|---|---|
| (type) | $\tau, \rho ::= \mathtt{B}_a$ | *indexed base type* |
| | $\mid \tau \times \rho$ | *pair type* |
| | $\mid \sigma \to \tau$ | *function type* |
| (schema) | $\sigma ::= \mathtt{B}_a \mid \forall \vec{i}.\ \sigma \to \tau$ | |

- we suppose functions are uncurried, to simplify notions

# Sized-types revisited

*ingredients*

$$
\begin{array}{llll}
\text{(type)} & \tau, \rho ::= & \mathtt{B}_a & \textit{indexed base type} \\
 & & \mid \tau \times \rho & \textit{pair type} \\
 & & \mid \sigma \to \tau & \textit{function type} \\
\text{(schema)} & \sigma ::= & \mathtt{B}_a \mid \forall \vec{i}.\, \sigma \to \tau &
\end{array}
$$

- we suppose functions are uncurried, to simplify notions
- quantification to the right of arrow does not add in strength

# Sized-types revisited

## *ingredients*

| | | |
|---|---|---|
| (type) | $\tau, \rho ::= \mathtt{B}_a$ | *indexed base type* |
| | $\mid \tau \times \rho$ | *pair type* |
| | $\mid \sigma \to \tau$ | *function type* |
| (schema) | $\sigma ::= \mathtt{B}_a \mid \forall \vec{i}.\ \sigma \to \tau$ | |
| (size index) | $a, b ::= i$ | *index variable* |
| | $\mid f(a_1, \ldots, a_k)$ | *function application* |

- we suppose functions are uncurried, to simplify notions
- quantification to the right of arrow does not add in strength

# Sized-types revisited

## *ingredients*

| | | |
|---|---|---|
| (type) | $\tau, \rho ::= \mathrm{B}_a$ | *indexed base type* |
| | $\mid \tau \times \rho$ | *pair type* |
| | $\mid \sigma \to \tau$ | *function type* |
| (schema) | $\sigma ::= \mathrm{B}_a \mid \forall \vec{i}.\, \sigma \to \tau$ | |
| (size index) | $a, b ::= i$ | *index variable* |
| | $\mid f(a_1, \ldots, a_k)$ | *function application* |

- we suppose functions are uncurried, to simplify notions
- quantification to the right of arrow does not add in strength
- meaning to index functions $f$ given by a **weakly monotonic** interpretation $[\![f]\!] : \mathbb{N}^k \to \mathbb{N}$

# Type-checking

## *auxiliary notions*

- each **function** $\texttt{f}$ **declared** by one or more closed schemas $\sigma$, in notation $\texttt{f} :: \sigma$, obeying to the following restrictions:
  1. datatypes to the left of arrow annotated by variables

     $\texttt{half} :: \forall i. \mathsf{N}_{2 \cdot i} \rightarrow \mathsf{N}_i \quad \Rightarrow \quad \texttt{half} :: \forall i. \mathsf{N}_i \rightarrow \mathsf{N}_{i/2}$
  2. all these variables must be pairwise distinct

     $\texttt{f} :: \forall i. \mathsf{N}_i \rightarrow \mathsf{N}_i \rightarrow \tau \quad \Rightarrow \quad \texttt{f} :: \forall ij. \mathsf{N}_i \rightarrow \mathsf{N}_j \rightarrow \tau'$
  3. schema closes over all variables occurring in negative position

     $\texttt{g} :: \forall ij. \, (\mathsf{N}_i \rightarrow \mathsf{N}_{i+j}) \rightarrow \tau \quad \Rightarrow \quad \texttt{g} :: \forall j. \, (\forall i. \, \mathsf{N}_i \rightarrow \mathsf{N}_{i+j}) \rightarrow \tau$

# Type-checking

*auxiliary notions*

- each **function** $\texttt{f}$ **declared** by one or more closed schemas $\sigma$, in notation $\texttt{f} :: \sigma$, obeying to the following restrictions:
  1. datatypes to the left of arrow annotated by variables

     $\texttt{half} :: \forall i.\mathsf{N}_{2 \cdot i} \to \mathsf{N}_i \quad \Rightarrow \quad \texttt{half} :: \forall i.\mathsf{N}_i \to \mathsf{N}_{i/2}$
  2. all these variables must be pairwise distinct

     $\texttt{f} :: \forall i.\mathsf{N}_i \to \mathsf{N}_i \to \tau \quad \Rightarrow \quad \texttt{f} :: \forall ij.\mathsf{N}_i \to \mathsf{N}_j \to \tau'$
  3. schema closes over all variables occurring in negative position

     $\texttt{g} :: \forall ij.\ (\mathsf{N}_i \to \mathsf{N}_{i+j}) \to \tau \quad \Rightarrow \quad \texttt{g} :: \forall j.\ (\forall i.\ \mathsf{N}_i \to \mathsf{N}_{i+j}) \to \tau$
- **constructor declarations** *count*, e.g.

$$\texttt{Succ} :: \forall i.\ \mathsf{N}_i \to \mathsf{N}_{i+1}$$

# Type-checking

## *auxiliary notions*

- each **function** $\mathtt{f}$ **declared** by one or more closed schemas $\sigma$, in notation $\mathtt{f} :: \sigma$, obeying to the following restrictions:
  1. datatypes to the left of arrow annotated by variables
     $\mathtt{half} :: \forall i.\mathsf{N}_{2 \cdot i} \to \mathsf{N}_i \quad \Rightarrow \quad \mathtt{half} :: \forall i.\mathsf{N}_i \to \mathsf{N}_{i/2}$
  2. all these variables must be pairwise distinct
     $\mathtt{f} :: \forall i.\mathsf{N}_i \to \mathsf{N}_i \to \tau \quad \Rightarrow \quad \mathtt{f} :: \forall ij.\mathsf{N}_i \to \mathsf{N}_j \to \tau'$
  3. schema closes over all variables occurring in negative position
     $\mathtt{g} :: \forall ij.\ (\mathsf{N}_i \to \mathsf{N}_{i+j}) \to \tau \quad \Rightarrow \quad \mathtt{g} :: \forall j.\ (\forall i.\ \mathsf{N}_i \to \mathsf{N}_{i+j}) \to \tau$
- **constructor declarations** *count*, e.g.

$$\mathtt{Succ} :: \forall i.\ \mathsf{N}_i \to \mathsf{N}_{i+1}$$

- **typing judgement** are of the form

$$x_1 : \sigma_1, \cdots, x_n : \sigma_n \vdash s : \tau$$

# Type-checking

## *excerpt*

$$\frac{[\![a]\!] \leq [\![b]\!]}{\mathsf{B}_a \sqsubseteq \mathsf{B}_b} \qquad \frac{\sigma_2 \sqsubseteq \sigma_1 \quad \tau_1 \sqsubseteq \tau_2}{\sigma_1 \to \tau_1 \sqsubseteq \sigma_2 \to \tau_2} \qquad \frac{\tau_1 \sqsubseteq \tau_2\{\vec{a}/\vec{j}\} \quad \vec{i} \notin \mathsf{FV}(\forall \vec{j}.\tau_2)}{\forall \vec{i}.\tau_1 \sqsubseteq \forall \vec{j}.\tau_2}$$

Figure: subtyping.

$$\frac{\Gamma(x) = \forall \vec{i}.\tau}{\Gamma \vdash x : \tau\{\vec{a}/\vec{i}\}} \qquad \frac{\mathtt{f} :: \forall \vec{i}.\tau}{\Gamma \vdash \mathtt{f} : \tau\{\vec{a}/\vec{i}\}}$$

$$\frac{\Gamma \vdash s : (\forall \vec{i}.\rho) \to \tau \quad \Gamma \vdash t : \rho \quad \vec{i} \notin \mathsf{FV}(\Gamma)}{\Gamma \vdash s\, t : \tau} \qquad \frac{\Gamma \vdash s : \rho \quad \rho \sqsubseteq \tau}{\Gamma \vdash s : \tau}$$

Figure: type-checking.

# Subject reduction

Definition

A program P is **well-typed** if for all equations $\mathtt{f}\ p_1 \cdots\ p_n = r$ of P,

$$\Gamma \vdash_{\mathsf{FP}} \mathtt{f}\ p_1 \cdots\ p_n : \tau \implies \Gamma \vdash r : \tau\ ,$$

holds for all contexts $\Gamma$ and types $\tau$.

# Subject reduction

**Definition**

A program P is **well-typed** if for all equations $\texttt{f}\ p_1 \cdots p_n = r$ of P,

$$\Gamma \vdash_{\mathsf{FP}} \texttt{f}\ p_1 \cdots p_n : \tau \implies \Gamma \vdash r : \tau \ ,$$

holds for all contexts $\Gamma$ and types $\tau$.

- the footprint judgement assigns to terms *"most general"* type, e.g.,

$$x : \mathsf{N}, xs : \mathsf{L}_i, ys : \mathsf{L}_j \vdash_{\mathsf{FP}} \texttt{append}\ (\texttt{Cons}\ x\ xs)\ ys : \mathsf{L}_{(i+1)+j}$$

where $\texttt{append} :: \forall ij.\ \mathsf{L}_i \to \mathsf{L}_j \to \mathsf{L}_{i+j}$

# Subject reduction

**Definition**

A program P is **well-typed** if for all equations $\mathtt{f}\ p_1 \cdots\ p_n = r$ of P,

$$\Gamma \vdash_{\mathsf{FP}} \mathtt{f}\ p_1 \cdots\ p_n : \tau \implies \Gamma \vdash r : \tau\ ,$$

holds for all contexts $\Gamma$ and types $\tau$.

- the footprint judgement assigns to terms *"most general"* type, e.g.,

$$x : \mathsf{N}, xs : \mathsf{L}_i, ys : \mathsf{L}_j \vdash_{\mathsf{FP}} \mathtt{append}\ (\mathtt{Cons}\ x\ xs)\ ys : \mathsf{L}_{(i+1)+j}$$

  where $\mathtt{append} :: \forall ij.\ \mathsf{L}_i \to \mathsf{L}_j \to \mathsf{L}_{i+j}$
- it can be understood as a function $\mathsf{footprint}(s) := (\Gamma, \tau)$

# Subject reduction

**Definition**

A program P is **well-typed** if for all equations $\mathtt{f}\ p_1 \cdots p_n = r$ of P,

$$\Gamma \vdash_{\mathsf{FP}} \mathtt{f}\ p_1 \cdots p_n : \tau \implies \Gamma \vdash r : \tau \ ,$$

holds for all contexts $\Gamma$ and types $\tau$.

- the footprint judgement assigns to terms *"most general"* type, e.g.,

  $$x : \mathsf{N}, xs : \mathsf{L}_i, ys : \mathsf{L}_j \vdash_{\mathsf{FP}} \mathtt{append}\ (\mathtt{Cons}\ x\ xs)\ ys : \mathsf{L}_{(i+1)+j}$$

  where $\mathtt{append} :: \forall ij.\ \mathsf{L}_i \to \mathsf{L}_j \to \mathsf{L}_{i+j}$
- it can be understood as a function $\mathrm{footprint}(s) := (\Gamma, \tau)$

**Theorem (Subject reduction)**

*Suppose* P *is well-typed. If* $\vdash s : \tau$ *and* $s \to_{\mathsf{P}} t$ *then* $\vdash t : \tau$.

# Inference

## *overview*

- index language extended with second-order index variables $A$

$$a ::= i \mid f(a_1, \ldots, a_k) \mid A$$

- second-order index variable $A$ represents index term

# Inference

## *overview*

- index language extended with second-order index variables $A$

$$a ::= i \mid f(a_1, \ldots, a_k) \mid A$$

- second-order index variable $A$ represents index term
- **input** simply-typed program

```
concatMap :: (N → L) → L → L          lam :: (N → L) → N → L → L
concatMap f = foldr (lam f) Nil       lam f x = append (f x)
```

# Inference

*overview*

- index language extended with second-order index variables $A$

$$a ::= i \mid f(a_1, \ldots, a_k) \mid A$$

- second-order index variable $A$ represents index term
- **input** simply-typed program

$$\texttt{concatMap} :: (\mathsf{N} \to \mathsf{L}) \to \mathsf{L} \to \mathsf{L} \qquad \texttt{lam} :: (\mathsf{N} \to \mathsf{L}) \to \mathsf{N} \to \mathsf{L} \to \mathsf{L}$$

$$\texttt{concatMap}\, f = \texttt{foldr}\ (\texttt{lam}\, f)\ \texttt{Nil} \qquad \texttt{lam}\, f\, x = \texttt{append}\ (f\, x)$$

- **outputs**
  1. size-annotated type declarations
  2. semantic interpretation functions $[\![\cdot]\!]$

# Inference

## *step ①: annotation*

decorate simple types with uninterpreted indices

$$\texttt{append} :: \forall ij.\ \mathsf{L}_i \rightarrow \mathsf{L}_j \rightarrow \mathsf{L}_{apd(i,j)}$$

$$\texttt{lam} :: \forall jkl.\ (\forall i.\mathsf{N}_i \rightarrow \mathsf{L}_{f(i,j)}) \rightarrow \mathsf{N}_k \rightarrow \mathsf{L}_l \rightarrow \mathsf{L}_{lm(j,k,l)}$$

$$\texttt{foldr} :: \forall klm.\ (\forall ij.\mathsf{N}_i \rightarrow \mathsf{L}_j \rightarrow \mathsf{L}_{g(i,j,k)}) \rightarrow \mathsf{L}_l \rightarrow \mathsf{L}_m \rightarrow \mathsf{L}_{fld(k,l,m)}$$

# Inference

## *step ②: constraint generation*

for all equations in P, generate two sets of constraints, e.g. for

$$\texttt{lam}\, f\, x = \texttt{append}\, (f\, x)$$

as follows

# Inference

## *step ②: constraint generation*

for all equations in P, generate two sets of constraints, e.g. for

$$\texttt{lam}\, f\, x = \texttt{append}\, (f\, x)$$

as follows

1. **footprint** gives environment and template type of left-hand side

$$\text{footprint}(\texttt{lam}\, f\, x) = (\underbrace{\{f : \forall i.\mathsf{N}_i \to \mathsf{L}_{f(i,j)}, x : \mathsf{N}_k\}}_{:=\Gamma}, \underbrace{\mathsf{L}_l \to \mathsf{L}_{lm(j,k,l)}}_{:=\tau})$$

# Inference

## *step ②: constraint generation*

for all equations in P, generate two sets of constraints, e.g. for

$$\texttt{lam}\, f\, x = \texttt{append}\, (f\, x)$$

as follows

1. **footprint** gives environment and template type of left-hand side

$$\text{footprint}(\texttt{lam}\, f\, x) = (\underbrace{\{f : \forall i.\mathsf{N}_i \to \mathsf{L}_{f(i,j)}, x : \mathsf{N}_k\}}_{:=\Gamma}, \underbrace{\mathsf{L}_l \to \mathsf{L}_{lm(j,k,l)}}_{:=\tau})$$

2. **inference** $\text{infer}(\Gamma, r)$ on right-hand side $r$ generates template for $r$ and first set of constraints

$$\text{infer}(\Gamma, \texttt{append}\, (f\, x)) = (\mathsf{L}_{A_3} \to \mathsf{L}_{apd(A_2,A_3)}, \{f(A_1, j) \leq A_2, k \leq A_1\})$$

# Inference

## *step ②: constraint generation*

for all equations in P, generate two sets of constraints, e.g. for

$$\texttt{lam}\; f\; x = \texttt{append}\; (f\; x)$$

as follows

1. **footprint** gives environment and template type of left-hand side

$$\text{footprint}(\texttt{lam}\; f\; x) = (\underbrace{\{f : \forall i. \mathsf{N}_i \to \mathsf{L}_{f(i,j)}, x : \mathsf{N}_k\}}_{:=\Gamma}, \underbrace{\mathsf{L}_l \to \mathsf{L}_{lm(j,k,l)}}_{:=\tau})$$

2. **inference** $\text{infer}(\Gamma, r)$ on right-hand side $r$ generates template for $r$ and first set of constraints

$$\text{infer}(\Gamma, \texttt{append}\; (f\; x)) = (\mathsf{L}_{A_3} \to \mathsf{L}_{apd(A_2,A_3)}, \{f(A_1, j) \leq A_2, k \leq A_1\})$$

3. **well-typedness** condition enforced by second set of constraints

$$\text{subtypeOf}(\mathsf{L}_{A_3} \to \mathsf{L}_{apd(A_1,A_2)}, \tau) = \{l \leq A_3, apd(A_1, A_2) \leq lm(j, k, l)\}$$

# Inference

## *step ②: constraint generation*

$$\frac{}{\{a \leq b\} \vdash_{\mathsf{ST}} \mathsf{B}_a \sqsubseteq \mathsf{B}_b} \qquad \frac{C_1 \vdash_{\mathsf{ST}} \sigma_2 \sqsubseteq \sigma_1 \quad C_2 \vdash_{\mathsf{ST}} \tau_1 \sqsubseteq \tau_2}{C_1 \cup C_2 \vdash_{\mathsf{ST}} \sigma_1 \to \tau_1 \sqsubseteq \sigma_2 \to \tau_2}$$

$$\frac{C \vdash_{\mathsf{ST}} \tau_1 \sqsubseteq \tau_2\{\vec{a}/\vec{j}\} \quad \vec{i} \notin \mathsf{FV}(\forall \vec{j}.\tau_2)}{C \vdash_{\mathsf{ST}} \forall \vec{i}.\tau_1 \sqsubseteq \forall \vec{j}.\tau_2}$$

Figure: subtyping.

$$\frac{\Gamma(x) = \forall \vec{i}.\tau}{\emptyset; \Gamma \vdash_{\mathsf{I}} x : \tau\{\vec{a}/\vec{i}\}} \qquad \frac{\mathtt{f} :: \forall \vec{i}.\tau}{\emptyset; \Gamma \vdash_{\mathsf{I}} \mathtt{f} : \tau\{\vec{a}/\vec{i}\}}$$

$$\frac{C_1; \Gamma \vdash_{\mathsf{I}} s : (\forall \vec{i}.\rho) \to \tau \quad C_2; \Gamma \vdash_{\mathsf{I}} t : \rho' \quad C_3 \vdash_{\mathsf{ST}} \rho' \sqsubseteq \rho \quad \vec{i} \notin \mathsf{FV}(\Gamma)}{C_1, C_2, C_3; \Gamma \vdash_{\mathsf{I}} s\ t : \tau}$$

Figure: type inference.

# Inference

## *step ②: constraint generation*

$$\frac{}{\{a \leq b\} \vdash_{ST} \mathrm{B}_a \sqsubseteq \mathrm{B}_b} \qquad \frac{C_1 \vdash_{ST} \sigma_2 \sqsubseteq \sigma_1 \quad C_2 \vdash_{ST} \tau_1 \sqsubseteq \tau_2}{C_1 \cup C_2 \vdash_{ST} \sigma_1 \to \tau_1 \sqsubseteq \sigma_2 \to \tau_2}$$

$$\frac{C \vdash_{ST} \tau_1 \sqsubseteq \tau_2\{\vec{a}/\vec{j}\} \quad \vec{i} \notin \mathrm{FV}(\forall\vec{j}.\tau_2)}{C \vdash_{ST} \forall\vec{i}.\tau_1 \sqsubseteq \forall\vec{j}.\tau_2}$$

Figure: subtyping.

$$\frac{\Gamma(x) = \forall\vec{i}.\tau}{\emptyset;\Gamma \vdash_I x : \tau\{\vec{a}/\vec{i}\}} \qquad \frac{\mathtt{f} :: \forall\vec{i}.\tau}{\emptyset;\Gamma \vdash_I \mathtt{f} : \tau\{\vec{a}/\vec{i}\}}$$

$$\frac{C_1;\Gamma \vdash_I s : (\forall\vec{i}.\rho) \to \tau \quad C_2;\Gamma \vdash_I t : \rho' \quad C_3 \vdash_{ST} \rho' \sqsubseteq \rho \quad \vec{i} \notin \mathrm{FV}(\Gamma)}{C_1, C_2, C_3;\Gamma \vdash_I s\ t : \tau}$$

Figure: type inference.

# Inference

## *step ②: constraint generation*

$$\frac{}{\{a \leq b\} \vdash_{\mathsf{ST}} \mathrm{B}_a \sqsubseteq \mathrm{B}_b} \qquad \frac{C_1 \vdash_{\mathsf{ST}} \sigma_2 \sqsubseteq \sigma_1 \quad C_2 \vdash_{\mathsf{ST}} \tau_1 \sqsubseteq \tau_2}{C_1 \cup C_2 \vdash_{\mathsf{ST}} \sigma_1 \to \tau_1 \sqsubseteq \sigma_2 \to \tau_2}$$

$$\frac{C \vdash_{\mathsf{ST}} \tau_1 \sqsubseteq \tau_2\{\vec{A}/\vec{j}\} \quad \vec{i} \notin \mathsf{FV}(\forall \vec{j}.\tau_2) \quad \vec{A}\ \textit{fresh}}{C \vdash_{\mathsf{ST}} \forall \vec{i}.\tau_1 \sqsubseteq \forall \vec{j}.\tau_2}$$

Figure: subtyping.

$$\frac{\Gamma(x) = \forall \vec{i}.\tau \quad \vec{A}\ \textit{fresh}}{\emptyset;\Gamma \vdash_{\mathsf{I}} x : \tau\{\vec{A}/\vec{i}\}} \qquad \frac{\mathtt{f} :: \forall \vec{i}.\tau \quad \vec{A}\ \textit{fresh}}{\emptyset;\Gamma \vdash_{\mathsf{I}} \mathtt{f} : \tau\{\vec{A}/\vec{i}\}}$$

$$\frac{C_1;\Gamma \vdash_{\mathsf{I}} s : (\forall \vec{i}.\rho) \to \tau \quad C_2;\Gamma \vdash_{\mathsf{I}} t : \rho' \quad C_3 \vdash_{\mathsf{ST}} \rho' \sqsubseteq \rho \quad \vec{i} \notin \mathsf{FV}(\Gamma)}{C_1, C_2, C_3;\Gamma \vdash_{\mathsf{I}} s\ t : \tau}$$

Figure: type inference.

# Inference

## *step ②: constraint generation*

$$\frac{}{\{a \leq b\} \vdash_{ST} \mathsf{B}_a \sqsubseteq \mathsf{B}_b} \qquad \frac{C_1 \vdash_{ST} \sigma_2 \sqsubseteq \sigma_1 \quad C_2 \vdash_{ST} \tau_1 \sqsubseteq \tau_2}{C_1 \cup C_2 \vdash_{ST} \sigma_1 \to \tau_1 \sqsubseteq \sigma_2 \to \tau_2}$$

$$\frac{C \vdash_{ST} \tau_1 \sqsubseteq \tau_2\{\vec{A}/\vec{j}\} \quad \vec{i} \notin FV(\forall \vec{j}.\tau_2) \quad \vec{A} \textit{ fresh}}{C, \vec{i} \notin_{sol} \mathsf{SOVars}(\tau_1) \cup \mathsf{SOVars}(\tau_2) \vdash_{ST} \forall \vec{i}.\tau_1 \sqsubseteq \forall \vec{j}.\tau_2}$$

Figure: subtyping.

$$\frac{\Gamma(x) = \forall \vec{i}.\tau \quad \vec{A} \textit{ fresh}}{\emptyset; \Gamma \vdash_I x : \tau\{\vec{A}/\vec{i}\}} \qquad \frac{\mathtt{f} :: \forall \vec{i}.\tau \quad \vec{A} \textit{ fresh}}{\emptyset; \Gamma \vdash_I \mathtt{f} : \tau\{\vec{A}/\vec{i}\}}$$

$$\frac{C_1; \Gamma \vdash_I s : (\forall \vec{i}.\rho) \to \tau \quad C_2; \Gamma \vdash_I t : \rho' \quad C_3 \vdash_{ST} \rho' \sqsubseteq \rho \quad \vec{i} \notin FV(\Gamma)}{C_1, C_2, C_3, \vec{i} \notin_{sol} \mathsf{SOVars}(\Gamma) \cup \mathsf{SOVars}(\rho); \Gamma \vdash_I s\ t : \tau}$$

Figure: type inference.

# Inference

## *step ③: constraint solving*

- find a model $(\vartheta, [\![\cdot]\!])$ for collected constraints $C$ consisting of
  - $\vartheta$ assigns index terms to second-order variables
  - $[\![\cdot]\!]$ assigns meaning to index functions

$$(\vartheta, [\![\cdot]\!]) \vDash C \quad :\Longleftrightarrow \quad \begin{cases} [\![l\vartheta]\!] \leq [\![r\vartheta]\!] & \text{for all } l \leq r \in C \\ i \notin \mathsf{FV}(\vartheta(A)) & \text{for all } (i \notin A) \in C \end{cases}$$

# Inference

## *step ③: constraint solving*

- find a model $(\vartheta, [\![\cdot]\!])$ for collected constraints $C$ consisting of
  - $\vartheta$ assigns index terms to second-order variables
  - $[\![\cdot]\!]$ assigns meaning to index functions

$$(\vartheta, [\![\cdot]\!]) \vDash C \quad :\Longleftrightarrow \quad \begin{cases} [\![l\vartheta]\!] \leq [\![r\vartheta]\!] & \textit{for all } l \leq r \in C \\ i \notin \mathsf{FV}(\vartheta(A)) & \textit{for all } (i \notin A) \in C \end{cases}$$

1. eliminate second-order variables by **skolemization**

$$\begin{array}{rcl} f(A_1, j) \leq A_2 & & f(sk_1(k), j) \leq sk_2(k, j) \\ k \leq A_1 & & k \leq sk_1(k) \\ l \leq A_3 & \Longrightarrow & l \leq sk_3(l) \\ apd(A_1, A_2) \leq lm(j, k, l) & & apd(sk_1(k), sk_2(k, j)) \leq lm(j, k, l) \\ \dots & & \dots \end{array}$$

# Inference

## *step ③: constraint solving*

- find a model $(\vartheta, [\![\cdot]\!])$ for collected constraints $C$ consisting of
  - $\vartheta$ assigns index terms to second-order variables
  - $[\![\cdot]\!]$ assigns meaning to index functions

$$(\vartheta, [\![\cdot]\!]) \vDash C \quad :\Longleftrightarrow \quad \begin{cases} [\![l\vartheta]\!] \leq [\![r\vartheta]\!] & \textit{for all } l \leq r \in C \\ i \notin \mathsf{FV}(\vartheta(A)) & \textit{for all } (i \notin A) \in C \end{cases}$$

1. eliminate second-order variables by **skolemization**

$$\begin{aligned} f(A_1, j) &\leq A_2 \\ k &\leq A_1 \\ l &\leq A_3 \\ apd(A_1, A_2) &\leq lm(j,k,l) \\ &\dots \end{aligned} \quad \Longrightarrow \quad \begin{aligned} f(sk_1(k), j) &\leq sk_2(k,j) \\ k &\leq sk_1(k) \\ l &\leq sk_3(l) \\ apd(sk_1(k), sk_2(k,j)) &\leq lm(j,k,l) \\ &\dots \end{aligned}$$

2. first-order constraints solvable with our tool `GUBS`

# Soundness and relative completeness

Theorem (Soundness and Completeness)

*Program* P *is well-typed if and only if* $(\vartheta, [\![\cdot]\!]) \vDash C$ *for some* $(\vartheta, [\![\cdot]\!])$ *and constraints* C *generated as explained before.*

# Soundness and relative completeness

Theorem (Soundness and Completeness)

*Program* P *is well-typed if and only if* $(\vartheta, [\![\cdot]\!]) \vDash C$ *for some* $(\vartheta, [\![\cdot]\!])$ *and constraints* C *generated as explained before.*

Essential proof steps.

1. $C \vdash_{\mathsf{ST}} \tau \sqsubseteq \rho \iff \tau\vartheta \sqsubseteq \rho\vartheta$, e.g.,

$$\frac{}{\{a \leq b\} \vdash_{\mathsf{ST}} \mathrm{B}_a \sqsubseteq \mathrm{B}_b} \iff \frac{[\![a\vartheta]\!] \leq [\![b\vartheta]\!]}{\mathrm{B}_{a\vartheta} \sqsubseteq \mathrm{B}_{b\vartheta}}$$

since $(\vartheta, [\![\cdot]\!]) \vDash \{a \leq b\} \implies [\![a\vartheta]\!] \leq [\![b\vartheta]\!]$

# Soundness and relative completeness

**Theorem (Soundness and Completeness)**

*Program P is well-typed if and only if* $(\vartheta, [\![\cdot]\!]) \vDash C$ *for some* $(\vartheta, [\![\cdot]\!])$ *and constraints C generated as explained before.*

**Essential proof steps.**

1. $C \vdash_{ST} \tau \sqsubseteq \rho \iff \tau\vartheta \sqsubseteq \rho\vartheta$, e.g.,

$$\frac{}{\{a \leq b\} \vdash_{ST} \mathrm{B}_a \sqsubseteq \mathrm{B}_b} \iff \frac{[\![a\vartheta]\!] \leq [\![b\vartheta]\!]}{\mathrm{B}_{a\vartheta} \sqsubseteq \mathrm{B}_{b\vartheta}}$$

   since $(\vartheta, [\![\cdot]\!]) \vDash \{a \leq b\} \implies [\![a\vartheta]\!] \leq [\![b\vartheta]\!]$

2. $C; \Gamma \vdash_I s : \tau \iff \Gamma \vdash s : \tau\vartheta$, e.g.

$$\frac{\Gamma(x) = \forall \vec{i}.\tau \quad \vec{A} \text{ fresh}}{\emptyset; \Gamma \vdash_I x : \tau\{\vec{A}/\vec{i}\}} \iff \frac{\Gamma(x) = \forall \vec{i}.\tau}{\Gamma \vdash x : \tau\{\vartheta(\vec{A})/\vec{i}\}}$$

□

# Conclusion

- **fully polymorphic sized-type system**
- **ticking transformation** reduces runtime-complexity to size analysis

# Conclusion

- **fully polymorphic sized-type system**
- **ticking transformation** reduces runtime-complexity to size analysis
- **fully working implementation** on top of Hindley-Milner type system, including constraint solver

`http://cl-informatik.uibk.ac.at/zini`

# Conclusion

- **fully polymorphic sized-type system**
- **ticking transformation** reduces runtime-complexity to size analysis
- **fully working implementation** on top of Hindley-Milner type system, including constraint solver

  `http://cl-informatik.uibk.ac.at/zini`

- currently, whole program analysis, but ...
  - each (mutual) recursive definition gives rise to an SCC
  - bottom-up per SCC-analysis implemented