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 tutorial proof checker (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. $F\omega$—a type system for termination
4. Examples: the type system at work
5. Productivity via coinduction
6. Achieved results and future work
Termination

- 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

Slide 6

Termination for theorem proving

- Inductive theorem provers: e.g., Agda, Coq, LEGO, Twelf.
- Some proofs are tree-shaped derivations, e.g., proof that \([a, 0] = [b, 0]\).

\[
\begin{align*}
0 &= 0 \\
a &= b \\
(0 :: []) &= (0 :: []) \\
a :: (0 :: []) &= b :: (0 :: [])
\end{align*}
\]

- Some proofs are recursive programs, manipulating derivations.
- E.g., proof of \((l_1 = l_2) \rightarrow (l_2 = l_3) \rightarrow (l_1 = l_3)\).
- Only terminating programs denote valid proofs.
- E.g., program \(\text{trans} d_1 d_2 = \text{trans} d_1 d_2\) has to be rejected.
Termination of Functions Over Inductive Types

- 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

- 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
  \[
  \text{List Int} = \{ [] \} \cup \{ x :: xs \mid x \in \text{Int} \text{ and } xs \in \text{List Int} \}
  \]
- Abstracting away the names
  \[
  \text{List Int} = 1 + \text{Int} \times \text{List Int}
  \]
- Definable as least fixed-point $\mu F$ of some type operator $F$
  \[
  \text{List Int} := \mu (\lambda X. 1 + \text{Int} \times X)
  \]
Iterating to the least fixed point

- The least fixed point is reachable from below by ordinal iteration:
  \[
  \begin{align*}
  \mu^0 F &= \emptyset \\
  \mu^{\alpha+1} F &= F(\mu^\alpha F) \\
  \mu^\lambda F &= \bigcup_{\alpha<\lambda} \mu^\alpha F
  \end{align*}
  \]

  \begin{align*}
  \mu^2 F &= \mu^{\alpha+1} F \\
  \mu^3 F &= \mu^{\omega+1} F \\
  \mu^4 F &= \mu^{\omega+1} F
  \end{align*}

  \mu^\omega F

Slide 9

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

\[
\begin{align*}
[] & \in \text{List}^{\alpha+1} \text{Int} \\
(\cdot:) & \in \text{Int} \rightarrow \text{List}^\alpha \text{Int} \rightarrow \text{List}^{\alpha+1} \text{Int}
\end{align*}
\]
Recursive functions over inductive types

- E.g., we want to define list summation \( \text{sum} \in \text{List}^{\omega} \text{Int} \to \text{Int} \).
- Recursive program:
  
  \[
  \text{sum} \ [\ ] = 0 \\
  \text{sum} \ (x :: xs) = x + \text{sum} \ xs
  \]

Slide 11

- Via fixed-point combinator \( \text{fix} \ f = f (\text{fix} \ f) \).

  \[
  \text{sum} = \text{fix} \ (\lambda \text{sum}. \lambda l. \text{match} \ l \text{ with} \\
  \text{nil} \mapsto 0 \\
  (x :: xs) \mapsto x + \text{sum} \ xs)
  \]

- How to prove that \( \text{sum} \) is well defined, i.e., terminating?

A recursion principle from transfinite induction

- Rule for transfinite induction:
  
  \[
  \frac{P(0) \quad P(\alpha) \to P(\alpha + 1)}{P(\beta) \quad (\forall \alpha < \lambda. P(\alpha)) \to P(\lambda)}
  \]

Slide 12

- Use transfinite induction to define a recursive program:
  
  \[
  \frac{\text{fix} \ f \in A^0 \quad f \in A^\alpha \to A^{\alpha+1} \quad (\forall \alpha < \lambda. \text{fix} \ f \in A^\alpha) \to \text{fix} \ f \in A^\lambda}{\text{fix} \ f \in A^\beta}
  \]

- For \( \text{sum} \in \text{List}^{\omega} \text{Int} \to \text{Int} \), instantiate \( A^\alpha = \text{List}^{\alpha} \text{Int} \to \text{Int} \) and \( \beta = \omega \).
Handling base and limit case

- Recursion principle:

\[
\begin{align*}
\text{fix } f \in A^0 & \quad f \in A^\alpha \rightarrow A^{\alpha+1} \quad (\text{fix } f \in \bigcap_{\alpha<\lambda} A^\alpha) \rightarrow \text{fix } f \in A^\lambda \\
\implies & \quad \text{fix } f \in A^\beta
\end{align*}
\]

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

- Specialized rule

\[
\begin{align*}
\forall \alpha. f \in A^\alpha \rightarrow A^{\alpha+1} \\
\implies & \quad \text{fix } f \in A^\beta \quad A^\alpha \text{ admissible}
\end{align*}
\]

From semantics to syntax

- Recapitulation of semantic types we used:

\[
\begin{align*}
\forall \alpha. f \in A^\alpha \rightarrow A^{\alpha+1} \\
\implies & \quad \text{fix } f \in A^\beta \quad A^\alpha \text{ admissible}
\end{align*}
\]

\[
\text{sum} \in \text{List}^\omega \text{Int} \rightarrow \text{Int}
\]

\[
\text{nil} \in \text{List}^{\alpha+1} \text{Int}
\]

\[
(::) \in \text{Int} \rightarrow \text{List}^\alpha \text{Int} \rightarrow \text{List}^{\alpha+1} \text{Int}
\]

- We only talk about ordinal variables \((\alpha, \beta)\), successor, and closure ordinal (in this case, \(\omega\))!

- We can turn these semantic rules into syntax without an ordinal notation system (e.g., Cantor normal form).
\( F_\omega \): 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:
  \[
  \kappa ::= * \quad \text{types } A, B \\
  \text{ord} \quad \text{ordinals } a, b \\
  \kappa_1 \overset{p}{\rightarrow} \kappa_2 \quad p\text{-variant type constructors } F, G
  \]

- Constructors can be covariant \((p = +)\), contravariant \((p = -)\), and non-variant \((p = 0, \text{“don’t know”})\).

\( F_\omega \): constructors

- Types and type constructors:
  \[
  F, G ::= X \mid \lambda X.F \mid F G \mid \rightarrow \mid \forall \kappa \mid \mu a \\
  a, b ::= i \mid a + 1 \mid \infty
  \]

**Slide 16**

- Defined types:
  \[
  \forall X: \kappa. A = \forall \kappa.(\lambda X.A) \\
  1 = \forall X: *. X \rightarrow X \\
  A + B = \forall X: *. (A \rightarrow X) \rightarrow (B \rightarrow X) \rightarrow X \\
  A \times B = \forall X: *. (A \rightarrow B \rightarrow X) \rightarrow X
  \]
\( \mathcal{F}_\omega \): sized inductive types

- Sized polymorphic lists and tree ordinals:
  \[
  \text{List} : \ ord \downarrow * \downarrow * \\
  \text{List} := \lambda a. \lambda A. \mu a (\lambda X. 1 + A \times X)
  \]
  \[
  \text{Ord} : \ ord \downarrow * \\
  \text{Ord} := \lambda a. \mu a (\lambda X. 1 + X + (\text{Nat}^\infty \rightarrow X))
  \]
- Sized de Bruijn terms:
  \[
  \text{Lam} : \ ord \downarrow * \downarrow * \\
  \text{Lam} := \lambda a. \mu a (\lambda X. \lambda A. A + (X A \times X A) + X (1 + A))
  \]
- Lam is an example of a non-regular type / heterogeneous type / nested type / inductive constructor.

\( \mathcal{F}_\omega \): judgements on constructors

- Judgements
  \[
  F : \kappa \quad \text{constructor } F \text{ has kind } \kappa \\
  F = G : \kappa \quad \text{constructors } F, G \text{ are } \beta\eta\text{-equal} \\
  F \leq G : \kappa \quad F \text{ is a higher-order subtype of } G
  \]
- Kinding of type constructor constants
  \[
  \rightarrow : * \downarrow * \downarrow * \quad \text{function space} \\
  \forall_{\kappa} : (\kappa \rightarrow *) \downarrow * \quad \text{quantification} \\
  \mu_{\kappa} : \text{ord} \downarrow (\kappa \downarrow \kappa) \downarrow \kappa \quad \text{inductive constructors}
  \]
\(F_\omega\): higher-order subtyping

- Subtyping for ordinal expressions:
  \[
  \begin{align*}
  a \leq b : \text{ord} & \quad \frac{a + 1 \leq b + 1 : \text{ord}}{a \leq b + 1 : \text{ord}} \\
  a : \text{ord} & \quad a \leq \infty : \text{ord}
  \end{align*}
  \]

- Point-wise ordering of type constructors
  \[
  \frac{F \leq F' : \kappa \quad \rho, \kappa' \quad G : \kappa}{FG \leq F'G : \kappa'}
  \]

- Co/contra-variant subtyping
  \[
  \begin{align*}
  F : \kappa \leadsto \kappa' & \quad G \leq G' : \kappa \quad \frac{F : \kappa \leadsto \kappa'}{FG \leq FG' : \kappa'} \\
  F : \kappa \twoheadleftarrow \kappa' & \quad \frac{F : \kappa \twoheadleftarrow \kappa'}{FG \leq FG' : \kappa'}
  \end{align*}
  \]

- Subtyping for inductive constructors:
  \[
  \begin{align*}
  a \leq b : \text{ord} & \quad \frac{\mu^a F \leq \mu^b F : \kappa}{F : \kappa \leadsto \kappa}
  \end{align*}
  \]

\(F_\omega\): terms

- Terms:
  \[
  r, s, t ::= x \mid \lambda x.t \mid r \cdot s \mid \text{fix}
  \]

- Typing judgment \(t : A\).

- Inductive type folding and unfolding:
  \[
  \begin{align*}
  t : F(\mu^a F) & \quad t : F(\mu^{a+1} F) \\
  t : \mu^{a+1} F & \quad \frac{t : F(\mu^a F)}{t : F(\mu^{a+1} F)}
  \end{align*}
  \]

- Recursion rule:
  \[
  \frac{a : \text{ord}}{\text{fix} : (\forall i. A^i \rightarrow A^{i+1}) \rightarrow A^a \text{ admissible}}
  \]
Examples

• Typing of \texttt{sum}:

\[
\begin{align*}
\texttt{sum} & : \ \text{List}^\infty \text{Int} \rightarrow \text{Int} \\
\texttt{sum} & = \ \text{fix} \ (\lambda \texttt{sum}: \text{List}^i \text{Int} \rightarrow \text{Int} . \ \lambda l : \text{List}^{i+1} . \\
& \quad \ \text{match } l \ \text{with} \\
& \quad \ \text{nil} \ \mapsto \ 0 \\
& \quad \ (x :: (xs : \text{List}^i)) \ \mapsto \ x + \texttt{sum} \ xs
\end{align*}
\]

Slide 21

• Syntax with implicit \texttt{fix} and size annotations:

\[
\begin{align*}
\texttt{sum} \ \texttt{([], [])}^{i+1} & = \ 0 \\
\texttt{sum} \ (x :: xs^i)^{i+1} & = \ x + \texttt{sum} \ xs^i
\end{align*}
\]

Merge sort: splitting phase

Slide 22
Merge sort: merging phase

Slide 23

Splitting: definition

split : ∀A : *, List A → List A × List A
split [] = ([], [])
split (y :: l) = let (xs, ys) = split l in
((y :: ys), xs)

Slide 24
Splitting: termination

\[ \text{split} : \forall i : \text{ord} . \forall A : *, \text{List}^i A \rightarrow \text{List}^i A \times \text{List}^i A \]

\[ \text{split} \; \square = (\square, \square) \]

\[ \text{split} \; (y :: l)^{i+1} = \text{let} \; (xs, ys) = \text{split}^i \; l \; \text{in} \]

\[ ((y :: ys)^{i+1}, xs^{\leq i+1}) \]

Slide 25

- To compute \text{split} at stage \( i + 1 \), \text{split} is only used at stage \( i \).
- Hence, \text{split} is terminating.

Splitting: bounded output

\[ \text{split} : \forall i : \text{ord} . \forall A : *, \text{List}^i A \rightarrow \text{List}^i A \times \text{List}^i A \]

\[ \text{split} \; \square^{i+1} = (\square^{i+1}, \square^{i+1}) \]

\[ \text{split} \; (y :: l)^{i+1} = \text{let} \; (xs^i, ys^i) = \text{split}^i \; l \; \text{in} \]

\[ ((y :: ys)^{i+1}, xs^i^{\leq i+1}) \]

Slide 26

- We additionally can infer that \text{split} is non-size increasing.
- Using \text{split}, we can define merge sort...
Merging: definition

merge produces a sorted list from two sorted input lists

\[
\text{merge} : \quad \text{List Int} \to \text{List Int} \to \text{List Int}
\]

\[
\text{merge} [\ ] \quad l = l
\]

\[
\text{merge} (x :: xs) \quad l = \text{merge}' l
\]

where \( \text{merge}' : \) \quad \text{List A} \to \text{List A} \n
\[
\text{merge}' [\ ] = x :: xs
\]

\[
\text{merge}' (y :: ys) = \begin{cases} \text{if } x \leq y \text{ then} \\
& x :: \text{merge} \; \text{xs} \; (y :: ys) \\
& \text{else } y :: \text{merge}' \; ys
\end{cases}
\]


Merging: termination

merge terminates by lexicographic ordering

\[
\text{merge} : \forall i : \text{ord}. \text{List^i Int} \to \text{List^\infty Int} \to \text{List^\infty Int}
\]

\[
\text{merge} [\ ] \quad l = l
\]

\[
\text{merge} (x :: xs)^{i+1} \quad l = \text{merge}' l
\]

where \( \text{merge}' : \) \quad \text{List A} \to \text{List A} \n
\[
\text{merge}' [\ ] = x :: xs
\]

\[
\text{merge}' (y :: ys) = \begin{cases} \text{if } x \leq y \text{ then} \\
& x :: \text{merge} \; xs^i (y :: ys) \\
& \text{else } y :: \text{merge}' \; ys
\end{cases}
\]
Merging: termination

merge terminates by lexicographic ordering

merge : \( \forall i : \text{ord}. \) List\(^i\) Int \( \rightarrow \) List\(^\infty\) Int \( \rightarrow \) List\(^\infty\) Int

\[
\begin{align*}
\text{merge} \; [\;] &= l \\
\text{merge} \; (x :: xs)^{i+1} \; l &= \text{merge}' \; l
\end{align*}
\]

where \( \text{merge}' : \forall j : \text{ord}. \) List\(^j\) A \( \rightarrow \) List\(^\infty\) A

\[
\begin{align*}
\text{merge}' \; [\;] &= x :: xs \\
\text{merge}' \; (y :: ys)^{j+1} &= \text{if } x \leq y \text{ then } x :: \text{merge} \; xs^{i} \; (y :: ys)^{j+1} \leq \infty \\
&\quad \text{else } y :: \text{merge}' \; ys^{j}
\end{align*}
\]

Merge sort: definition

msort : List Int \( \rightarrow \) List Int

\[
\begin{align*}
\text{msort} \; [\;] &= [\;] \\
\text{msort} \; (x :: l) &= \text{msort}' \; x \; l
\end{align*}
\]

msort' : Int \( \rightarrow \) List Int \( \rightarrow \) List Int

\[
\begin{align*}
\text{msort}' \; x \; [\;] &= [x] \\
\text{msort}' \; x \; (y :: l) &= \text{let } (xs , ys) = \text{split} \; l \text{ in } \\
&\quad \text{merge} \; (\text{msort}' \; x \; xs) \\
&\quad \text{msort}' \; y \; ys)
\end{align*}
\]
**Merge sort: termination**

\[
\text{msort} : \text{List}^{\infty} \text{Int} \rightarrow \text{List}^{\infty} \text{Int}
\]

\[
\text{msort} [\ ] = [\ ]
\]

\[
\text{msort} (x :: l) = \text{msort}' x l
\]

\[
\text{msort}' : \forall i : \text{ord. Int} \rightarrow \text{List}^i \text{Int} \rightarrow \text{List}^{\infty} \text{Int}
\]

\[
\text{msort}' x [\ ]^{i+1} = [x]
\]

\[
\text{msort}' x (y :: l^i) = \text{let } (xs^i, ys^i) = \text{split } l^i \text{ in}
\]

\[
\begin{align*}
\text{merge} & (\text{msort}' x xs^i) \\
& (\text{msort}' y ys^i)
\end{align*}
\]

---

**Leaving Hindley-Milner typing**

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

\[ \text{msort}' \text{ split } x \; [\ldots] = [x] \]
\[ \text{msort}' \text{ split } x \; (y :: l) = \text{let } (xs, ys) = \text{split } l \; \text{ in} \]
\[ \quad \text{merge (msort}' \; x \; xs) \]
\[ \quad \text{(msort}' \; y \; ys) \]

- The variable \text{split} can only be instantiated with \textit{non size increasing} functions
- This is naturally expressed with a rank-2 \textit{size polymorphic} type

Merge sort: abstract split (II)

\[ \text{msort}' : (\forall i : \text{ord}. \forall A : \ast. \text{List}^i A \rightarrow \text{List}^i A \times \text{List}^i A) \rightarrow \]
\[ \forall i : \text{ord}. \text{Int} \rightarrow \text{List}^i \text{Int} \rightarrow \text{List}^{\ast i} \text{Int} \]
\[ \text{msort}' \text{ split } x \; [\ldots]^{i+1} = [x] \]
\[ \text{msort}' \text{ split } x \; (y :: l^i) = \text{let } (xs^i, ys^i) = \text{split } l^i \; \text{ in} \]
\[ \quad \text{merge (msort}' \; x \; xs^i) \]
\[ \quad \text{(msort}' \; y \; ys^i) \]

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

- Tree ordinals
  \[\text{Ord}^\alpha = \mu^\alpha (\lambda X. 1 + X + (\text{Nat}^\infty \rightarrow X))\]

- Definable constructors

Slide 35

- ozero : \(\forall i : \text{ord}. \text{Ord}^i\)
- osucc : \(\forall i : \text{ord}. \text{Ord}^i \rightarrow \text{Ord}^{i+1}\)
- olim : \(\forall i : \text{ord}. (\text{Nat} \rightarrow \text{Ord}^i) \rightarrow \text{Ord}^{i+1}\)

An element of infinite height

\[\text{Ord}^1 \quad \text{Ord}^2 \quad \text{Ord}^3 \quad \ldots \subseteq \text{Ord}^\omega\]

Slide 36

\[
\begin{array}{c}
\text{ozero} \\
\text{osucc} \\
\text{olim}
\end{array}
\]

\[\text{Ord}^{\omega+1}\]
Example: addition for tree ordinals

- Constructors:
  
  \[
  \begin{align*}
  \text{ozo} & : \forall i : \text{ord}. \text{Ord}^i \\
  \text{osucc} & : \forall i : \text{ord}. \text{Ord}^i \to \text{Ord}^{i+1} \\
  \text{olim} & : \forall i : \text{ord}. (\text{Nat} \to \text{Ord}^i) \to \text{Ord}^{i+1}
  \end{align*}
  \]

Slide 37

- Addition:
  
  \[
  \begin{align*}
  \text{add} : \text{Ord}^\infty & \to \forall i : \text{ord}. \text{Ord}^i \to \text{Ord}^\infty \\
  \text{add } x \text{ ozero} & = x \\
  \text{add } x (\text{osucc } y^i)^{i+1} & = \text{osucc } (\text{add } x y^i) \\
  \text{add } x (\text{olim } f^i)^{i+1} & = \text{olim } (\lambda n. \text{add } x (f n)^i)
  \end{align*}
  \]

Productivity

- Productivity is dual to termination
- A productive process should continuously turn input into output
- Examples: editor, operating system, stream

Slide 38

- Important in embedded and functional reactive programming
Infinite structures

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

  \[
  (:) : A \to \forall i. \text{Stream}^i A \to \text{Stream}^{i+1} A
  \]

  \[
  \text{hd} : \forall i. \text{Stream}^{i+1} A \to A
  \]

  \[
  \text{tl} : \forall i. \text{Stream}^{i+1} A \to \text{Stream}^i A
  \]

- Subtyping: \( \text{Stream}^\infty A \leq \ldots \text{Stream}^{i+1} A \leq \text{Stream}^i A \)

\( F_\omega \): extension by coinduction

- Add type constructor \( \nu_\kappa : \text{ord} \to (\kappa \uplus \kappa) \uplus \kappa \).
- Example \( \text{Stream}^a = \lambda A. \nu^a (\lambda X. A \times X) \).
- Recursion rule also usable for corecursion!

\[
\begin{align*}
a & : \text{ord} \\
\text{fix} & : (\forall i. A^i \to A^{i+1}) \to A^a A^i \text{ admissible}
\end{align*}
\]

- Example: defining infinite sequence \( \text{upfrom} 0 = [0, 1, 2, \ldots] \)

\[
\begin{align*}
\text{upfrom} & : \text{Int} \to \text{Stream}^\infty \text{Int} \\
\text{upfrom} & := \text{fix} (\lambda \text{upfrom}. \lambda n. (n, \text{upfrom}(n + 1)))
\end{align*}
\]
Related works on type-based termination

- 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

- *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

- *A predicative strong normalization proof for a λ-calculus with interleaving inductive types* (Abel, Altenkirch, TYPES’99)
- *Co(ellation) 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

- 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 λµ-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

\[
\begin{align*}
\text{Tm} & = (\text{Tm} \times \text{Tm}) + (\text{Tm} \to \text{Tm}) \\
\text{app} & : \text{Tm}^i \to \text{Tm}^i \to \text{Tm}^{i+1} \\
\text{abs} & : (\text{Tm}^2 \to \text{Tm}^i) \to \text{Tm}^{i+1}
\end{align*}
\]

- 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?
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)

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).
Acknowledgements

• 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