Paper Review: Proving the Correctness of Reactive Systems Using Sized Types

Jifeng Wu

2023-03-27

Outline

- Background and Motivation👈
- Definitions and Notations
- Using the Type System

Background and Motivation

- What are Reactive Systems?👉
- What Problems Do Sized Types Intend to Solve?

What are Reactive Systems?

Reactive programming is a declarative programming paradigm concerned with *data streams* and the propagation of change.

In reactive programming, an attractive program structure is a set of concurrent processes known as *stream processors*. A stream processor is process that *consumes an input stream and produces an output stream*.

Wikipedia contributors. "Reactive programming." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 16 Mar. 2023. Web. 22 Mar. 2023.

What are Reactive Systems?

A class of embedded programs can be expressed as a reactive system where the control software must continuously react to the input streams by producing elements on the output streams. This is the core of more expressive formalisms that accommodate asynchronous events, non-determinism, etc.

Hallgren, T., Carlsson, M. (1995). Programming with Fudgets. In: Jeuring, J., Meijer, E. (eds) Advanced Functional Programming. AFP 1995. Lecture Notes in Computer Science, vol 925. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-59451-5_5

Background and Motivation

- What are Reactive Systems?
- What Problems Do Sized Types Intend to Solve?👈

Background and Motivation

Fundamental correctness property of an embedded functional program: the computation of each stream element terminates.

Define the datatype of streams of natural numbers 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 programs that use this datatype are:

- `head (Mk n s) = n`, of type `St -> Nat`
- `tail (Mk n s) = s`, of type `St -> Nat`

Background and Motivation

A more interesting program is:

```
letrec ones = Mk 1 ones in ones
```

which computes an infinite stream of 1's, and the program is *productive* (a request for the first i elements of the stream is guaranteed to be processed in finite time).

Background and Motivation

```python
import inspect
import sys


def mk(integer, stream, indent_level=0):
    indent = '    ' * indent_level
    print(indent, inspect.currentframe(), file=sys.stderr)

    yield integer
    yield from stream


def create_ones(indent_level=0):
    indent = '    ' * indent_level
    print(indent, inspect.currentframe(), file=sys.stderr)

    yield from mk(1, create_ones(indent_level + 1), indent_level + 1)
```

Background and Motivation

```
In [2]: ones = create_ones()

In [3]: next(ones)
 <frame at 0x7f6358731200, file '<ipython-input-1-bad85f883e32>', line 15, code create_ones>
     <frame at 0x7f6358737040, file '<ipython-input-1-bad85f883e32>', line 7, code mk>
Out[3]: 1

In [4]: next(ones)
     <frame at 0x2995df0, file '<ipython-input-1-bad85f883e32>', line 15, code create_ones>
         <frame at 0x298ee60, file '<ipython-input-1-bad85f883e32>', line 7, code mk>
Out[4]: 1

In [5]: next(ones)
         <frame at 0x7f635a03a5e0, file '<ipython-input-1-bad85f883e32>', line 15, code create_ones>
             <frame at 0x7f635a1e99f0, file '<ipython-input-1-bad85f883e32>', line 7, code mk>
Out[5]: 1
```

Background and Motivation

However,

```
letrec ones' = Mk 1 (tail ones') in ones'
```

is not productive; it cannot compute the first i elements of the stream for any i > 1.

Background and Motivation

```python
def tail(stream, indent_level=0):
    indent = '    ' * indent_level
    print(indent, inspect.currentframe(), file=sys.stderr)

    discarded = next(stream)

    print(indent, f'{discarded} discarded in {inspect.currentframe()}', file=sys.stderr)
    yield from stream


def create_ones_(indent_level=0):
    indent = '    ' * indent_level
    print(indent, inspect.currentframe(), file=sys.stderr)

    yield from mk(
        1,
        tail(
            create_ones_(
                indent_level + 1
            ),
            indent_level + 1
        ),
        indent_level + 1
    )
```

Background and Motivation

```
ones_ = create_ones_()
for i in ones_:
    print(f'Got {i} from ones_', file=sys.stderr)
```

Infinite unbounded recursion happens after 'Got 1 from ones_':

```
 <frame at 0x7f8abf1199f0, file 'ones_.py', line 32, code create_ones_>
     <frame at 0x7f8abf0283e0, file 'ones_.py', line 7, code mk>
Got 1 from ones_
     <frame at 0x7f8abf028210, file 'ones_.py', line 22, code tail>
     <frame at 0x7f8abf119040, file 'ones_.py', line 32, code create_ones_>
         <frame at 0x7f8abf028780, file 'ones_.py', line 7, code mk>
     1 discarded in <frame at 0x7f8abf028210, file 'ones_.py', line 26, code tail>
         <frame at 0x7f8abf0285b0, file 'ones_.py', line 22, code tail>
         <frame at 0x7f8abf170740, file 'ones_.py', line 32, code create_ones_>
             <frame at 0x7f8abf028b20, file 'ones_.py', line 7, code mk>
         1 discarded in <frame at 0x7f8abf0285b0, file 'ones_.py', line 26, code tail>
```

Background and Motivation

```
letrec ones' = Mk 1 (tail ones') in ones'
```

Assume that after unfolding the recursion a number of times, we obtain a stream s with `i + 1` elements.

The recursive call then computes `(tail s)` which has `i` elements and adds one element to produce a stream that has `i + 1` elements.

Each recursive call is attempting to construct a stream with no more elements than the previous call and no new elements are added to the stream after the first one.

Background and Motivation

Represent streams with at least i elements with the family of types St^i.

We can express more infromative types for `Mk`, `head`, and `tail`:

- `data St = Mk Nat St.` : $\forall i.\, Nat \to St^i \to St^{i+1}$
- `head (Mk n s) = n` : $\forall i.\, St^{i+1} \to Nat$
- `tail (Mk n s) = s` : $\forall i.\, St^{i+1} \to St^i$

Background and Motivation

```
letrec ones = Mk 1 ones in ones
```

- `ones` $: St^i$
- `Mk 1 ones` $: St^{i+1}$

```
letrec ones' = Mk 1 (tail ones') in ones'
```

- `ones'` $: St^{i+1}$
- `tail ones'` $: St^i$
- `Mk 1 (tail ones')` $: St^{i+1}$

Background and Motivation

Using this representation, we can also establish that some functions are unsafe as their computation requires unbounded space.

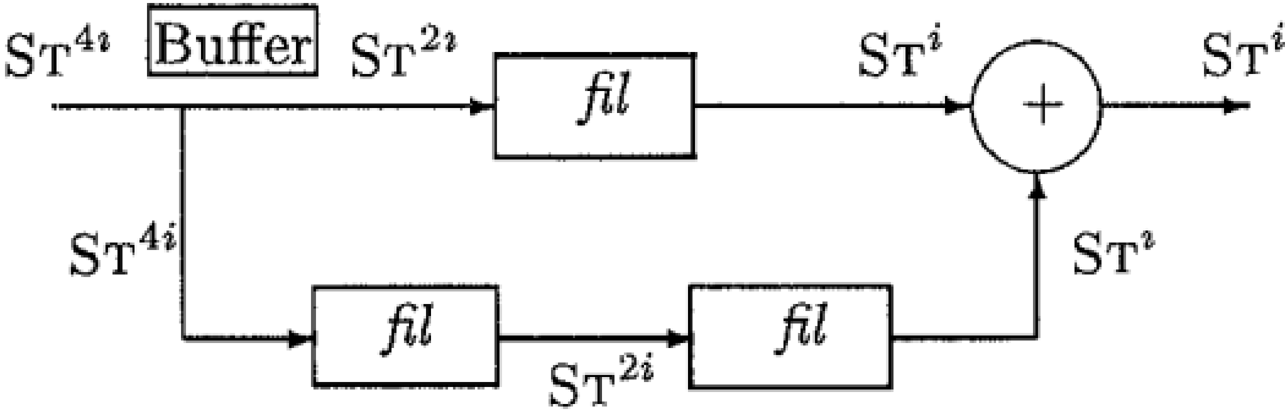
```
letrec fil (Mk n1 (Mk n2 s)) = Mk (avg n1 n2) (fil s) in λs.stream-add (fil s) (fil (fil s))
```

According to `fil (Mk n1 (Mk n2 s)) = Mk (avg n1 n2) (fil s)`, if a stream `s` contained the elements $n_1, n_2, n_3, n_4, \ldots$, `fil s` would return a stream containing the elements $\frac{n_1+n_2}{2}, \frac{n_3+n_4}{2}, \frac{n_5+n_6}{2}, \ldots$. Thus, the type of `fil` should be $\forall i.\, St^{2i} \to St^{i}$.

Background and Motivation

For every i output elements, we need $4i$ input elements. These elements are all consumed along the bottom path. However, only $2i$ elements are consumed along the top path, and the remaining elements must be buffered.

It is impossible
to implement a buffer of size $2i$
for all i.



Outline

- Background and Motivation
- Definitions and Notations👉
- Using the Type System

Definitions and Notations

- Datatypes👉
- Type System

Datatypes

- `data`: the user is interested in *finite elements* of the datatype
  - `data Nat = Zero + Succ Nat`
  - `data List t = Nil + Cons t (List t)`
- `codata`: the user is also interested in the *infinite elements* of the datatype
  - `codata St t = Mk t (St t)`

Datatypes

With each datatype name, our system associates a family of types.

- For `data`, it is represented with a *size index subscript* representing the elements of the datatype with the given size bound.
  - Nat_3 represents the natural numbers $\{0, 1, 2\}$
  - $List_3 t$ represents all lists with fewer than 3 elements of type t.
- For `codata`, it is represented with a *size index superscript*.
  - $St^3 t$ represents all streams with at least 3 elements of type t.
  - $St^\omega t$ represents all infinitie streams with elements of type t.

Datatypes

Valid size indexes include:

- natural numbers
- natural number *size variables*, e.g. i
- the special index ω
- *linear functions* of the size variables

The precise type for $factorial$ would be $\forall i.\, Nat_i \to Nat_{i!}$.
However, the best type we can express for $factorial$ is
$\forall i.\, Nat_i \to Nat_\omega$.

Datatypes

There are several sections in the paper which I couldn't understand:

- 3.2 Semantics of Expressions
- 3.3 The Universe of Types
- 3.4 Continuity and Ordinals
- 3.5 Semantics of Types
- 3.6 Testing for $\perp$
- 3.7 ω-Types

Definitions and Notations

- Datatypes
- Type System 👈

Definitions and Notations

The subtyping relation

Perks:

- `data` datatypes require $i \leq j$ for D_i to be a subtype of D_j.
- `codata` datatypes rquire $j \leq i$ for C_i to be a subtype of C_j.

$$\frac{}{\sigma \rhd \sigma} \qquad \frac{\sigma_1 \rhd \sigma_2 \qquad \sigma_2 \rhd \sigma_3}{\sigma_1 \rhd \sigma_3}$$

$$\frac{}{D_i\overline{\tau_k} \rhd D_j\overline{\tau_k}} \ \text{if } i \leq j \text{ or } j = \omega \qquad \frac{}{C_i\overline{\tau_k} \rhd C_j\overline{\tau_k}} \ \text{if } j \leq i \text{ or } i = \omega$$

$$\frac{\sigma_3 \rhd \sigma_1 \qquad \sigma_2 \rhd \sigma_4}{\sigma_1 \to \sigma_2 \rhd \sigma_3 \to \sigma_4}$$

$$\frac{\tau_i \rhd \tau_i' \ (i \text{ in covariant position}) \qquad \tau_j' \rhd \tau_j \ (j \text{ in contravariant position})}{D_k \ \tau_1 \ \ldots \ \tau_n \rhd D_k \ \tau_1' \ \ldots \ \tau_n'}$$

$$\frac{\tau_i \rhd \tau_i' \ (i \text{ in covariant position}) \qquad \tau_j' \rhd \tau_j \ (j \text{ in contravariant position})}{C^k \ \tau_1 \ \ldots \ \tau_n \rhd C^k \ \tau_1' \ \ldots \ \tau_n'}$$

Figure 2: The subtyping relation

Definitions and Notations

The main innovation in our type inference system is the rule for typing recursive declarations.

The second premise states that should `x` occur *within* `M` with type τ, it should have the "next" size ($\tau[i+1/i]$) *on the left hand side*.

- `data` computations terminate
- `codata` computations are productive

$$\frac{\begin{array}{l} \mathit{all}\ (\tau[0/i]) \\ \Gamma \vdash \lambda x.M : \forall i.\tau \rightarrow \tau[i+1/i] \\ \Gamma \cup \{x, \forall i.\tau\} \vdash N : \tau_1 \end{array}}{\Gamma \vdash (\mathbf{letrec}\ x = M\ \mathbf{in}\ N) : \tau_1}\quad i \notin FV(\Gamma)$$

Definitions and Notations

The first premise, known as the "bottom check", ensures that we can start the iteration of the functional in the first place.

- Set all natural number *size variables*, e.g. i, to 0, to get a type $\tau[0/i]$.
- If $\tau[0/i]$ is not empty, check whether *any* term is a valid term of $\tau[0/i]$.

$$\frac{\begin{array}{l} all\ (\tau[0/i]) \\ \Gamma \vdash \lambda x.M : \forall i.\tau \rightarrow \tau[i+1/i] \\ \Gamma \cup \{x, \forall i.\tau\} \vdash N : \tau_1 \end{array}}{\Gamma \vdash (\mathbf{letrec}\ x = M\ \mathbf{in}\ N) : \tau_1} \quad i \notin FV(\Gamma)$$

Definitions and Notations

The "bottom check" is very important.

```
s = if head s then Mk True s else Mk False s
```

Assuming that the right hand side `s` is of type St^{i+1} (required due to `head s`), the left hand side `s` is of type St^{i+2}.

- $St^{i+1} \to St^{i+2}$ satisfies the second premise. The functional is making progress at each recursive call.
- However, the "bottom check" fails.
  - $\tau[0/i] = St^1$.
  - St^1 is not empty.
  - $\perp$ (empty stream) is not a term of St^1.

Definitions and Notations

Rules for Type Inference (1)

- Var: variable
- Abs: procedure
- App: application
- Con: constructor
  - the type of the constructor σ_{Con} is inferred from the datatype declaration

$$\frac{}{\Gamma \cup \{x : \sigma\} \vdash x : \sigma}\ (\mathrm{Var}) \qquad \frac{\Gamma \cup \{x : \tau_1\} \vdash M : \tau_2}{\Gamma \vdash (\lambda x.M) : \tau_1 \to \tau_2}\ (\mathrm{Abs}) \qquad \frac{\Gamma \vdash M : \tau_1 \to \tau_2 \qquad \Gamma \vdash N : \tau_1}{\Gamma \vdash MN : \tau_2}\ (\mathrm{App})$$

$$\frac{}{\Gamma \vdash \mathrm{Con} : \sigma_{\mathsf{Con}}}\ (\mathrm{Con})$$

$$\frac{\Gamma \vdash M : \tau_1 \qquad \vdash^{\mathrm{Pat}} (P_i, \tau_1) : \Gamma_i \qquad \Gamma \cup \Gamma_i \vdash M_i : \tau_2 \qquad (i \in \{1..n\})}{\Gamma \vdash \textbf{case}\ M\ \textbf{of}\ (P_1 \to M_1) \ldots (P_n \to M_n) : \tau_2}\ (\mathrm{Case})$$

Definitions and Notations

Rules for Type Inference (2)

- Gen, Inst: Generalize and instantiate *type schemes*.
- GenS, InstS: Generalize and instantiate *size variables*.

$$\frac{\Gamma \vdash M : \forall t.\sigma}{\Gamma \vdash M : \sigma[\tau/t]}\ (\text{Inst}) \qquad \frac{\Gamma \vdash M : \sigma \qquad t \notin FV(\Gamma)}{\Gamma \vdash M : \forall t.\sigma}\ (\text{Gen})$$

$$\frac{\Gamma \vdash M : \forall i.\sigma}{\Gamma \vdash M : \sigma[S/i]}\ (\text{InstS}) \qquad \frac{\Gamma \vdash M : \sigma \qquad i \notin FV(\Gamma)}{\Gamma \vdash M : \forall i.\sigma}\ (\text{GenS})$$

Definitions and Notations

Rules for Type Inference (3)

- Coer: Coerce a type to a less precise one by relaxing the bounds on the sizes.
- ω: Instantiate size variables that index ω-undershooting type expressions to ω.
- Rec: Ensure that the resulting computations are terminating and/or productive when typing recursive declarations.

$$\frac{\Gamma \vdash M : \sigma_1 \qquad \sigma_1 \rhd \sigma_2}{\Gamma \vdash M : \sigma_2}\ (\text{Coer}) \qquad \frac{\Gamma \vdash M : \forall i.\pi_i}{\Gamma \vdash M : \pi_\omega}\ \pi_i \text{ is } \omega\text{-undershooting} \quad (\omega)$$

$$\frac{\mathit{all}\,(\tau[0/i]) \quad \Gamma \vdash \lambda x.M : \forall i.\tau \to \tau[i+1/i] \quad \Gamma \cup \{x : \forall i\overline{k}.\forall\overline{t}.\tau\} \vdash N : \tau_1}{\Gamma \vdash (\mathbf{letrec}\ x = M\ \mathbf{in}\ N) : \tau_1}\ \ i \notin FV(\Gamma), \quad \overline{k}, \overline{t} \subseteq FV(\tau) \backslash FV(\Gamma) \quad (\text{Rec})$$

Figure 3: Rules for Type Inference

Outline

- Background and Motivation
- Definitions and Notations
- Using the Type System 👈

Using the Type System

Assume that `append` has type
$\forall i. \forall j. \forall t. List_i t \to List_{j+1} t \to List_{i+j} t.$

We can prove that `reverse` has type $\forall i. \forall t. List_i t \to List_i t$:

```
reverse xs =
    case xs of
        Nil -> Nil
        Cons y ys -> append (reverse ys) (Cons y Nil)
```

Using the Type System

- `Nil -> Nil`: assume `xs` has type $List_i t$. In this case, `reverse xs` has type $List_i t$, and `reverse` has type $\forall i. \forall t. List_i t \rightarrow List_i t$. ✅
- `Cons y ys -> append (reverse ys) (Cons y Nil)` case, assume `reverse` has type $\forall i. \forall t. List_i t \rightarrow List_i t$ on the right hand side, `ys` has type $List_i t$.
  - `y` has type t, `xs` has type $List_{i+1} t$.
  - `reverse ys` has type $List_i t$, `Cons y Nil` has type $List_2 t$ (because it is a list with length 1).
  - `append (reverse ys) (Cons y Nil)` has type $List_{i+1} t$, `reverse` has type $\forall i. \forall t. List_{i+1} t \rightarrow List_{i+1} t$ on the left hand side. ✅
- Bottom check: $List_0 t \rightarrow List_0 t$. $List_0 t$ is empty, pass. ✅

Using the Type System

Automated Type Checker

- Requires all let-bound variables to be annotated with sized type signatures, but infers the types for all other expressions.
- Proof strategy:
  - Uses Var, Abs, App, Con, Case, Rec according to the structure of the program.
  - Inst, InstS with Var, Con.
  - Gen, GenS with Rec.

$$\frac{}{\Gamma \cup \{x : \sigma\} \vdash x : \sigma}\ (\mathrm{Var}) \qquad \frac{\Gamma \cup \{x : \tau_1\} \vdash M : \tau_2}{\Gamma \vdash (\lambda x.M) : \tau_1 \to \tau_2}\ (\mathrm{Abs}) \qquad \frac{\Gamma \vdash M : \tau_1 \to \tau_2 \qquad \Gamma \vdash N : \tau_1}{\Gamma \vdash MN : \tau_2}\ (\mathrm{App})$$

$$\frac{}{\Gamma \vdash \mathrm{Con} : \sigma_{\mathsf{Con}}}\ (\mathrm{Con})$$

$$\frac{\Gamma \vdash M : \tau_1 \qquad \vdash^{\mathrm{Pat}} (P_i, \tau_1) : \Gamma_i \qquad \Gamma \cup \Gamma_i \vdash M_i : \tau_2 \qquad (i \in \{1..n\})}{\Gamma \vdash \mathbf{case}\ M\ \mathbf{of}\ (P_1 \to M_1) \ldots (P_n \to M_n) : \tau_2}\ (\mathrm{Case})$$

$$\frac{\Gamma \vdash M : \forall t.\sigma}{\Gamma \vdash M : \sigma[\tau/t]}\ (\mathrm{Inst}) \qquad \frac{\Gamma \vdash M : \sigma \qquad t \notin FV(\Gamma)}{\Gamma \vdash M : \forall t.\sigma}\ (\mathrm{Gen})$$

$$\frac{\Gamma \vdash M : \forall i.\sigma}{\Gamma \vdash M : \sigma[S/i]}\ (\mathrm{InstS}) \qquad \frac{\Gamma \vdash M : \sigma \qquad i \notin FV(\Gamma)}{\Gamma \vdash M : \forall i.\sigma}\ (\mathrm{GenS})$$

$$\frac{\Gamma \vdash M : \sigma_1 \qquad \sigma_1 \rhd \sigma_2}{\Gamma \vdash M : \sigma_2}\ (\mathrm{Coer}) \qquad \frac{\Gamma \vdash M : \forall i.\pi_i}{\Gamma \vdash M : \pi_\omega}\ \pi_i \text{ is } \omega\text{-undershooting} \quad (\omega)$$

$$\frac{\mathit{all}\,(\tau[0/i]) \quad \Gamma \vdash \lambda x.M : \forall i.\tau \to \tau[i+1/i] \quad \Gamma \cup \{x : \forall i\overline{k}.\forall\overline{t}.\tau\} \vdash N : \tau_1}{\Gamma \vdash (\mathbf{letrec}\ x = M\ \mathbf{in}\ N) : \tau_1}\quad i \notin FV(\Gamma), \quad \overline{k}, \overline{t} \subseteq FV(\tau) \backslash FV(\Gamma) \quad (\mathrm{Rec})$$

Figure 3: Rules for Type Inference

Using the Type System

- Generates a system of inequalities based on
  - Size constraints whenever Coer is used.
  - All size variables are `Nat`'s (integers >=0).
- Solved using a constraint solver.

$$
\begin{array}{l}
\{0 <= k24, \quad 0 <= k10, \quad 0 <= k25, \quad 0 <= k4, \\
0 <= k6, \quad 0 <= k28, \quad 0 <= k, \quad 0 <= l, \\
k24 + 1 <= k10 + 1, \quad , \quad k25 + 1 <= k4 \\
k6 <= k24 + 1 + k25 \\
k10 + 1 <= k24 + 1, \quad k4 <= k25 + 1 \\
k24 + 1 + k25 <= k6, \quad k4 <= l + 1 \\
k10 <= k24, \quad k4 <= k25 + 1 \\
k24 + k25 <= k28, \quad k28 + 1 <= k6 \\
k4 <= k6, \quad k <= k24, \quad l + 1 <= k25 + 1 \\
k24 + k25 <= k + l, \quad k10 + 1 <= k + 1 \\
k + 1 + l <= k6, \quad k24 <= k, \quad k25 + 1 <= l + 1 \\
k + l <= k24 + k25, \quad k + 1 <= k10 + 1 \\
l + 1 <= k4, \quad k6 <= k + 1 + l\}
\end{array}
$$

Using the Type System

Three major steps of the algorithm:

- Hindley Milner Inference: check that the program is type correct in terms of ordinary types.
- Size Inference.
- Constraint Solving and Bottom Check.

Using the Type System

However, I don't quite understand the implementation details in 6.2. Specifically, whose sizes are these in the constraints?

$$\langle C, \emptyset, t', s \rangle \quad \text{where}$$

$$\begin{aligned} C = \{ & \textsc{List}_{k10+1}\alpha \to \textsc{List}_{k4}\alpha \to \textsc{List}_{k6}\alpha \leq \\ & \textsc{List}_{k24+1}\alpha' \to \textsc{List}_{k25+1}\alpha' \to \textsc{List}_{k24+1+k25}\alpha', \\ & \textsc{List}_{k24+1}\alpha' \to \textsc{List}_{k25+1}\alpha' \to \textsc{List}_{k24+1+k25}\alpha' \leq \\ & \textsc{List}_{k10+1}\alpha \to \textsc{List}_{k4}\alpha \to \textsc{List}_{k6}\alpha, \\ & \textsc{List}_{k10}\alpha \leq \textsc{List}_{k24}\alpha, \quad \textsc{List}_{k4}\alpha \leq \textsc{List}_{k25+1}\alpha, \\ & \alpha \leq \alpha, \quad \textsc{List}_{k24+k25}\alpha \leq \textsc{List}_{k28}\alpha, \\ & \textsc{List}_{k28+1}\alpha \leq \textsc{List}_{k6}\alpha, \quad \textsc{List}_{k4}\alpha \leq \textsc{List}_{k6}\alpha \} \end{aligned}$$

$$\begin{aligned} t' = {} & (\textsc{List}_{k24}\alpha' \to \textsc{List}_{k25+1}\alpha' \to \textsc{List}_{k24+k25}\alpha') \to \\ & \textsc{List}_{k10+1}\alpha \to \textsc{List}_{k4}\alpha \to \textsc{List}_{k6}\alpha \end{aligned}$$

$$\begin{aligned} s = \forall kl.\forall a.(& \textsc{List}_{k}a \to \textsc{List}_{l+1}a \to \textsc{List}_{k+l}a) \to \\ & \textsc{List}_{k+1}a \to \textsc{List}_{l+1}a \to \textsc{List}_{k+1+l}a \end{aligned}$$