Refining Infinite Stream Behaviours By Bound Functions

Walter Dosch

Institut für Softwaretechnik und Programmiersprachen
Medizinische Universität zu Lübeck
Seelandstraße 1a, D–23569 Lübeck

phone ++49-451-3909-501
fax ++49-451-3909-503
e-mail dosch@isp.mu-luebeck.de
URL http://www.isp.mu-luebeck.de

Abstract. Stream processing functions form an abstract model of distributed systems where the components concurrently process their input streams and asynchronously communicate their results. The finite resp. infinite behaviour of a deterministic component is described by a function on finite resp. on infinite input histories. We present formal methods for systematically refining infinite behaviours by bound functions to finite behaviours. We illustrate the approach by series of case studies, viz. a shift register, a synchronization component and an iteration component.

1 Introduction

The design of asynchronous distributed systems is based on the behaviour of its components. A component is the physical or logical encapsulation of a device providing a specified service. A component consumes a stream of incoming messages and produces a stream of outgoing messages [4, 7]. The extensional behaviour of a deterministic component is described by a function mapping input histories to output histories [12, 13] . This model of stream processing functions underlies our approach to the description, refinement, and implementation of software and hardware components.

Communication histories can be finite or infinite. Finite histories form a partial order under the prefix relation. Progression of time effects a prolongation of the history. Infinite streams model the input and output of continuously processing components. Such an infinite stream is completely determined by the set of its finite prefixes. The finite behaviour of a component is described by a monotonic function on finite histories. By a standard theorem, it uniquely extends to a continuous function on infinite histories. This lays the basis for a design method where the finite behaviour of a component is specified first, then the infinite behaviour is inferred from it.

Often it is more convenient to specify the infinite behaviour of a component in the first step. Then the infinite behaviour must be refined to a finite

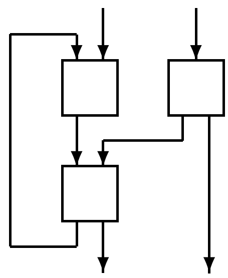

Fig. 1. Network of computing agents

behaviour which supports an algorithmic description and enables further refinement steps. There is a wide choice of finite behaviours that approximate the infinite behaviour specified. A naïve approach would literally transfer the axioms characterizing the function on infinite stream to finite streams. The resulting specifications may, however, be inconsistent [8] or fail to meet the intended finite behaviour.

In this paper we present a systematic method for refining infinite stream behaviours to finite stream behaviours. The essential design decision consists in imposing suitable bound functions that establish the monotonicity of the finite approximation. An important class of bound functions refers to the number of elements processed from the input stream or sent to the output stream. We investigate the algebraic properties of bound functions that work at the front of the stream. The results form the basis for refining infinite behaviours to finite behaviours in a sound way.

We adopt a transformational approach to system design [1, 10] . In a design step, the refined description is not invented and verified backwards; rather the refined description is formally deduced following sound calculation rules. We refrain from giving a general treatment of bound functions for stream processing functions which would end up in a classification of recursion schemes. Rather we explicate widely applicable design methods by presenting a series of characteristic case studies.

The paper is organized as follows. In Section 2 we shortly survey basic notions about streams and stream processing functions. Section 3 presents the algebraic properties of bound functions that take a fixed number of elements from the front of the stream. In Section 4, we refine the infinite behaviour of a shift register. Section 5 presents the derivation of a synchronization component. In Section 6 we treat an iterator which repeatedly applies a function to the stream of serially transmitted arguments. The conclusion surveys the approach and outlines future research.

2 Streams and Stream Processing Functions

In this section, we shortly survey the basic notions about streams and stream processing functions [16] to render the paper self-contained. The expert reader should immediately proceed to Section 3 .

2.1 Streams

Streams model the temporal succession of data in a network of loosely coupled computing agents. Given an alphabet $\mathcal{A}$, the equation

$$\mathcal{A}^\star = \{\langle\rangle\} \ \cup \ \mathcal{A} \times \mathcal{A}^\star \tag{1}$$

defines the set $\mathcal{A}^\star$ of *finite streams* $A = \langle a_1, a_2, \ldots, a_k \rangle$ of length $k \geq 0$ with elements $a_i \in \mathcal{A}$ $(i \in [1, k])$. Throughout the paper, streams are denoted by capital letters, their elements by small letters.

Finite streams are generated from the *empty stream* $\langle\rangle$ by repeatedly attaching an element to the front of the stream: $\langle a_1, a_2, \ldots, a_k \rangle = a_1 \lhd a_2 \cdots \lhd a_k \lhd \langle\rangle$. The *concatenation* of two finite streams $A = \langle a_1, a_2, \ldots, a_k \rangle$ and $B = \langle b_1, b_2, \ldots, b_l \rangle$ yields the stream $A \,\&\, B = \langle a_1, a_2, \ldots, a_k, b_1, b_2, \ldots, b_l \rangle$.

The set $\mathcal{A}^\star$ of finite streams over $\mathcal{A}$ forms a *partial order* under the *prefix relation*. Here a stream A *approximates* a stream B, denoted by $A \sqsubseteq B$, iff $A \,\&\, R = B$ holds for some stream $R \in \mathcal{A}^\star$. The prefix relation models operational progress in time: the shorter stream forms an initial part of the communication history. The empty stream is the least element which can be extended to every history.

Continuously working components process infinite streams. The partial order $(\mathcal{A}^\star, \sqsubseteq)$ of finite streams has no limit points to denote infinite histories. Therefore we construct a complete partial order by *ideal completion*. An *ideal* is a nonempty downward closed chain in $\mathcal{A}^\star$. The *domain* $\mathcal{A}^\infty$ *of streams* [17] consists of all ideals in $\mathcal{A}^\star$ ordered by set inclusion. It can be partitioned into the set $\mathcal{A}^\star$ of *finite streams* and the set $\mathcal{A}^\omega$ of *infinite streams*:

$$\mathcal{A}^\infty = \mathcal{A}^\star \cup \mathcal{A}^\omega \tag{2}$$

Infinite streams are uniquely determined by the ideal of their finite prefixes. Thus they can conveniently be handled through their finite approximations [14, 15] .

2.2 Stream Processing Functions

A *stream processing function* $F : \mathcal{A}^\infty \to \mathcal{B}^\infty$ maps an input stream to an output stream. It models a deterministic component with one input and one output channel. The types $\mathcal{A}$ and $\mathcal{B}$ determine the syntactic interface of the component. The generalization to stream processing functions with several arguments and results is straightforward. We will not enter the area of nondeterminism [3, 11] in this paper.

In the sequel, we concentrate on monotonic functions where further input leads to further output. A function $F : \mathcal{A}^\infty \to \mathcal{B}^\infty$ is called (*prefix*) *monotonic*, if $A \sqsubseteq B$ implies $F(A) \sqsubseteq F(B)$ for all $A, B \in \mathcal{A}^\infty$. A monotonic function $F : \mathcal{A}^\infty \to \mathcal{B}^\infty$ is called *continuous*, if $F(\sqcup K) = \sqcup F(K)$ holds for all chains K in $\mathcal{A}^\infty$ where $\sqcup$ denotes the least upper bound.

The input/output function $F : \mathcal{A}^\infty \to \mathcal{B}^\infty$ describes the *behaviour* of the component. Its restriction to finite streams is called the *finite behaviour* $F^\star : \mathcal{A}^\star \to \mathcal{B}^\infty$, its restriction to infinite streams the *infinite behaviour* $F^\omega : \mathcal{A}^\omega \to \mathcal{B}^\infty$.

Every monotonic function $f : \mathcal{A}^\star \to \mathcal{B}^\infty$ induces a stream processing function $\widehat{f} : \mathcal{A}^\infty \to \mathcal{B}^\infty$ where $\widehat{f}(I)$ is the least ideal comprising $f(I)$. This establishes a sound method for inferring infinite behaviours from finite behaviours.

In the paper, we study the opposite design direction. We explore methods how to refine a given infinite behaviour $F^\omega : \mathcal{A}^\omega \to \mathcal{B}^\infty$ to a finite behaviour $F^\star : \mathcal{A}^\star \to \mathcal{B}^\infty$ such that $F^\star \cup F^\omega = \widehat{F^\star}$.

3 Bound Functions

In this section we explore the algebraic properties of bound functions that take a fixed number of elements from the front of the stream.

The functional $take : \mathbb{N} \to [\mathcal{D}^\star \to \mathcal{D}^\star]$ delivers a prefix of the stream consisting of a specified number of elements provided they exist. The functional is defined by structural induction both on natural numbers and streams:

$$take(0)(X) = \langle\rangle \tag{3}$$

$$take(n)(\langle\rangle) = \langle\rangle \tag{4}$$

$$take(n+1)(x \lhd X) = x \lhd take(n)(X) \tag{5}$$

The *bound functions* $take(n)$ $(n \in \mathbb{N})$ smoothly cooperate with concatenation:

$$take(n)(X \,\&\, Y) = \begin{cases} take(n)(X) & \text{if } n \leq |X| \\ X \,\&\, take(n - |X|)(Y) & \text{if } n \geq |X| \end{cases} \tag{6}$$

Therefore all bound functions $take(n)$ $(n \in \mathbb{N})$ are prefix monotonic

$$take(n)(X) \sqsubseteq take(n)(X \,\&\, Y) \tag{7}$$

and possess a unique continuous extension $Take(n) : \mathcal{D}^\infty \to \mathcal{D}^\infty$ to infinite streams.

For every $n \in \mathbb{N}$, the bound function $Take(n)$ yields a prefix of the input stream:

$$Take(n)(X) = X \;\text{ iff }\; X \in \mathcal{D}^\star \text{ and } n \geq |X| \tag{8}$$

$$Take(n)(X) \sqsubset X \;\text{ iff }\; (X \in \mathcal{D}^\star \text{ and } n < |X|) \text{ or } X \in \mathcal{D}^\infty \tag{9}$$

So every bound function $Take(n)$ $(n \in \mathbb{N})$ approximates the identity function:

$$Take(n) \sqsubset id_{\mathcal{D}^\infty \to \mathcal{D}^\infty} \,. \tag{10}$$

Bound functions cutting initial segments of different length are related in the prefix order:

$$Take(m) \sqsubseteq Take(m+n) \tag{11}$$

The family $(Take(n))_{n \in \mathbb{N}}$ forms an ascending chain in $[\mathcal{D}^\infty \to \mathcal{D}^\infty]$ with least upper bound

$$\bigsqcup_{n \in \mathbb{N}} Take(n) = id_{\mathcal{D}^\infty \to \mathcal{D}^\infty} \,. \tag{12}$$

This is the key concept for refining infinite stream behaviours in a calculational way. The algebra of programs is studied in [2], fold/unfold transformations were introduced in [6] .

In the next sections we present three applications for the calculational refinement of infinite stream behaviours, viz. a shift register, a synchronization agent and an iterator.

4 Shift Register

A shift register of fixed length $k \geq 1$ stores k data elements in a first-in first-out manner, compare Fig. 2 . The infinite behaviour $Shift : \mathcal{D}^k \to [\mathcal{D}^\omega \to \mathcal{D}^\infty]$ of the

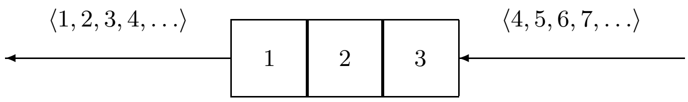

Fig. 2. Input/output function of a 3-place shift register

shift register is given by

$$Shift(Q)(X) = Q \,\&\, X \,. \tag{13}$$

For a state $Q \in \mathcal{D}^k$ and an infinite input history $X \in \mathcal{D}^\omega$, the shift register will first output the internal queue Q and then reproduce the infinite input stream X . The external behaviour summarizes the data flow through the interfaces of the component. It abstracts from the state transitions of the shift register for the single elements; this provides freedom for different implementations.

The infinite behaviour of the component does not coincide with its finite behaviour. From a finite input stream, the shift register retains up to k final elements; so it will not reproduce a finite input.

In a first design step, the specification is refined to a reasonable finite behaviour that guarantees the defined infinite behaviour. To this end, we impose a bound function on the length of the output stream. So we define the finite behaviour $shift : \mathcal{D}^k \to [\mathcal{D}^\star \to \mathcal{D}^\infty]$ by

$$shift(Q)(X) = take(|X|)(Q \,\&\, X) \,. \tag{14}$$

By the limit properties of the bound functions, the finite behaviour refines the infinite behaviour. After this refinement step, the recursive algorithm is easily derived by structural induction on the input stream, compare Appendix A :

$$shift(q \lhd Q)(\langle\rangle) = \langle\rangle \tag{15}$$

$$shift(q \lhd Q)(x \lhd X) = q \lhd shift(Q \rhd x)(X) \tag{16}$$

Here the operation $\rhd$ attaches an element to the rear of a queue.

5 Synchronisation

A synchronisation component has two input channels and two output channels of the same types, compare Fig. 3 . It synchronizes the communication streams

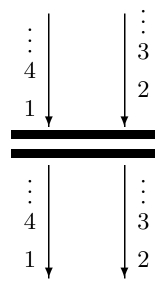

Fig. 3. Input/output function of a synchronisation component

arriving asynchronously on the two input ports in the following way: data on one input line may pass the synchronisation barrier iff there arrives data on the other input line as well.

The infinite behaviour is described by a function $Syn : \mathcal{D}^\omega \times \mathcal{E}^\omega \to \mathcal{D}^\infty \times \mathcal{E}^\infty$ with

$$Syn(X, Y) = (X, Y)\,. \tag{17}$$

The synchronisation component reproduces each two infinite streams, since every data arriving on one input line will eventually pass the barrier. Note again that histories abstract from transitions and record only the data flow [18] .

The specification (17) of the synchronisation component is violated by finite input histories. If both input streams are finite, then the final segment of the longer input stream will not show up as output.

The finite behaviour is described by a function $syn : \mathcal{D}^\star \times \mathcal{E}^\star \to \mathcal{D}^\infty \times \mathcal{E}^\infty$ having the same interface types. We define the finite behaviour imposing a balance on the length of input and output:

$$syn(X, Y) = (take(|Y|)(X), take(|X|)(Y)) \tag{18}$$

For sake of brevity, we only treat the uniform case where both input streams are finite. After this refinement step, we easily derive an algorithm by induction on both input streams, compare Appendix B:

$$syn(\langle\rangle, Y) = (\langle\rangle, \langle\rangle) \tag{19}$$

$$syn(X, \langle\rangle) = (\langle\rangle, \langle\rangle) \tag{20}$$

$$syn(x \triangleleft X, y \triangleleft Y) = (x, y) \lll syn(X, Y) \tag{21}$$

Above the operator $(a, b) \lll (A, B) = (a \triangleleft A, b \triangleleft B)$ attaches a pair of elements to a pair of streams. The synchronisation function strict in both arguments. The algorithm precisely describes the transition behaviour of the component. If there are elements arriving on both input ports, they may pass. If data arrives on one input port only, it will have to wait for a possibly never arriving data on the other input port.

6 Iterator

As the final application, we consider an iterator component with one input and one output port, compare Fig. 4 . The component repeatedly applies an n-ary

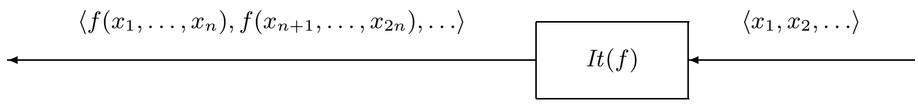

Fig. 4. Input/output function of an iterator component with serial input

function $(n \geq 1)$ to the arguments arriving serially in the input stream. The infinite behaviour is uniquely defined by the functional $It : [\mathcal{A}^n \to \mathcal{B}] \to [\mathcal{A}^\omega \to \mathcal{B}^\infty]$ with

$$It(f)(x_1 \triangleleft \cdots \triangleleft x_n \triangleleft X) = f(x_1, \ldots, x_n) \triangleleft It(f)(X)\,. \tag{22}$$

The finite behaviour must also cope with partial input. For an arbitrary infinite stream $Y \in \mathcal{D}^\omega$ we specify the finite behaviour $it : [\mathcal{A}^n \to \mathcal{B}] \to [\mathcal{A}^\star \to \mathcal{B}^\infty]$ by

$$it(f)(X) = Take(|X| \div n)(It(f)(X \,\&\, Y))\,, \tag{23}$$

where $\div$ denotes integer division. The function $it(f)$ is well defined, since it does not depend on the particular prolongation Y . After a short derivation, compare Appendix C , we obtain for all $i < n$:

$$it(f)(\langle x_1, \ldots, x_i\rangle) = \langle\rangle \tag{24}$$

$$it(f)(x_1 \triangleleft \cdots \triangleleft x_n \triangleleft X) = f(x_1, \ldots, x_n) \triangleleft it(f)(X)\,. \tag{25}$$

The component accumulates exactly n inputs. For the output, we retrieved the recursion equation (22) .

7 Conclusion

We investigated formal methods for refining infinite stream behaviours to finite stream behaviours. The calculational derivation consists of many simple mechanical steps which are well supported by a program transformation system. Yet, the crucial design decision consists in choosing suitable bound functions. The bound function determines the transition behaviour of the component, in particular its reaction on missing input.

The finite and the infinite behaviour of components allow complementary insights. Both formalisms shows advantages and disadvantages with respect to compositionality, abstractness, verification, synthesis, and tool support. Currently we investigate a complementary approach where the rear of the output stream is truncated. In general, bound functions are well suited for designing components with uniform input/output behaviour. For asynchronous component we elaborate an order-theoretic approach. Here the finite behaviour is the maximal common prefix of all infinite behaviours resulting from possible extensions of the finite input stream.

Finally we note that a design involves different refinement relations. Future studies will investigate how the refinement relation introduced in this paper cooperates with other notions of refinement, e.g. interface refinement [5] , communication refinement or state refinement [9] .

Acknowledgements I thank Kathrin Schröder for excellent typing and Annette Stümpel for carefully reading drafts of the manuscript.

References

1. F.L. Bauer, B. Möller, H. Partsch, P. Pepper: *Formal Program Construction by Transformations — Computer–Aided, Intuition Guided Programming.* IEEE Transactions on Software Engineering *15*:2, 165–180 (1989)
2. R.S. Bird, O. de Moor: *The Algebra of Programming.* Hemel Hempstead: Prentice Hall 1997
3. J.D. Brock, W.B. Ackerman: *Scenarios: A Model of Non-Determinate Computation.* In: J. Diaz, I. Ramos: Formalization of Programming Concepts. Lecture Notes in Computer Science *107*. Berlin: Springer 1981, 252–259
4. M. Broy: *(Inter-)Action Refinement: The Easy Way.* In: M. Broy (Hrsg.): Program Design Calculi. Springer NATO ASI Series F: Computer and System Sciences *118*. Berlin: Springer 1993, 121–158
5. M. Broy, K. Stølen: *Specification and Refinement of Finite Data Flow Networks — A Relational Approach.* In: H. Langmaack, W.P. de Roever, J. Vytopil (eds.): Formal Techniques in Real-Time and Fault-Tolerant Computing. Lecture Notes in Computer Science *863*. Berlin: Springer 1994
6. R. Burstall, J. Darlington: *A Transformation System for Developping Recursive Programs.* Journal of the ACM *24*, 44–67 (1977)
7. W. Dosch: *Semantic Design Decisions for Data Flow Languages.* Education in Computing — Computers in Education *4:1*, 177–197 (1992)

8. W. Dosch, A. Stümpel: *Transformational Design of a Merge Agent For Ordered Streams.* In: R. Y. Lee (ed.): Proceedings of the 17th Annual International Conference on Computer Science, Vol. B. San Diego, August 6–8, 1999. Virginia Beach: Maximilian Press Publishers 1999, 171–176
9. W. Dosch: *Transforming Stream Processing Functions into State Based Components.* In: H. Fouchal, R. Lee (eds.): International Conference on Software Engineering Applied to Networking, Parallel and Distributed Computing (SNPD'00). Reims, May 18–21, 2000. Mt. Pleasant, MI: International Association for Computer and Information Science 2000, 27–34
10. M. Feather: *A Survey and Classification of Some Program Transformation Approaches and Techniques.* In: L.G.L.T. Meertens (ed.): Proceedings IFIP TC2 Working Conference on Program Specification and Transformation. Amsterdam: North Holland 1987, 165–195
11. B. Jonsson: *A Fully Abstract Trace Model For Data Flow and Asynchronous Networks.* Distributed Computing *7*, 197–212 (1994)
12. G. Kahn: *The Semantics of a Simple Language for Parallel Programming.* In: J.L. Rosenfeld (ed.): Information Processing 1974. Amsterdam: North-Holland 1974, 471–475
13. G. Kahn, D.B. MacQueen: *Coroutines and Networks of Parallel Processes.* In: B. Gilchrist (ed.): Information Processing 1977. Amsterdam: North-Holland 1977, 993–998
14. B. Möller: *Ideal Streams.* In: E.-R. Olderog (ed.): Programming Concepts, Methods and Calculi. IFIP Transactions A-56. Amsterdam: Elsevier/North-Holland 1994, 39–58
15. B. Möller: *Ideal Stream Algebra..* In: B. Möller, J.V. Tucker (eds.): Prospects For Hardware Foundations. Lecture Notes in Computer Science *1546*. Berlin: Springer 1998, 69–116
16. R. Stephens: *A Survey of Stream Processing.* Acta Informatica *34*, 491–541 (1997)
17. V. Stoltenberg-Hansen, I. Lindström, E.R. Griffor: *Mathematical Theory of Domains.* Cambridge Tracts on Theoretical Computer Science *22*. Cambridge: Cambridge University Press 1994
18. W.W. Wadge, E.A. Ashcroft: *Lucid — The Dataflow Programming Language.* APIC Studies in Data Processing *22*. New York: Academic Press 1985

Appendix A

Induction basis

$$
\begin{aligned}
& \mathit{shift}(Q)(\langle\rangle) \\
& \quad \{\text{unfold } \mathit{shift}\ (14)\} \\
= {} & \mathit{take}(|\langle\rangle|)(Q \,\&\, \langle\rangle) \\
& \quad \{\text{simplify}\} \\
= {} & \mathit{take}(0)(Q) \\
& \quad \{\text{unfold } \mathit{take}\ (3)\} \\
= {} & \langle\rangle
\end{aligned}
$$

Induction step

$$
\begin{aligned}
& \mathit{shift}(q \lhd Q)(x \lhd X) \\
& \quad \{\text{unfold } \mathit{shift}\ (14)\} \\
= {} & \mathit{take}(|x \lhd X|)((q \lhd Q) \,\&\, (x \lhd X)) \\
& \quad \{\text{simplify and rearrange}\} \\
= {} & \mathit{take}(1 + |X|)(q \lhd ((Q \rhd x) \,\&\, X)) \\
& \quad \{\text{unfold } \mathit{take}\ (5)\} \\
= {} & q \lhd \mathit{take}(|X|)((Q \rhd x) \,\&\, X) \\
& \quad \{\text{fold } \mathit{shift}\ (14) \text{ observing } |q \lhd Q| = |Q \rhd x|\} \\
= {} & q \lhd \mathit{shift}(Q \rhd x)(X)
\end{aligned}
$$

Appendix B

Induction bases

$$\begin{aligned}
& syn(\langle\rangle, Y) \\
& \quad \{\text{unfold } syn \ (18)\} \\
= \ & (take(|Y|)(\langle\rangle), take(|\langle\rangle|)(Y)) \\
& \quad \{\text{unfold } take \text{ twice (4) and (3)}\} \\
= \ & (\langle\rangle, \langle\rangle)
\end{aligned}$$

$$\begin{aligned}
& syn(X, \langle\rangle) \\
& \quad \{\text{unfold } syn \ (18)\} \\
= \ & (take(|\langle\rangle|)(X), take(|X|)(\langle\rangle)) \\
& \quad \{\text{unfold } take \text{ twice (3) and (4)}\} \\
= \ & (\langle\rangle, \langle\rangle)
\end{aligned}$$

Induction step

$$\begin{aligned}
& syn(x \triangleleft X, y \triangleleft Y) \\
& \quad \{\text{unfold } syn \ (18)\} \\
= \ & (take(|y \triangleleft Y|)(x \triangleleft X), take(|x \triangleleft X|)(y \triangleleft Y)) \\
& \quad \{\text{simplify length}\} \\
= \ & (take(1 + |Y|)(x \triangleleft X), take(1 + |X|)(y \triangleleft Y)) \\
& \quad \{\text{unfold } take \text{ (5) twice}\} \\
= \ & (x \triangleleft take(|Y|)(X), y \triangleleft take(|X|)(Y)) \\
& \quad \{\text{introduce attach operator } \blacktriangleleft \text{ for pairs}\} \\
= \ & (x, y) \lll (take(|Y|)(X), take(|X|)(Y)) \\
& \quad \{\text{fold } syn \ (18)\} \\
= \ & (x, y) \lll syn(X, Y)
\end{aligned}$$

Appendix C

Induction bases. Assume $i < n$.

$$\begin{aligned}
& it(f)(\langle x_1, \ldots, x_i\rangle) \\
& \quad \{\text{unfold } it \ (23)\} \\
= \ & Take(|\langle x_1, \ldots, x_i\rangle| \div n)(It(f)(\langle x_1, \ldots, x_i\rangle \,\&\, Y)) \\
& \quad \{\text{simplify } i \div n = 0\} \\
= \ & Take(0)(It(f)(\langle x_1, \ldots, x_i\rangle \,\&\, Y)) \\
& \quad \{\text{unfold } Take\} \\
= \ & \langle\rangle
\end{aligned}$$

Induction step

$$\begin{aligned}
& it(f)(x_1 \triangleleft \cdots \triangleleft x_n \triangleleft X) \\
& \quad \{\text{unfold } it \ (23)\} \\
= \ & Take(|x_1 \triangleleft \cdots \triangleleft x_n \triangleleft X| \div n)(It(f)(x_1 \triangleleft \cdots \triangleleft x_n \triangleleft X \,\&\, Y)) \\
& \quad \{\text{simplify } (n + |X|) \div n = 1 + |X| \div n\} \\
= \ & Take(1 + |X| \div n)(It(f)(x_1 \triangleleft \cdots \triangleleft x_n \triangleleft X \,\&\, Y)) \\
& \quad \{\text{unfold } It \ (22)\} \\
= \ & Take(1 + |X| \div n)(f(x_1, \ldots, x_n) \triangleleft It(f)(X \,\&\, Y)) \\
& \quad \{\text{unfold } Take\} \\
= \ & f(x_1, \ldots, x_n) \triangleleft Take(|X| \div n)(It(f)(X \,\&\, Y)) \\
& \quad \{\text{fold } it \ (23)\} \\
= \ & f(x_1, \ldots, x_n) \triangleleft it(f)(X) \,.
\end{aligned}$$