Turing-Completeness of Polymorphic Stream Equation Systems

Christian Sattler

Joint Work with Florent Balestrieri

University of Nottingham

May 30, 2012
Semantic Setting

- Partial streams $\text{Str}_D := \mathbb{N} \rightarrow D_\perp$ over datatype $D$
Semantic Setting

- Partial streams $\text{Str}_D := \mathbb{N} \rightarrow D \perp$ over datatype $D$
- Polymorphic stream functions: families $f_D : \text{Str}^m_D \rightarrow \text{Str}_D$ polymorphic (or natural, parametric, ...) in $D$
Semantic Setting

- Partial streams $\text{Str}_D := \mathbb{N} \rightarrow D_\bot$ over datatype $D$
- Polymorphic stream functions: families $f_D : \text{Str}_D^m \rightarrow \text{Str}_D$ polymorphic (or natural, parametric, ...) in $D$:

$$
\begin{align*}
\text{Str}_{D_1} \times \ldots \times \text{Str}_{D_1} & \xrightarrow{f_{D_1}} \text{Str}_{D_1} \\
[\text{map}(g), \ldots, \text{map}(g)] & \downarrow \\
\text{Str}_{D_2} \times \ldots \times \text{Str}_{D_2} & \xrightarrow{f_{D_2}} \text{Str}_{D_2}
\end{align*}
$$

for all $g : D_1 \rightarrow (D_2)_\bot$ (where $\text{map}(g)(s)(i) := g(s(i)))$. 
Polymorphic stream functions are effectively crippled: They may only discard, duplicate, and reorder stream elements.

Example: \( f(s, t) = s(0) :: t(0) :: t(1) :: s(1) :: s(2) :: t(2) :: \ldots \)
Polymorphic stream functions are effectively crippled: They may only discard, duplicate, and reorder stream elements.

Example: \( f(s, t) = s(0) :: t(0) :: t(1) :: s(1) :: s(2) :: t(2) :: \ldots \)

Polymorphic \( f_D : \text{Str}_D^m \rightarrow \text{Str}_D \) can equivalently be represented by an indexing function \( \bar{f} : \mathbb{N} \rightarrow (\{0, \ldots, m - 1\} \times \mathbb{N})_\perp \): For \( k \in \mathbb{N} \) and \( (j, i) := \bar{f}(k) \), the \( k \)-th output element of \( f_D \) is given by the \( i \)-th stream element of the \( j \)-th input argument of \( f_D \).

For the above example: \( \bar{f} \) starts with \( (0, 0), (1, 0), (1, 1), (0, 1), (0, 2), (1, 2), \ldots \)
Polymorphic stream functions are effectively crippled: They may only discard, duplicate, and reorder stream elements.

Example: \( f(s, t) = s(0) :: t(0) :: t(1) :: s(1) :: s(2) :: t(2) :: \ldots \)

Polymorphic \( f_D : \text{Str}_D^m \rightarrow \text{Str}_D \) can equivalently be represented by an indexing function \( \bar{f} : \mathbb{N} \rightarrow (\{0, \ldots, m-1\} \times \mathbb{N})_\perp \):

For \( k \in \mathbb{N} \) and \( (j, i) := \bar{f}(k) \), the \( k \)-th output element of \( f_D \) is given by the \( i \)-th stream element of the \( j \)-th input argument of \( f_D \).

For the above example: \( \bar{f} \) starts with \( (0, 0), (1, 0), (1, 1), (0, 1), (0, 2), (1, 2), \ldots \)

In case of \( f \) unary we can write \( \bar{f} : \mathbb{N} \rightarrow \mathbb{N}_\perp \).

This gives a direct notion of \emph{computability} for polymorphic stream functions.
A *(stream equation)* system is a set of equations

\[
\begin{align*}
    f_0(s_0, \ldots, s_{m_0-1}) &= \rho_1, \\
    \ldots \\
    f_{n-1}(s_0, \ldots, s_{m_{n-1}-1}) &= \rho_n
\end{align*}
\]

where

\[
\begin{align*}
    \rho, \sigma &::= s_i \mid \text{tail}(\sigma) \mid \text{head}(\sigma) :: \sigma' \mid f_j(\sigma_0, \ldots, \sigma_{m_j-1}).
\end{align*}
\]
A *(stream equation) system* is a set of equations

\[
\begin{align*}
    f_0(s_0, \ldots, s_{m_0-1}) &= \rho_1, \\
    &\ldots \\
    f_{n-1}(s_0, \ldots, s_{m_n-1}) &= \rho_n
\end{align*}
\]

where

\[
\rho, \sigma ::= s_i \mid \text{tail}(\sigma) \mid \text{head}(\sigma) :: \sigma' \mid f_j(\sigma_0, \ldots, \sigma_{m_j-1}).
\]

➤ Interpret head(·) :: · (head-cons) and tail canonically

➤ Kleene’s theorem guarantees unique (partial) least fixpoint solutions \((f_j)_D : \text{Str}^m_D \rightarrow \text{Str}_D\).
Examples

Interleaving:

\[
\text{zip}_n(s_0, \ldots, s_{n-1}) = \text{head}(s_0) :: \text{zip}_n(s_1, \ldots, s_{n-1}, \text{tail}(s_0))
\]

\[
= \text{head}(s_0) \ldots \text{head}(s_{n-1}) ::
\]

\[
\text{zip}_n(\text{tail}(s_0), \ldots, \text{tail}(s_{n-1}))
\]
Examples

Interleaving:

\[ \text{zip}_n(s_0, \ldots, s_{n-1}) = \text{head}(s_0) :: \text{zip}_n(s_1, \ldots, s_{n-1}, \text{tail}(s_0)) \]

\[ = \text{head}(s_0) \ldots \text{head}(s_{n-1}) :: \]

\[ \text{zip}_n(\text{tail}(s_0), \ldots, \text{tail}(s_{n-1})) \]

Projection:

\[ \text{proj}_n(s) = \text{head}(s) :: \text{proj}_n(\text{tail}^n(s)) \]
Operational Semantics: Small-Step

Introduce functional relation $\sigma ! k$ for computing stream position $k \in \mathbb{N}$ in $\sigma$:

$$\text{tail}(\sigma) ! k \rightarrow \sigma ! k + 1,$$

$$\text{head}(\sigma) :: \sigma' ! k \rightarrow \begin{cases} \sigma ! 0 & \text{if } k = 0, \\ \sigma' ! k - 1 & \text{else}, \end{cases}$$

$$f_j(\sigma_0, \ldots, \sigma_{n_j - 1}) ! k \rightarrow \rho_j[\sigma_i/s_i]_{i \in \{0, \ldots, n_j - 1\}} ! k$$
Introduce functional relation $\sigma \mapsto k$ for computing stream position $k \in \mathbb{N}$ in $\sigma$:

$$
tail(\sigma) \mapsto k \rightarrow \sigma \mapsto k + 1,
$$

$$
\text{head}(\sigma) :: \sigma' \mapsto k \rightarrow \begin{cases} 
\sigma \mapsto 0 & \text{if } k = 0, \\
\sigma' \mapsto k - 1 & \text{else}, 
\end{cases}
$$

$$
f_j(\sigma_0, \ldots, \sigma_{n_j - 1}) \mapsto k \rightarrow \rho_j[\sigma_i/s_i]_{i \in \{0, \ldots, n_j - 1\}} \mapsto k
$$

We are now able to explicitly define

$$
f_{j,\mathbb{D}}(s_0, \ldots, s_{m_j - 1}) = \begin{cases} 
\mathit{s_w}(i) & \text{if } f_j(s_0, \ldots, s_{m_j - 1}) \mapsto^* \mathit{s_w} \mapsto i, \\
\bot & \text{else.}
\end{cases}
$$
Equivalent view: rewriting system on terms $\text{head}(\text{tail}^k(\sigma))$ based on

\[
\begin{align*}
\text{head}(\text{head}(\sigma) :: \sigma') & \rightarrow \text{head}(\sigma), \\
\text{tail}(\text{head}(\sigma) :: \sigma') & \rightarrow \sigma', \\
 f_j(\sigma_0, \ldots, \sigma_{n_j-1}) & \rightarrow \rho_j[\sigma_i/S_i]_{i \in \{0, \ldots, n_j-1\}}
\end{align*}
\]

with deterministic outermost rewriting strategy.

Or simply: an infinitary rewriting system.
Examples: Indexing Functions

Interleaving:

$$zip_n(s_0, \ldots, s_{n-1}) = head(s_0) :: zip_n(s_1, \ldots, s_{n-1}, tail(s_0))$$

$$zip_n(k) = (k \mod n, \lfloor n/k \rfloor)$$
Examples: Indexing Functions

Interleaving:

\[ \text{zip}_n(s_0, \ldots, s_{n-1}) = \text{head}(s_0) :: \text{zip}_n(s_1, \ldots, s_{n-1}, \text{tail}(s_0)) \]

\[ \text{zip}_n(k) = (k \mod n, \lfloor n/k \rfloor) \]

Projection:

\[ \text{proj}_n(s) = \text{head}(s) :: \text{proj}_n(\text{tail}^n(s)) \]

\[ \text{proj}_n(k) = nk \]
A Warning

Important
Polymorphism is a debilitating restriction!
A Warning

Important
Polymorphism is a debilitating restriction!
Even with only two distinguishable elements $0, 1 \in \mathbb{D}$, we can directly represent Turing machines on alphabet $\{0, 1\}$ [Roșu, 2006; Simonsen, 2009]:

- One stream for the tape left of the head
- One stream for the tape right of the head
- State transition rules based on case distinction (matching) of head value and current state (fixed number of bits)
A Warning

Important
Polymorphism is a debilitating restriction!
Even with only two distinguishable elements $0, 1 \in \mathbb{D}$, we can directly represent Turing machines on alphabet \{0, 1\} \cite{Rosu2006, Simonsen2009}:

- One stream for the tape left of the head
- One stream for the tape right of the head
- State transition rules based on case distinction (matching) of head value and current state (fixed number of bits)

With only polymorphic stream functions, it is not clear at all how to encode machines, not being able to match on data.
Can we decide *productivity* (equivalently, semantic totality, convergence, ...)?
Can we decide *productivity* (equivalently, semantic totality, convergence, . . .)?

- No, it is $\Pi^0_2$-complete [Endrullis, Grabmayer, Hendriks, 2009] - proof by reduction from Fractran.
Big Questions

Can we decide *productivity* (equivalently, semantic totality, convergence, ...)?
  
  - No, it is $\Pi_2^0$-complete [Endrullis, Grabmayer, Hendriks, 2009] - proof by reduction from Fractran.

What does the class of definable stream functions look like?
Can we decide *productivity* (equivalently, semantic totality, convergence, . . .)?

- No, it is $\Pi^0_2$-complete [Endrullis, Grabmayer, Hendriks, 2009] - proof by reduction from Fractran.

What does the class of definable stream functions look like?

- *All* computable polymorphic stream functions are definable - proof idea similar to the above, but with reduction from counter machines.
- A polymorphic stream function is *computable* iff its indexing function of type $\mathbb{N} \rightarrow (\{0, \ldots, n - 1\} \times \mathbb{N}) \perp$ is computable.
Definability: Basic Idea

- Encode register state $R_0, R_1, R_2, \ldots \in \mathbb{N}$ as

$$f_{\text{instruction}} ! 2^{R_0} 3^{R_1} 5^{R_2} \ldots$$

- $\text{proj}_{p_i}$ increments register $i$.
- $\text{zip}_{p_i}$ decrements register $i$.

Most importantly: $\text{zip}_{p_i}$ also acts as a dispatching device, sending states to different paths depending on the residue of the index modulo $p_i$, i.e. whether $R_i$ is zero or not.

We have branching!
Definability: Basic Idea

- Encode register state $R_0, R_1, R_2, \ldots \in \mathbb{N}$ as
  
  $$f_{\text{instruction}} = 2^{R_0}3^{R_1}5^{R_2} \ldots .$$

- proj$_{p_i}$ increments register $i$.
- zip$_{p_i}$ decrements register $i$.
- Most importantly: zip$_{p_i}$ also acts as a *dispatching device*, sending states to different paths depending on the residue of the index modulo $p_i$, i.e. whether $R_i$ is zero or not.
  - We have branching!
The central point of the previous construction, as well the construction of Endrullis et al., is the use of interleaving as the only available device for control flow handling.
The central point of the previous construction, as well the construction of Endrullis et al., is the use of interleaving as the only available device for control flow handling.

What if we removed interleaving from the language, i.e. restricted systems to only unary function definitions?
Definability: Basic Idea

The central point of the previous construction, as well the construction of Endrullis et al., is the use of interleaving as the only available device for control flow handling.

What if we removed interleaving from the language, i.e. restricted systems to only unary function definitions?

How do you encode machines without the possibility of branching?
Main Result: Unary Definability

What does the class of polymorphic stream functions definable by a unary system look like?

▶ It is still equal to the class of computable unary polymorphic stream functions!

Proof highly technical: multiple layers of emulation with crazy encodings

Corollary: Productivity is undecidable even for only unary stream equations.
Main Result: Unary Definability

What does the class of polymorphic stream functions definable by a unary system look like?

- It is *still equal* to the class of computable unary polymorphic stream functions!
Main Result: Unary Definability

What does the class of polymorphic stream functions definable by a unary system look like?

- It is *still equal* to the class of computable unary polymorphic stream functions!
- Proof highly technical: multiple layers of emulation with crazy encodings.
Main Result: Unary Definability

What does the class of polymorphic stream functions definable by a unary system look like?

▶ It is *still equal* to the class of computable unary polymorphic stream functions!

▶ Proof highly technical: multiple layers of emulation with crazy encodings

Corollary: Productivity is undecidable even for only unary stream equations.
Main Result: Sketch of a Sketch

1. Encode generalized Collatz functions.
Main Result: Sketch of a Sketch

1. Encode generalized Collatz functions.
2. Use these to simulate a single loop iteration of a While-program with a single top-level loop.
Main Result: Sketch of a Sketch

1. Encode generalized Collatz functions.
2. Use these to simulate a single loop iteration of a While-program with a single top-level loop.
3. Modify such While-programs to encode their result in the number of loop iterations until termination.
Main Result: Sketch of a Sketch

1. Encode generalized Collatz functions.
2. Use these to simulate a single loop iteration of a While-program with a single top-level loop.
3. Modify such While-programs to encode their result in the number of loop iterations until termination.
4. Loop the program and decode the result from the run time.
Main Result: Generalized Collatz Functions

Encoding of $g(n \cdot q + i) = a_i \cdot q + b_i$ (for $i = 0, \ldots, n - 1$):

\[\text{add}(s) = \text{head}(\text{tail}^{n \cdot a_0 + 0}(s)) :: \ldots :: \text{head}(\text{tail}^{n \cdot a_{n-1} + (n-1)}(s)) :: \text{add}(\text{tail}^n(s)) ,\]

\[\text{u}(s) = \text{head}(\text{tail}^{n \cdot b_0 + 0}(s)) :: \ldots :: \text{head}(\text{tail}^{n \cdot b_{n-1} + (n-1)}(s)) :: \text{u}(\text{add}(s)) ,\]

\[\text{div}(s) = \text{head}(s) :: \ldots :: \text{head}(s) :: \text{div}(\text{tail}(s)) ,\]

\[n \text{ times}\]

\[\nu(s) = \text{u}(\text{div}(s))\]

\[\bar{\nu} = g\]
Unary Definability: Minimum System Size

In total, we need to define eleven non-mutual unary equations, of which nine are essential.

It is natural to ask: what is the minimal number of equations required for unary definability?
Unary Definability: Minimum System Size

In total, we need to define eleven non-mutual unary equations, of which nine are essential.

It is natural to ask: what is the minimal number of equations required for unary definability?

We managed to find a construction using only four non-mutual unary equations (very ugly, even more technical).
Define \( a_{0,1} := 0 \), \( a_n := \lfloor \log_2(\log_2(n)) \rfloor + 1 \) for \( n \geq 2 \).

Let \( b : \mathbb{N} \rightarrow \mathbb{N} \) be the infinite concatenation, for \( n = 0, 1, 2, \ldots \), of

\[ a_n, a_n + 1, \ldots, n. \]

There is no unary system of size 3 with \( b \) as indexing function.
Unary Definability: Summary

- Four equations can define any computable stream function.
- Three equations do not suffice.

Productivity is still undecidable (Π₀²-complete) for only two equations. Surprisingly, productivity for a single equation is decidable! [Proofs mostly rather lengthy and technical]
Unary Definability: Summary

- Four equations can define any computable stream function.
- Three equations do not suffice.
- Productivity is still undecidable ($\Pi^0_2$-complete) for only two equations.
Unary Definability: Summary

- Four equations can define any computable stream function.
- Three equations do not suffice.
- Productivity is still undecidable ($\Pi^0_2$-complete) for only two equations.
- Surprisingly, productivity for a single equation is decidable!
Unary Definability: Summary

- Four equations can define any computable stream function.
- Three equations do not suffice.
- Productivity is still undecidable ($\Pi^0_2$-complete) for only two equations.
- Surprisingly, productivity for a single equation is decidable!

[Proofs mostly rather lengthy and technical]
Conclusions

- Delineated exact boundaries of decidability and productivity for (unary) polymorphic stream equations
- Found an elegant model of Turing-completeness in a severely restricted environment