Type-Based Termination and Productivity Checking

Andreas Abel
Dept. of Comp. Sci., Chalmers

TCS Oberseminar, LMU Munich
July 19, 2005

Work supported by: GKLI (DFG), TYPES, APPSEM-II and CoVer (SSF)

Short CV

1999 Diploma from this university
Major in computer science, minor in mathematics
Diplomathesis: *termination checker* foetus *for structural recursion*

1999-2003 Ph.D. student at this chair in the PhD program *Logic in Computer Science*:

2000/01 Visit to Frank Pfenning at Carnegie-Mellon, Pittsburgh, USA: Development of a *tut*orial proof *ch*ecker (Tutch) for constructive logics

Short CV (cont.)

2004-today Postdoc at Chalmers, Göteborg, Sweden
Verifying Haskell programs using First-Order Logic and Type Theory

Oct 2005(?!) Ph.D. from this university *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*

Talk outline

1. Introduction to termination
2. Inductive types and a recursion principle
3. $\mathsf{F}_{\widehat{\omega}}$—a type system for termination
4. Examples: the type system at work
5. Productivity via coinduction
6. Achieved results and future work

Termination

Slide 5

- Question: *Will the run of a program eventually halt?*
- Undecidable for Turing-complete programming languages (Halteproblem).
- No termination checker can give a definitive answer for all programs.
- Problem still interesting for:
  - optimization and program specialization
  - total correctness of programs
  - theorem proving

Termination for theorem proving

Slide 6

- Inductive theorem provers: e.g., Agda, Coq, LEGO, Twelf.
- Some proofs are *tree-shaped deriviations*, e.g., proof that $[a, 0] = [b, 0]$.

$$\dfrac{a = b \qquad \dfrac{0 = 0 \qquad [\,] = [\,]}{(0 :: [\,]) = (0 :: [\,])}}{a :: (0 :: [\,]) = b :: (0 :: [\,])}$$

- Some proofs are *recursive programs*, manipulating derivations.
- E.g., proof of $(l_1 = l_2) \to (l_2 = l_3) \to (l_1 = l_3)$.
- Only *terminating* programs denote valid proofs.
- E.g., program $\mathsf{trans}\, d_1\, d_2 = \mathsf{trans}\, d_1\, d_2$ has to be rejected.

Termination of Functions Over Inductive Types

Slide 7

- For termination, only structure of trees is interesting.
- Structure of these trees can be represented by *inductive types*.
- More inductive types:
  - lists
  - binary trees
  - natural numbers
  - tree ordinals

Inductive types

Slide 8

- Semantical perspective: types are *value sets*.
- Example: integer lists
  - $[\,]$ is an int. list
  - if x is an int. and xs an int. list, then $x :: xs$ is an int. list
- Least solution of type equation

$$\mathsf{List\ Int} = \{[\,]\} \cup \{x :: xs \mid x \in \mathsf{Int}\ \text{ and } xs \in \mathsf{List\ Int}\}$$

- Abstracting away the names

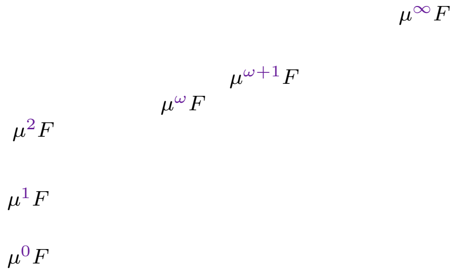
$$\mathsf{List\ Int} = 1 + \mathsf{Int} \times \mathsf{List\ Int}$$

- Definable as least fixed-point $\mu^\infty F$ of some type operator F

$$\mathsf{List\ Int} := \mu^\infty(\lambda X.\, 1 + \mathsf{Int} \times X)$$

Iterating to the least fixed point

Slide 9



Iterating to the least fixed point

Slide 10

- The least fixed point is reachable from below by ordinal iteration:

$$\begin{aligned} \mu^{0} F &= \emptyset \\ \mu^{\alpha+1} F &= F\,(\mu^{\alpha}\,F) \\ \mu^{\lambda} F &= \textstyle\bigcup_{\alpha<\lambda} \mu^{\alpha} F \end{aligned}$$

- E.g., $\mathsf{List}^{\alpha}\ \mathsf{Int} := \mu^{\alpha}(\lambda X.\, 1 + \mathsf{Int} \times X)$ contains integer lists of length $< \alpha$.
- $\mathsf{List}^{\omega}\ \mathsf{Int}$ is already the least fixed point.
- List constructors definable:

$$\begin{aligned} [\,] &\in \mathsf{List}^{\alpha+1}\,\mathsf{Int} \\ (::) &\in \mathsf{Int} \to \mathsf{List}^{\alpha}\,\mathsf{Int} \to \mathsf{List}^{\alpha+1}\,\mathsf{Int} \end{aligned}$$

Recursive functions over inductive types

Slide 11

- E.g., we want to define list summation $\mathsf{sum} \in \mathsf{List}^{\omega}\,\mathsf{Int} \to \mathsf{Int}$.
- Recursive program:

$$\begin{array}{lll} \mathsf{sum} \quad [\,] & = & 0 \\ \mathsf{sum} \quad (x :: xs) & = & x + \mathsf{sum}\, xs \end{array}$$

- Via fixed-point combinator $\mathsf{fix}\, f = f\,(\mathsf{fix}\, f)$.

$$\mathsf{sum} = \mathsf{fix}\ \underbrace{(\lambda sum.\lambda l.\ \mathsf{match}\ l\ \mathsf{with} \begin{array}{lll} \mathsf{nil} & \mapsto & 0 \\ (x :: xs) & \mapsto & x + sum\, xs) \end{array}}_{f}$$

- How to prove that sum is well defined, i.e., terminating?

A recursion principle from transfinite induction

Slide 12

- Rule for transfinite induction:

$$\frac{P(0) \qquad P(\alpha) \to P(\alpha+1) \qquad (\forall \alpha < \lambda.\, P(\alpha)) \to P(\lambda)}{P(\beta)}$$

- Use transfinite induction to define a recursive program:

$$\frac{\mathsf{fix}\, f \in A^{0} \qquad f \in A^{\alpha} \to A^{\alpha+1} \qquad (\forall \alpha < \lambda.\, \mathsf{fix}\, f \in A^{\alpha}) \to \mathsf{fix}\, f \in A^{\lambda}}{\mathsf{fix}\, f \in A^{\beta}}$$

- For $\mathsf{sum} \in \mathsf{List}^{\omega}\,\mathsf{Int} \to \mathsf{Int}$, instantiate $A^{\alpha} = \mathsf{List}^{\alpha}\,\mathsf{Int} \to \mathsf{Int}$ and $\beta = \omega$.

Handling base and limit case

Slide 13

- Recursion principle:

$$\frac{\mathsf{fix}\, f \in A^0 \qquad f \in A^\alpha \to A^{\alpha+1} \qquad (\mathsf{fix}\, f \in \bigcap_{\alpha<\lambda} A^\alpha) \to \mathsf{fix}\, f \in A^\lambda}{\mathsf{fix}\, f \in A^\beta}$$

- Restrict admissible types A^α such that
  - $\mathsf{fix}\, f \in A^0$ is trivial, e.g., $A^\alpha = \mu^\alpha F \to C$,
  - $(\bigcap_{\alpha<\lambda} A^\alpha) \subseteq A^\lambda$.
- Specialized rule

$$\frac{\forall \alpha.\, f \in A^\alpha \to A^{\alpha+1}}{\mathsf{fix}\, f \in A^\beta}\, A^\alpha \text{ admissible}$$

From semantics to syntax

Slide 14

- Recapitulation of semantic types we used:

$$\frac{\forall \alpha.\, f \in A^\alpha \to A^{\alpha+1}}{\mathsf{fix}\, f \in A^\beta}\, A^\alpha \text{ admissible}$$

$$\begin{array}{lcl}
\mathsf{sum} & \in & \mathsf{List}^\omega\, \mathsf{Int} \to \mathsf{Int} \\
\mathsf{nil} & \in & \mathsf{List}^{\alpha+1}\, \mathsf{Int} \\
(::) & \in & \mathsf{Int} \to \mathsf{List}^\alpha\, \mathsf{Int} \to \mathsf{List}^{\alpha+1}\, \mathsf{Int}
\end{array}$$

- We only talk about ordinal variables (α, β), successor, and closure ordinal (in this case, ω)!
- We can turn these semantic rules into syntax *without an ordinal notation system* (e.g., Cantor normal form).

$\mathsf{F}_{\widehat{\omega}}$: a type system for termination

- A language with three levels:
  - *Terms* (programs) which have types.
  - *Type constructors*: a language to construct types.
  - *Kinds*, the “types” of type constructor.

Slide 15

- Kinds:

$$\begin{array}{rcll} \kappa & ::= & * & \text{types } A, B \\ & \mid & \mathsf{ord} & \text{ordinals } a, b \\ & \mid & \kappa_1 \xrightarrow{p} \kappa_2 & p\text{-variant type constructors } F, G \end{array}$$

- Constructors can be covariant ($p = +$), contravariant ($p = -$), and non-varaint ($p = \circ$, “don’t know”).

$\mathsf{F}_{\widehat{\omega}}$: constructors

- Types and type constructors:

$$\begin{array}{rcl} F, G & ::= & X \mid \lambda X.F \mid F\,G \mid \to \mid \forall_\kappa \mid \mu^a \\ a, b & ::= & i \mid a + 1 \mid \infty \end{array}$$

Slide 16

- Defined types:

$$\begin{array}{rcl} \forall X\!:\!\kappa.A & = & \forall_\kappa(\lambda X.A) \\ 1 & = & \forall X\!:\!*.\, X \to X \\ A + B & = & \forall X\!:\!*.\, (A \to X) \to (B \to X) \to X \\ A \times B & = & \forall X\!:\!*.\, (A \to B \to X) \to X \end{array}$$

$\mathsf{F}_{\widehat{\omega}}$: sized inductive types

Slide 17

- Sized polymorphic lists and tree ordinals:

$$\begin{aligned}\mathsf{List} &: \mathsf{ord} \overset{+}{\to} * \overset{+}{\to} * \\ \mathsf{List} &:= \lambda a.\lambda A.\, \mu^{a}\, (\lambda X.\, \mathbf{1} + A \times X)\end{aligned}$$

$$\begin{aligned}\mathsf{Ord} &: \mathsf{ord} \overset{+}{\to} * \\ \mathsf{Ord} &:= \lambda a.\, \mu^{a}\, (\lambda X.\, \mathbf{1} + X + (\mathsf{Nat}^{\infty} \to X))\end{aligned}$$

- Sized de Bruijn terms:

$$\begin{aligned}\mathsf{Lam} &: \mathsf{ord} \overset{+}{\to} * \overset{+}{\to} * \\ \mathsf{Lam} &:= \lambda a.\ \mu^{a}\, (\lambda X.\lambda A.\, A + (X\ A \times X\ A) + X\ (\mathbf{1} + A))\end{aligned}$$

- Lam is an example of a non-regular type / heterogeneous type / nested type / inductive constructor.

$\mathsf{F}_{\widehat{\omega}}$: judgements on constructors

Slide 18

- Judgements

$F : \kappa$	constructor F has kind κ
$F = G : \kappa$	constructors F, G are $\beta\eta$-equal
$F \leq G : \kappa$	F is a higher-order subtype of G

- Kinding of type constructor constants

$\to\ : * \overset{-}{\to} * \overset{+}{\to} *$	function space
$\forall_{\kappa}\ : (\kappa \overset{\circ}{\to} *) \overset{+}{\to} *$	quantification
$\mu_{\kappa}\ : \mathsf{ord} \overset{+}{\to} (\kappa \overset{+}{\to} \kappa) \overset{+}{\to} \kappa$	inductive constructors

$\mathsf{F}_{\widehat{\omega}}$: higher-order subtyping

Slide 19

- Subtyping for ordinal expressions:

$$\frac{a \leq b : \mathsf{ord}}{a+1 \leq b+1 : \mathsf{ord}} \qquad \frac{a \leq b : \mathsf{ord}}{a \leq b+1 : \mathsf{ord}} \qquad \frac{a : \mathsf{ord}}{a \leq \infty : \mathsf{ord}}$$

- Point-wise ordering of type constructors

$$\frac{F \leq F' : \kappa \xrightarrow{p} \kappa' \qquad G : \kappa}{F\,G \leq F'\,G : \kappa'}$$

- Co/contra-variant subtyping

$$\frac{F : \kappa \xrightarrow{+} \kappa' \qquad G \leq G' : \kappa}{F\,G \leq F\,G' : \kappa'} \qquad \frac{F : \kappa \xrightarrow{-} \kappa' \qquad G \leq G' : \kappa}{F\,G' \leq F\,G : \kappa'}$$

- Subtyping for inductive constructors:

$$\frac{a \leq b : \mathsf{ord} \qquad F : \kappa \xrightarrow{+} \kappa}{\mu^a F \leq \mu^b F : \kappa}$$

$\mathsf{F}_{\widehat{\omega}}$: terms

Slide 20

- Terms:

$$r, s, t ::= x \mid \lambda x.t \mid r\,s \mid \mathsf{fix}$$

- Typing judgment $t : A$.
- Inductive type folding and unfolding:

$$\frac{t : F(\mu^a F)}{t : \mu^{a+1} F} \qquad \frac{t : \mu^{a+1} F}{t : F(\mu^a F)}$$

- Recursion rule:

$$\frac{a : \mathsf{ord}}{\mathsf{fix} : (\forall i.\, A^i \to A^{i+1}) \to A^a}\; A^i \text{ admissible}$$

Examples

Slide 21

- Typing of sum:

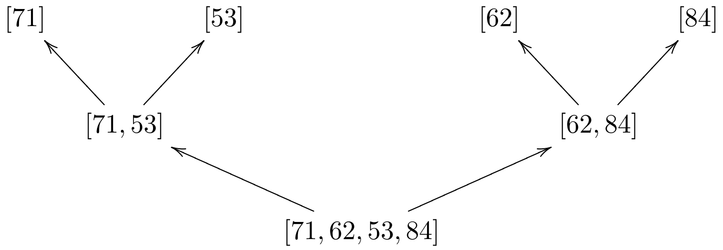
$$\begin{aligned} \mathsf{sum} \quad &: \quad \mathsf{List}^{\infty}\,\mathsf{Int} \to \mathsf{Int} \\ \mathsf{sum} \quad &= \quad \mathsf{fix}\ (\lambda sum\,{:}\,\mathsf{List}^{i}\,\mathsf{Int} \to \mathsf{Int}.\ \lambda l\,{:}\,\mathsf{List}^{i+1}. \\ &\qquad\quad \mathsf{match}\ l\ \mathsf{with} \\ &\qquad\qquad \mathsf{nil} \qquad\qquad\qquad \mapsto \quad 0 \\ &\qquad\qquad (x :: (xs : \mathsf{List}^{i})) \quad \mapsto \quad x + sum\ xs) \end{aligned}$$

- Syntax with implicit fix and size annotations:

$$\begin{aligned} \mathsf{sum} \quad ([\,])^{i+1} \qquad &= \quad 0 \\ \mathsf{sum} \quad (x :: xs^{i})^{i+1} \quad &= \quad x + \mathsf{sum}\ xs^{i} \end{aligned}$$

Merge sort: splitting phase

Slide 22



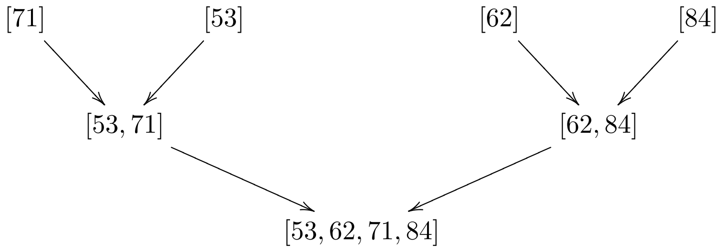
$[71]$ $[53]$ $[62]$ $[84]$

$[71, 53]$ $[62, 84]$

$[71, 62, 53, 84]$

Merge sort: merging phase

Slide 23



Splitting: definition

$\mathsf{split} :\quad \forall A{:}{*}.\ \mathsf{List}\ A \to \mathsf{List}\ A \times \mathsf{List}\ A$

$\mathsf{split}\ [\,] \quad = ([\,]\ ,\ [\,]\ )$

$\mathsf{split}\ (y :: l\ ) \quad = \mathsf{let}\ (xs\ ,\ ys\ ) = \mathsf{split}\, l\ \ \mathsf{in}$

$\quad ((y :: ys)\ ,\ xs\ )$

Slide 24

Splitting: termination

$\mathsf{split} : \forall i\!:\!\mathsf{ord}.\, \forall A\!:\!*.\ \mathsf{List}^{i} A \to \mathsf{List}\ A \times \mathsf{List}\ A$

$$\begin{aligned}
\mathsf{split}\ [\,] &= ([\,]\quad, [\,]\quad) \\
\mathsf{split}\ (y :: l^{i})^{i+1} &= \mathsf{let}\ (xs\ , ys\ ) = \mathsf{split}\, l^{i}\ \mathsf{in} \\
&\ ((y :: ys)\quad, xs\qquad)
\end{aligned}$$

Slide 25

- To compute split at stage $i+1$, split is only used at stage i.
- Hence, split is terminating.

Splitting: bounded output

$\mathsf{split} : \forall i\!:\!\mathsf{ord}.\, \forall A\!:\!*.\ \mathsf{List}^{i} A \to \mathsf{List}^{i} A \times \mathsf{List}^{i} A$

$$\begin{aligned}
\mathsf{split}\ [\,]^{i+1} &= ([\,]^{i+1}, [\,]^{i+1}) \\
\mathsf{split}\ (y :: l^{i})^{i+1} &= \mathsf{let}\ (xs^{i}, ys^{i}) = \mathsf{split}\, l^{i}\ \mathsf{in} \\
&\ ((y :: ys)^{i+1}, xs^{i \le i+1})
\end{aligned}$$

Slide 26

- We additionally can infer that split is non-size increasing.
- Using split, we can define merge sort...

Merging: definition

Slide 27

merge produces a sorted list from two sorted input lists

$\mathsf{merge} : \mathsf{List}\ \mathsf{Int} \to \mathsf{List}\ \mathsf{Int} \to \mathsf{List}\ \mathsf{Int}$

$$
\begin{aligned}
&\mathsf{merge}\ [\,]\ && l = l \\
&\mathsf{merge}\ (x :: xs)\ && l = \mathsf{merge}'\ l
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{where}\ \ \mathsf{merge}' : \mathsf{List}\ A \to \mathsf{List}\ A \\
&\qquad \mathsf{merge}'\ [\,] = x :: xs \\
&\qquad \mathsf{merge}'\ (y :: ys) = \mathsf{if}\ x \le y\ \mathsf{then} \\
&\qquad\qquad x :: \mathsf{merge}\ xs\ (y :: ys) \\
&\qquad\qquad \mathsf{else}\ \ y :: \mathsf{merge}'\ ys
\end{aligned}
$$

Merging: termination

Slide 28

merge terminates by lexicographic ordering

$\mathsf{merge} : \forall i : \mathsf{ord}.\ \mathsf{List}^{i}\ \mathsf{Int} \to \mathsf{List}^{\infty}\ \mathsf{Int} \to \mathsf{List}^{\infty}\ \mathsf{Int}$

$$
\begin{aligned}
&\mathsf{merge}\ [\,]\ && l = l \\
&\mathsf{merge}\ (x :: xs^{i})^{i+1}\ && l = \mathsf{merge}'\ l
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{where}\ \ \mathsf{merge}' : \mathsf{List}\ A \to \mathsf{List}\ A \\
&\qquad \mathsf{merge}'\ [\,] = x :: xs \\
&\qquad \mathsf{merge}'\ (y :: ys) = \mathsf{if}\ x \le y\ \mathsf{then} \\
&\qquad\qquad x :: \mathsf{merge}\ xs^{i}\ (y :: ys) \\
&\qquad\qquad \mathsf{else}\ \ y :: \mathsf{merge}'\ ys
\end{aligned}
$$

Merging: termination

Slide 29

merge terminates by lexicographic ordering

$$\mathsf{merge} : \forall i{:}\mathsf{ord}.\ \mathsf{List}^{i}\ \mathsf{Int} \to \mathsf{List}^{\infty}\ \mathsf{Int} \to \mathsf{List}^{\infty}\ \mathsf{Int}$$

$$\begin{array}{lcl}
\mathsf{merge}\ [\,] & l & = & l \\
\mathsf{merge}\ (x :: xs^{i})^{i+1} & l & = & \mathsf{merge}'\ l
\end{array}$$

$$\mathsf{where}\ \ \mathsf{merge}' : \forall j{:}\mathsf{ord}.\ \mathsf{List}^{j}\ A \to \mathsf{List}^{\infty}\ A$$

$$\begin{array}{lcl}
\mathsf{merge}'\ [\,] & = & x :: xs \\
\mathsf{merge}'\ (y :: ys^{j})^{j+1} & = & \mathsf{if}\ \ x \le y\ \ \mathsf{then} \\
 & & \qquad x :: \mathsf{merge}\ xs^{i}\ (y :: ys)^{j+1 \le \infty} \\
 & & \mathsf{else}\ \ y :: \mathsf{merge}'\ ys^{j}
\end{array}$$

Merge sort: definition

Slide 30

$$\mathsf{msort}\ \ : \mathsf{List}\ \ \ \mathsf{Int} \to \mathsf{List}\ \ \ \mathsf{Int}$$

$$\begin{array}{ll}
\mathsf{msort}\ \ [\,] & = [\,] \\
\mathsf{msort}\ \ (x :: l) & = \mathsf{msort}'\ \ x\ l
\end{array}$$

$$\mathsf{msort}' : \qquad \mathsf{Int} \to \mathsf{List}\ \ \mathsf{Int} \to \mathsf{List}\ \ \ \mathsf{Int}$$

$$\begin{array}{ll}
\mathsf{msort}'\ x\ [\,] & = [x] \\
\mathsf{msort}'\ x\ (y :: l\ \ ) & = \mathsf{let}\ (xs\ \ , ys\ \ ) = \mathsf{split}\ \ l\ \ \mathsf{in} \\
 & \quad\ \mathsf{merge}\ (\mathsf{msort}'\ \ x\ xs\ \ ) \\
 & \qquad\qquad (\mathsf{msort}'\ \ y\ ys\ \ )
\end{array}$$

Merge sort: termination

$$\begin{array}{lll}
\mathsf{msort} & : \mathsf{List}^{\infty}\,\mathsf{Int} \to \mathsf{List}^{\infty}\,\mathsf{Int} & \\
\mathsf{msort} & [\,] & = [\,] \\
\mathsf{msort} & (x :: l) & = \mathsf{msort}'\ x\ l
\end{array}$$

$$\begin{array}{ll}
\mathsf{msort}' : \forall i\!:\!\mathsf{ord}.\ \mathsf{Int} \to \mathsf{List}^{i}\,\mathsf{Int} \to \mathsf{List}^{\infty}\,\mathsf{Int} & \\
\mathsf{msort}'\ x\ [\,]^{i+1} & = [x] \\
\mathsf{msort}'\ x\ (y :: l^{i}) & = \mathsf{let}\ (xs^{i}, ys^{i}) = \mathsf{split}\ \ l^{i}\ \mathsf{in} \\
 & \quad\ \mathsf{merge}\ (\mathsf{msort}'\ \ x\ xs^{i}) \\
 & \qquad\qquad\ (\mathsf{msort}'\ \ y\ ys^{i})
\end{array}$$

Slide 31

Leaving Hindley-Milner typing

- So far, termination could have been checked without types
- The size relation of split could have been recorded separately
- But now let us parametrize merge sort over a split function...

Slide 32

Merge sort: abstract split

$$\begin{aligned}
&\mathsf{msort}'\ \mathit{split}\ x\ [\,] &&= [x]\\
&\mathsf{msort}'\ \mathit{split}\ x\ (y :: l\ ) &&= \mathsf{let}\ (xs\ , ys\ ) = \mathit{split}\ l\ \ \mathsf{in}\\
& && \quad \mathsf{merge}\ (\mathsf{msort}'\ \ x\ xs\ )\\
& && \quad\qquad (\mathsf{msort}'\ \ y\ ys\ )
\end{aligned}$$

Slide 33

- The variable *split* can only be instantiated with *non size increasing* functions
- This is naturally expressed with a rank-2 *size polymorphic* type

Merge sort: abstract split (II)

$$\begin{aligned}
&\mathsf{msort}'\ : (\forall i\!:\!\mathsf{ord}.\,\forall A\!:\!*.\ \mathsf{List}^i A \to \mathsf{List}^i A \times \mathsf{List}^i A) \to\\
&\qquad \forall i\!:\!\mathsf{ord}.\ \mathsf{Int} \to \mathsf{List}^i\,\mathsf{Int} \to \mathsf{List}^\infty\,\mathsf{Int}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{msort}'\ \mathit{split}\ x\ [\,]^{i+1} &&= [x]\\
&\mathsf{msort}'\ \mathit{split}\ x\ (y :: l^i) &&= \mathsf{let}\ (xs^i, ys^i) = \mathit{split}\ l^i\ \mathsf{in}\\
& && \quad \mathsf{merge}\ (\mathsf{msort}'\ \ x\ xs^i)\\
& && \quad\qquad (\mathsf{msort}'\ \ y\ ys^i)
\end{aligned}$$

Slide 34

- We drop the restriction of Hughes, Pareto, and Sabry and Barthe et. al. that sizes should be inferable

Tree ordinals

- Tree ordinals

$$\mathsf{Ord}^{a} = \mu^{a}\,(\lambda X.\,\mathbf{1} + X + (\mathsf{Nat}^{\infty} \to X))$$

- Definable constructors

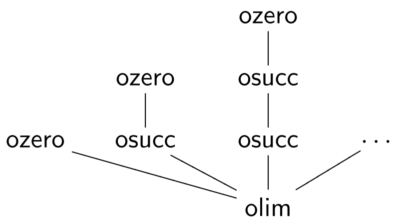
Slide 35

$$\begin{aligned}
\mathsf{ozero} &\;:\; \forall i\!:\!\mathsf{ord}.\ \mathsf{Ord}^{i} \\
\mathsf{osucc} &\;:\; \forall i\!:\!\mathsf{ord}.\ \mathsf{Ord}^{i} \to \mathsf{Ord}^{i+1} \\
\mathsf{olim} &\;:\; \forall i\!:\!\mathsf{ord}.\ (\mathsf{Nat} \to \mathsf{Ord}^{i}) \to \mathsf{Ord}^{i+1}
\end{aligned}$$

An element of infinite height

$$\mathsf{Ord}^{1} \qquad \mathsf{Ord}^{2} \qquad \mathsf{Ord}^{3} \qquad \cdots \qquad \subseteq \mathsf{Ord}^{\omega}$$

Slide 36



$\mathsf{Ord}^{\omega+1}$

Example: addition for tree ordinals

- Constructors:

$$\begin{array}{lcl} \mathsf{ozero} & : & \forall i\!:\!\mathsf{ord}.\ \mathsf{Ord}^{i} \\ \mathsf{osucc} & : & \forall i\!:\!\mathsf{ord}.\ \mathsf{Ord}^{i} \to \mathsf{Ord}^{i+1} \\ \mathsf{olim} & : & \forall i\!:\!\mathsf{ord}.\ (\mathsf{Nat} \to \mathsf{Ord}^{i}) \to \mathsf{Ord}^{i+1} \end{array}$$

Slide 37

- Addition:

$$\mathsf{add} : \mathsf{Ord}^{\infty} \to \forall i\!:\!\mathsf{ord}.\ \mathsf{Ord}^{i} \to \mathsf{Ord}^{\infty}$$

$$\begin{array}{lcl} \mathsf{add}\ x\ \ \mathsf{ozero} & = & x \\ \mathsf{add}\ x\ (\mathsf{osucc}\ y^{i})^{i+1} & = & \mathsf{osucc}\ (\mathsf{add}\ x\ y^{i}) \\ \mathsf{add}\ x\ (\mathsf{olim}\ f^{\cdot \to i})^{i+1} & = & \mathsf{olim}\ (\lambda n.\ \mathsf{add}\ x\ (f\ n)^{i}) \end{array}$$

Productivity

- Productivity is *dual* to termination
- A productive process should continuously turn input into output
- Examples: editor, operating system, stream
- Important in embedded and functional reactive programming

Slide 38

Infinite structures

- On infinite objects like streams, we are interested in the *definedness* rather than the size.
- $s : \mathsf{Stream}^{a} A$ means s is defined upto depth a.
- Objects which are defined upto depth ∞ are called *productive*.
- Basic stream operations:

Slide 39

$$\begin{array}{lcl} (::) & : & A \to \forall i\!:\!\mathsf{ord}.\, \mathsf{Stream}^{i} A \to \mathsf{Stream}^{i+1} A \\ \mathsf{hd} & : & \forall i\!:\!\mathsf{ord}.\, \mathsf{Stream}^{i+1} A \to A \\ \mathsf{tl} & : & \forall i\!:\!\mathsf{ord}.\, \mathsf{Stream}^{i+1} A \to \mathsf{Stream}^{i} A \end{array}$$

- Subtyping: $\mathsf{Stream}^{\infty} A \leq \ldots \mathsf{Stream}^{i+1} A \leq \mathsf{Stream}^{i} A$

$\mathsf{F}_{\widehat{\omega}}$: extension by coinduction

- Add type constructor $\nu_\kappa : \mathsf{ord} \xrightarrow{-} (\kappa \xrightarrow{+} \kappa) \xrightarrow{+} \kappa$.
- Example $\mathsf{Stream}^{a} = \lambda A.\, \nu^{a}\, (\lambda X.\, A \times X)$.
- Recursion rule also usable for corecursion!

Slide 40

$$\frac{a : \mathsf{ord}}{\mathsf{fix} : (\forall i.\, A^{i} \to A^{i+1}) \to A^{a}} \; A^{i} \text{ admissible}$$

- Example: defining infinite sequence $\mathsf{upfrom}\, 0 = [0, 1, 2, \ldots]$

$$\begin{array}{lcl} \mathsf{upfrom} & : & \mathsf{Int} \to \mathsf{Stream}^{\infty}\, \mathsf{Int} \\ \mathsf{upfrom} & := & \mathsf{fix}\ (\lambda upfrom. \lambda n.\ \underbrace{(n,\ \overbrace{upfrom(n+1))}^{\mathsf{Stream}^{i}\, \mathsf{Int}}}_{\mathsf{Stream}^{i+1}\, \mathsf{Int}}) \end{array}$$

Related works on type-based termination

Slide 41

- Hughes, Pareto, Sabry (1996)
  Proving the correctness of reactive system using sized types
- Amadio and Coupet-Grimal (1998)
  Analysis of a guard condition in type theory
- Barthe, Frade, Giménez, Pinto, Uustalu (2004)
  Type-based termination of recursive definitions
- Buchholz (2003), *Recursion on non-wellfounded trees*

Own works on termination

Slide 42

- *Specification and verification of a formal system for structural recursion* (TYPES'99)
- *A predicative analysis of structural recursion*
  (with Altenkirch, JFP, 2002)
- *Termination and guardedness checking with continuous types*
  (TLCA'03)
- *Termination checking with types* (ITA, 2004)
- *A polymorphic λ-calculus with sized higher-order types*
  (Ph.D. thesis, almost ready for submission)

Works on iteration and recursion

Slide 43

- *A predicative strong normalization proof for a λ-calculus with interleaving inductive types* (Abel, Altenkirch, TYPES'99)
- *Co(iteration) for higher-order nested datatypes* (Abel, Matthes, TYPES'02)
- *Generalized iteration and coiteration for higher-order nested datatypes* (Abel, Matthes, Uustalu, FoSSaCS'03)
- *Fixed points of type constructors and primitive recursion* (Abel, Matthes, CSL'04)
- *Generalized iteration and coiteration for higher-order and nested datatypes* (Abel, Matthes, Uustalu, TCS, 2005)

Works on dependent type theory

Slide 44

- Meta-theoretical:
  Untyped algorithmic equality for Martin-Löf's Logical Framework with Surjective Pairs (Abel, Coquand, TLCA'05)
- Case studies:
  - *A third-order representation of the $\lambda\mu$-calculus* (MERLIN'01)
  - *Weak normalization for the simply-typed λ-calculus in Twelf* (LFM'04)
  - *Verifying Haskell programs in constructive type theory* (Abel, Benke, Bove, Hughes, Norell, Haskell'05)

Short-term research goals

- Adopt type-based termination to *dependent types*
- Investigate type-based termination for higher-order abstract syntax
  - Challenge: negative inductive types

Slide 45

$$\begin{aligned} \mathsf{Tm} &= (\mathsf{Tm} \times \mathsf{Tm}) + (\mathsf{Tm} \to \mathsf{Tm}) \\ \mathsf{app} &: \mathsf{Tm}^{i} \to \mathsf{Tm}^{i} \to \mathsf{Tm}^{i+1} \\ \mathsf{abs} &: (\mathsf{Tm}^{?} \to \mathsf{Tm}^{i}) \to \mathsf{Tm}^{i+1} \end{aligned}$$

  - Type-based termination not directly applicable.
  - Can it be adopted to negative types?

Longer-term research goals

- Can type-based termination be adopted to languages with references?
- Integrate with heap type system
- Combinable with Hofmann/Jost system?

Slide 46

Works on theorem proving

- *Human-readable machine-verifiable proofs for teaching constructive logic* (Abel, Chang, Pfenning, PTP'01)
- *Connecting a logical framework to a first-order prover* (Abel, Coquand, Norell, FroCoS'05)

Slide 47

Long term research: proof documents

- Future of theorem proving:
  - User writes legible, formal proof document
  - Trivial steps are filled in by machine
- How should the proof language look like?
- What can be considered a trivial step?
- How to integrate automation?

This is a community effort (TYPES).

Slide 48

Acknowledgements

Slide 49

- Technical discussions on my thesis:

Klaus Aehlig Thorsten Altenkirch Martin Hofmann
John Hughes Ralph Matthes Tarmo Uustalu

- Stipends

GKLI CoVer

- Colleagues at Munich and Chalmers for support