Linear Dependent Types in a Subrecursive Setting

Ugo Dal Lago

Bounded Linear Logic Workshop
December 4th, 2013
Part I

Linear Dependent Types: Motivations
Characterizing Complexity Classes

$\mathcal{L}$

$\mathcal{C}$
Characterizing Complexity Classes
Characterizing Complexity Classes

\[ L \quad S \quad C \quad P \]
Proving $[S] = \mathcal{P}$

- $\mathcal{P} \subseteq [S]$
  - For every function $f$ which can be computed within the bounds prescribed by $\mathcal{P}$, there is a $P \in S$ such that $[P] = f$.

- $[S] \subseteq \mathcal{P}$
  - Semantically
    - For every $P \in S$, $[P] \in \mathcal{P}$ is proved by showing that an algorithm computing $[P]$ exists which works within the prescribed resource bounds.
    - $P \in \mathcal{L}$ does not necessarily exhibit a nice computational behavior.
    - Example: category-theoretic proofs [Hofmann09].
  - Operationally
    - Sometimes, $\mathcal{L}$ can be endowed with an effective operational semantics.
    - Let $\mathcal{L}_\mathcal{P} \subseteq \mathcal{L}$ be the set of those programs which work within the bounds prescribed by $\mathcal{C}$.
    - $[S] \subseteq \mathcal{P}$ can be shown by proving $S \subseteq \mathcal{L}_\mathcal{P}$. 
Proving $[S] = \mathcal{P}$

- $\mathcal{P} \subseteq [S]$
  - For every function $f$ which can be computed within the bounds prescribed by $\mathcal{P}$, there is a $P \in S$ such that $[P] = f$.

- $[S] \subseteq \mathcal{P}$
  - **Semantically**
    - For every $P \in S$, $[P] \in \mathcal{P}$ is proved by showing that an algorithm computing $[P]$ exists which works within the prescribed resource bounds.
    - $P \in \mathcal{L}$ does not necessarily exhibit a nice computational behavior.
    - Example: category-theoretic proofs [Hofmann09].
  - **Operationally**
    - Sometimes, $\mathcal{L}$ can be endowed with an effective operational semantics.
    - Let $\mathcal{L}_\mathcal{P} \subseteq \mathcal{L}$ be the set of those programs which work within the bounds prescribed by $\mathcal{C}$.
    - $[S] \subseteq \mathcal{P}$ can be shown by proving $S \subseteq \mathcal{L}_\mathcal{P}$. 
If Soundness is Proved Operationally...
If Soundness is Proved Operationally...
$P \in \mathcal{L}$

\[ S \quad \begin{cases} 
\text{Yes, } P \in \mathcal{LP} \\
\text{Don't know}
\end{cases} \]
ICC Systems as Static Analyzers

\[ P \in \mathcal{L} \quad \rightarrow \quad S \]

- Yes, \( P \in \mathcal{L_P} \) + bounds
- Don’t know
ICC: Intensional Expressive Power
ICC: Intensional Expressive Power
ICC: Intensional Expressive Power
ICC: Intensional Expressive Power
ICC: Intensional Expressive Power
ICC: Intensional Expressive Power

Safe Recursion [BC92]
Light Linear Logic [Girard97]
Bounded Recursion on Notation [Cobham63]
Bounded Arithmetic [Buss80]
Program Logics

- Judgments:

\[
\{ P \} C \{ Q \}
\]

precondition

postcondition

program

- Some rules:

\[
\{ P[E/x] \} \quad x := E \quad \{ P \} \\
\{ P \} \text{skip} \quad \{ P \}
\]

\[
\frac{\{ P \} \quad C \quad \{ Q \} \quad \{ Q \} \quad D \quad \{ R \}}{\{ P \} \quad C;D \quad \{ R \}}
\]

\[
R \Rightarrow P \quad \{ P \} \quad C \quad \{ Q \} \quad Q \Rightarrow S \\
\{ R \} \quad C \quad \{ S \}
\]
Program Logics

- Judgments:

\[
\{P\} C \{Q\}
\]

precondition \rightarrow \text{program} \rightarrow \text{postcondition}

- Some rules:

\[
\{P[E/x]\} \ x := E \ {P} \quad \{P\} \text{skip} \ {P}
\]

\[
\begin{array}{c}
\{P\} C \ {Q} \quad \{Q\} D \ {R} \\
\hline
\{P\} C; D \ {R}
\end{array}
\]

R \Rightarrow \ P \quad \{P\} C \ {Q} \quad Q \Rightarrow \ S

\[
\{R\} C \ {S}
\]
The formal system is sound.
  - If true formulas of PA are used as side-conditions.

It’s also relatively complete [Cook78].
  - All true assertions can be derived if all true PA formulas can be used as side-conditions.

Concrete axiom systems can be derived by throwing in a concrete sound formal system $\mathcal{F}$ for PA.
  - They are sound.
  - They are incomplete, due to Gödel incompleteness.
  - $\mathcal{F}$ is solely responsible for their incompleteness.

A variety of FH logics enjoy the properties above.
  - Including some for higher-order programs [Honda2000]...
  - ... and some in which the complexity of programs and not only their extensional behavior is taken into account.
ICC Systems

Degree of Completeness

Property Complexity

Degree of Completeness
Degree of Completeness
Property Complexity

ICC Systems
Some Examples

- **Simply Types**
  - “Well-typed programs do not go wrong”.
  - Type inference and type checking are often decidable.

- **Dependent Types**
  - Type checking is decidable.
  - Interesting, extensional properties can be specified.

- **Intersection Types**
  - Sound and complete for termination.
  - In certain cases, the complexity of evaluation of a $\lambda$-term can be read from its type derivation [deCarvalho2008], [Lengrand2011].
  - Type inference is *not* decidable.
  - Studying programs as *functions* requires considering an infinite family of type derivations.
Part II

The System
“[BLL] allows for the definition of all polynomial time computable functions and [...] can define datatypes via their Church encodings. Like in Cobham’s system explicit size bounds need to be maintained; the difference is that enforcement of these bounds is intrinsically guaranteed by the type system and does neither rely on external reasoning nor on an ad hoc solutions such as cutting off at size overflow [...]. Unfortunately, BLL has received very little attention since its publication; a further elaboration might prove worthwhile.”

— Martin Hofmann, Programming Languages Capturing Complexity Classes, ACM SIGACT News.
Bounded Linear Logic
Bounded Linear Logic

- One of the earliest examples of a system capturing polynomial time **functions** [GSS1992].
  - Extensionally!
  - For every polytime function there is at least one proof in BLL computing it.

- Types:

\[
A ::= \alpha(p_1, \ldots, p_n) \mid A \otimes A \mid A \rightarrow A \mid \forall \alpha. A \mid !x<pA
\]

- How many “polytime proofs” does BLL capture?
  - There’s evidence they are many [DLHofmann2010].
- Type checking can be **problematic**. As an example:

\[
\frac{\Gamma, !x<pA, !y<qA\{p + y/x\} \vdash B \quad p + q \leq r}{\Gamma, !x<rA \vdash B} \quad X
\]
One of the earliest examples of a system capturing polynomial time \textbf{functions} [GSS1992].

- Extensionally!
- For every polytime function there is \textbf{at least one} proof in BLL computing it.

Types:

\[ A ::= \alpha(p_1, \ldots, p_n) | A \otimes A | A \multimap A | \forall \alpha.A | !_{x<p}A \]

How many “polytime proofs” does BLL capture?

- There’s evidence they are \textbf{many} [DLHofmann2010].

Type checking can be \textbf{problematic}. As an example:

\[
\Gamma, !_{x<p}A, !_{y<q}A\{p + y/x\} \vdash B \quad p + q \leq r \quad X
\]

\[
\Gamma, !_{x<r}A \vdash B
\]
\[ !A \rightsquigarrow !A \otimes !A \]

\[ !x<p+qA \rightsquigarrow !x<pA \otimes !x<qA\{x + p/x\} \]
BLL vs. MELL

\[ !A \rightarrow !A \otimes !A \]

\[ \Downarrow \]

\[ !x < p + q A \rightarrow !x < p A \otimes !x < q A \{ x + p/x \} \]
\( !A \rightarrow A \)

\( \Downarrow \)

\( !_{x<1} A \rightarrow A\{0/x\} \)
BLL vs. MELL

\(!A \leadsto A\)

\(!x < 1 A \leadsto A\{0/x\}\)
\[
!A \rightarrow !!A
\]

\[
!x < \sum_{y < p} qA \rightarrow !y < p!x < qA \{x + \sum_{z < y} q\{z/y\}/x\}
\]
BLL vs. MELL

\[ !A \rightarrow \circ!!A \]

\[ \downarrow \]

\[ !x < \sum_{y < p} qA \rightarrow !y < p!x < qA \{ x + \sum_{z < y} q\{ z/y \}/x \} \]
BLL is not Intensionally Complete

- Everything works in polynomial time.
  - Take:

    \[
    \log(x) \rightarrow 1^{\log_2(|x|)}; \\
    \exp(x) \rightarrow 1^{2^{|x|}}.
    \]

- Consider:

  \[
  id(x) \rightarrow \exp(\log(x)).
  \]
  - \(id\) is polytime, but cannot be typed in BLL, because \(\exp\) cannot.

- How about hereditarily polytime algorithms?
  - Polynomials seem to be “too uniform” anyway.
  - Hard to capture algorithms which are polytime but not blindly polytime.

- A more general notion of a bound is needed!
Why?

\[ !_a < I A \]

\[ \downarrow \]

\[ A\{0/a\} \otimes \ldots \otimes A\{I - 1/a\} \]
Why?

\[ a < I A \]

\[ \Downarrow \]

\[ A\{0/a\} \otimes \ldots \otimes A\{I - 1/a\} \]
A type system for the $\lambda$-calculus with constants and full higher-order recursion. (i.e. PCF).

Greatly inspired by BLL.

Indices are not necessarily polynomials, but terms from a signature $\Sigma$.

Symbols in $\Sigma$ are given a meaning by an equational program $\mathcal{E}$.

Side conditions: $K_1 \leq H_1, \ldots, K_n \leq H_n \models^{\mathcal{E}} I \leq J$

Types and modal types:

$$A, B ::= \text{Nat}[I, J] \mid F \rightarrow A$$  basic types

$$F, G ::= [a < I] \cdot A$$  modal types
A type system for the \( \lambda \)-calculus with constants and full higher-order recursion. (i.e. PCF).

Greatly inspired by BLL.

Indices are not necessarily polynomials, but terms from a signature \( \Sigma \).

Symbols in \( \Sigma \) are given a meaning by an equational program \( \mathcal{E} \).

Side conditions: \( K_1 \leq H_1, \ldots, K_n \leq H_n \models \mathcal{E} \quad I \leq J \)

Types and modal types:

\[
A, B ::= \text{Nat}[I] | F \rightarrow A \quad \text{basic types}
\]

\[
F, G ::= [a < I] \cdot A \quad \text{modal types}
\]
A type system for the \( \lambda \)-calculus with constants and full higher-order recursion. (i.e. PCF).

Greatly inspired by BLL.

Indices are not necessarily polynomials, but terms from a signature \( \Sigma \).

Symbols in \( \Sigma \) are given a meaning by an equational program \( \mathcal{E} \).

Side conditions: \( K_1 \leq H_1, \ldots, K_n \leq H_n \models^\mathcal{E} I \leq J \)

Types and modal types:

\[
A, B ::= \{ a \in \text{Nat} \mid I \} \mid F \rightarrow A \quad \text{basic types}
\]

\[
F, G ::= [a < I] \cdot A \quad \text{modal types}
\]
The Meaning of Types

\[ [a < 1] \cdot A \rightarrow B \]

\[ \Downarrow \]

\[ (A\{0/a\} \otimes \ldots \otimes A\{1 - 1/a\}) \rightarrow B \]
The Meaning of Types

\[ [a < I] \cdot A \rightarrow B \]

\[ \Downarrow \]

\[ (A\{0/a\} \otimes \ldots \otimes A\{I - 1/a\}) \rightarrow B \]
\( \Gamma, x : [a < I] \cdot A \vdash^\varepsilon_0 x : B\{0/a\} \)

\( \vdash^\varepsilon [a < I] \cdot A \equiv [a < 1] \cdot B \)

**Weight**
\[ \vdash^E \text{Nat}[I + 1, J + 1] \subseteq \text{Nat}[K, H] \]
\[ \Gamma \vdash^E M : \text{Nat}[I, J] \]
\[ \Gamma \vdash^E \text{succ}(M) : \text{Nat}[K, H] \]
$ \Gamma \vdash^\varepsilon \Sigma \subseteq \Gamma \uplus \sum_{a < I} \Delta$

$\Gamma \vdash^\varepsilon M : [a < I] \cdot A \rightarrow B$

$\Delta \vdash^\varepsilon N : A$

$\Sigma \vdash^\varepsilon J + \sum_{a \leq I} K + I \quad MN : B \quad A$
dlPCF: Some Rules

Sum of Modal Types

\[
\begin{align*}
\Gamma \vdash^\varepsilon \Sigma &\subseteq \Gamma \oplus \sum_{a<i} \Delta \\
\Gamma \vdash^\varepsilon M : [a < I] \cdot A \rightarrow B \\
\Delta \vdash^\varepsilon N : A \\
\sum \vdash^\varepsilon_{j + \sum a \leq I} MN : B
\end{align*}
\]
Bounded Sum of Modal Types

\[ \vdash^\varepsilon \Sigma \subseteq \Gamma \uplus \sum_{a < I} \Delta \]
\[ \Gamma \vdash^\varepsilon J M : [a < I] \cdot A \rightarrow B \]
\[ \Delta \vdash^\varepsilon K N : A \]

\[ \Sigma \vdash^\varepsilon J + \sum_{a \leq I} K + I MN : B \]
\[
b; x : [a < I] \cdot A \xrightarrow{\xi} M : B
\]

\[
\begin{array}{c}
B\{0/b\} \\
\quad [a < I\{0/b\}] \cdot A\{0/b\} \\
\quad \quad A\{0/b\}\{0/a\} \quad \cdots \quad A\{0/b\}\{I\{0/b\} - 1/a\} \\
\quad \quad \equiv \quad \quad \equiv \\
B\{1/b\} \\
\quad [a < I\{1/b\}] \cdot A\{1/b\} \\
\quad \quad \vdots \\
\end{array}
\]

\[
\begin{array}{c}
\quad \cdots \\
\quad B\{I\{0/b\}/b\} \\
\quad [a < I\{(I\{0/b\})/b\}] \cdot A\{(I\{0/b\})/b\} \\
\quad \vdots
\end{array}
\]
\[ b; x : [a < 1] \cdot A \xrightarrow{\mathcal{E}} M : B \]

\[
\begin{array}{c}
B\{0/b\} \\
\quad [a < 1\{0/b\}] \cdot A\{0/b\}
\end{array}
\]

\[
\begin{array}{ccc}
A\{0/b\}\{0/a\} & \cdots & A\{0/b\}\{I\{0/b\} - 1/a\}
\end{array}
\]

\[
\begin{array}{c}
B\{1/b\} \\
\quad [a < 1\{1/b\}] \cdot A\{1/b\}
\end{array}
\]

\[
\begin{array}{c}
B\{I\{0/b\}/b\} \\
\quad [a < 1\{(I\{0/b\})/b\}] \cdot A\{(I\{0/b\})/b\}
\end{array}
\]

\[
\begin{array}{c}
\vdots
\end{array}
\]
\[ b; x : [a < I] \cdot A \vdash_{\beta}^\xi M : B \]

\[ B\{0/b\} \]

\[ [a < I\{0/b\}] \cdot A\{0/b\} \]

\[ A\{0/b\}\{0/a\} \quad \cdots \quad A\{0/b\}\{I\{0/b\} - 1/a\} \]

\[ \equiv \quad \cdots \quad \equiv \]

\[ B\{1/b\} \quad \cdots \quad B\{I\{0/b\}/b\} \]

\[ [a < I\{1/b\}] \cdot A\{1/b\} \quad [a < I\{(I\{0/b\})/b\}] \cdot A\{(I\{0/b\})/b\} \]

\[ \vdots \quad \vdots \]
\[
b; x : [a < I] \cdot A \vdash^\xi_j M : B
\]

\[
B\{0/b\}
\]

\[
[a < I\{0/b\}] \cdot A\{0/b\}
\]

\[
\begin{align*}
A\{0/b\}\{0/a\} & \equiv B\{1/b\} \\
\cdots & \equiv \cdots \\
A\{0/b\}\{I\{0/b\} - 1/a\} & \equiv B\{I\{0/b\}/b\}
\end{align*}
\]

\[
[a < I\{1/b\}] \cdot A\{1/b\} \\
[a < I\{(I\{0/b\})/b\}] \cdot A\{(I\{0/b\})/b\}
\]

\[
\vdots
\]
\[ b; x : [a < 1] \cdot A \vdash^\varepsilon M : B \]

\[ B\{0/b\} \]

\[ [a < I\{0/b\}] \cdot A\{0/b\} \]

\[ A\{0/b\}\{0/a\} \quad \cdots \quad A\{0/b\}\{I\{0/b\} - 1/a\} \]

\[ \equiv \]

\[ B\{1/b\} \quad \cdots \quad B\{I\{0/b\}/b\} \]

\[ [a < I\{1/b\}] \cdot A\{1/b\} \quad [a < I\{(I\{0/b\})/b\}] \cdot A\{(I\{0/b\})/b\} \]

\[ \vdots \quad \vdots \]
\[
\begin{align*}
  b; x : [a < I] \cdot A \vdash^\xi M : B \\
  B\{0/b\} \\
  [a < I\{0/b\}] \cdot A\{0/b\} \\
  A\{0/b\}\{0/a\} \quad \cdots \quad A\{0/b\}\{I\{0/b\} - 1/a\} \\
  \equiv \quad \cdots \quad \equiv \\
  B\{1/b\} \quad \cdots \quad B\{I\{0/b\}/b\} \\
  [a < I\{1/b\}] \cdot A\{1/b\} \quad [a < I\{I\{0/b\}/b\}] \cdot A\{I\{0/b\}/b\} \\
  \vdots \quad \vdots 
\end{align*}
\]
\( b; x: [a < 1] \cdot A \vdash \frac{\xi}{j} M : B \)
\( b; x : [a < I] \cdot A \vdash^j M : B \)
\[ \vdash_I M : [J] \cdot \text{Nat}[a] \rightarrow \text{Nat}[K, H] \]

What does this mean?

- \( M \) computes a function from natural numbers to natural numbers.
- Something **extensional**:
  - On input a natural number \( a \), \( M \) returns a natural number in \([K, H]\)
- Something more **intensional**:
  - The cost of evaluation of \( M \) on an input \( a \) is \( I + J \).
- Two questions:
  - Is this **correct**?
  - How many programs can be captured this way?
- Type soundness is somehow inherited from BLL.
\[ \vdash_I M : \llbracket J \rrbracket \cdot \text{Nat}[a] \to \text{Nat}[K, H] \]

What does this mean?

- **\( M \)** computes a function from natural numbers to natural numbers.
- Something **extensional**:
  - On input a natural number \( a \), \( M \) returns a natural number in \([K, H]\).
- Something more **intensional**:
  - The cost of evaluation of \( M \) on an input \( a \) is \( I + J \).
- Two questions:
  - Is this **correct**?
  - How many programs can be captured this way?
- Type soundness is somehow inherited from BLL.
\[ \vdash_I M : [J] \cdot \text{Nat}[a] \rightarrow \text{Nat}[K, H] \]

**What does this mean?**

- *M* computes a function from natural numbers to natural numbers.
- Something **extensional**:
  - On input a natural number *a*, *M* returns a natural number in [K, H]
- Something more **intensional**:
  - The cost of evaluation of *M* on an input *a* is *I + J*.
- Two questions:
  - Is this **correct**?
  - How many programs can be captured this way?
- Type soundness is somehow inherited from BLL.
\( \vdash I M : [J] \cdot \text{Nat}[a] \rightarrow \text{Nat}[K, H] \)

What does this mean?

- \( M \) computes a function from natural numbers to natural numbers.
- Something \textit{extensional}:
  - On input a natural number \( a \), \( M \) returns a natural number in \( [K, H] \)
- Something more \textit{intensional}:
  - The cost of evaluation of \( M \) on an input \( a \) is \( I + J \).
- Two questions:
  - Is this \textit{correct}?
  - How many programs can be captured this way?
- Type soundness is somehow inherited from BLL.
\( \vdash \text{\textit{M}} : [\text{\textit{J}}] \cdot \text{Nat}[a] \rightarrow \text{Nat}[K, H] \)

**What does this mean?**

- \( \text{\textit{M}} \) computes a function from natural numbers to natural numbers.
- Something **extensional**:
  - On input a natural number \( a \), \( \text{\textit{M}} \) returns a natural number in \([K, H]\)
- Something more **intensional**:
  - The cost of evaluation of \( \text{\textit{M}} \) on an input \( a \) is \( I + J \).
- Two questions:
  - Is this **correct**?
  - How many programs can be captured this way?
- Type soundness is somehow inherited from BLL.
What does this mean?

- $M$ computes a function from natural numbers to natural numbers.
- Something **extensional**:
  - On input a natural number $a$, $M$ returns a natural number in $[K, H]$
- Something more **intensional**:
  - The cost of evaluation of $M$ on an input $a$ is $I + J$.
- Two questions:
  - Is this **correct**?
  - How many programs can be captured this way?
- Type soundness is somehow inherited from BLL.
\[
\vdash I \quad M : [J] \cdot \text{Nat}[a] \rightarrow \text{Nat}[K, H]
\]

What does this mean?

- \( M \) computes a function from natural numbers to natural numbers.
- Something **extensional**: 
  - On input a natural number \( a \), \( M \) returns a natural number in \([K, H]\)
- Something more **intensional**: 
  - The cost of evaluation of \( M \) on an input \( a \) is \( I + J \).
- Two questions:
  - Is this **correct**?
  - How many programs can be captured this way?
- Type soundness is somehow inherited from BLL.
Completeness for Programs

- The following holds only when $\mathcal{E}$ is **universal**.
- $(\|A\|)$ is the PCF type underlying $A$, i.e. its skeleton.
- The type system can be lifted to configurations of an abstract machine à la Krivine.

**Lemma (Weighted Subject Expansion)**

*If $D$ has weight $I$ and type $A$ and $C$ is typable with type $(\|A\|)$. Then, $C \rightarrow D$ implies that $C$ has weight $J$ and type $A$, where $\models J \leq I + 1$."

**Theorem (Relative Completeness for Programs)**

*Let $M$ be a PCF program such that $M \downarrow^n m$. Then, there exist two index terms $I$ and $J$ such that $\llbracket I \rrbracket^U \leq n$ and $\llbracket J \rrbracket^U = m$ and such that the term $M$ is typable in $\text{dlPCF}$ as $\vdash^U_I M : \text{Nat}[J]$. "*
Completeness for Programs

- The following holds only when $E$ is universal.
- $(|A|)$ is the PCF type underlying $A$, i.e. its skeleton.
- The type system can be lifted to configurations of an abstract machine à la Krivine.

Lemma (Weighted Subject Expansion)

If $D$ has weight $I$ and type $A$ and $C$ is typable with type $(|A|)$. Then, $C \rightarrow D$ implies that $C$ has weight $J$ and type $A$, where $J \leq I + 1$.

Theorem (Relative Completeness for Programs)

Let $M$ be a PCF program such that $M \downarrow^n m$. Then, there exist two index terms $I$ and $J$ such that $[[I]]^U \leq n$ and $[[J]]^U = m$ and such that the term $M$ is typable in dlPCF as $\vdash^U I M : \text{Nat}[J]$.
Completeness for Programs

- The following holds only when $E$ is universal.
- $\langle A \rangle$ is the PCF type underlying $A$, i.e. its skeleton.
- The type system can be lifted to configurations of an abstract machine à la Krivine.

**Lemma (Weighted Subject Expansion)**

If $D$ has weight $I$ and type $A$ and $C$ is typable with type $\langle A \rangle$. Then, $C \rightarrow D$ implies that $C$ has weight $J$ and type $A$, where $J \leq I + 1$.

**Theorem (Relative Completeness for Programs)**

Let $M$ be a PCF program such that $M \downarrow^n m$. Then, there exist two index terms $I$ and $J$ such that $[I]^U \leq n$ and $[J]^U = m$ and such that the term $M$ is typable in $d\ell$PCF as $\vdash^U M : \text{Nat}[J]$. 
Completeness for Functions

- It strongly relies on the universality of $U$.
- Suppose that $\{\pi_n\}_{n \in \mathbb{N}}$ is an r.e. family of type derivations:
  - For the same term $M$;
  - Having the same PCF skeleton (as type derivations);
  Then we can turn them into a single, parametric type derivation.

**Theorem (Relative Completeness for Functions)**

Suppose that $M$ is a PCF term such that $\vdash M : \text{Nat} \rightarrow \text{Nat}$. Moreover, suppose that there are two (total and computable) functions $f, g : \mathbb{N} \rightarrow \mathbb{N}$ such that $M \downarrow^{g(n)} f(n)$. Then there are terms $I, J, K$ with $\llbracket I + J \rrbracket \leq g$ and $\llbracket K \rrbracket = f$, such that

$$\vdash_U M : [J] \cdot \text{Nat}[a] \rightarrow \text{Nat}[K].$$
Completeness for Functions

- It strongly relies on the universality of \( U \).
- Suppose that \( \{\pi_n\}_{n \in \mathbb{N}} \) is an r.e. family of type derivations:
  - For the same term \( M \);
  - Having the same PCF skeleton (as type derivations);

Then we can turn them into a single, parametric type derivation.

Theorem (Relative Completeness for Functions)

Suppose that \( M \) is a PCF term such that \( \vdash M : \text{Nat} \rightarrow \text{Nat} \).
Moreover, suppose that there are two (total and computable) functions \( f, g : \mathbb{N} \rightarrow \mathbb{N} \) such that \( M \, \text{nat} \downarrow g(n) \, f(n) \). Then there are terms \( I, J, K \) with \( [I + J] \leq g \) and \( [K] = f \), such that

\[ \vdash_1^U \, M : [J] \cdot \text{Nat}[a] \rightarrow \text{Nat}[K]. \]
Linear Dependent Types [DLGabboardi2011]

![Graph showing Degree of Completeness vs Property Complexity with BLL and $d\ell$PCF]

- Degree of Completeness
- Property Complexity
- BLL
- $d\ell$PCF

Degree of Completeness
Property Complexity
BLL
$d\ell$PCF
Part III

Recent Contributions
We have a powerful system for complexity analysis, which is relatively complete.
  - Thus undecidable!

So what?

- It’s the same situation as in program logics…
- But there are algorithms for weakest preconditions, there…
- We need something similar also here…
Type Inference
(joint with Petit)

- We have a powerful system for complexity analysis, which is relatively complete.
  - Thus undecidable!

- **So what?**
  - It’s the same situation as in program logics,…
  - But there are algorithms for weakest preconditions, there,…
  - We need something similar also here,…
Type Inference
(joint with Petit)

- We have a powerful system for complexity analysis, which is relatively complete.
  - Thus undecidable!

- So what?
- It’s the same situation as in program logics…
  - But there are algorithms for weakest preconditions, there…
  - We need something similar also here…
We have a powerful system for complexity analysis, which is relatively complete.

- Thus undecidable!

**So what?**

- It’s the same situation as in program logics...
- But there are algorithms for weakest preconditions, there...

- We need something similar also here...
We have a powerful system for complexity analysis, which is relatively complete.
  - Thus undecidable!

So what?
- It’s the same situation as in program logics...
- But there are algorithms for weakest preconditions, there...
- We need something similar also here...
Type Inference
(joint with Petit)
Type Inference
(joint with Petit)

\[ \vdash^\mathcal{E}_I t : \sigma \iff \forall E \in \mathcal{I}. \models^\mathcal{E} E \]
A necessary condition for completeness: universality of $\mathcal{E}$.

What if we restrict PCF to a subrecursive language similar to Gödel’s T?

We don’t need the complicated machinery needed for handling recursion in PCF, anymore.

We can easily handle references through effects.

Judgments become:

$$\Gamma \vdash \mathcal{E}_M : A, \varphi$$

Two interesting results on type inference:

- Linearity on higher-order variables implies termination of $\mathcal{E}$;
- If all occurrences of recursion in the underlying program are first-order, then $\mathcal{E}$ is primitive recursive.
A necessary condition for completeness: universality of $\mathcal{E}$.

What if we restrict PCF to a subrecursive language similar to Gödel’s $\mathcal{T}$?

We don’t need the complicated machinery needed for handling recursion in PCF, anymore.

We can easily handle references through effects.

Judgments become:

$$\Gamma \vdash_1 \xi \ M : A, \varphi$$

Two interesting results on type inference:

- Linearity on higher-order variables implies termination of $\mathcal{E}$;
- If all occurrences of recursion in the underlying program are first-order, then $\mathcal{E}$ is primitive recursive.
Linear Dependency and Session Types (Ongoing)

- Linear logic and session types are strongly related (e.g. [CairesPfenning10]).
- The type of a (potentially non-terminating) session between a bank and a customer could be, e.g.

\[
\mu \alpha.\text{Nat} \rightarrow \text{Nat} \oplus \alpha
\]

- With linear dependency, one could decorate the type as follows:

\[
\mu \alpha(a).\text{Nat}(b) \rightarrow \text{Nat}(K) \oplus_I \alpha(J)
\]

where I, H are boolean index term.
- If the rewrite system

\[
f(a) \rightarrow \text{if } I \text{ then } f(J) \text{ else end}
\]

is terminating, then the session above is finite.
Questions?