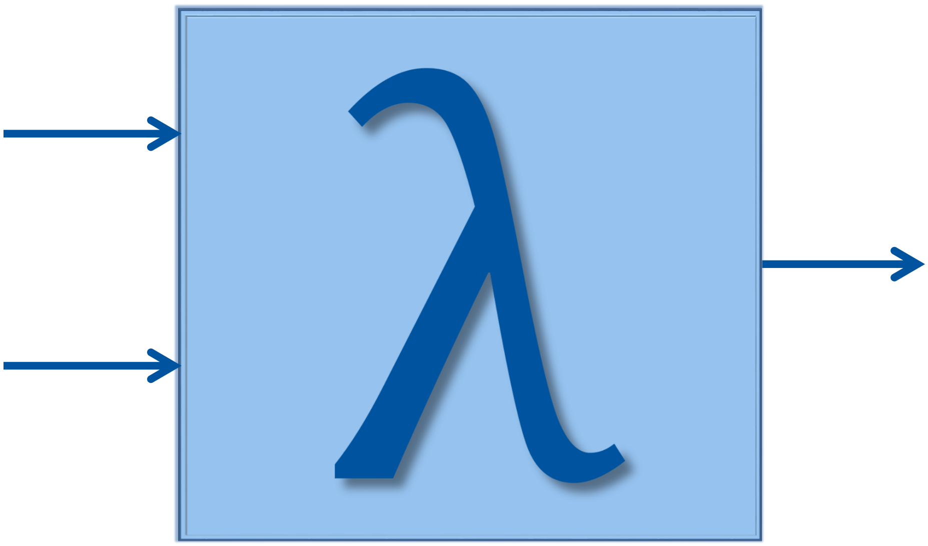
RWTH Aachen University
Software Engineering Group

# Towards an Isabelle Theory for distributed, interactive systems - the untimed case

## Technical Report




**Jens Christoph Bürger**
**Hendrik Kausch**
**Deni Raco**
**Jan Oliver Ringert**
**Bernhard Rumpe**
**Sebastian Stüber**
**Marc Wiartalla**

Aachen, January 19, 2020

# Abstract

This report describes a specification and verification framework for distributed interactive systems. The framework encodes the untimed part of the formal methodology FOCUS [BS01] in the proof assistant Isabelle [Pau90] using domain-theoretical concepts. The key concept of FOCUS, the stream data type, together with the corresponding prefix-order, is formalized as a pointed complete partial order. Furthermore, a high-level API is provided to hide the explicit usage of domain theoretical concepts by the user in typical proofs. Realizability constraints for modeling component networks with potential feedback loops are implemented. Moreover, a set of commonly used functions on streams are defined as least fixed points of the corresponding functionals and are proven to be prefix-continuous.

As a second key concept the stream processing function (SPF) is introduced describing a statefull, deterministic behavior of a message-passing component. The denotational semantics of components in this work is a defined set of stream processing functions, each of which maps input streams to output streams.

Furthermore, an extension of the framework is presented by using an isomorphic transformation of tuples of streams to model component interfaces and allowing composition. The structures for modeling component networks are implemented by giving names to channels and defining composition operators. This is motivated by the advantage that a modular modeling of component networks offers, based on the correctness of components of the decomposed system and using proper composition operators, the correctness of the whole system is automatically derived by construction.

To facilitate automated reasoning, a set of theorems is proven covering the main properties of these structures. Moreover, essential proof methods such as stream-induction are introduced and support these by further theorems. These examples demonstrate the principle usability of the modeling concepts of FOCUS and the realized verification framework for distributed systems with security and safety issues such as cars, airplanes, etc. Finally, a running example extracted from a controller in a car is realized to demonstrate and validate the framework.

# Contents

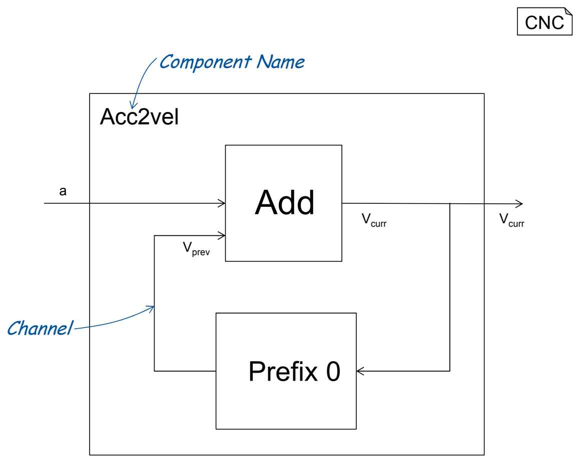
# Chapter 1

# Introduction

Distributed systems can be described as a physically or logically distributed collection of *components* which may only communicate by exchanging messages over communication channels, i.e, components do not share a global memory and their direct communication might be limited by the absence of communication channels. Examples of distributed systems can be found in telecommunication networks, cloud applications, control devices in cars, high-performance computing etc.

The design of distributed systems [BS01; Cou+12; Lee16] has proven to be much more error prone than that of sequential software. For this reason, formal methods, like CSP [Hoa78; Hei+15], CCS [Mil89], Petri Nets [Pet66; Rei12], or the $\pi$-calculus [Mil99], are often used for precise system specification and verification. The presence of a formal specification has proven to lead to better implementations, as potential sources of error are detected earlier [Hal90; Mao+17]. The method for system specification we use here is called FOCUS [Rum96; BS01] and is based heavily on the data flow paradigm. Some key works that influenced FOCUS are Petri Nets [Pet66], and Kahn networks [Kah74]. Other methodologies for modeling distributing systems usually differ from this approach by depending on a global state and shared memory.

The core concept of FOCUS is the *stream*. A stream is a potentially infinite message sequence from an alphabet and models a communication channel history starting at a




| Channel | Stream value |
|---|---|
| $a$ | 1 1 1 0 0 1 2 ... |
| $v_{curr}$ | 1 2 3 3 3 4 6 ... |
| $v_{pref}$ | 0 1 2 3 3 3 4 6 ... |

Figure 1.1: Running Example: Cruise Control

certain point in time until infinity. This communication history is also often called an observation. To make an analogy, one can picture a person sitting by a channel without a watch and writing down the messages that pass by.

We consider a component in a distributed system as a unit of computation that processes messages interactively. We will use the component network shown in Figure 1.1 as a running example of a component network throughout this document. It is extracted from a cruise control system and demonstrates the stepwise increment of the velocity depending on the acceleration. One component initializes the sequence by a 0. The other component performs the addition and has a well-defined interface: input channels $\mathrm{a}$ and $v_{prev}$ and output channel $v_{cur}$ denoted by arrows in the figure below. We verify the correct behavior in Appendix F.

Components can also be nondeterministic which means that they might have multiple behaviors for the same input. Furthermore, they might also have multiple input and output channels. We will define the semantics [HR04] of such a nondeterministic component using sets of functions following [Rum96]. Alternative semantic definitions can also be found in [BR07; BS01; RR11; Rin14].

Because streams model history, not every function $f$ that maps streams to streams actually models a real-life interactive component. However, the subset of these functions $f$ is characterised by two properties that exactly resemble the possible behavior of real-life interactive components. Both are well known from mathematics, namely *monotonicity* and *continuity*. We now give an intuitive explanation of these requirements, and then in the next chapter, we give a formal definition.

If a component has emitted a message, it cannot take it back. So any reaction that happens in the future after message was emitted can only be the emission of further messages. That means mathematically, that an enlargement of the input sequence of the component can only lead to an enlargement of the output sequence of messages. This property of stream processing functions is called *monotonicity* and is necessary, because our functions can look at the whole history in its arguments and describe the whole history of the output at the same time.

Second, to describe liveness and related properties, infinite streams need to be considered to describe full histories. However, each emitted message must actually be admitted as a reaction of a finite sequence of input messages. It is illegal to look at the complete input history to emit a message – which obviously then would be emitted after a finite period of time. Technically this is ensured by enforcing $f$ to be *continuous*, which allows to define behavior, like $f$, by inductively looking at *approximations* of the input to produce *approximations* of the output [Kle52].

Finally, *stream processing functions* are defined as functions from stream (tuples) to stream (tuples) which are continuous [RR11].

Next we use *sets of functions* to be able to give a specification a semantics that actually allows several different behaviors, based on possible nondeterminism of the components implementation, or based on insufficient information available during development time.

One of the most important properties of FOCUS [Bro+92; BS01] is that it provides sound and mighty composition operators. Composition of continuous functions is continuous as well. So a computation unit can be hierarchically decomposed into a collection of continuous stream processing functions.

There is a straightforward extension of composition to sets of functions, which then aso allows us to decompose specifications (sets of behaviors).

The second fundamental property is that refinement of component specifications is semantically reflected by the concept of set inclusion between function sets. And most importantly:

| |
|---|
| Refinement of a component in a decomposed structure automatically leads to refinement of the composition [BR07]. |

This important property is actually the reason, why streams are such a helpful technique to formalize behavior of distributed components. We can abstractly specify behavior, decompose the specification (may be hierarchically as long as desired), refine each individual sub-specification until an implementation component is reached, and then can be sure that the composition of the implementations is correct by design. To our knowledge, no other approach can do this so powerful as FOCUS.

In this work the above mentioned constructs have been encoded in the interactive proof assistant *Isabelle* [Pau90; www18; PB10]. While streams and stream processing functions heavily rely on inductive definition and therefore on fixpoint theory, we provide a high-level API and hide specific usage of domain theoretical concepts from the user. Therefore, proving theorems on streams very often does not have to deal with the domain theoretical induction concepts in the proof engine, but can deal with abstract high level definitions and theorems. Domain-theory, on which this work relies, has already been sufficiently formalized in Isabelle in [Reg94]. In [Huf12], a comprehensive introduction to Isabelle/HOLCF as a theorem proving system focusing on the domain-theory formalization is given, and the so-called *domain* package, which facilitates the work with domain theoretical concepts, is introduced. Nevertheless, an optimal formalization of the stream theory in Isabelle hasn't been achieved yet. A variant of a stream data type is presented in HOLCF [Mül+99]. The works [GR06], [Stü16], [Bür17], [Slo17], [Wia17], [Kau17], [Zel17], [Mül18] present some ideas which form the foundation of the current work.

Spichkova [Spi08] independently formalized parts of FOCUS in Isabelle/HOL. System specifications can either be translated manually or developed directly in Isabelle/HOL. The implementation covers timed streams and also proposes ways to handle time-synchronous streams. Based on some results Trachtenhertz [Tra09] has formalized semantics for the description techniques of AutoFOCUS in Isabelle/HOL. The framework focuses on temporal specifications of functional properties. The work aims at supporting the development process from design phase to an executable specification. As an industrial case study, an adaptive cruise control system is formalized. The tool-chain AutoFOCUS [HF11] uses this HOL-formalization of FOCUS to check properties of component networks.

In a different line of work relying on automated rather than interactive theorem proving, Huber et al. [HSE97] report on automated verification of the refinement of AutoFOCUS components described by state transition diagrams and a sequence diagram notation using the model checkers SMV [Bur+92], and $\mu$-cke [Bie97] based on trace inclusion. Similarly, [Rin14] shows a translation of components with nondeterministic automata implementations to the theorem prover Mona [EKM98; www13].

Another related work to formalize model component networks is the Ptolemy Project [Lee09] where the authors create a framework for actor-oriented design. The encoding of possible infinite streams in a computer program is nontrivial. A methodology for the encoding of possibly endless sequence data structures in theorem provers is presented in the work of Devillers, Griffioen and Müller [DGM97].

Compared with the works mentioned above, the benefit of our approach is that it offers a comfortable environment for reasoning about component networks occurring in architecture description languages, i.e., MontiArc [HRR12], or component-behavior implementation languages like MontiArcAutomaton [RRW14]. On top of that, it provides a semantic [HR04] domain for component-and-connector modeling languages.

## 1.1 Goals and Results

The key contributions of this work are:

- An optimized Isabelle realization of streams.
- A group of over 100 useful functions on streams, bundles, stream processing functions and their composition. The corresponding continuity proofs, which constitute a vital part of this contribution, are found in the appendix.
- A collection of about 1000 theorems providing a high-level API for proofs on streams and stream processing functions.
- A formalization of realizability constraints for untimed modeling of components and component networks with potential feedback loops.
- A formalization of the general composition operator for (sets of) stream processing functions.
- An extension of the framework for tuples of streams to model component interfaces and allowing composition is implemented, as well as essential theorems about these.
- An implementation of untimed stream processing functions.
- An evaluation on a case study used as a running example.

Further contributions are:

- An Isabelle theory for natural numbers with the largest element $\infty$. (Section 4.3)
- An Isabelle theory for enhancing sets with the inclusion-order. (Section 4.2)

Figure 1.2 gives an overview of our theories, such as stream bundle (SB), stream-processing function (SPF), and sets of functions denoted as stream processing specification (SPS). Theory imports are represented as arrows in the figure.

We begin this work with a small introduction to domain theory in Chapter 2 to explain the mathematical concepts that are required to understand the definition of streams and stream processing functions.

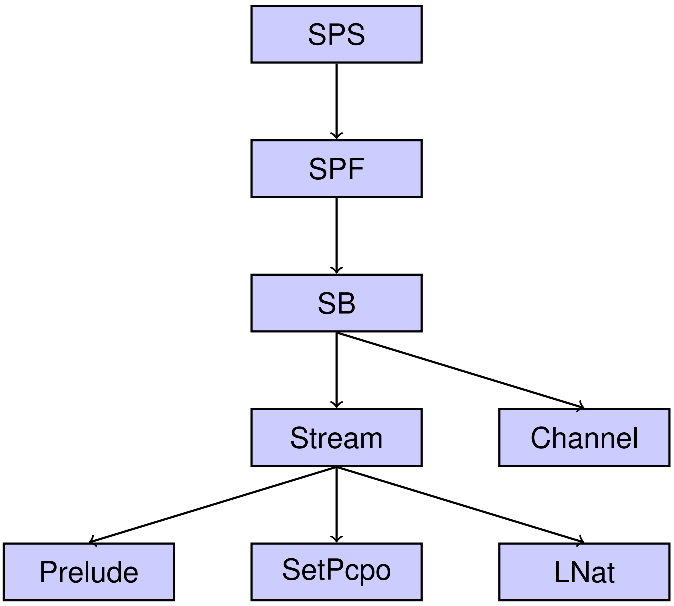



Figure 1.2: Overview of the Theory Structure

In Chapter 3 we will introduce the basics of the proof assistant Isabelle [NPW02]. For a more detailed introduction to domain theory and its formalization in theorem provers, we recommend reading [Nip13], and [NPW02]. Both works provide an in-depth introduction to functional programming, higher-order logic, HOLCF (higher-order logic of continuous functions) as well as the theorem prover Isabelle that we use for our FOCUS formalization.

In Chapter 4 we will then discuss our extension theories for the HOLCF library. The mathematical definition of streams in FOCUS as well as our formalization in Isabelle will be presented in Chapter 5. After that, the concept of stream bundle [Rum96] that help to improve the scalability of FOCUS models will be introduced. Furthermore, we will present the corresponding implementation in Isabelle. Based on the concept of stream bundle we will then describe stream-processing functions and their formalization in Chapter 7. Finally, we will explain our formalization the general composition operator for (sets of) Stream Processing Functions and their implementation in Chapter 8.

**Acknowledgements** We thank Peter Sommerhoff and Patricia Wessel for writing a first draft of the foundations chapter of this work.

# Chapter 2

# Foundations of Domain Theory

The mathematical field of domain theory plays a major role in denotational semantics which defines the meaning of programming language elements based on known mathematical structures [HR04; Bro13]. In particular, denotational semantics allows us to define the meaning of possibly recursive functions which build the foundation for streams and stream processing function. So to fully understand the concept of streams, we need to understand the mathematics used to define them first, especially domains, functions, and fixed-points.

## 2.1 Partial Orders

Ordered sets are one of the core concepts in domain theory. Since orders are special relations, we have to define relations first.

**Definition 2.1.** A (binary) *relation* $R$ over a set $S$ is a subset of the Cartesian product $S \times S$.

We can now define partial orders as follows:

**Definition 2.2.** A *partial order (po)* over a set $S$ is a relation $R$, denoted as $\sqsubseteq$, that satisfies the following [SK95]:

- $\sqsubseteq$ is reflexive: $\forall x \in S.\ x \sqsubseteq x$
- $\sqsubseteq$ is transitive: $\forall x, y, z \in S.\ x \sqsubseteq y \wedge y \sqsubseteq z \ \Rightarrow\ x \sqsubseteq z$
- $\sqsubseteq$ is antisymmetric: $\forall x, y \in S.\ x \sqsubseteq y \wedge y \sqsubseteq x \ \Rightarrow\ x = y$

Based on partial orders, we can now define the concept of chains and least upper bounds.

**Definition 2.3.** An *(ascending) chain* in a partially ordered set $S$ is a sequence of elements $[x_1, x_2, x_3, \dots]$ such that $x_1 \sqsubseteq x_2 \sqsubseteq x_3 \sqsubseteq \dots$. [SK95]

Please note that a chain has at most countably many elements. For the following two definitions let $S, S'$ be sets such that $S' \subseteq S$. $S$ need not be countable.

**Definition 2.4.** An *upper bound* of $S'$ in $S$ is an element $b \in S$ such that $\forall x \in S'.x \sqsubseteq b$ holds. [SK95]

**Definition 2.5.** A *least upper bound (lub)* (denoted $\bigsqcup S$) is an upper bound of $S$ such that for any upper bound $b'$, $b \sqsubseteq b'$ holds. [SK95]

Since we have now defined the requirements of partial orders, we can give an example of such an order.

**Example 2.1.** *Let $A$ be a set, then the subset relation $\subseteq$ is a partial order on the powerset $\wp(A)$, as the subset relation satisfies the three properties a partial order must have. Furthermore, there exists a least upper bound for every subset of $\wp(A)$.*

Depending on the use case, e.g. constructing the communication history on channels recursively, the requirements for partial orders alone may not be strong enough to serve as semantic domains. For example, an ordered set does not necessarily posses a least element, which is useful as an initial approximation, and ordered sets might not have least upper bounds, which are desirable as tight approximations. To overcome this issue, we can define two, more restrictive types of partial orders.

**Definition 2.6.** A *complete partial order (cpo)* on a set $S$ is a partial order $\sqsubseteq$ such that every ascending chain in $S$ has a least upper bound which is included in $S$. [SK95]

**Definition 2.7.** A *pointed complete partial order (pcpo)* on a set $S$ is a cpo such that an element $\perp \in S$ exists for which $\forall x \in S.\perp \sqsubseteq x$ holds. [SK95]

## 2.2 Domains

In our work, a domain is a pointed complete partial order (pcpo). One of the simplest domain types are the elementary or flat domains [SK95] which express results of programs in denotational semantics and allow the precise specification of a program's meaning. We can convert any set $S$, e.g., Boolean values $S = \{0, 1\}$ or Integer values $S = \mathbb{Z}$, into such an elementary domain by adding an artificial bottom element $\perp$ and defining a discrete partial order $\sqsubseteq_{S_\perp}$ that satisfies the following:

$$\forall x, y \in S.\ x \sqsubseteq y \overset{\text{def}}{\Leftrightarrow} (x = y) \vee (x = \perp)$$

It should be noted, that for sets like the integers the partial order $\sqsubseteq_{\mathbb{Z}_\perp}$ is not equal to the default ordering $\leq_{\mathbb{Z}}$. For example, $1 \leq_{\mathbb{Z}} 2$ holds but $1 \sqsubseteq_{\mathbb{Z}_\perp} 2$ not.

Based on multiple domains, we can construct more complex ones using domain constructors. The presumably most well-known constructor is the Cartesian product that can be used to generate product domains.

**Definition 2.8.** The product domain $X \times Y$ with the ordering $\sqsubseteq_{X \times Y}$ of two pcpos $X$ and $Y$ with the orderings $\sqsubseteq_X$ and $\sqsubseteq_Y$ is defined as follows:

$$X \times Y := \{(x, y) \,|x \in X,\ y \in Y\}$$

$$(x_1, y_1) \sqsubseteq_{X \times Y} (x_2, y_2) \overset{\text{def}}{\Leftrightarrow} x_1 \sqsubseteq_X x_2 \wedge y_1 \sqsubseteq_Y y_2$$

**Lemma 2.1.** The ordering $\sqsubseteq_{X \times Y}$ is a pcpo on $X \times Y$ with least element $(\perp, \perp)$ [SK95].

## 2.3 Functions

Sets of functions with an appropriate order can also be a domain. Before we can show how function domains can be constructed, we have to define the concept of partial and total functions first.

**Definition 2.9.** A function $f : X \to Y$ is a *total* function if for every $x \in X$ the value $f(x) \in Y$ is defined.

If $f$ is not total, $f$ is called *partial*.

It should be noted that we can transform every partial function $f_p : A \to B$ into a total function $f_t$. This can for example be achieved by adding a new and distinct element $\bot_B$ to the codomain and defining $f_t$ as follows:

$$f_t(x) := \begin{cases} f_p(x) & \text{if } f_p(x) \text{ is defined} \\ \bot_B & \text{otherwise} \end{cases}$$

In functional languages this lifting process can be realized by changing data type `c` of a function $f$ to `c option`:

```
datatype 'a option = None | Some 'a
```

Here **`datatype`** is a keyword for creating data types, `'a` denotes a type variable, and `None` and `Some` are constructors. Undefined inputs of a function are then mapped to `None` whereas a defined value $y$ is mapped to `Some y`.

In the following, we will denote the signature of such lifted functions as $A \rightharpoonup C$, where $A$ is an arbitrary type. As this lifting is always possible, we will from now on restrict ourselves on total functions.

#### Function Domains

Since we have now characterized total and partial functions, we can define function domains [SK95].

**Definition 2.10.** Let $X$ and $Y$ be pcpos with the orders $\sqsubseteq_X$ and $\sqsubseteq_Y$. The function domain $\mathrm{Fun}(X, Y)$ is defined as the set of all *total functions* with domain $X$ and codomain $Y$. The ordering on this set $\sqsubseteq$ is then defined such that the following holds:

$$\forall f, g \in \mathrm{Fun}(X, Y). \quad (\forall x \in X.\ f(x) \sqsubseteq g(x)) \ \Leftrightarrow \ f \sqsubseteq g$$

Based on this definition we can deduce the following:

**Lemma 2.2.** The ordering $\sqsubseteq_{\mathrm{Fun}(X,Y)}$ is a pcpo on $\mathrm{Fun}(X, Y)$ [SK95].

To make use of function domains in denotational semantics, we have to further restrict the set $\mathrm{Fun}(X, Y)$ as it may contain functions that are not realizable. For example, $\mathrm{Fun}(\mathbb{N}_\bot \to \mathbb{N}_\bot, \mathbb{B})$ includes a function $H$ which delivers $\mathrm{true}$ if and only if the function $f$ it is applied to, always delivers a defined value, which is not computable.

The first restriction we can define for function domains is monotonicity.

**Definition 2.11.** A function $f \in \mathrm{Fun}(X, Y)$ is *monotonic* if

$$\forall x_1, x_2 \in X.\ x_1 \sqsubseteq x_2 \ \Rightarrow\ f(x_1) \sqsubseteq f(x_2)$$

holds.

The second restriction is the continuity.

**Definition 2.12.** A function $f \in \mathrm{Fun}(X, Y)$ is *continuous* if for every chain $A$ the following holds:

$$f\left(\bigsqcup\{a_i|1 \le i\}\right) = \bigsqcup\{f(a_i)|1 \le i\}$$

So a monotonic function preserves the ordering and a continuous function preserves least upper bounds. The following two lemmas will be of particular importance later:

**Lemma 2.3.** The concatenation of two continuous functions is again continuous. [SK95]

**Lemma 2.4.** A continuous function is monotonic. [SK95]

**Example 2.2.** *Let $A := \{a^i|i \in \mathbb{N}_\infty\}$ where $\mathbb{N}_\infty := \mathbb{N} \cup \{\infty\}$, and $\sqsubseteq_A$ be the prefix ordering on words e.g., $a \sqsubseteq_A aa$. Then $\sqsubseteq_A$ is a pcpo on $A$ where $\perp = a^0$.*

*Furthermore, let $f_1, f_2, f_3 \in \mathrm{Fun}(A, A)$ such that $\forall x \in \{1, 2, 3\} : \forall i \in \mathbb{N} :\ f_x(a^i) := a$. The mappings of the infinitely long $a$ word are defined as shown below:*

- $f_1(a^\infty) := \perp$
- $f_2(a^\infty) := aa$
- $f_3(a^\infty) := a$

*Note that there exists an ascending chain $Y := [a^0, a^1, a^2, \ldots, ]$ such that $\bigsqcup\{y_i|1 \le i\} = a^\infty$, but for all $x \in \{1, 2, 3\}$ we have $\bigsqcup\{f_x(Y_i)|1 \le i\} = a$.*

*Then $f_1$ is neither monotonic nor continuous. The functions $f_2$ and $f_3$ are monotonic, but only $f_3$ is continuous.*

## 2.4 Fixed-Points

The semantics of recursive functions [Kle52], as well as the stream flowing in feedback loops [Bro+92] will be described by so-called *least fix point (lfp)*.

**Definition 2.13.** The *fixed-point* of a function $f : D \to D$ is a $d \in D$ such that $f(d) = d$.

Thus, applying $f$ to one its fixed points will return that fixed-point.

**Example 2.3.** *A function may have no fixed-point, an unique fixed-point, or multiple fixed-points:*

- $f : \mathbb{N} \to \mathbb{N}, x \mapsto x + 1$ *has no fixed-point.*
- $f : \mathbb{N} \to \mathbb{N}, x \mapsto 2x$ *has the unique fixed-point $x = 0$.*
- $f : \mathbb{N} \to \mathbb{N}, x \mapsto x$ *has infinitely many fixed-points, i.e. all $x \in \mathbb{N}$.*

### 2.4.1 Motivation: Recursive Definitions

Recursion is a principle used not only in programming but also in mathematical definitions. However, the meaning of such recursive definitions needs to be clearly defined. Consider for example the following recursive function:

**Example 2.4.**

$$f : \mathbb{N} \to \mathbb{N}, x \mapsto [\textit{if } x = 0 \textit{ then } 42 \textit{ else if } x = 1 \textit{ then } f(x+2) \textit{ else } f(x-2)]$$

As we can see, the function $f$ will return 42 for even numbers as input but is undefined otherwise. For instance, $f(4) = f(2) = f(0) = 42$, whereas $f(5) = f(3) = f(1) = f(3) = f(1) = \ldots$.

To define the meaning of such recursively defined functions, we consider functionals. A functional $F$ is a function which maps functions. They are often also called higher-order functions. Here, the goal is to define the meaning of $f$, out of all the functions which satisfy the recursive equation, as the least fixed-point of a corresponding functional $F$. Due to this, we define $F : (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$ as follows:

**Example 2.5.**

$$F\ f\ x\ := [\textit{if } x = 0 \textit{ then } 42 \textit{ else if } x = 1 \textit{ then } f(x+2) \textit{ else } f(x-2)]$$

We assume here that function application associates to the left, i.e. F f x $= (F(f))(x)$. Now, any fixed-point of $F$ fulfills the recursive function definition of $f$. However, notice that the closed form of $f$ is not uniquely defined. This comes back to the fact that a function can have multiple fixed-points, as exemplified above.

Consider for example the function $g : \mathbb{N} \to \mathbb{N}$ with $g(x) := 42$ for all $x \in \mathbb{N}$ which is a fixed-point of $F$:

**Example 2.6.**

$$F\ g\ x := [\textit{if } x = 0 \textit{ then } 42 \textit{ else if } x = 1 \textit{ then } 42 \textit{ else } 42] = 42 = g\ x$$

In other words, $F\ g = g$, which means that $g$ is indeed a fixed-point of $F$. However, we do not want to accept this as the semantic of the original recursive function $f$ because $g$ is over approximating $f$. More specifically, $f$ is not defined for odd numbers, whereas $g$ is defined for every natural number.

There are two problems we have to solve to make this approach work:

1. Make sure that the functional has at least one fixed-point
2. If it has multiple fixed-points, choose the “best” fixed-point under some criteria

This will lead us to the usage of continuous functionals, which have at least one fixed-point. Next, we will choose the *least* fixed-point with respect to $\sqsubseteq$, which will be the “best” fixed-point for our purposes.

### 2.4.2 Fixed-Point Theorems

Knaster-Tarski's Fixed-Point Theorem [SK95] implies that any *monotonic* function $f : D \to D$ on a pcpo $D$ has a unique least fixed-point.

**Lemma 2.5.** From Kleene's Fixed-Point Theorem [Kle52], we can follow that any *continuous* function $f : D \to D$ on a pcpo $D$ has as a least fixed point $\text{fix}(f)$ that satisfies the following:

$$\text{fix}(f) = \bigsqcup\{f^n(\bot) \mid i \geq 0\} = \bigsqcup_{n\in\mathbb{N}} f^n(\bot)$$

This means that the least fixed point is effectively computable by starting with $\bot$ and iteratively applying $f$.

*Proof.* First, we show that $f$ has a fixed-point. Remember that continuity implies monotonicity. With monotonicity, $\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \dots$ forms an ascending chain in $D$ which has an upper bound in $D$, say $u := \bigsqcup\{f^i(\bot) \mid i \geq 0\}$. Thus:

$$\begin{aligned}
f(u) &= f(\bigsqcup\{f^i(\bot) \mid i \geq 0\}) \\
&= \bigsqcup\{f^{i+1}(\bot) \mid i \geq 0\} && (f \text{ continuous}) \\
&= \bigsqcup\{f^i(\bot) \mid i > 0\} \\
&= u && (f^0(\bot) = \bot \text{ does not change least upper bound})
\end{aligned}$$

So the least upper bound $u$ is a fixed-point of $f$.

Second, we show that $u$ is indeed the *least* fixed-point. Assume we have another fixed-point $v \in D$. Then:

$$\begin{aligned}
&\bot \sqsubseteq v \\
&\Rightarrow f(\bot) \sqsubseteq f(v) = v && f \text{ monotonic, } v \text{ fixed-point} \\
&\Rightarrow f^i(\bot) \sqsubseteq f^i(v) = v \ \forall i \geq 0 && \text{induction} \\
&\Rightarrow f^i(\bot) \sqsubseteq v \ \forall i \geq 0 \\
&\Rightarrow u \sqsubseteq v
\end{aligned}$$

With this, we have shown that a continuous function over domains has a unique least fixed-point that can be iteratively approximated. □

Please note, however, that [Kle52] can only be applied to countable chains. For larger domains, such as power sets of functions, where uncountable chains might be neccesary to reach the least fixed point, Knaster-Tarski is appropriate. Here, we prefer the least fixed point over other fixed points because in our semantic definitions it represents the least amount of information necessary to be consistent with a specified behavior. Being able to compute the least fixed-point suffices when dealing with issues of computability. For streams and SPFs computability is of course an important concept. SPSs, however, denote the semantics of a specification. Specifications describe potentially large sets of computable SPFs and are therefore not bound to the least fixed point only. In particular we will use monotonic, but non-continuous functionals to explain SPSs. For more detail on how to construct continuous functionals, see [SK95].

### 2.4.3 Relation Between Monotonic/Continuous Functions and Least Fixed-Points

To better understand the relations between the concepts of monotonic functions, continuous function, and least fixed-points, the illustration in Figure 2.1 along with examples for each class in the Venn diagram helps. Let $A$ be defined as in Example 2.2.

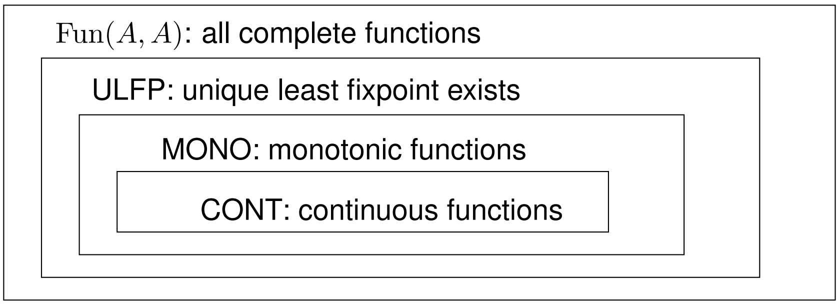



Figure 2.1: Function classes in $\mathrm{Fun}(A, A)$

To show that the classes are proper subsets of each other, we give an example function $f \in \mathrm{Fun}(A, A)$ for each class that is not included in the more restrictive classes. The classification and examples are as follows:

1. $\mathrm{Fun}(A, A)$ portrays the set of all functions in the function domain. An example of a function in $\mathrm{Fun}(A, A)$ but not in ULFP (because f has no fixed point at all) is:

$$f(x) = \text{ if } x = a \text{ then } aa \text{ else } a$$

2. ULFP is the set of all functions that have an unique least fixed-point (lfp). As the diagram implies, ULFP is a proper subset of $\mathrm{Fun}(A, A)$. An example of a function in this class that is not included in MONO (follows by definition of monotonicity) is the following:

$$f(x) = \text{ if } x = a^\infty \text{ then } \perp \text{ else } a$$

3. MONO is the set of all monotonic functions. A monotonic but non-continuous function can be defined as shown below (follows by definition of continuity):

$$f(x) = \text{ if } x = a^\infty \text{ then } aa \text{ else } a$$

4. CONT is the set of all continuous functions. An example of such a continuous function is a constant function as shown below:

$$f(x) = a$$

So monotonicity implies the existence of a unique least fixed-point (lfp), and continuity implies that the least fixed-point is equal to $\bigsqcup\{f^i(\perp) | i \in \mathbb{N}\}$. As already mentioned earlier, this means that for continuous functions the lfp can be iteratively approximated. Using our examples we can now also conclude, that the implications do not necessarily hold in the opposite direction, i.e., there are *non*-monotonic functions with a unique lfp.

### 2.4.4 Predicates and Admissibility

To make a statement about properties of elements, we use predicates. A predicate is a function, which evaluates to the Boolean values $\mathrm{true}$ or $\mathrm{false}$.

**Definition 2.14.** We call a predicate $P : A \to \mathbb{B}$ *admissible* [Huf12] if it holds for the lub of a chain in $A$ whenever it holds for the elements of the chain:

$$\mathrm{adm}(P) \iff \forall C : \ (\mathrm{chain}(C) \implies (\forall x \in C.P(x)) \implies P(\bigsqcup C))$$

Admissibility is helpful when the predicate shall be shown inductively over a chain.

**Example 2.7.** *The list predicate* $P : [\mathbb{N}] \to \mathbb{B},\ P(s) \mapsto (\mathrm{length}(s) >= 0)$ *is admissible.*

**Example 2.8.** *The list predicate* $P : [\mathbb{N}] \to \mathbb{B},\ P(s) \mapsto (\mathrm{length}(s) < \infty)$ *is not admissible.*

If we regard predicates as function with target domain , where $false \sqsubseteq true$, then admissibility is identical to continuity.

### 2.4.5 Fixed-Point Induction

Statements about recursive functions can now be established by induction on the construction of the least fixed-point.

Consider a predicate $P$ that describes a property of a recursive function $f$. To show that $P$ holds for $f$, we show

1. $P$ holds for all elements in the ascending chain $\{F^i(\bot) \mid i \geq 0\}$ (using induction).
2. $P$ is admissible

### 2.4.6 Construction of Admissible Predicates and Continuous Functions

As a remark, we note that admissible predicates and continuous functions can easily be constructed from smaller ones, E.g. if $P$ and $Q$ are admissible, so are $P \wedge Q$ and $P \vee Q$.

If $f$ and $g$ are continuous, so is $f \circ g$. This leads to structural induction on definition of functions and predicates. However, negation and quantification may violate this structural induction.

# Chapter 3

# Introduction to Isabelle/HOLCF

Throughout the remainder of this book, we will use the proof assistant Isabelle [NPW02] and its implementation language ML for our FOCUS formalization. We will now give a brief overview of the tool and refer the interested reader to a more comprehensive documentation on Isabelle in [www18].

Isabelle is an interactive proof assistant. The syntax of is similar to functional languages like ML or Haskell. However in contrast to such pure functional programming languages, we can proof properties directly in Isabelle. We formalize these proofs, data structures as well as functions in so called theory files.

```
theory ExampleTheory
imports Main
begin
    (* definitions and lemmas *)
end
```

In this example, the theory `ExampleTheory` imports just the `Main` theory which acts as a facade of all predefined HOL theories. It should be noted, that in contrast to Java or Python files, theory files are strictly read from the top to the bottom without the possibility of forward referencing.

## 3.1 Isabelle/HOL

There are a variety of libraries available that extend Isabelle's pretty small logical core and simplify working with the tool. One of the most frequently used libraries is Isabelle/HOL [NPW02] that extends the logical core by the concept of higher order logic as well as data structures like sets, and lists.

### 3.1.1 Isabelle's Type System

In Isabelle all variables are typed. Polymorphic types can be denoted via formal type parameters like `'a` or `'b`. Although Isabelle can in most cases automatically determine the type of a variable or constant `x`, we can also fix the type to a specific one which is denoted as `(x::<typename>)`.

The type of a total function with $n$ input parameters of types $\tau_1, \ldots, \tau_n$ and return type $\tau$ is denoted as $\tau_1 \Rightarrow \tau_2 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \tau$. As usual, $\Rightarrow$ associates to the right. For instance, the type of the identity function is `'a ⇒ 'a`.

Isabelle/HOL also defines types like `nat` for the natural numbers and `bool` for Boolean values. Furthermore, predefined type constructors, i.e., `list` and `set`, can be used in postfix syntax to create composite types, such as `nat list` or `bool set`.

### 3.1.2 Defining Types

We can also create our own data types in Isabelle. A simple data type can be defined using the syntax shown below:

```
datatype Operator = Plus | Minus
```

The statement above declares the new data type `Operator` with exactly two constructors separated by a | character: `Plus` and `Minus`. Both constructors are nullary, i.e., they do not have any parameters. Thus, their type is `()⇒Operator`. Note that the `Operator` type viewed as a set consists of exactly two elements. However, a **datatype** based type definition does not instantiate an order on the data type elements.

Data types can also be parameterized using formal type parameter as follows:

```
datatype 'a Box = EmptyBox | Wrap 'a
```

To use this `Box` data type, the formal type parameter `'a` must be instantiated first. For instance, `Wrap (Suc 0)` is of type `nat Box`. Here, `EmptyBox` is a nullary constructor and `Wrap` is a unary constructor. Note that for a type $\tau$ with $n$ elements, the type $\tau$ `Box` has exactly $n+1$ elements, i.e., `bool Box` has 3 elements.

Next, we can declare recursive data types by using the declared type on the right-hand side of the type definition:

```
datatype 'a strictlist = Empty | Prepend "'a" "'a strictlist"
```

Again, this generates two constructors with the following types:

```
Empty :: () ⇒ 'a strictlist
Prepend :: 'a ⇒ 'a strictlist ⇒ 'a strictlist
```

To construct instances of custom data types more concisely, we can define constructor abbreviations in the data type declaration:

```
datatype 'a strictlist = Empty ("[]") |
                         Prepend "'a" "'a strictlist" (infixl "@")
```

This way, we can denote an empty list as `[]` and write `x @ xs` instead of `Prepend x xs`. Notice that **infixl** declares `@` as a left-associative infix operator, i.e.

```
x₁ @ x₂ @ ... @ xs = ((x₁ @ x₂) @ ...) @ xs
```

Lastly, we can name formal constructor parameters such that Isabelle generates selectors for them:

```
datatype 'a strictlist =
  Empty ("[]") |
  Prepend (head :: "'a") (tail :: "'a strictlist") (infixl "@")
```

In this case Isabelle creates the following two selectors:

```
head :: 'a strictlist ⇒ 'a
tail :: 'a strictlist ⇒ 'a strictlist
```

So when `xs` is a non-empty list of type `'a strictlist`, we can access the first list element via `head xs` and the rest of the list with `tail xs`.

## 3.2 Function and Class Definitions

To define simple non-recursive functions in Isabelle, we can use the **definition** command. For instance, a generic identity function can be defined as shown below:

```
definition id :: "'a ⇒ 'a" where
"id ≡ (λx. x)"
```

Alternatively, if we just want to use such a definition to abbreviate a more complex formula, we can use the **abbreviation** keyword. On the one hand, an abbreviation has the advantage that it is automatically replaced by its definition and vice versa if necessary. On the other hand, this automatic replacement might not always be desired since it can negatively influence Isabelle's proof strategies like the simplifier.

Recursive or pattern matching based functions can be defined using the **fun** or **primrec** command. For instance, a function that delivers $1$ if applied to $0$ and otherwise behaves like the identity function can be formalized as shown below:

```
fun succ_zero :: "nat ⇒ nat" where
"succ_zero 0 = 1" |
"succ_zero x = x"
```

As we can see, the syntax of such definitions is yet again similar to Haskell. The same also holds true for class definitions. However, classes in Isabelle also allow us to specify properties of functions. A class for types with an equality function can for example be specified as follows:

```
class Eq =
  fixes myEq :: "'a ⇒ 'a ⇒ bool"
  assumes reflexivity:  "myEq a a = True"
  assumes symmetry:     "myEq a b = eq b a"
  assumes transitivity: "myEq a b ∧ eq b c ⟶ eq a c"
```

## 3.3 Domains in Isabelle

In the previous we saw how the **datatype** constructor can be used to define custom data types. However, such data type definitions have some limitations, i.e., they only consist of values that can be constructed with finitely many applications of the constructors. Furthermore, the **datatype** command does not establish an order on the data type but orders are necessary for inductive reasoning over the data type. In the following we will present three ways to overcome these issues namely the lifting of data types, the domain constructor and subtypes.

### Lifting Datatypes to Domains

For any ordinary HOL type we can define a (trivial) complete partial order by giving it a discrete ordering. In HOLCF this construction is formalized using the `'a discr` type [Huf12]:

```
datatype 'a discr = Discr "'a"
```

To still be able to access the elements of the lifted data type, an inverse of the `Discr` constructor is defined as shown below:

```
definition undiscr :: "'a discr ⇒ 'a" where
"undiscr x ≡ (case x of Discr y ⇒ y)"
```

The ordering on `'a discr` is defined as a flat ordering, i.e., $(x \sqsubseteq y) = (x = y)$. Thus, `'a discr` is an instance of the `discrete cpo` class [Huf12].

It follows straightforwardly by the corresponding definitions that every function `f :: 'a discr ⇒ 'b` is continuous and every predicate `P :: 'a discr ⇒ bool` is admissible.

Furthermore, we can also lift a given type with a complete partial ordering to a type with a pointed cpo by adding a new bottom element. Let $D$ be a cpo (which may or may not have a least element), then the lifted pcpo $D_\perp$ consists of a bottom element $\perp$ and wrapped elements of the original type.

In HOLCF, the lifting of cpos to pcpos can be achieved by using the `'a u` type which is also often abbreviated as `'a`$_\perp$:

```
datatype 'a u = Ibottom | Iup 'a
```

The order on this data type is defined such that the following holds:

$$a \sqsubseteq b \Leftrightarrow (a = \texttt{Ibottom}) \vee (\exists x, y.\ a = \texttt{Iup}\ \texttt{x} \wedge b = \texttt{Iup}\ \texttt{y} \wedge x \sqsubseteq y)$$

One can show that the type `'a`$_\perp$ is a pcpo, if the type `'a` is substituted with, has a partial order.

### The Domain Type-Constructor

In Section 2.2 we already explained how domains can be constructed from simpler domains using domain constructors. To achieve the same in Isabelle, we can use the domain package [Huf12] which produces data types that are instances of the pcpo class and hence have a pcpo ordering. The syntax of such a data type definition is similar to a type definition using the **datatype** keyword.

However, the domain package defines the data type constructors as strict continuous functions and automatically adds a bottom element $\perp$. An example for such a domain definition is the lazy list data type:

```
domain 'a list = Cons "'a discr u" (lazy "'a list")
```

The empty list is here implicitly defined as the bottom element $(\perp)$ of the data type. The **lazy** keyword makes the constructors non-strict in specific arguments. In this example we defined `Cons` to be non-strict in its second argument to allow the definition of infinitely long lists.

To facilitate proofs that involve such **domain** types, Isabelle also adds several rewrite rules to Isabelle's simplifier (`simp`), and generates the necessary theorems and functions for case distinctions and induction proofs. [Huf12] provides a good overview of all theorems and functions that are automatically generated by the domain package.

### CPOs on Subtypes

An even simpler way to create a cpo type is to define it as a subset of an existing cpo type. Under certain conditions, a subtype can inherit the ordering structure from the existing ordering it is based on which means that the subtype is again a member of the cpo class. In Isabelle this process can be automated using the **pcpodef** and **cpodef** commands [Huf12]. Both commands are based on the **typedef** command [NPW02] that allows defining a new type as an isomorphic and nonempty subset of an existing type. For example, we can define a new type `zeroToFive` that is isomorphic to the set of all integers that are smaller or equal than $5$:

```
typedef zeroToFive = "{x::int. 0 ≤ ∧ x ≤ 5}"
  by (auto)
```

The proof is necessary since we must prove that the newly created types is non-empty.

After showing the set on the right side of the definition is non-empty, the **typedef** package automatically creates useful theorems as well as the functions (`Rep_zeroToFive`, `Abs_zeroToFive`) to convert elements of `zeroToFive` to elements of the `int` data type and vice versa. We can then use those function to define new functions on `zeroToFive` based on existing function on the `int` type:

```
definition zeroToFive_add:: "zeroToFive ⇒ zeroToFive ⇒ zeroToFive"
where "zeroToFive_add x y =
  Abs_zeroToFive (Rep_zeroToFive x + Rep_zeroToFive y)"
```

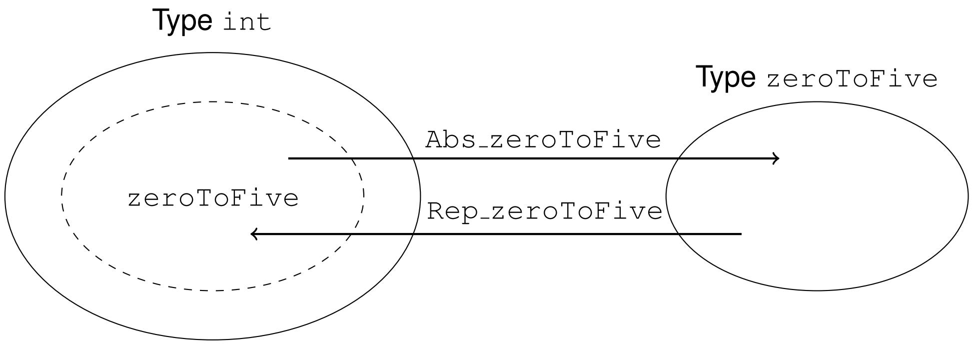



Figure 3.1: Graphical Representation of the typedef Mechanism

The newly defined addition operator on the new type relies on (and hides) the primitive + operator on integers. Later this principle helps to reach an important achievement of this work; creating a high-level API to hide the low-level domain-theoretical concepts from the user.

The lemmas generated by the **typedef** package can also be used to show properties like the commutativity of this function.

The **cpodef** and **pcpodef** commands, that automatically construct an ordering on the new subtype, have an identical syntax as **typedef**. If we want to use **cpodef** to define the zeroToFive type, we additionally would have to show that predicate $\lambda$ x. x $\leq$ 5 is admissible. In case we want to use **pcpodef** we additionally would have to show that the subtype has a least element.

## 3.4 Continuous Functions and Fixed Points

As we already saw in Chapter 2, the concepts of mononicity and continuity play an important role in the field of function domains.

Continuous functions in HOLCF are formalized by using the cfun data type which is instantiated using the **cpodef** command:

```
definition "cfun = {f::'a => 'b. cont f}"

cpodef ('a, 'b) cfun ("(_ →/ _)" [1, 0] 0) =
  "cfun :: ('a => 'b) set"
  unfolding cfun_def by (auto intro: cont_const adm_cont)
```

Thus, the type of continuous functions from $A$ to $B$ is denoted as A → B. To automatically lift anonymous functions to their continuous counterparts, the small letter $\lambda$ in the definition of such a function can be replaced by $\Lambda$. However, such a lifting is of course only successful if the function that should be lifted is in fact continuous.

Based on the cfun type, functions like the **fix** operator which calculates the lfp [Mül+99] can be defined:

```
primrec iterate :: "nat ⇒ ('a::cpo → 'a::pcpo) → ('a → 'a)" where
```

```
"iterate 0 = (Λ F x. x)" |
"iterate (Suc n) = (Λ F x. F·(iterate n·F·x))"

definition fix :: "('a ::pcpo → 'a) → 'a" where
"fix = (Λ F. ⨆i. iterate i·F·⊥)"
```

As we can see, this operator is directly based on the fixed point theorem by Kleene (c.f. Lemma 2.5). It should also be noted, that the restriction of the type variable `'a` to members of the pcpo class is essential as otherwise the existence of the bottom element that is required in the definition cannot be guaranteed.

## 3.5 Proofs in Isabelle

Isabelle allows us to formalize and prove mathematical statements (lemmas) about structures. Those proofs are then automatically checked by Isabelle. Furthermore, Isabelle includes tools like `sledgehammer` or `nitpick` that can help to find proofs or counter examples. However, the power of these tools is limited.

To demonstrate how poofs in Isabelle work, we will now show that the function `succ_zero` (Section 3.2) applied to a natural number $x$, returns a natural number that is always greater or equal than $x$ This can be formalized and proven as shown below:

```
fun succ_zero :: "nat ⇒ nat" where
"succ_zero 0 = 1" |
"succ_zero x = x"

lemma succ_zero_le: "x ≤ succ_zero x"
  apply (case_tac x)
  apply simp
  by simp
```

The proof of the `succ_zero_le` lemma is conducted by successively applying rules until we have transformed the claim of the lemma into a tautology. This proof strategy is also called backward chaining. After we have successfully conducted the proof of lemma, we can also use it to prove other lemmas.

Assumptions that are necessary for the proof of a lemma can be formalized using the **assumes** and **shows** keyword as follows:

```
lemma succ_zero_eq: assumes "1 ≤ x"
 shows "x = succ_zero x"
  apply (case_tac x)
  using assms apply auto[1]
  by simp
```

Note, that the **shows** keyword is necessary to separate the actual claim from the assumptions of the lemma. In the proof, assumptions can then be referenced using `assms` keyword.

Alternatively we can also express the lemma above in a more concise manner:

```
lemma succ_zero_eq:
  "⟦1 ≤ x⟧⟹x = succ_zero x"
  <<proof>>
```

As we can see, such proofs are not easily readable, since intermediate steps are not explicitly visible. Furthermore, the underlying proof principle of backward reasoning can also make the proofs hard to understand. To overcome these issues, a new proof language Isar [Wen02] was introduced that allows conducting proofs in a more human readable manner. For instance, the `succ_zero_le` can be proven with Isar as shown below:

```
lemma succ_zero_ge: "x ≤ succ_zero x"
proof(cases "x = 0")
  case True
  thus ?thesis
    by simp
next
  case False
  thus ?thesis
    by (metis False eq_iff succ_zero.elims)
qed
```

As Isar provides means to efficiently prove and handle large proofs, we will use it extensively in our theories. However, we will often abbreviate the proofs of fully verified lemmas with `<<proof>>` for the sake of readability.

# Chapter 4

# Extensions of HOLCF

During the creation of our framework, we proved a number of general theorems. To simplify future reuse, we decided to outsource them in separate theories. In the following we will present three of those theories namely `Prelude`, `SetPcpo` and `LNat`. For each theory we will briefly explain the main results and leave the rest, as well as the proofs, to be found in Appendix A for the interested reader. Based on the theories in this chapter, we will then present the implementation of streams in the next chapter.lly valid lemmas.

## 4.1 Prelude

The `Prelude` theory directly imports HOLCF's [Reg94; Huf12] main theory facade and decorates it with frequently used lemmas. In this section, we will present the most frequently used functions, and explain some key theorems which will be needed later to implement the streams. The full set of ca. 60 theorems in `Prelude` and their proofs can be found in the Appendix A.

| Notation | Signature | Functionality |
|---|---|---|
| | rel2map: $\wp(M \times N) \Rightarrow (M \rightharpoonup N)$ | convert relation to function |
| | literate: $\mathbb{N} \Rightarrow (M \Rightarrow M) \Rightarrow M \Rightarrow [M]$ | create list by iterating |
| $\alpha.l$ | lrcdups: $[M] \Rightarrow [M]$ | remove duplicates from a list |
| | getinj: $\wp(M) \Rightarrow \mathbb{N} \Rightarrow \wp(\mathbb{N} \times M)$ | enumerate elements of a set |
| $M_\perp$ | updis: $M \rightarrow M_\perp$ | make an arbitrary type flat |
| | upApply: $(M \Rightarrow N) \Rightarrow (M_\perp \rightarrow N_\perp)$ | transfer function to flattend type |
| | upApply2: $(M \Rightarrow N \Rightarrow O) \Rightarrow (M_\perp \rightarrow N_\perp \rightarrow N_\perp)$ | like upApply, with two inputs |

Table 4.1: Functions defined in `Prelude`; M, N, O are arbitrary types, $_\perp$ denotes flat orders, $\rightharpoonup$ denotes partial functions

The first theorem we present simplifies continuity proofs and gives another intuition how the concepts of admissibility, monotonicity and continuity are related with each other:

```
lemma adm2cont:
  fixes f:: "'a::cpo ⇒ 'b::cpo"
  assumes "monofun f" and "⋀k. adm (λY. (f Y) ⊑ k)"
```

```
  shows "cont f"
  <<proof>>
```

For continuity proofs it has furthermore proven useful to add another continuity introduction lemma based on the `contI` lemma [Huf12]:

```
lemma contI2:
  "⟦monofun (f::'a::cpo ⇒ 'b::cpo);
      (∀Y. chain Y⟶f (⨆i. Y i) ⊑ (⨆i. f (Y i)))⟧⟹cont f"
  <<proof>>
```

## 4.2 Properties of Set Orderings

Some functions on streams return sets. To define them as continuous functions, we need to show some properties of the inclusion order on countable sets. Due to the duality of sets and predicates, it has also proven useful to define the implication-relation as an order on Boolean values as well. In this section the primary results regarding the two above-mentioned orders is to show that they are pcpo's. The full set of ca. 15 theorems in `SetPcpo` and the corresponding proofs can be found in the Appendix A.2.

We first show that inclusion on sets is a partial order by instantiating the set type as a member of the `po` class:

```
instantiation set :: (type) po
begin
  definition less_set_def: "(op ⊑) = (op ⊆)"
instance
  <<proof>>
end
```

Furthermore, we can also show that for a chain of sets the union of all chain elements is the least upper bound of the set:

```
lemma Union_is_lub: "A <<| ⋃ A"
  <<proof>>
```

Another variant of the lemma above is to show that the `lub` and `Union` operator on sets are equal:

```
lemma lub_eq_Union: "lub = Union"
  <<proof>>
```

Now we can show that the inclusion order on sets is complete:

```
instance set :: (type) cpo
  <<proof>>
```

Sets are also pcpo's, pointed with the empty set as the minimal element.

```
instance set :: (type) pcpo
  <<proof>>
```

For sets the bottom element is indeed equal to the empty set.

```
lemma UU_eq_empty: "⊥ = {}"
  <<proof>>
```

After we have now successfully shown that the inclusion order on sets is pcpo, we can do the same for Boolean values.

As before we first prove that the order on Boolean values is a po:

```
instantiation bool :: po
begin
  definition less_bool_def: "(op ⊑) = (op⟶)"
instance
  <<proof>>
end
```

Chains of Boolean values are always finite:

```
instance bool :: chfin
<<proof>>
```

So there always exists a chain element that is equal to the least upper bound of the chain of Boolean values.

As a direct consequence we now know that every chain has a least element, since every chain consists of at least one element. Thus, the ordering on the Boolean values also forms a cpo:

```
instance bool :: cpo ..
```

Here, the two points `..` show that the correctness is immediately recognized.

The order on Boolean values is also pointed with `False` acting as the minimal element.

```
instance bool :: pcpo
  <<proof>>
```

This enables us to prove a set of useful theorems about admissible predicates, such as

```
lemma adm_in: "adm (λA. x ∈ A)"
  <<proof>>
```

## 4.3 Lazy Natural Numbers

To encode the length of possibly infinitely long streams, we must create a data type for natural numbers with a top element (infinity). For that purpose, we extend the already existing theory of lazy natural numbers (lNats) in the `LNat` theory. The full set of the defined functions and approximately 100 theorems along with their proofs can be found in the Appendix A.3.

### 4.3.1 Definition

We can define the `lnat` data type using the **domain** constructor:

```
domain lnat = lnsuc (lazy lnpred::lnat)
```

As mentioned earlier this also automatically adds a bottom element $\perp$ to the data type and establishes a pointed complete partial order $\sqsubseteq$. Furthermore, the `lnpred` destructor is defined which serves as an inverse function of the `lnsuc` constructor function. We can then show that the order on `lnat` is total and has 0 as its least element by instantiating `Lnat` as a member of the `ord` and `zero` class.

```
instantiation lnat :: "{ord, zero}"
begin
  definition lnzero_def: "(0::lnat) ≡ ⊥"
  definition lnless_def: "(m::lnat) < n ≡ m ⊑ n ∧ m ≠ n"
  definition lnle_def:   "(m::lnat) ≤ n ≡ m ⊑ n"
instance ..
end
```

We define `lntake` as an abbreviation for `lnat take`, which is generated by the **domain** package. To conveniently denote such natural numbers in our theories, we define two helper functions `Inf'` and `Fin`.

The function `Inf'` (abbreviated with $\infty$) is a nullary function that returns the maximum of all elements in the `lnat` type.

```
definition Inf'  ::  "lnat"   ("∞") where
"Inf' ≡ fix·lnsuc"
```

As we can see it is defined as the fixed point over the continuous successor function `lnsuc`. This is possible since infinity is the only, and hence also the least fix point of the successor function.

For finite numbers we define the helper function `Fin` that given a natural number $n$ returns the corresponding lazy natural number.

```
definition Fin   ::  "nat ⇒ lnat" where
"Fin k ≡ lntake k·∞"
```

Another useful and frequently used function is `lnmin` which determines the minimum of the given two `lnat`.

```
definition lnmin ::  "lnat → lnat → lnat" where
"lnmin ≡ fix·(Λ h. strictify·(Λ m. strictify·(Λ n.
                    lnsuc·(h·(lnpred·m)·(lnpred·n)))))"
```

### 4.3.2 Properties of the Data Type

Since we have now defined the basics of the `lnat` type, we can evaluate its properties. We begin with the fundamental property that the order corresponds to the order on the `nat` type:

```
lemma less2nat_lemma "∀k. (Fin n ≤ Fin k) ⟶ (n ≤ k)"
  <<proof>>
```

Furthermore, we can prove some basic properties of the order like reflexivity and transitivity:

```
lemma refl_lnle: "(x::lnat) ≤ x"
  <<proof>>

lemma trans_lnle: "⟦x ≤ y; y ≤ z⟧ ⟹ (x::lnat) ≤ z"
  <<proof>>
```

Also for every element in an infinite `lnat` chain, we can find a bigger element in the same chain.

```
lemma inf_chainl2:
"⟦chain Y; ¬ finite_chain Y⟧ ⟹ ∃j. Y k ⊑ Y j ∧ Y k ≠ Y j"
  <<proof>>
```

To demonstrate the closure of the structure, we can show that the least upper bound of any infinite `lnat` chain is $\infty$.

```
lemma unique_inf_lub: "⟦chain Y; ¬ finite_chain Y⟧ ⟹ Lub Y = ∞"
  <<proof>>
```

Furthermore, and to prove the distinctness between maximum and minimum elements in sets, we show that the order on the type is a linear order where for each two elements $x$ and $y$ either $x \leq y$ or $y \leq x$ holds.

```
instantiation lnat :: linorder
begin
  instance
  <<proof>>
end
```

We can also prove that the `lnat` data type belongs to the `wellorder` class. This class includes all types where every non-empty subset of that type possesses a least element. To prove the membership we have to show that the order on the type is linear and satisfies the following predicate:

```
lemma lnat_well:
assumes "(⋀x. (⋀y. y < x ⟹ P y) ⟹ P x) ⟹ P a"
shows "P a"
  <<proof>>
```

After proving some auxiliary lemmas, we can finally show:

```
instance lnat :: wellorder
  <<proof>>
```

As we will see later, this property is of particular importance in our definition of SB lengths (c.f. Section 6.5).

# Chapter 5

# Streams

This chapter introduces the concept of streams in FOCUS [BS01] along with important functions, properties, as well as proof methods in Isabelle. Furthermore, we will also give an introduction to our stream formalization in Isabelle.

Streams are used to model communication channels of components in interactive and distributed systems. Formalizing these as finite or infinite streams over a pcpo allows formal verification of such components and their interaction behavior. We recall our view of a component as a computation unit that nonstop processes messages. Such a component then has a well-defined interface, consisting of input and output channels as well as a behavior.

We come back to our running example. The addition component (add, or +) has two input channels accepting natural numbers as well as one output channel. In a functional language like Haskell the behavior of the addition component can then be described as follows:

```
add :: [Nat] → [Nat] → [Nat]
add (x:xs) (y:ys) = (x + y) : add (xs) (ys)
```

It should be noted that in this example we used the built-in list data type instead of our own stream data type to improve the understandability. However, as we will see in Section 5.2.4 the list and our stream data type are closely related to each other.

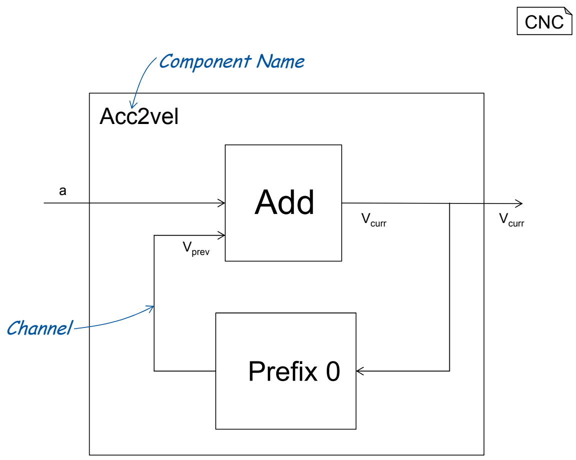


Figure 5.1: Running Example: Cruise Control

## 5.1 Mathematical Definition and Construction

Let $M$ be a set of possible messages (e.g. $M = \mathbb{N}$).

**Definition 5.1.** The set of all finite streams over this domain is denoted by:

$$M^* := \{\epsilon\} \cup \{\langle m_1, m_2, \ldots, m_i \rangle \mid \forall j \in [1, i].\ m_j \in M\}$$

Since we also want to model interactive systems as well as verify their behavior, we also need to introduce infinite streams.

**Definition 5.2.** The set of all infinite streams over $M$ is

$$M^\infty := \{\langle m_1, m_2, \ldots \rangle \mid m_i \in M\}$$

Furthermore, we define $M^\omega := M^* \cup M^\infty$ to be the set of all (untimed) streams. Please note that $M^\omega$ completes $M^*$ to a cpo with respect to prefixing.

To construct streams, the concatenation operator $\bullet$ with signature $M^\omega \Rightarrow M^\omega \Rightarrow M^\omega$ is defined. Based on the concatenation operator on streams, we can define an ordering $\sqsubseteq$ on the set of streams.

**Definition 5.3.** The prefix ordering [Rum96] on streams $\sqsubseteq$ is defined such that the following holds:

$$\forall x, y \in M^\omega.\ x \sqsubseteq y \Leftrightarrow \exists s \in M^\omega.\ \ x \bullet s = y$$

Then we can deduce the following.

**Lemma 5.1.** The prefix ordering $\sqsubseteq$ is a pcpo on the set of streams with the empty stream as its least element, denoted by $\epsilon$ [Rum96].

### 5.1.1 Properties of Streams

We will now discuss fundamental properties of streams and selected functions defined on them. The next lemma allows us to reduce equality proofs for infinite streams to equality proofs for finite streams and induction on the natural numbers.

**Lemma 5.2** (Take lemma)**.** Two streams are equal if their prefixes of arbitrary length are equal:

$$(\forall n.\text{take } n\ s_1 = \text{take } n\ s_2) \Rightarrow s_1 = s_2$$

The take lemma will be useful for induction on streams. For induction on streams, many concepts presented so far come together, more specifically the take lemma, admissibility, domain theory, as well as properties of and functions on streams.

**Lemma 5.3** (Induction on finite Streams)**.** For finite streams we can formulate the induction rule for a predicate $P$ as follows:

$$(P\ \bot \wedge (\forall a, s.P\ s \Rightarrow P\ (a : s))) \Rightarrow \forall x \in M^*.\ P\ x$$

Thus, if the predicate $P$ holds for the bottom element, and we can prepend elements to a stream without losing validity of $P$, then $P$ also holds for all finite streams.

**Lemma 5.4** (Induction on infinite Streams)**.** For infinite streams, we must require admissibility of the predicate $P$ for the induction to work. This way, it is ensured that $P$ remains valid when taking the least upper bound of a chain of streams:

$$(\text{adm}(P) \wedge P\ \bot \wedge (\forall a, s.P\ s \Rightarrow P\ (a : s))) \Rightarrow \forall x \in M^\omega.\ P\ x$$

These induction rules rely on the take lemma and the properties of stream concatenation. A more detailed discussion can be found in [GR06; Huf12].

## 5.2 Streams in Isabelle

The `stream` data type is defined using the **`domain`** constructor as shown below:

```
domain 'a stream = lsconc (lshd :: "'a discr u")
                   (lazy srt :: "'a stream") (infixr "&&" 65)
```

Please note an implementation detail: to make sure that the constructors and their inverse functions are continuous (which means we can make use of a rich lemmata library for automatic reasoning about continous functions), the alphabet of messages itself (thus not just the data type of streams) is given a bottom element (which will not play any role in the rest of this work) and also a flat ordering(since continuous functions are defined on pcpo's). Here, each stream element is of the type `'a discr u`, where `discr` enhances a type with a discrete ordering (making it a cpo) and `u` (indicating "up") lifts the type to a pcpo. Analogue to the previous section, we also define an infix abbreviation $\&\&$ for the `lsconc` constructor. Furthermore, we specify the selectors `lshd` and `srt` to access the head and rest of a stream.

As already mentioned in Section 3.3, the domain constructor automatically defines a bottom element of the data type. To avoid confusion we will denote this bottom element, which we also refer to as the empty stream, by $\epsilon$.

In the following Tables 5.1 and 5.2 about 40 of the most commonly used functions on streams are listed. Furthermore, we also proved about 600 lemmas during the creation of the `Stream` theory. The most important ones will be introduced in this chapter without proofs. The complete set of lemmas and all the proofs can be found in Appendix B. Some of our implemented functions are the implementations of the signatures of [RR11], while others were necessary during the verification of several case studies.

To improve the readability, we introduce type synonyms for continuous function from streams to streams.

```
type_synonym ('a, 'b) spf = "('a stream → 'b stream)"
type_synonym 'm spfo = "('m, 'm) spf"
```

Further helpful functions, which do not belong directly to the API, are defined. They are listed in table 5.2.

### 5.2.1 Running Example: The Addition-Component

To define the addition component from fig. 5.1 by construction from an elementary function, we introduce the following definitions. Similar to `map` on lists, we introduce a function `smap` which applies a function to all elements of a stream:

```
definition smap :: "('a ⇒ 'b) ⇒ ('a,'b) spf" where
"smap f ≡ fix·(Λ h s. (↑(f (shd s)) • (h·(srt·s))))"
```

To simplify the definition of components with multiple input channels, we introduce `szip`, which enables us to zip two streams into one:

```
definition szip :: "'a stream → 'b stream → ('a × 'b) stream" where
"szip ≡ fix·(Λ h s1 s2. ↑(shd s1, shd s2) •
    (h·(srt·s1)·(srt·s2))))"
```

We furthermore create the `merge` function, which takes as input a function `f` and two streams `s1` and `s2`, and merges their elements according to `f`:

```
definition merge::"('a  ⇒ 'b ⇒ 'c) ⇒ 'a stream
                   → 'b stream → 'c stream" where
"merge f ≡ Λ s1 s2. smap (λ s3. f (fst s3) (snd s3))·(szip·s1·s2)"
```

Here `snd` is a selector that returns the second element of a stream.

Now we can easily define the add-function that essentially applied the elementary plus-operation to two streams using the `merge` operator:

```
definition add:: "nat stream → nat stream → nat stream" where
"add = merge plus"
```

Let $s, s' \in M^\omega, m \in M, n \in \mathbb{N}_\infty, A \subseteq M$. As before continuous function are denoted by $\rightarrow$, $\Rightarrow$ denotes non continuous functions and $\rightharpoonup$ partial functions:

| Notation | Signature | Functionality |
|---|---|---|
| $\varepsilon$ | sbot: $M^\omega$ | empty stream |
| $m{:}s$ | lscons: $M \times M^\omega \rightarrow M^\omega$ | append first element |
| $s \sqsubseteq s'$ | below: $M^\omega \times M^\omega \rightarrow \mathbb{B}$ | prefix relation |
| $\uparrow$ | sup': $M \Rightarrow M^\omega$ | construct a stream by a single element |
| $s\|_n$ | stake: $\mathbb{N}_\infty \Rightarrow M^\omega \rightarrow M^\omega$ | retrieve the first $n$ elements of a stream |
| $s \bullet s'$ | sconc: $M^\omega \Rightarrow M^\omega \rightarrow M^\omega$ | concatenation of streams |
| | sValues: $M^\omega \rightarrow \wp(M)$ | the set of all messages in a stream |
| | shd: $M^\omega \Rightarrow M$ | first element of stream |
| | srt: $M^\omega \rightarrow M^\omega$ | stream without first element |
| $\#s$ | slen: $M^\omega \rightarrow \mathbb{N}_\infty$ | length of stream |
| | sdrop: $\mathbb{N}_\infty \Rightarrow M^\omega \rightarrow M^\omega$ | remove first $n$ elements |
| $s.n$ | snth: $\mathbb{N} \Rightarrow M^\omega \Rightarrow M$ | $n^{th}$ element of stream |
| $s \cdot n$ | sntimes: $\mathbb{N}_\infty \Rightarrow M^\omega \Rightarrow M^\omega$ | stream iterated $n$ times |
| $s \cdot \infty$ | sinftimes: $M^\omega \Rightarrow M^\omega$ | stream iterated $\infty$ times |
| | smap: $(I \Rightarrow O) \Rightarrow I^\omega \rightarrow O^\omega$ | element-wise function application |
| | siterate: $(M \Rightarrow M) \Rightarrow M \Rightarrow M^\omega$ | infinite iteration of function |
| $A \ominus s$ | sfilter: $\wp(M) \Rightarrow M^\omega \rightarrow M^\omega$ | filtering function |
| | stakewhile: $(M \Rightarrow \mathbb{B}) \Rightarrow (M^\omega \rightarrow M^\omega)$ | prefix where predicate holds |
| | sdropwhile: $(M \Rightarrow \mathbb{B}) \Rightarrow (M^\omega \rightarrow M^\omega)$ | drop prefix while predicate holds |
| | szip: $I^\omega \rightarrow O^\omega \rightarrow (I \times O)^\omega$ | zip two streams into one stream |
| $\alpha.s$ | srcdups: $M^\omega \rightarrow M^\omega$ | remove consecutive duplicates |
| | fup2map: $(I \Rightarrow O_\perp) \Rightarrow (I \rightharpoonup O)$ | conversion of f. to partial f. |
| | slookahd: $I^\omega \rightarrow (I \Rightarrow O) \rightarrow O$ | apply function to head of stream |
| | sfoot: $M^\omega \Rightarrow M$ | last elem. of not empty, finite stream |
| | merge: $(I \Rightarrow O \Rightarrow M) \Rightarrow I^\omega \rightarrow O^\omega \rightarrow M^\omega$ | merges streams acc. to the function |
| | sprojfst: $(I \times O)^\omega \rightarrow I^\omega$ | first stream of two zipped streams |
| | sprojsnd: $(I \times O)^\omega \rightarrow O^\omega$ | second stream of two zipped streams |
| | stwbl: $(M \Rightarrow \mathbb{B}) \Rightarrow (M^\omega \rightarrow M^\omega)$ | stakewhile + first violating element |
| | srtdw: $(M \Rightarrow \mathbb{B}) \Rightarrow (M^\omega \rightarrow M^\omega)$ | dropwhile and then remove head |
| | sscanl: $(O \Rightarrow I \Rightarrow O) \Rightarrow O \Rightarrow (I^\omega \rightarrow O^\omega)$ | state-based specifications |
| | siterateBlock: $(M^\omega \Rightarrow M^\omega) \Rightarrow M^\omega \Rightarrow M^\omega$ | alternative definition similar to siterate |

Table 5.1: API: Operations on untimed streams

We check the correctness of the addition-function on streams by proving the following lemma. The $n$-th element of the stream created by applying `add` to two streams $s_1$ and $s_2$ is the same as adding up the $n$-th elements of $s_1$ and $s_2$:

```
lemma add_snth: "Fin n <#xs⟹Fin n < #ys⟹
   snth n (add·xs·ys) = snth n xs + snth n ys"
```

| Notation | Signature | Functionality |
|---|---|---|
| | s2list: $M^\omega \Rightarrow M^*$ | convert a stream to a list |
| | slpf2spf: $(I^* \Rightarrow O^*) \Rightarrow (I^\omega \rightarrow O^\omega)$ | list-processing f. to stream-proc. f. |
| | sislivespf: $(I^\omega \rightarrow O^\omega) \Rightarrow \mathbb{B}$ | liveness predicate for SPFs |
| | sspf2lpf: $(I^\omega \rightarrow O^\omega) \Rightarrow (I^* \Rightarrow O^*)$ | stream-processing f. to a list-proc. f. |
| | add: $\mathbb{N}^\omega \rightarrow \mathbb{N}^\omega \rightarrow \mathbb{N}^\omega$ | element-wise addition function |
| $< \cdot >$ | list2s: $M^* \Rightarrow M^\omega$ | convert list to stream |
| | niterate: $\mathbb{N} \Rightarrow (M \Rightarrow M) \Rightarrow (M \Rightarrow M)$ | helper function for siterate |

Table 5.2: Further operations on untimed streams

```
  <<proof>>
```

### 5.2.2 The Take-Functional and Induction on Streams

Since we used the `domain` constructor to define the `stream` data type, a take function is automatically created. For the stream definition above, the continuous function

```
stream_take :: nat ⇒ 'a stream → 'a stream
```

is generated, where `stream_take n s` returns a stream consisting of the first $n$ elements of the stream $s$. For instance, `stream_take 0 s` returns $\epsilon$, and `stream_take 3 s` evaluates to $m_1$ : $m_2$ : $m_3$ : $\epsilon$.

To increase the readability, we define an abbreviation for the take function on streams as shown below:

```
abbreviation stake :: "nat ⇒ 'a spfo" where
"stake ≡ stream_take"
```

A stream is then equal to the least upper bound of its prefixes.

```
lemma reach_stream: "(⨆i. stake i·s) = s"
  <<proof>>
```

Furthermore, the `stake` operator is monotonic in its first argument.

```
lemma stake_mono: assumes "i ≤ j"
  shows "stake i·s ⊑ stake j·s"
  <<proof>>
```

This result is of particular importance in the proof of the induction over stream length rule.

```
lemma ind:
"⟦adm P; P ϵ; ⋀a s. P s ⟹P (↑a • s)⟧⟹P x"
  <<proof>>
```

### 5.2.3 Concatenation of Streams

Another important function is the `concatenation` operator on streams:

```
definition sconc :: "'a stream ⇒ 'a stream → 'a stream" where
"sconc ≡ fix·(Λ h. (λ s1. Λ s2.
                    if s1 = ϵ then s2 else  (lshd·s1) && (h
                       (srt·s1)·s2)))"
```

We can show that the operator is continuous in its second argument but please note that is is not continuous in its first argument.

```
lemma cont_sconc:
  "⋀s1 s2.
     cont (λh. if s1 = ϵ then s2 else (lshd·s1) && (h (srt·s1)·s2))"
  <<proof>>
```

We abbreviate the concatenation operator with • and can show that the concatenation of streams is associative:

```
lemma assoc_sconc: "(s1•s2)•s3 = s1•(s2•s3)"
  <<proof>>
```

If a stream is nonempty, the concatenation of its head and rest delivers a stream that is equal to the original one:

```
lemma surj_scons: "x≠ϵ ⟹↑(shd x) • (srt·x) = x"
  <<proof>>
```

An operator that infinitely concatenates a stream with itself can be defined as shown below:

```
definition sinftimes  :: "'a stream ⇒ 'a stream" ("_∞") where
"sinftimes ≡ fix·(Λ h. (λs.
                        if s = ϵ then ϵ else (s • (h s))))"
```

As a result we can now easily define infinite streams consisting only of the message 1 as follows:

```
definition s2 :: "nat stream" where
"s2 = <[1]>∞"
```

### 5.2.4 Reusing List Theories

To simplify the instantiation of streams, we define a function `list2s`, with brackets as an abbreviation, that converts the built-in lists from Isabelle into streams. If the list is empty, the empty stream is returned:

```
primrec list2s :: "'a list ⇒ 'a stream" where
list2s_0:   "list2s [] = ϵ" |
list2s_Suc: "list2s (a#as) = updis a && (list2s as)"
```

```
definition s1 :: "nat stream" where
"s1=<[1,2,3]>"
```

Based on `list2s`, we can also define a partial order on the `list` data type using `list2s`, and the prefix ordering on streams:

```
instantiation list :: (countable) po
begin
  definition sq_le_list:
    "s ⊑ t ≡ (list2s s ⊑ list2s t)"
  instance
  <<proof>>
end
```

Concatenating streams corresponds to the concatenation of lists:

```
lemma listConcat: "<l1> • <l2> = <(l1 @ l2)>"
  <<proof>>
```

To convert back from a stream to a list, we introduce another function `s2list`. If a stream has infinite length, the result is undefined:

```
definition s2list :: "'a stream ⇒ 'a list" where
"s2list s ≡ if #s ≠ ∞ then SOME l. list2s l = s else undefined"
```

Also, the function `s2list` is left-inverse to `list2s`, which means that converting a list into a stream and afterwards re-converting this stream into a list results in the original list:

```
lemma "s2list (list2s l) = l"
  <<proof>>
```

To convert list-processing functions into stream-processing function, we define `slpf2spf`:

```
definition slpf2spf ::"('in,'out) lpf ⇒ ('in,'out) spf" where
"slpf2spf f ≡
    if monofun f
       then Λ s. (⨆k. list2s (f (s2list (stake k·s))))
       else undefined"
```

A monotonic list-processing function induces a monotonic stream-processing function by applying it to the $k$ messages long prefix of the stream.

```
lemma mono_slpf2spf:
"monofun f⟹monofun (λs. list2s (f (s2list (stake k·s))))"
  <<proof>>
```

Finally, an important result is also that `slpf2spf` is continuous:

```
lemma slpf2spf_cont:
"monofun f⟹
     (Λ s. (⨆k. list2s (f (s2list (stake k·s))))·s
          = (⨆k. list2s (f (s2list (stake k·s))))"
  <<proof>>
```

### 5.2.5 The Length Operator

Another typical operator on lists is the length-operator. For streams, it is defined as the number of its elements/messages or $\infty$ for infinite streams:

```
definition slen :: "'a stream → lnat" where
"slen ≡ fix·(Λ h. strictify·(Λ s. lnsuc·(h·(srt·s))))"
```

We can define the length operator even more elegantly as a composition of continuous functions [Huf12] using the `fixrec` keyword:

```
fixrec slen2 :: "'a stream → lnat" where
"slen2·⊥ = ⊥" |
"x≠⊥⟹slen2·(x&&xs) = lnsuc·(slen2·xs)"
```

As a result the function is automatically continuous: After proving some auxiliary lemmas, we can show that both definitions are equivalent:

```
lemma slen_eq: "slen2 = slen"
  <<proof>>
```

Since continuity implies montonicity, we can show that the length function for streams is monotonic:

```
lemma mono_slen: "x ⊑ y⟹#x ≤ #y"
  <<proof>>
```

Besides these technical properties we can also evaluate that the length operator works as expected. Appending a stream consisting of only one element increases the length by 1:

```
lemma slen_scons: "#(↑a•as) = lnsuc·(#as)"
  <<proof>>
```

Finally, if the stream has infinite length, appending elements to the stream does not change the stream.

```
lemma sconc_fst_inf: "#x= ∞ ⟹ x•y = x"
  <<proof>>
```

### 5.2.6 The Domain Operator

To retrieve the set of all messages in a stream, we define the operator `sValues` using the `snth` function which, as described in the table above, retrieves the $n$-th element of a stream:

```
definition sValues :: "'a stream → 'a set" where
"sValues ≡ Λ s. {snth n s | n. Fin n < #s}"
```

We can the show that the function is continuous:

```
lemma "cont (λs. {snth n s | n. Fin n < #s})"
  <<proof>>
```

The message domain of the concatenation of streams is the union of the respective message domains:

```
lemma svalues_sconc2un: "#x = Fin k⟹sValues·(x • y) = sValues·x ∪
    sValues·y"
  <<proof>>
```

### 5.2.7 Defining Functions with Explicitly Memorized State

To define state-based functions, we define `sscanl` as the least upper bound of the corresponding primitive recursive function. This **primrec** function takes a natural number (indicating the number of elements to scan), a reducing function, an initial element, and an input stream. It then returns a stream consisting of the partial reductions of the input stream:

```
primrec SSCANL :: "nat ⇒ ('o ⇒ 'i ⇒ 'o) ⇒ 'o ⇒ 'i stream ⇒ 'o
    stream" where
"SSCANL 0 f q s = ϵ" |
"SSCANL (Suc n) f q s =
     (if s=ϵ
         then ϵ
         else ↑(f q (shd s)) • (SSCANL n f (f q (shd s)) (srt·s)))"
```

We obtain the scanline function with its usual signature by taking the least upper bound of the function above. It behaves similar to `map`, but also takes the previously generated output element as additional input to the function. For the first computation, an initial value is provided:

```
definition sscanl :: "('o ⇒ 'i ⇒ 'o) ⇒ 'o ⇒ ('i, 'o) spf" where
"sscanl f q ≡ Λ s. ⨆i. SSCANL i f q s"
```

So the first argument is the reducing function, which can be for example be $+$. The second parameter is the initial value, and the third argument the input stream. We demonstrate the definition by defining a helper function, which returns the $n$-th element of the output of `scanline`:

```
primrec sscanl_nth :: "nat ⇒ ('a ⇒ 'a  ⇒ 'a) ⇒ 'a  ⇒ 'a stream ⇒
    'a" where
"sscanl_nth 0 f q s =  f q (shd s)" |
"sscanl_nth (Suc n) f q s = sscanl_nth n f (f q (shd s)) (srt·s)"
```

We can now show that it corresponds to the output of the original `sscanl` function:

```
lemma sscanl2sscanl_nth:
"Fin n<#s⟹snth n (sscanl f q·s) = sscanl_nth n f q s"
  <<proof>>
```

After proving some auxiliary lemmas, we can then show the continuity of `sscanl`:

```
lemma cont_lub_SSCANL: "cont (λs. ⨆i. SSCANL i f q s)"
  <<proof>>
```

### 5.2.8 Map, Filter, Zip, Project, Merge and Removing Duplicates

The function `smap` on streams, which we defined while introducing our running example, works similarly to the map-function for lists:

```
lemma smap2map: "smap g·(<ls>) = <(map g ls)>"
  <<proof>>
```

Applying smap in two passes, first $h$ is applied, afterwards $g$ is equivalent to mapping $g \circ h$ in a single pass.

```
lemma smaps2smap: "smap g·(smap h·xs) =  smap (λ x. g (h x))·xs"
  <<proof>>
```

The `smap` function is a homomorphism on streams with respect to concatenation:

```
lemma smap_split: "smap f·(a • b) = (smap f·a) • (smap f·b)"
  <<proof>>
```

For multiple specifications it has proven useful to introduce a function `sfilter`, which can be used to filter elements from a stream. Given a set and a stream as input, the functions removes all elements from the stream which are not contained in the set:

```
definition sfilter :: "'a set ⇒ 'a spfo" where
"sfilter M ≡ fix·(Λ h s. slookahd·s·(λ a.
                (if (a ∈ M) then ↑a • (h·(srt·s)) else h·(srt·s))))"
```

Applying the message filter function twice with $M$ and $S$ as message sets is equivalent to applying it once with $M \cap S$ as the message set:

```
lemma int_sfilterl1: "sfilter S·(sfilter M·s) = sfilter (S ∩ M)·s"
  <<proof>>
```

We also introduce `sprojfst` which returns the first stream of two zipped streams:

```
definition sprojfst :: "(('a × 'b),'a) spf" where
"sprojfst ≡ Λ x. smap fst·x"
```

If the stream has infinite length, `sprojfst` applied to the two zipped streams returns the first stream:

```
lemma sprojfst_szipl1:
"∀x. #x = ∞⟶sprojfst·(szip·i·x) = i"
  <<proof>>
```

Particularly in telecommunication applications it has been proven useful to introduce the function `srcdups`, which removes successive duplicates from a stream:

```
definition srcdups :: "'a spfo" where
"srcdups ≡ fix·(Λ h s. slookahd·s·(λ a.
           ↑a •  h·(sdropwhile (λ z. z = a)·(srt·s))))"
```

For example:

```
srcdups ⟨[1,2,2,3]⟩ = ⟨[1,2,3]⟩
```

The $n$-th element of two merged streams after applying a function $f$ is the same as applying $f$ to the $n$-th elements of the two single streams.

```
lemma merge_snth:
"Fin n <#xs⟹Fin n < #ys
   ⟹snth n (merge f·xs·ys) = f (snth n xs) (snth n ys)"
  <<proof>>
```

The duplicate removing function `srcdups` is idempotent:

```
lemma srcdups2srcdups: "srcdups·(srcdups·s) = srcdups·s"
  <<proof>>
```

Finally, we can prove the following equality concerning `srcdups` and `smap`:

```
lemma srcdups_smap_com:
"srcdups·(smap f·(srcdups·s)) = smap f·(srcdups·s)
 ⟹srcdups·(smap f·s)= smap f·(srcdups·s)"
  <<proof>>
```

### 5.2.9 Infinite Streams and Kleene Theorem

The following key lemma `rek2sinftimes` is based on the Kleene-Theorem:

```
lemma rek2sinftimes: assumes "xs = x • xs" and "x≠ϵ"
shows "xs = sinftimes x"
  <<proof>>
```

The infinite repetition of a stream $x$ is the least fixed point of $\Lambda s.\ x \bullet s$:

```
lemma fix2sinf: "fix·(Λ s. x • s) = x ∞"
  <<proof>>
```

## 5.3 Further Kinds of Streams

Some examples of special types of dataflow networks could be [RR11]:

- sensors, control units, and actuators in automobiles exchanging data values and control signals,
- real-time software controlling actions of actuators depending on sensors' data,

- interaction between objects via message passing in object oriented software systems or
- messages transmitted between web services in cloud computing applications.

For some systems, an airbag system, timing is an important requirement. To model time-sensitive systems we thus need a notion of time. In this paragraph we will describe briefly three variants of streams for time-sensitive modeling.

*Variant 1:* A possible formalization of time-sensitive systems is by extending the message alphabet with a dummy element *tick* (denoted as $\surd$) [Rum96]. The blocks between each two ticks are equidistant time intervals, where the duration of an interval can be chosen depending on the modelled system. One variant of this timed model is to allow a finite sequence of messages in each time slice. The messages on each time slice are still ordered, but the time distance between each two consecutive messages on the same time slice is not specified.

*Variant 2:* Another model (time synchronous) allows at most one message per block. This is not appropriate for all forms of timed specifications, because the number of messages per time slice in version 1 is not bounded and thus the number of micro-steps in variant 2, needed to appropriately refine the steps in version 1, is unknown. Second, each fine grained micro-time slice enforces a very fine grained use of $\surd$'s in timed specifications. Delay, e.g. is then much more often to be taken into account and a fair merge is very complex. However, if the time model is appropriate, it can also be represented by another (isomorphic) model: by extending the message alphabet by a dummy element *eps* (denoted as $\sim$), instead of *tick*. In this interpretation we again assume a discrete global clock, but each element of the stream is either a message arriving during one time frame, or an $\sim$ (interpreted here as "no message has arrived", having also the length of one time frame). Variant 2 can technically also be used to model synchronous, permanent signals that change at most every time step. These are e.g. occurring in chips with synchronous clocks.

*Variant 3:* If a signal is permanently available then the special element $\sim$ never occurs. Each stream then is of type $M^\omega$, with the special interpretation that each message represents one time step.

A further variant of timed stream are superdense streams by using $\mathbb{R}_+$ as time axis [Lee16].

We will focus in this work on untimed streams and explain in depth functions manipulating untimed (bundles of) streams.

# Chapter 6

# Stream Bundles

We are interested in modular and compositional modeling of component networks. Modularity means that we can define the behavior and check the correctness of components in isolation. Compositionality means that the use of proper composition operators guarantees the correctness of the whole system when constructed from correct components.

To facilitate composition, we enhance our modeling of component networks by naming channels and defining composition operators which connect channels of the same name and type.

The user can then define the type of a channel via a function which for each channel returns a set of allowed messages, i.e., the domain of the channel type. To model the input (or output) streams of a component, we work with an isomorphic transformation of the tuples of streams (instead of just working on tuples): namely with *mappings* from channel names to streams. Such a *mapping* is then called *stream bundle* [Rum96] if the messages of the streams mapped to the channels are allowed to flow on it. Thus, we can compose components and define generalized composition operators connecting same-named/same-typed channels without worrying about setting preconditions for the interface compatibility.

## 6.1 Mathematical Definition

There are multiple ways to formalize stream bundles (SBs). One approach is to define them as total functions from a specific channel set to streams as shown below.

**Definition 6.1** (Stream Bundle)**.** Let $C$ be a set of channel names, $M_c$ the set of allowed messages for a channel $c \in C$ and $M = \bigcup_{c \in C} M_c$. The stream bundle type is then defined as [Rum96]:

$$C^{\Omega} := \{s \in C \to M^{\omega} \mid \forall c \in C.\ s(c) \in M_c^{\omega}\}$$

Hence, stream bundles are functions that map channel names to streams. We furthermore restrict which types of messages can flow on a channel.

To perform induction over SBs similar to streams, we define a bundle datatype with exactly one message element on each channel.

**Definition 6.2** (Stream Bundle Element)**.** Let $C$ be a set of channel names, $M_c$ the set of allowed messages for a channel $c \in C$ and $M = \bigcup_{c \in C} M_c$. The stream bundle element type is then defined as:

$$C^{\surd} := \{s \in C \to M \mid \forall c \in C.\ s(c) \in M_c\}$$

## 6.2 System specific Datatypes

Components might have different channels and types, even user-defined types like an enum are possible. Hence, there is no general type definition assigning a message type to every possible channel. Thus, the datatypes have to be specific to the system under consideration. A possible simple example system is shown in fig. 6.1. It consists of a temperature sensor component and a guarding sensor that might cause an alarm depending on the temperature. The temperature sensor has no input channels, it depends completely on events outside of our modeled system. It outputs the temperature as an `int`-value. The output channel of the sensor is the input channel of the guard component, it then checks if a temperature threshold is exceeded to then raises an alarm.

The following section introduces two placeholder datatypes that will be used for defining SBs, SPFs and SPSs. The datatypes in this theory are only placeholder types. In concrete system development the placeholder will be refined with concrete definitions. Still, we need an instantiation of these types to define the main parts of the general framework.

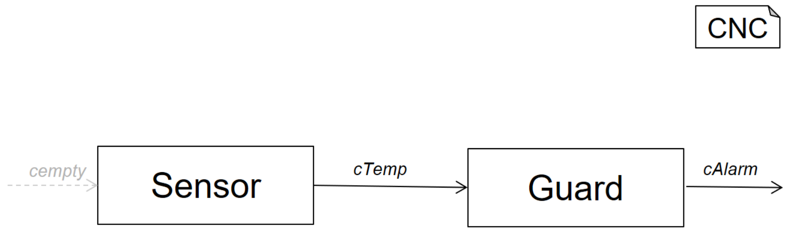



Figure 6.1: Example component network

### 6.2.1 Channel Datatype

The channel datatype is fixed for every system. The temperature alarm system fig. 6.1 would have the channel type `cempty | cTemp | cAlarm`. This datatype contains every used channel and at least one dummy "channel" for defining components with no input or no output channels. The `cempty` element in the channel datatype is a technical work-around since there are no empty types in Isabelle. Thus, even the type of an empty channel set has to contain an element.

For now the channel datatype is defined as only one element:

```
datatype channel = DummyChannel
```

To ensure that the dummy channel type is never used for proving anything not holding over every channel type, the constructor is immediately hidden.

**hide_const** `DummyChannel`

### 6.2.2 Message Datatype

Analogous to the channel datatype, the message datatype contains the messages that channels can transmit. Hence, every kind of message has to be described here. The messages for our sensor system would be defined as $\mathcal{I}$ `int | ` $\mathcal{B}$ `bool`. This message type contains all messages transmittable in a system.

**datatype** `M = DummyMessage`

To ensure that the dummy message type is never used for proving anything not holding for a different message type, the constructor is also immediately hidden.

**hide_const** `DummyMessage`

Since the stream type is used for defining stream bundles and any message type of a stream has to be countable, the globale message datatype has to be instantiated as countable.

**instance** `M :: countable`

In addition, each channel is typed and therefore, can be restricted to allow only a subset of messages from `M` on its stream. Thus, each channel can be mapped to a set of messages from datatype `M`.

**definition** `ctype` :: "`channel ⇒ M set`"

Such a mapping is described by the `ctype` function. Only messages included in the `ctype` are allowed to be transmitted on the respective channel. For the sensor system, channel `c1` would be allowed to transmit all $\mathcal{I}$ `int` and `c2` all $\mathcal{B}$ `bool` messages. The `cempty` channel can never transmit any message, hence, `ctype` of `cempty` would be empty.

We do assume, that there always exists at least one channel, on which no message can flow. Hence, every case-study also has to fulfill this assumption.

**theorem** `ctypeempty_ex: "∃c. ctype c = {}"`

Only with such an assumption we can define an "empty" stream bundle. The possibility to have an empty stream bundle is important for various reasons. Beside being able to define "sensors" and "sinks" as SPFs, also the general composition of components may result in components without in or output channels. Thus, we restrict the user to channel types, that contain a never transmitting channel. A sensor example would be the temperature sensor, a logging component might be described as a sink, because it has no output into the system itself.

### 6.2.3 Domain Classes

In this section we restrict the possible domains of components through the usage of classes. The main idea is to never construct a component which has channels with an empty ctype and channels with non-empty ctype simultaneously fig. 6.2. The domain of such a component would be equivalent to all its channels with non-empty ctype. This restriction is easily achievable by introducing a class which only allows specific subsets of the global channel type. Furthermore, all types for this class will have a domain which excludes all channels with an empty ctype.

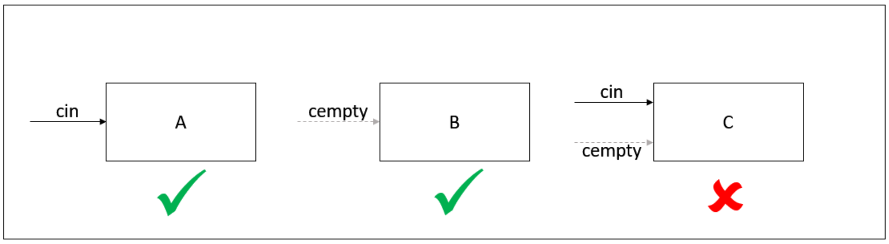



Figure 6.2: Allowed Components

### Preliminaries for Domain Classes

For understandable assumptions in our classes we first define the channel set, that contains all channels with an empty ctype.

```
definition cEmpty :: "channel set" where
"cEmpty = {c. ctype c = {}}"
```

cEmpty contains all channels on which no message is allowed to be transmitted.

### Classes

The first class introduced ensures that a mapping to the global channel type exists. Such a mapping can then be further restricted to limit the possible types exactly to desired types.

```
class rep =
  fixes Rep :: "'a ⇒ channel"
begin
  abbreviation "Abs ≡ inv Rep"
end
```

The following class restricts the mapping from the rep class to be injective and to also comply with our main idea. Through its injectivity, the type is isomorphic to a subset of our channel type.

```
class chan = rep +
  assumes chan_botsingle:
      "range Rep ⊆ cEmpty ∨
       range Rep ∩ cEmpty = {}"
  assumes chan_inj[simp]:"inj Rep"
begin
  theorem abs_rep_id[simp]:"Abs (Rep c) = c"
end
```

With Rep we require a representation function, that maps a type of chan to the channel type. The first class assumption ensures our channel separation and the second the injectivity. Furthermore, our abstraction function Abs is the inverse of Rep. A type of the class chan can be viewed as a subset of channel

#### Class Functions

We will now define a function for types of chan. It returns the Domain of the type. As a result of our class assumptions and of interpreting empty channels as non existing, our domain is empty, if and only if the input type contains channel(s) from cEmpty. A type can be defined as the input of a function by using itself type in the signature. Then, input chDom TYPE ('cs) results in the domain of 'cs.

```
definition chDom::"'cs::chan itself ⇒ channel set" where
"chDom a ≡ range (Rep::'cs ⇒ channel) - cEmpty"
```

The following abbreviation checks, if a type of chan is empty.

```
abbreviation chDomEmpty ::"'cs::chan itself ⇒ bool" where
"chDomEmpty cs ≡ chDom cs = {}"
```

As mentioned before, types of chan can be interpreted as a subset of channels, where on every channel either no message can be transmitted, or on every channel some message is allowed to be transmitted. The properties provided by the framework do use domain assumptions for some properties. The chan class can also be divided in two sub classes that automatically fulfill domain assumptions. Then, many properties of the framework hold immediately without proofing assumptions. This is useful for case-studies because the automatic prover tools can find and use applicable properties easier. Thus, two classes dividing the chan class are defined.

#### Class somechan

Types of somechan can transmit at least one message on every channel.

```
class somechan = rep +
  assumes chan_notempty: "(range Rep) ∩ cEmpty = {}"
      and chan_inj[simp]:"inj Rep"
begin end
```

The class somechan is a subclass of class chan.

```
subclass (in somechan) chan
```

Hence, we know chDom TYPE('c) ≠ {} and chDom TYPE('c) = range Rep.

#### Class emptychan

Types of emptychan can not transmit any message on any channel.

```
class emptychan = rep +
  assumes chan_empty:"(range Rep) ⊆ cEmpty"
      and chan_inj[simp]:"inj Rep"
begin end
```

Analogous to class somechan, also class emptychan is a subclass of class chan.

```
subclass (in emptychan) chan
```

Hence, the Domain is empty.

```
theorem emptychanempty[simp]:"chDomEmpty TYPE('cs::emptychan)"
```

In the following chapters, a "domain" is defined by chDom of a type and domain types are types of class chan, because each type of class chan corresponds to a specific domain.

### 6.2.4 Interconnecting Domain Types

There are three interesting interconnections between domains. Intuitively, the union operator takes all channels from both domains and the minus operator only channels that are in the first, but not the second domain. Since we also have to check for channels from cEmpty, its not that trivial. This would not be necessary, if the type-system of Isabelle would allow empty types.

Furthermore, the type-system of Isabelle has no dependent types which would allow types to be based on their value [Mou+15]. This also effects this framework, because a type 'cs1 ∪ 'cs2 is always different from type 'cs2 ∪ 'cs1, without assuming anything about the definition of ∪. This also makes evaluating types harder. Even type 'cs ∪ 'cs is not directly reducible to type 'cs by evaluating ∪. Of course the same holds for the − type.

#### Union Type

The union of two domains should contain every channel of each domain. So the union of two empty domains should also be empty. But because the type itself can never be empty, we again have to use channels in cEmpty to define the union.

```
typedef ('cs1,'cs2) union (infixr "∪" 20) =
        "if chDomEmpty TYPE ('cs1) ∧ chDomEmpty TYPE ('cs2)
            then cEmpty
            else chDom TYPE('cs1) ∪ chDom TYPE('cs2)"
```

Because channels in cEmpty are interpreted as no real channels, the union of two empty domains is defined as the channel set cEmpty. The next step is to instantiate the union of two members of class chan as a member of class chan. This is rather easy, because either the union results in cEmpty, so there are no channels where a message can be transmitted, or it results in the union of the domains without channels from cEmpty. Hence, the representation function Rep is defined as the representation function Rep_union generated from the typedef-keyword. The output type union type of two input chan types is always a member of chan as shown in following instantiation.

```
instantiation union :: (chan, chan) chan
begin
  definition "Rep == Rep_union"
instance
end
```

After the instantiation, class definition like the `chDom` function can be used. To verify the correctness of our definition we obtain the domain of the union type and prove, that it is indeed the union of the two sub domains.

```
theorem chdom_union[simp]:"chDom TYPE('cs1 ∪ 'cs2) =
                           chDom TYPE ('cs1) ∪ chDom TYPE('cs2)"
```

### Minus Type

Subtracting one domain from another results in the empty domain. But analogous to the union, our resulting type always contains channels. Subtracting a set from one of its subsets would result in an empty type. Hence, our result for this case is again `cEmpty`.

```
typedef ('cs1,'cs2) minus (infixr "-" 20) =
        "if chDom TYPE('cs1) ⊆ chDom TYPE('cs2)
             then cEmpty
             else chDom TYPE('cs1) - chDom TYPE('cs2)"
```

The result from the subtraction of two `chan` types is also in the class. The proof is, like above, straightforward.

```
instantiation minus :: (chan, chan) chan
begin
  definition "Rep == Rep_minus"
instance
end
```

For verifying the minus operator we again take a look at the resulting domain in the following theorem.

```
theorem chdom_minus[simp]:"chDom TYPE('cs1 - 'cs2) =
                           chDom TYPE ('cs1) - chDom TYPE('cs2)"
```

If we subtract domain `'cs2` from domain `'cs1` the resulting domain should contain no channels from `'cs2`.We also verify this correctness property.

```
theorem [simp]:"chDom TYPE('cs1 - 'cs2) ∩ chDom TYPE ('cs2) = {}"
```

## 6.3 Stream Bundle Elements

Before we define the stream bundle (SB) datatype, we define a type for stream bundle elements. The difference between both types is, that SBs map channels to streams but a stream bundle element maps channels to a single message.
A stream bundle element is a function from a `chan` type to a message `M` in `ctype` and quite useful in our later theories. But how can we define a non partial function, if the domain of our type is empty? Then the function can never map to any message and would be partial. To still retain the totality property in all possible cases, we define a stream bundle element as some total function, if the domain is not empty, and as nothing (`None`) if the domain is empty. The totality leads to shorter proofs because less cases have to be checked.

```
fun sbElem_well :: "('cs ⇒ M) option ⇒ bool" where
"sbElem_well    None    = chDomEmpty TYPE('cs)" |
```

```
"sbElem_well (Some sbe) = (∀c. sbe c ∈ ctype(Rep c))"
```

Predicate sbElem_well exactly describes our requirements. The option type in our predicate allows us to have the (None) function, iff the domain of our channel type is empty. For all non-empty domains, a total function is a sbElem, iff it only maps to messages in the ctype of the channel. With those preparations we now define the sbElem type:

```
typedef 'cs sbElem ("(_√)") =
        "{f::('cs ⇒ M) option. sbElem_well f}"
```

The suffix (_√) abbreviates 'cs sbElem to 'cs√.

The order of sbElems then has to be a discrete one. Else its order would be inconsistent to our prefix order on streams and also the resulting SB order.

```
instantiation  sbElem::(chan) discrete_cpo
begin
  definition below_sbElem::"'cs√ ⇒ 'cs√ ⇒ bool" where
  "below_sbElem sbe1 sbe2 ≡ sbe1 = sbe2"
  instance
end
```

The following three theorems describe the behaviour of the sbElem type for empty and non-empty domains. Hence, they verify the desired properties of our type.

```
theorem sbtypeepmpty_sbenone[simp]:
  fixes sbe::"'cs√"
  assumes "chDomEmpty TYPE ('cs)"
  shows "sbe = Abs_sbElem None"
```

In case of the empty domain, any sbElem is None. Hence, we now have to look at the behaviour for non-empty domains.

```
theorem sbtypefull_none[simp]:
  fixes sbe::"'cs√"
  assumes "¬chDomEmpty TYPE ('cs)"
  shows "Rep_sbElem sbe ≠ None"
```

First we show that a sbElem with a non-empty domain never is None. Thus, it is easy to show that there always exists a total function, that is an sbElem, if the domain is empty. It follows directly from the non-emptiness of a type.

```
theorem sbtypenotempty_somesbe:
  assumes "¬chDomEmpty TYPE ('cs)"
  shows "∃f::'cs ⇒ M. sbElem_well (Some f)"
```

## 6.4 Stream Bundles Datatype

Streams are the backbone of this verification framework and stream bundles are used to model components with multiple input and output streams. Any stream in a stream bundle is identifiable through its channel. Hence, a SB is a function from channels to streams. Since the allowed messages on a channel may be restricted, the streams of a

SB only contain streams of elements from the type of their channel (ctype). Similar to sbElems, we formulate a predicate to describe the properties of a SB.

```
definition sb_well :: "('c::chan ⇒ M stream) ⇒ bool" where
"sb_well f ≡ ∀c. sValues·(f c) ⊆ ctype (Rep c)"
```

This definition uses sValues function defined for streams in section 5.2.6 to obtain a set, which contains every element occurring in a stream. If the values of each stream are a subset of the allowed messages on their corresponding channels, the function is a SB. Unlike our sbElem predicate, a differentiation for the empty domain is not necessary, because every non-empty stream for bundles with an empty domain would lead to a contradiction with the sb_well predicate.

Since we define SBs as total functions from channels to streams, the type can be instantiated as a pcpo. This provides additional general properties and allows the usage of the fix point operator. The resulting prefix order for SBs follows directly from the order of streams. A SB is prefix of another SB, if each of its streams is prefix of the corresponding streams of the other SB.

```
pcpodef 'c::chan sb("(_^Ω)")
         = "{f::('c::chan ⇒ M stream). sb_well f}"
```

### SB Type Properties

The ⊥ element of our SB type is a mapping to empty streams.

```
theorem bot_sb: "⊥ = Abs_sb (λc. ε)"
```

In case of an empty domain, no stream should be in a SB. Hence, every SB with an empty domain should be ⊥. This is proven in the following theorem.

```
theorem sbtypeepmpty_sbbot[simp]:
  fixes sb::"'cs^Ω"
  assumes "chDomEmpty TYPE ('cs)"
  shows "sb = ⊥"
```

## 6.5 Functions for Stream Bundles

This section defines and explains the most commonly used functions for SB. Also, the main properties of important functions will be discussed.

### Converter from sbElem to SB

First we construct a converter from sbElems to SB. This is rather straight forward, since we either have a function from channels to messages, which we can easily convert to a function from channels to streams. This consists only of streams with the exact message from the sbElem. In the case of an empty domain, we map None to the ⊥ element of SB.

```
lift_definition sbe2sb::"'c^√ ⇒ 'c^Ω" is
"λ sbe. case (Rep_sbElem sbe) of Some f ⇒ λc. ↑(f c)
                               | None   ⇒ ⊥ "
```

Through the usage of keyword `lift_definition` instead of `definition` we automatically have to proof that the output is indeed a SB.

### Extracting a single stream

The direct access to a stream on a specific channel is one of the most important functions in the framework and also often used for verifying properties. Intuitively, the signature of such a function should be `'cs ⇒ 'cs^Ω → M stream`, but we use a slightly more general signature. Two domain types could contain exactly the same channels, but we could not obtain the streams of a SB with the intuitive signature, when the type of the SB is different (see section 6.2.4). To avoid this, we can use the `Rep` and `Abs` functions of our domain types to convert between the them by representation and abstraction via the global channel type. This also facilitates later function definitions and reduces the total framework size by using abbreviations of one general function that only restrict the signature.

```
lift_definition sbGetCh :: "'cs1 ⇒ 'cs2^Ω → M stream" is
"λc sb. if Rep c∈chDom TYPE('cs2)
            then Rep_sb sb (Abs(Rep c))
            else ε"
```

Our general signature allows the input of any channel from the `channel` type. If the channel is in the domain of the input SB, we obtain the corresponding channel by converting the channel to an element of our domain type with the nesting of `Abs` and `Rep`. Is the channel not in the domain, the empty stream $\varepsilon$ is returned. The continuity of this function is also immediately proven.

The next abbreviations are defined to differentiate between the intuitive and the expanded signature of `sbGetCh`. The first abbreviation is an abbreviation for the general signature, the second restricts to the intuitive signature.

```
abbreviation sbgetch_magic_abbr :: "'cs1^Ω ⇒ 'cs2 ⇒ M stream"
(infix " ▸⋆ " 65) where "sb ▸⋆ c ≡ sbGetCh c·sb"
```

All properties proven for the general signature automatically hold for the restricted signature. In general one could also add additional abbreviations with different signatures at a later time and immediately use properties of less restricted signatures.

```
abbreviation sbgetch_abbr :: "'cs^Ω ⇒ 'cs ⇒ M stream"
(infix " ▸ " 65) where "sb ▸ c ≡ sbGetCh c·sb"
```

Obtaining a `sbElem` from a SB is not always possible. If the domain of a bundle is not empty but there is an empty stream on a channel, the resulting `sbElem` could not map that channel to a message from the stream. Hence, no slice of such a SB can be translated to a `sbElem`. The following predicate states, that the first slice of an SB with a non-empty domain can be transformed to a `sbElem`, because it checks, if all streams in the bundle are not empty.

```
definition sbHdElemWell::"'c^Ω ⇒ bool" where
"sbHdElemWell  ≡ λ sb. (∀c. sb  ▸  c ≠ ε)"
```

```
abbreviation sbIsLeast::"'cs^Ω ⇒ bool" where
```

```
"sbIsLeast sb ≡ ¬sbHdElemWell sb"
```

The negation of this property is called `sbIsLeast`, because these SB do not contain any complete slices.

When using the intuitive variant of `sbGetCh`, it obtains a stream from a channel. It should never be able to do anything else. This behavior is verified by the following theorem. Obtaining a stream from its SB is the same as obtaining the output from the function realizing the SB.

```
theorem sbgetch_insert2:"sb ▸ c = (Rep_sb sb) c"
```

If a SB `sb1` is prefix of another SB `sb2`, the order also holds for each streams on every channel.

```
theorem sbgetch_sbelow[simp]:"sb1 ⊑ sb2 ⟹ sb1 ▸ c ⊑ sb2 ▸ c"
```

Now we can show the equality and order property of two SB though the relation of their respective streams. In both cases we only have to check channels from the domain, hence the properties automatically hold for SB with an empty domain.

```
theorem sb_belowI:
  fixes    sb1 sb2::"'cs^Ω"
  assumes "⋀ c. Rep c∈chDom TYPE('cs) ⟹  sb1 ▸ c ⊑ sb2 ▸ c"
  shows    "sb1 ⊑ sb2"
```

If all respectively chosen streams of one bundle are prefix of the streams of another bundle, the prefix relation holds for the bundles as well.

```
theorem sb_eqI:
  fixes    sb1 sb2::"'cs^Ω"
  assumes "⋀c. Rep c∈chDom TYPE('cs) ⟹ sb1 ▸ c = sb2 ▸ c"
  shows    "sb1 = sb2"
```

If all respectively chosen streams of one bundle are equal to the streams of another bundle, these bundles are the same.

Lastly, the conversion from a `sbElem` to a SB should never result in a SB which maps its domain to $\varepsilon$.

```
theorem sbgetch_sbe2sb_nempty:
  fixes    sbe::"'cs^√"
  assumes "¬chDomEmpty TYPE('cs)"
  shows    "sbe2sb sbe  ▸  c ≠ ε"
```

### Bundle Equality

Checking the equality of bundles with same domains is wanted, even if the types are different. The following operator checks the equality of bundles.

```
definition sbEQ::"'cs1^Ω ⇒ 'cs2^Ω ⇒ bool" where
"sbEQ sb1 sb2 ≡ chDom TYPE('cs1) = chDom TYPE('cs2) ∧
                (∀c. sb1 ▸ c = sb2 ▸⋆ c)"
```

The operator checks the domain equality of both bundles and then the equality of its streams. For easier use, an infix abbreviation ≜ is defined.

```
abbreviation sbeq_abbr :: "'cs1Ω  ⇒  'cs2Ω ⇒  bool"
(infixr "≜" 70) where "sb1 ≜ sb2 ≡ sbEQ sb1 sb2"
```

### Concatenation

Concatenating two SB is equivalent to concatenating their streams whilst minding the channels. The output is also a SB, because the values of both streams are in `ctype`, therefore, the same holds for the union. The compatibility of the input bundles is ensured by the signature of the function.

```
lift_definition sbConc:: "'csΩ ⇒ 'csΩ → 'csΩ" is
"λsb1 sb2. Abs_sb(λc. (sb1 ▸ c) • (sb2 ▸ c))"
```

For easier usability, we introduce a concatenation abbreviation.

```
abbreviation sbConc_abbr :: "'csΩ  ⇒  'csΩ ⇒  'csΩ"
(infixr "•Ω" 70) where "sb1 •Ω sb2 ≡ sbConc sb1·sb2"
```

After concatenating two SB, the resulting SB has to contain the streams from both SB in the correct order. Hence, obtaining a stream by its channel from the concatenation of two SB is equivalent to obtaining the stream by the same channel from the input SB and then concatenating the streams from the first input bundle with the stream from the second input bundle.

```
theorem sbconc_getch [simp]:
  shows "(sb1 •Ω sb2)  ▸ c = (sb1 ▸ c) • (sb2 ▸ c)"
```

It follows, that concatenating a SB with the ⊥ bundle in any order, results in the same SB.

```
theorem sbconc_bot_r[simp]: "sb •Ω ⊥ = sb"
theorem sbconc_bot_l[simp]: "⊥ •Ω sb = sb"
```

### Length of SBs

We define the length of a SB as follows:

- A SB with an empty domain is infinitely long
- A SB with an non-empty domain is as long as its shortest stream

The definition for the empty domain was designed with the timed case in mind. This definition can be used to define causality.

```
definition sbLen::"'csΩ ⇒ lnat"where
"sbLen sb ≡ if chDomEmpty TYPE('cs) then ∞
                else LEAST n . n∈{#(sb ▸ c) | c. True}"
```

Our `sbLen` function works exactly as described. It returns ∞, if the domain is empty. Else it chooses the minimal length of all the bundles streams.

Since the length of a bundle is used for defining causality in the framework, the desired behaviour is verified by many lemmas. We will introduce a few important properties as theorems.

The abbreviation # is a shortcut for `sbLen`.

The length of two concatenated bundles is greater or equal to the added length of both bundles. If both bundles have a minimal stream on the same channel, the resulting length would be equal.

```
theorem sblen_sbconc: "#sb1 + #sb2 ≤ #(sb1 •Ω sb2)"
```

This rule captures all necessary assumptions to obtain the exact length of a SB with a non-empty domain:

- All streams must be at least equally long to the length of the SB
- There exists a stream with length equal to the length of the SB

```
theorem sblen_rule:
  fixes sb::"'csΩ"
  assumes "¬chDomEmpty TYPE('cs)"
      and "⋀c. k ≤ #(sb ▸ c)"
      and "∃c. #(sb ▸ c) = k"
  shows   "#sb = k"
```

If two SB are in an order and also infinitely long, they have to be equal. This holds because either the domain is empty or every stream of the bundles is infinitely long.

```
theorem sblen_sbeqI:
  fixes sb1 sb2::"'csΩ"
  assumes "sb1⊑sb2" and "#sb1 =∞"
  shows   "sb1 = sb2"
```

We can also show that the length of any SB that has a non-empty domain is equal to the length of one of its streams.

```
theorem sblen2slen:
  assumes "¬chDomEmpty TYPE('cs)"
  shows "∃c. #(sb :: 'csΩ) = #(sb ▸ c)"
```

The length of a `sbElem` is 1, if the domain is not empty

```
theorem sbelen_one[simp]:
  fixes   sbe::"'cs√"
  assumes "¬chDomEmpty TYPE('cs)"
  shows   " #(sbe2sb sbe) = 1"
```

### Dropping Elements

Through dropping a number of SB elements, it is possible to access any element in the SB or to get a later part. Dropping the first `n` Elements of a SB means dropping the first `n` elements of every stream in the SB.

```
lift_definition sbDrop::"nat ⇒ 'csΩ → 'csΩ"is
"λ n sb. Abs_sb (λc. sdrop n·(sb ▸ c))"
```

A special case of `sbDrop` is to drop only the first element of the SB. It is the rest operator on SB.

```
abbreviation sbRt :: "'csΩ → 'csΩ"  where
"sbRt ≡ sbDrop 1"
```

### Taking Elements

Through taking the first `n` elements of a SB, it is possible to reduce any SB to a finite part of itself. The output is always a prefix of the input.

```
lift_definition sbTake::"nat ⇒ 'csΩ →  'csΩ"is
"λ n sb. Abs_sb (λc. stake n·(sb ▸ c))"
```

A special case of `sbTake` is to take only the first element of the SB.

```
abbreviation sbHd :: "'csΩ → 'csΩ"  where
"sbHd ≡ sbTake 1"
```

Obtaining some stream form a SB after applying `sbTake`, is the same as applying `stake` after obtaining the stream from the SB.

```
theorem sbtake_getch[simp]:"sbTake n·sb ▸ c = stake n·(sb ▸ c)"
```

The output of `sbTake` is always `(⊑)` the input.

```
theorem sbtake_below[simp]: "sbTake i·sb ⊑ sb"
```

Concatenating the first `n` elements of a SB to the SB without the first `n` elements results in the same SB.

```
theorem sbconctakedrop[simp]:"sbConc (sbTake n·sb)·(sbDrop n·sb) = sb"
```

### Concatenating sbElems with SBs

Given a `sbElem` and a SB, we can append the `sbElem`to the SB. Of course we also have to consider the domain when appending the bundle:

- If the domain is empty, the output SB is ⊥
- If the domain is not empty, the output SB has the input `sbElem` as its first element.

Using only this operator allows us to construct all SBs where every stream has the same length. But since there is no restriction for the input bundle, we can map to any SB with a length greater 0.

```
definition sbECons::"'cs√ ⇒ 'csΩ → 'csΩ" where
"sbECons sbe = sbConc (sbe2sb sbe)"
```

Because we already constructed a converter from `sbElems` to SBs in section 6.5 and the concatenation in section 6.5, the definition of `sbECons` is straight forward. We also add another abbreviation for this function.

```
abbreviation sbECons_abbr::"'cs√ ⇒ 'csΩ ⇒ 'csΩ"(infixr "•√" 100)
where "sbe •√ sb ≡ sbECons sbe·sb"
```

The concatenation results in ⊥ when the domain is empty.

```
theorem sbtypeempty_sbecons_bot:
  fixes    sbe::"'cs√"
  assumes "chDomEmpty TYPE ('cs)"
  shows   "sbe •√ sb = ⊥"
```

It also holds, that the rest operator (section 6.5) of a with `sbECons` constructed SB is a destructor.

```
theorem sbrt_sbecons: "sbRt·(sbe •√ sb) = sb"
```

Obtaining the head of a SB constructed this way results in the first element converted to SB.

```
theorem sbh_sbecons: "sbHd·(sbe  •√ sb) = sbe2sb sbe"
```

Constructing a SB with `sbECons` increases its length by exactly 1. This also holds for empty domains, because we interpret the length of those SB as ∞.

```
theorem sbecons_len:
  shows "#(sbe •√ sb) = lnsuc·(# sb)"
```

### SB induction and case rules

This framework also offers proof methods using the `sbElem` constructor, that offer an easy proof process when applied correctly. The first method is a case distinction for SBs. It differentiates between the short SBs where an empty stream exists and all other SBs. The configuration of the lemma splits the goal into the cases `least` and `sbeCons`. It also causes the automatic usage of this case tactic for variables of type SB.

```
theorem sb_cases [case_names least sbeCons, cases type: sb]:
  assumes "sbIsLeast (sb'::'csΩ) ⟹ P"
  and     "⋀sbe sb. sb' = sbe •√ sb ⟹ ¬chDomEmpty TYPE ('cs)
           ⟹ P"
  shows   "P"
```

The second showcased proof method is the induction for SBs. Beside the admissibility of the predicate, the inductions subgoals are also divided into the cases `least` and `sbeCons`.

```
theorem sb_ind[case_names adm least sbeCons, induct type: sb]:
  fixes    x::"'csΩ"
  assumes "adm P"
  and     "⋀sb. sbIsLeast sb ⟹ P sb"
  and     "⋀sbe sb. P sb ⟹ ¬chDomEmpty TYPE ('cs)
           ⟹ P (sbe •√ sb)"
  shows   "P x"
```

Here we show a small example proof for our SB cases rule. First the ISAR proof is started by applying the proof tactic to the theorem. This automatically generates the proof structure with the two cases and their variables. These two generated cases match with our theorem assumptions from sb_cases. Our theorems statement then follows then directly by proving both generated cases.

```
theorem sbecons_eq:
  assumes "# sb ≠ 0"
  shows "sbHdElem sb •√ sbRt·sb = sb"
proof(cases sb)
  assume "sbIsLeast sb"
  thus "sbHdElem sb •√ sbRt·sb = sb"
    using assms by(simp only: assms sbECons_def sbHdElem sbcons)
next
  fix sbe and sb'
  assume "sb = sbe •√ sb'"
  thus "(sbHdElem sb) •√ sbRt·sb = sb"
    using assms by(simp only: assms sbhdelem_sbecons sbrt_sbecons)
qed
```

The first subgoals assumption after applying the case tactic is sbIsLeast sb and proving this case and the sbeCons case is often simpler than proving the theorem without case distinction.

The second subgoals assumes sb = sbe •√ sb'. This alows splitting the SB in two parts, where the first part is a sbElem. This helps if a function works element wise on its input.

The next theorem is an example for the induction rule. Similar to the cases rule there are automatically generated cases that correspond to the assumptions of sb_ind. Our theorem is proven after showing the three generated goals.

```
theorem shows "sbTake n·sb ⊑ sb "
proof(induction sb)
  case adm
  then show ?case
    by simp
next
  case (least sb)
  then show "sbIsLeast sb ⟹ sbTake n·sb ⊑ sb"
    by simp
next
  case (sbeCons sbe sb)
  then show "sbTake n·sb ⊑ sb ⟹ sbTake n·(sbe •√ sb) ⊑ sbe •√ sb"
    by simp
qed
```

### Converting Domains of SBs

Two SBs with a different type are not comparable, since only SBs with the same type have an order. This holds even if the domain of both types is the same. To make them comparable we introduce a type caster that converts the type of a SB. This casting makes two SB of different type comparable. Since it does change the type, it can also restrict or expand the domain of a SB. Newly added channels map to $\varepsilon$.

```
lift_definition sbTypeCast::"'cs1^Ω → 'cs2^Ω"is
"(λ sb. Abs_sb (λc.  sb ▸⋆ c ))"
```

Because restricting the domain of a SB is an important feature of this framework, we offer explicit abbreviations for such cases.

```
abbreviation sbTypeCast_abbr :: "'cs1Ω ⇒ 'cs2Ω"
  ( "_⋆" 200)   where "sb⋆ ≡ sbTypeCast·sb"
```

Without the general signature of `sbTypeCast`, the following abbreviations would need own definitions and could not share properties dirctly among themselves.

```
abbreviation sbrestrict_abbr_fst :: "('cs1 ∪ 'cs2)Ω ⇒ 'cs1Ω"
  ( "_⋆₁" 200) where "sb⋆₁ ≡ sbTypeCast·sb"
```

```
abbreviation sbrestrict_abbr_snd :: "('cs1∪'cs2)Ω ⇒ 'cs2Ω"
  ( "_⋆₂" 200) where "sb⋆₂ ≡ sbTypeCast·sb"
```

```
abbreviation sbTypeCast_abbr_fixed :: "'cs1Ω ⇒ 'cs3 itself ⇒ 'cs3Ω"
  ( "_|_" 201) where "sb | _ ≡ sbTypeCast·sb"
```

A SB with domain `('cs1 ∪ 'cs2) -'cs3` can be restricted to domain `('cs1 - 'cs3)` by using `sb | TYPE ('cs1 - 'cs3).`

Obtaining a stream from a converted SB is the same as not converting it but using the general `sbGetCh` operator to convert the channels type. Thus, converting the domain of a bundle is equivalent to converting the type of all its channels.

```
theorem sbtypecast_getch [simp]: "sb⋆ ▸ c = sb ▸⋆ c"
```

### Union of SBs

The union operator for streams merges two SB together. The output domain is equal to the union of its input domains. But again we use a slightly different signature for the general definition. It is equal to applying the converter after building the exact union of both bundles. If the input SBs share a channel, the output SBs stream on that channel is the stream from the first input SB.

```
definition sbUnion::"'cs1Ω → 'cs2Ω → ('cs1 ∪ 'cs2)Ω" where
"sbUnion ≡ Λ sb1 sb2. Abs_sb (λ c.
                   if Rep c ∈ chDom TYPE('cs1)
                   then sb1 ▸⋆ c
                   else sb2 ▸⋆ c)"
```

The first abbreviation has the intuitive signature of the bundle union operator.

```
abbreviation sbUnion_abbr :: "'cs1Ω ⇒ 'cs2Ω ⇒ ('cs1 ∪ 'cs2)Ω"
(infixr "⊎" 100) where "sb1 ⊎ sb2 ≡ sbUnion·sb1·sb2"
```

The following abbreviations restrict the input and output domains of `sbUnion` to specific cases. These are displayed by its signature. Abbreviation $⊎_\star$ is the composed function of `sbUnion` and `sbTypeCast`, thus, it converts the output domain.

```
abbreviation sbUnion_magic_abbr :: "'cs1Ω ⇒ 'cs2Ω ⇒ 'cs3Ω"
(infixr "⊎⋆" 100) where "sb1 ⊎⋆ sb2 ≡ (sb1 ⊎ sb2)⋆"
```

The third abbreviation only fills in the stream its missing in its domain 'cs1. It does not use stream on channels that are in domain cs2 but not cs1.

```
abbreviation sbUnion_minus_abbr :: "('cs1 - 'cs2)Ω ⇒ 'cs2Ω ⇒ 'cs1Ω"
(infixr "⊎_" 500) where "sb1 ⊎_ sb2 ≡ sb1 ⊎⋆ sb2"
```

### sbUnion Properties

Here we show how the union operator and its abbreviations works.

The union operator is commutative, if the domains of its input are disjoint.

```
theorem ubunion_commu:
  fixes   sb1 ::"'cs1Ω"
  and      sb2 ::"'cs2Ω"
  assumes "chDom TYPE ('cs1) ∩ chDom TYPE ('cs2) = {}"
  shows   "sb1 ⊎⋆ sb2 = sb2 ⊎⋆ sb1"
```

The union of two SBs maps each channel in the domain of the first input SB to the corresponding stream of the first SB.

```
theorem sbunion_getchl[simp]:
  fixes   sb1 ::"'cs1Ω"
  and      sb2 ::"'cs2Ω"
  assumes "Rep c ∈ chDom TYPE('cs1)"
  shows   "(sb1 ⊎ sb2) ▸⋆ c = sb1 ▸⋆ c"
```

This also holds for the second input SB, if the domains of both SBs are disjoint.

```
theorem sbunion_getchr[simp]:
  fixes   sb1 ::"'cs1Ω"
  and      sb2 ::"'cs2Ω"
  assumes "Rep c ∉ chDom TYPE('cs1)"
  shows   "(sb1 ⊎ sb2) ▸⋆ c = sb2 ▸⋆ c"
```

Restricting the unions domain to the first inputs domain is equal to the first input.

```
theorem sbunion_fst: "(sb1 ⊎ sb2)⋆1 = sb1"
```

Analogous this also holds for the second input, if the input domains are disjoint.

```
theorem sbunion_snd[simp]:
  fixes   sb1 ::"'cs1Ω"
  and      sb2 ::"'cs2Ω"
  assumes "chDom TYPE ('cs1) ∩ chDom TYPE ('cs2) = {}"
  shows   "(sb1 ⊎ sb2)⋆2 = sb2"
```

### Renaming of Channels

Renaming the channels of a SB is possible if the allowed transmitted messages on the original channel are a subset of the allowed messages on the new channel. The following function renames arbitrary many channels by giving a channel name mapping function. If any of the renamed channels allow less messages, the renamed SB is not defined.

```
lift_definition sbRenameCh::"('cs1 ⇒ 'cs2) ⇒ 'cs2^Ω → 'cs1^Ω" is
"λf sb. if (∀c. ctype (Rep (f c)) ⊆ ctype (Rep c))
          then Abs_sb (λc. sb ▸ (f c))
          else undefined"
```

If the renaming is possible, no stream is changed.

```
theorem sbrenamech_getch[simp]:
  assumes "⋀c. ctype (Rep (f c)) ⊆ ctype (Rep c)"
  shows "(sbRenameCh f·sb) ▸ c = sb ▸ (f c)"
```

In some cases only certain channels should be modified, while keeping all other channels. For this case we define an alternative version of `sbRename`. It takes an partial function as argument. Only the channels in the domain of the function are renamed.

```
definition sbRename_part::"('cs1 ⇀ 'cs2) ⇒ 'cs2^Ω → 'cs1^Ω" where
"sbRename_part f = sbRenameCh (λcs1. case (f cs1) of Some cs2 ⇒ cs2
                                                    | None ⇒ Abs (Rep cs1))"
```

The `getch` lemmata is seperated into two cases. The first case is when the channel is part of the mapping. This first assumption is directly taken from the normal `sbRename` definition. The second assumption ensures that unmodified channels also exist in the output bundle.

```
theorem sbrenamepart_getch_in[simp]:
    fixes f :: "('cs1 ⇀ 'cs2)"
    assumes "⋀c. c∈dom f ⟹ ctype (Rep (the (f c))) ⊆ ctype (Rep c)"
        and "⋀c. c∉dom f ⟹ (Rep c) ∈ chDom TYPE ('cs2)"
        and "c∈dom f"
    shows "(sbRename_part f·sb) ▸ c = sb ▸ the (f c)"
```

When the channel is not part of the mapping the rename-function is not used:

```
theorem sbrenamepart_getch_out[simp]:
    fixes f :: "('cs1 ⇀ 'cs2)"
    assumes "⋀c. c∈dom f ⟹ ctype (Rep (the (f c))) ⊆ ctype (Rep c)"
        and "⋀c. c∉dom f ⟹ (Rep c) ∈ chDom TYPE ('cs2)"
        and "c∉dom f"
    shows "(sbRename_part f·sb) ▸ c = sb ▸⋆ c"
```

### Lifting from Stream to Bundle

This section provides a bijective mapping from `'a` to SB. Type `'a` could for example be a `nat stream × bool stream`. A locale [Bal06] can be used to lift functions over streams to bundles. The number of channels is not fixed, it can be an arbitrary large number.

A `locale` is a special environment within Isabelle. In the beginning of the locale are multiple assumptions. Within the locale these can be freely used. To use the locale the user has to proof these assumptions later. After that all definitions and theorems in the locale are accessible. The locale can be used multiple times.

The definition `lConstructor` maps the `'a` element to a corresponding SB. The constructor has to be injective and maps precisely to all possible functions, that can be lifted to stream bundles. Since the setter and getter in this locale are always bijective, all SBs can be constructed.

The continuity of the setter is given by assuming the continuity of the constructor. Thus continuity of the getter follows from assuming that the constructor maintains non-prefix orders and from the continuity and surjectivity of the setter. Furthermore, assumptions over the length (#) exist.

```
locale sbGen =
 fixes lConstructor::" 'a::{pcpo,len} ⇒ 'cs::{chan}  ⇒ M stream"
 assumes c_type: "⋀a c. sValues·(lConstructor a c) ⊆ ctype (Rep c)"
   and c_inj: "inj lConstructor"
   and c_surj: "⋀f. sb_well f ⟹ f∈range lConstructor"
   and c_cont: "cont lConstructor"
   and c_nbelow: "⋀x y. ¬(x ⊑ y) ⟹
                          ¬(lConstructor x ⊑ lConstructor y)"
   and c_len:"⋀ a c. ¬chDomEmpty TYPE('cs)⟹ #a ≤ # (lConstructor a c)"
   and c_lenex:"⋀ a. ¬chDomEmpty TYPE('cs)⟹ ∃c.#a = #(lConstructor a c)"
```

The lifting of the setter and getter function to a continuous function is a short proof.

```
lift_definition setter::"'a → 'csΩ"
  is "Abs_sb o lConstructor"
```

```
lift_definition getter::"'csΩ → 'a"
  is "(inv lConstructor) o  Rep_sb"
```

Finally, the composed execution of setter and getter results in the identity.

```
theorem get_set[simp]: "getter·(setter·a) = a"
```

```
theorem set_get[simp]: "setter·(getter·sb) = sb"
```

The length of the resulting bundle is connected to the length of the user-supplied datatype 'a:

```
theorem setter_len: assumes "chDom TYPE('cs) ≠ {}"
  shows "#(setter·a) = #a"
```

### Overview of all functions

In table 6.1 are all function over the SB datatype depicted.

| Def | Abbrev | Signature | Description |
|---|---|---|---|
| Abs_sb | | $('cs \Rightarrow M^{\omega}) \Rightarrow 'cs^{\Omega}$ | Lift a function to a SB (3.3) |
| Rep_sb | | $'cs^{\Omega} \Rightarrow 'cs \Rightarrow M^{\omega}$ | Inverse Function of Abs_sb |
| bottom | $\bot$ | $'cs^{\Omega}$ | Least stream bundle |
| chDom | | 'cs itself $\Rightarrow$ channel set | Domain of the bundle (6.2.3) |
| sbe2sb | | $'cs^{\surd} \Rightarrow 'cs^{\Omega}$ | conversion of sbElems to SB (6.5) |
| sbGetCh | $(\ \blacktriangleright_{\star}\ )$ | $'cs1 \Rightarrow 'cs2^{\Omega} \rightarrow M^{\omega}$ | Get the stream on a channel (6.5) |
| | $(\ \blacktriangleright\ )$ | $'cs \Rightarrow 'cs^{\Omega} \rightarrow M^{\omega}$ | |
| sbIsLeast | | $'cs^{\Omega} \Rightarrow \mathcal{B}$ | True iff an empty stream exists |
| sbEQ | $(\triangleq)$ | $'cs1^{\Omega} \Rightarrow 'cs2^{\Omega} \Rightarrow bool$ | Equality of bundles (section 6.5) |
| sbConc | $(\bullet^{\Omega})$ | $'cs^{\Omega} \Rightarrow 'cs^{\Omega} \rightarrow 'cs^{\Omega}$ | Concatenation of bundles (6.5) |
| sbECons | $(\bullet^{\surd})$ | $'cs^{\surd} \Rightarrow 'cs^{\Omega} \rightarrow 'cs^{\Omega}$ | Concatenation with sbElem (6.5) |
| sbDrop | | $\mathbb{N} \Rightarrow 'cs^{\Omega} \rightarrow 'cs^{\Omega}$ | drops the first n elements (6.5) |
| sbRt | | $'cs^{\Omega} \rightarrow 'cs^{\Omega}$ | drops the first element |
| sbTake | | $\mathbb{N} \Rightarrow 'cs^{\Omega} \rightarrow 'cs^{\Omega}$ | takes the first n elements (6.5) |
| sbHd | | $'cs^{\Omega} \rightarrow 'cs^{\Omega}$ | takes the first element |
| sbHdElem | $\lfloor sb$ | $'cs^{\Omega} \Rightarrow 'cs^{\surd}$ | first element as a sbElem |
| sbTypeCast | $sb\star$ | $'cs1^{\Omega} \rightarrow 'cs2^{\Omega}$ | Type Conversion (6.5) |
| | $sb\star_1$ | $('cs1 \cup 'cs2)^{\Omega} \rightarrow 'cs1^{\Omega}$ | |
| | $sb\star_2$ | $('cs1 \cup 'cs2)^{\Omega} \rightarrow 'cs2^{\Omega}$ | |
| | $sb\star_{\rightleftharpoons}$ | $('cs1 \cup 'cs2)^{\Omega} \rightarrow ('cs2 \cup 'cs1)^{\Omega}$ | |
| | $sb\star_{-}$ | $'cs1^{\Omega} \rightarrow ('cs1 - 'cs2)^{\Omega}$ | |
| | $sb \mid \_$ | $'cs1^{\Omega} \Rightarrow 'cs3$ itself $\Rightarrow 'cs3^{\Omega}$ | |
| sbUnion | $(\uplus)$ | $'cs1^{\Omega} \rightarrow 'cs2^{\Omega} \rightarrow ('cs1 \cup 'cs2)^{\Omega}$ | merges two SB together (6.5) |
| | $(\uplus_{\star})$ | $'cs1^{\Omega} \rightarrow 'cs2^{\Omega} \rightarrow 'cs3^{\Omega}$ | |
| | $(\uplus_{-})$ | $('cs1 - 'cs2)^{\Omega} \rightarrow 'cs2^{\Omega} \rightarrow 'cs1^{\Omega}$ | |
| sbRenameCh | | $('cs1 \Rightarrow 'cs2) \Rightarrow 'cs2^{\Omega} \rightarrow 'cs1^{\Omega}$ | renaming channels of a SB (6.5) |
| sbRename_part | | $('cs1 \rightharpoonup 'cs2) \Rightarrow 'cs2^{\Omega} \rightarrow 'cs1^{\Omega}$ | renaming channels of a SB (6.5) |
| sbSetCh | | $'cs \Rightarrow M^{\omega} \Rightarrow 'cs^{\Omega} \rightarrow 'cs^{\Omega}$ | Overwrite channel |
| sbNTimes | | $\mathbb{N} \Rightarrow 'cs^{\Omega} \Rightarrow 'cs^{\Omega}$ | Iterate each stream n-times |
| sbInfTimes | | $'cs^{\Omega} \Rightarrow 'cs^{\Omega}$ | Iterate each stream $\infty$-times |
| sbFilter | | $M\ set \Rightarrow 'cs^{\Omega} \rightarrow 'cs^{\Omega}$ | Apply filter to each stream |
| sbTakeWhile | | $(M \Rightarrow bool) \Rightarrow 'cs^{\Omega} \rightarrow 'cs^{\Omega}$ | Prefix while predicate holds |
| sbDropWhile | | $(M \Rightarrow bool) \Rightarrow 'cs^{\Omega} \rightarrow 'cs^{\Omega}$ | Drop while predicate holds |
| sbRcdups | | $'cs^{\Omega} \rightarrow 'cs^{\Omega}$ | Remove successive duplicates |

Table 6.1: Functions for SBs

# Chapter 7

# Stream Processing Functions

A deterministic component is modeled by a *stream (bundle) processing function* (the type denoted as SPF), which is a continuous function mapping bundles to bundles. We will focus in section 7.3 on deterministic components. In chapter 8 we will also consider *nondeterministic* components, modeled as sets of stream processing functions. Most of their properties can be straightforwardly lifted. The full set of the definitions and lemmas can found in the appendix D.

## 7.1 Mathematical Definition

We define a stream processing function `f` as a continuous bundle to bundle function with fixed input and output channels [Rum96]. Monotonicity of the function implies that a component can not take back an already produced output. Continuity ensures that a component behaves the same on an infinite input as it would on its finite prefixes.

### Stream Processing Functions for Bundles

**Definition 7.1** (SPF based on total function)**.** Let $C$ be the set of all possible channels and $I, O \subseteq C$. We can define the SPF type $SPF_{I,O}$ that includes all (continuous) SPFs with input channels $I$ and output channels $O$ as shown below:

$$SPF_{I,O} := \{f \in I^{\Omega} \to O^{\Omega}\}$$

Based on that definition, we can then define the generic SPF type $SPF_{total}$ as follows:

$$SPF_{total} := \bigcup_{I,O \in C} SPF_{I,O}$$

### Stream-Processing Functions with direct Channels

For completeness, we also add the definition of the pure channel-based stream processing functions (C-SPF).

For system modeling, we are only interested in realizable (well-behaved) functions over streams. Continuity is our corresponding concept for being well-behaved. Continuous functions are defined over pcpo's. This justifies the definition of our data type as a pcpo.

The semantics of our component is a stream-processing function. Later, we will generalize this understanding to *sets* of stream-processing function.

**Definition 7.2.** A stream-processing function is a continuous function where the domain is the set of all streams over a set of input messages $M_{in}$ and the range is the set of all streams over a set of output messages $M_{out}$:

$$f : M_{in}^{\omega} \to M_{out}^{\omega}$$

Viewing a function as behavior definition of a component, monotonicity intuitively means that a component cannot take back any already made output. Continuity means that we can approximate the output of a component on an infinite input using the outputs on finite prefixes of that infinite input. Thus, both of these coincide with our intuitive view of software component behaviors. We recall that the continuity of stream-processing function implies monotonicity (c.f. Lemma 2.4).

## 7.2 Composition of SPFs

We recall that our form of modular modeling the network has the advantage that, after checking the correctness of individual components of the decomposed system and after composing them correctly, the desired properties of the whole system can be derived by construction.

It is one of the key benefits of using FOCUS that in contrast to other similar known formalisms refinement is fully compatible with composition [RR11]. We formalized various composition operators, which can be categorized into special operators and a general operator [RR11; BR07]. The special operators are the *sequential*, *parallel* and *feedback* operator. The *general* operator subsumes all special operators [Bro+92] as well as any network construction. They fulfill a set of properties such as commutativity, and under some easy to ensure preconditions also associativity which allows to flexibly compose large networks of components in a hierarchical way.

### Special Composition Operators

The three specific operators in FOCUS can be applied to either one or two SPFs.

#### Sequential Composition Operator

To compose two SPFs sequentially, the output channels of one component have to match the input channels of the other component exactly. They cannot share any other channels. Only if this requirement is met, we can compose the SPFs as displayed in figure 7.1.

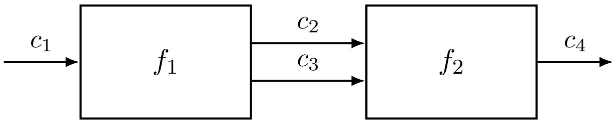



Figure 7.1: Sequential composition example

#### Parallel Composition Operator

The parallel composition of two SPFs is well-defined as long as the output channels of both functions are disjoint. However, no feedback occurs. In fig. 7.2 is an example of a simple parallel composition where neither the input nor the output channels are joint.

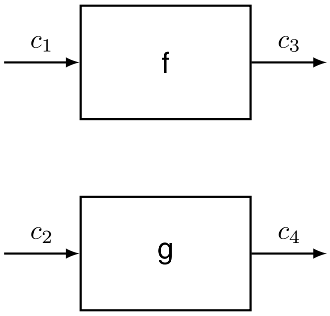



Figure 7.2: Parallel composition example

#### Feedback Composition Operator

The feedback composition operator $\mu$ connects shared input and output channels. Hence, it is appropriate for a SPF $f :: I^{\Omega} \rightarrow O^{\Omega}$ if there is at least one channel in the set $(I \cap O) = S$. An example component with a feedback channel can be examined in Figure 7.3. In the following we denote by $I - S$ the set of elements in $I$ that are not in $S$.

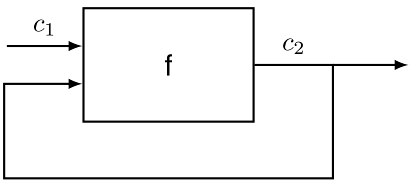



Figure 7.3: Feedback composition example

### General Composition Operator

In the previous section, we already saw how to use composition operators of FOCUS to create a network. Each of those operators performs a specific type of composition and connects the involved channels differently. So a user has to explicitly think about which composition operator is the correct one to chose. On top of that, such operators allow an implicit overwriting of channels which can decrease the understandability of the composed system for someone that was not directly involved in the specification process.

These two disadvantages of a classical composition can be resolved by using a general composition operator that connects channels with the same name [RR11]. Internally such an operator can be realized using a special kind of parallel composition operator that also considers streams on feedback channels. In contrast to the classical composition operators introduced earlier, the operator is not only capable of performing pure classical compositions forms like the feedback, parallel, or sequential composition but can also perform a mixture of them as shown in Figure 7.4.

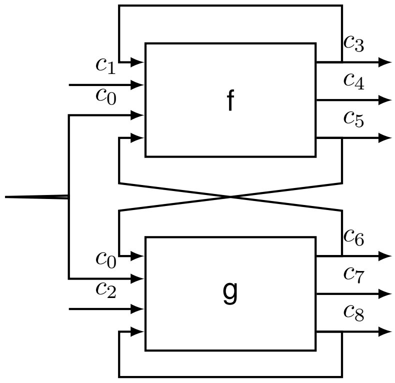



Figure 7.4: Complex Composition Scenario

## 7.3 Stream Processing Functions in Isabelle

For SPFs no new datatype is created. Instead we use the existing type for glscont functions from HOLCF. That way many definitions and lemmata are already available.

A SPF is written as `('I^Ω → 'O^Ω)`. It is an continuous function from the input bundle `('I^Ω)` to an output bundle `('O^Ω)`. The signature of the component is directly visible from the type-signatur of the SPF:

```
definition spfType ::"('IΩ → 'OΩ) itself ⇒
  (channel set × channel set)" where
"spfType _ = (chDom TYPE ('I), chDom TYPE ('O))"
```

```
definition spfDom ::"('IΩ → 'OΩ) itself ⇒ channel set" where
"spfDom = fst o spfType"
```

```
definition spfRan ::"('IΩ → 'OΩ) itself ⇒ channel set" where
"spfRan = snd o spfType"
```

The input output behaviour of a SPF is defined as a set of tuples of bundles where the first tuples element represents the input bundle and the second element the output bundle of an spf.

```
definition spfIO::"('I1Ω → 'O1Ω) ⇒ ('I1Ω × 'O1Ω) set" where
"spfIO spf = {(sb, spf·sb) | sb. True}"
```

### SPF Equality

Evaluate the equality of bundle functions with same input and output domains disregarding different types is possible by reusing the bundle equality ≜ operator.

```
definition spfEq::"('I1^Ω → 'O1^Ω) ⇒ ('I2^Ω → 'O2^Ω) ⇒ bool" where
"spfEq f1 f2 ≡ chDom TYPE('I1) = chDom TYPE('I2) ∧
               chDom TYPE('O1) = chDom TYPE('O2) ∧
                (∀sb1 sb2. sb1 ≜ sb2 ⟶ f1·sb1 ≜ f2·sb2)"
```

The operator checks the domain equality of input and output domains and then the bundle equality of its possible output bundles. For easier use, a infix abbreviation $\triangleq_f$ is defined.

```
abbreviation sbeq_abbr :: "('I1^Ω → 'O1^Ω) ⇒ ('I2^Ω → 'O2^Ω) ⇒ bool"
(infixr "≜_f" 101) where "f1 ≜_f f2 ≡ spfEq f1 f2"
```

## 7.4 General Composition Operators

The compositions output is completely determined by a fixed point. In essence, the composed SPF uses its input and previous output to compute the next output which is equivalent to its sub SPFs output. This is done until a fixed point is reached.

Our general composition operator is capable of all possible compositions. It is defined over the SPF type to allow the fix point calculation.

```
fixrec spfComp::"('I1^Ω → 'O1^Ω) → ('I2^Ω → 'O2^Ω)
 → ((('I1 ∪ 'I2) - ('O1 ∪ 'O2))^Ω → ('O1 ∪ 'O2)^Ω)" where
"spfComp·spf1·spf2·sbIn = spf1·((sbIn ⊎_ spfComp·spf1·spf2·sbIn)⋆₁)
                        ⊎ spf2·((sbIn ⊎_ spfComp·spf1·spf2·sbIn)⋆₂)"
```

The standard abbreviation of the composition operator is ⊗.

```
abbreviation spfComp_abbr::
"('I1^Ω → 'O1^Ω) ⇒ ('I2^Ω → 'O2^Ω)
 ⇒ ((('I1 ∪ 'I2) - ('O1 ∪ 'O2))^Ω → ('O1 ∪ 'O2)^Ω)"
(infixr "⊗" 70) where "spf1 ⊗ spf2 ≡ spfComp·spf1·spf2"
```

The compositions output domain is equal to output domain union of both input functions. Thus, the composition operator does not hide any internal channels in the output. This can still be achieved by using the `sbTypeCast` operator to restrict the output domain. A abbreviation for applying `sbTypeCast` to the composition operator is provided. It can be used for hiding channels.

```
abbreviation spfCompm_abbr (infixr "⊗⋆" 70) where
"spf1 ⊗⋆ spf2 ≡ sbTypeCast oo (spfComp·spf1·spf2) oo sbTypeCast"
```

The continuity of the composition operator holds by construction, because it only uses continuous functions.
The older version of this framework also provided a continuous composition operator, but its definition and continuity proof using the old SPF type was complicated and long. One of the consequences of the old SPF type was the necessity of a fix point operator over cpo that had to be defined for the composition operator.

Its commutativity is shown in the following theorem:

```
theorem spfcompcommu:
  fixes f::"'fIn^Ω → 'fOut^Ω"
  and g::"'gIn^Ω → 'gOut^Ω"
  assumes "chDom TYPE('fOut) ∩ chDom TYPE('gOut) = {}"
  shows "(f ⊗ g) ≜_f (g ⊗ f)"
```

The commutativity theorem needs a disjoint output domain assumption, because the `sbUnion` operator is only commutative for disjoint domains (see section 6.5). Furthermore, the commutativity is proven using the special equality for SPFs ($\triangleq_f$). Otherwise a type error would occur, because the output type `('fOut ∪ 'gOut)`$^\Omega$ is different to `('gOut ∪ 'fOut)`$^\Omega$

The composition definition returns the smallest fixpoint:

```
theorem spfcomp_belowI:
  fixes    f  ::"'fIn^Ω → 'fOut^Ω"
    and     g  ::"'gIn^Ω → 'gOut^Ω"
  assumes "f·(sb ⊎_ out⋆₁) ⊑ (out⋆₁)"
      and "g·(sb ⊎_ out⋆₂) ⊑ (out⋆₂)"
  shows    "(f⊗g)·sb ⊑ out"
```

To show equality further assumptions are required:.

```
theorem spfcomp_eqI:
  fixes    f  ::"'fIn^Ω → 'fOut^Ω"
  and      g  ::"'gIn^Ω → 'gOut^Ω"
  and      out::"('fOut ∪ 'gOut)^Ω"
  assumes "chDom TYPE ('fOut) ∩ chDom TYPE ('gOut) = {}"
  and      "f·(sb ⊎_ out⋆₁) =(out⋆₁)"
  and      "g·(sb ⊎_ out⋆₂) = (out⋆₂)"
  and      "⋀z. f·(sb ⊎⋆ z) = (z⋆₁) ∧ g·(sb ⊎⋆ z) = (z⋆₂) ⟹ out ⊑ z"
  shows    "((f⊗g)·sb) = out"
```

Sequential and feedback compositions are a special cases of the general composition `spfComp`. They are useful to reduce the complexity since they work without computing the fixpoint. If the domains of two functions fulfill the sequential composition assumptions, following theorem can be used for an easier output evaluation.

```
theorem spfcomp_serial2:
  fixes f::"'fIn^Ω → 'fOut^Ω"
    and g::"'gIn^Ω → 'gOut^Ω"
  assumes "chDom TYPE ('gIn) ⊆ chDom TYPE ('fOut)"
    and "chDom TYPE ('fOut) ∩ chDom TYPE ('gOut) = {}"
    and "chDom TYPE('gOut) ∩ chDom TYPE('gIn) = {}"
    and "chDom TYPE('fOut) ∩ chDom TYPE('fIn) = {}"
    and "chDom TYPE('gOut) ∩ chDom TYPE('fIn) = {}"
  shows "(f ⊗ g)·sb = f·(sb⋆) ⊎ g·(f·(sb⋆)⋆)"
```

To ease the use of this important case, there is an explicit definition of the sequential composition:

```
definition spfCompSeq::"('In^Ω → 'Intern^Ω) → ('Intern^Ω → 'Out^Ω)
  → ('In^Ω → 'Out^Ω)" where
"spfCompSeq ≡ Λ spf1 spf2 sb. spf2·(spf1·sb)"
```

In the sequential case the general composition `spfComp` is equivalent to `spfCompSeq`. The output of the general composition is restricted to `'Out`, because the general composition also returns the internal channels.

```
theorem spfcomp_to_sequential:
  fixes f::"'In^Ω → 'Intern^Ω"
    and g::"'Intern^Ω → 'Out^Ω"
  assumes "chDom TYPE ('In) ∩ chDom TYPE ('Intern) = {}"
    and "chDom TYPE('In) ∩ chDom TYPE('Out) = {}"
    and "chDom TYPE('Intern) ∩ chDom TYPE('Out) = {}"
  shows "(f ⊗ g)·(sb⋆) | TYPE('Out) = spfCompSeq·f·g·sb"
```

The same holds for parallel compositions, the output of parallel composed functions is independent from the other functions output.

```
theorem spfcomp_parallel:
  fixes f::"'fIn^Ω → 'fOut^Ω"
    and g::"'gIn^Ω → 'gOut^Ω"
  assumes "chDom TYPE ('fOut) ∩ chDom TYPE ('gOut) = {}"
      and "chDom TYPE ('fOut) ∩ chDom TYPE ('gIn) = {}"
      and "chDom TYPE ('fOut) ∩ chDom TYPE ('fIn) = {}"
      and "chDom TYPE('gOut) ∩ chDom TYPE('gIn) = {}"
      and "chDom TYPE('gOut) ∩ chDom TYPE('fIn) = {}"
    shows "(f ⊗ g)·sb = f·(sb⋆) ⊎ g·(sb⋆)"
```

Similar to the sequential composition, we add a definition for the parallel case:

```
definition spfCompPar:: "('In1^Ω → 'Out1^Ω) → ('In2^Ω → 'Out2^Ω) →
  ('In1 ∪ 'In2)^Ω → ('Out1 ∪ 'Out2)^Ω" where
"spfCompPar ≡ Λ spf1 spf2 sb. spf1·(sb⋆1) ⊎  spf2·(sb⋆2)"
```

The two components may share input channels, otherwise all ports are disjunct. There is no communication between the components:

```
theorem spfcomp_to_parallel:
  fixes f::"'fIn^Ω → 'fOut^Ω"
    and g::"'gIn^Ω → 'gOut^Ω"
  assumes "chDom TYPE ('fOut) ∩ chDom TYPE ('gOut) = {}"
      and "chDom TYPE ('fOut) ∩ chDom TYPE ('gIn) = {}"
      and "chDom TYPE ('fOut) ∩ chDom TYPE ('fIn) = {}"
      and "chDom TYPE('gOut) ∩ chDom TYPE('gIn) = {}"
      and "chDom TYPE('gOut) ∩ chDom TYPE('fIn) = {}"
    shows "(f ⊗ g)·(sb⋆) | TYPE('fOut ∪ 'gOut) = spfCompPar·f·g·sb"
```

The feedback composition is different to the previous cases because there is only one component instead of two. It is also more complicated since a fixpoint is computed:

```
definition spfCompFeed ::"('In^Ω → 'Out^Ω) → ('In-'Out)^Ω → 'Out^Ω" where
"spfCompFeed ≡ Λ spf sb. μ sbOut. spf·(sb ⊎_ sbOut)"
```

Since the general composition takes two components instead of one like the feedback definition, one component is "removed" by assuming the output is empty. That way it does not contribute to any behaviour.

```
theorem spfcomp_to_feedback:
  fixes f::"'fIn^Ω → 'fOut^Ω"
```

```
  and g::"'gIn^Ω → 'gOut^Ω"
assumes "chDom TYPE ('gOut) = {}"
  shows "(f ⊗ g)·(sb⋆) | TYPE('fOut) = spfCompFeed·f·sb"
```

## 7.5 Overview of SPF Functions

In table 7.1 the functions over SPFs are shown. Notice that these are not all functions. Many functions from table 6.1 are also SPFs. Take for example `sbRt`. It is an continuous function from input bundle to output bundle. Hence it can be used as a component, especially in combination with the sequential composition operator.

| Def | Abbrev | Signature | Description |
|---|---|---|---|
| `spfType` | | $('I^\Omega \rightarrow 'O^\Omega)$ itself $\Rightarrow$ (channel set $\times$ channel set) | Signature of Component (7.3) |
| `spfDom` | | $('I^\Omega \rightarrow 'O^\Omega)$ itself $\Rightarrow$ channel set | Input Channels (7.3) |
| `spfRan` | | $('I^\Omega \rightarrow 'O^\Omega)$ itself $\Rightarrow$ channel set | Output Channels (7.3) |
| `spfIO` | | $('I1^\Omega \rightarrow 'O1^\Omega) \Rightarrow ('I1^\Omega \times 'O1^\Omega)$ set | Input/Output behaviour (7.3) |
| `spfEq` | $(\triangleq_f)$ | $('I1^\Omega \rightarrow 'O1^\Omega) \Rightarrow ('I2^\Omega \rightarrow 'O2^\Omega) \Rightarrow$ bool | Equality of SPF (7.3) |
| `spfConvert` | | $('a^\Omega \rightarrow 'b^\Omega) \rightarrow 'c^\Omega \rightarrow 'd^\Omega$ | Type Conversion |
| `spfComp` | $(\otimes)$ | $('I1^\Omega \rightarrow 'O1^\Omega) \rightarrow ('I2^\Omega \rightarrow 'O2^\Omega) \rightarrow (('I1 \cup 'I2) - ('O1 \cup 'O2))^\Omega \rightarrow ('O1 \cup 'O2)^\Omega$ | General Composition (7.4) |
| | $(\otimes_\star)$ | $('I1^\Omega \rightarrow 'O1^\Omega) \rightarrow ('I2^\Omega \rightarrow 'O2^\Omega) \rightarrow ('I3^\Omega \rightarrow 'O3^\Omega)$ | Composition with Typecast (7.4) |
| `spfCompSeq` | | $('In^\Omega \rightarrow 'Intern^\Omega) \rightarrow ('Intern^\Omega \rightarrow 'Out^\Omega) \rightarrow ('In^\Omega \rightarrow 'Out^\Omega)$ | Sequential Composition (7.4) |
| `spfCompPar` | | $('In1^\Omega \rightarrow 'Out1^\Omega) \rightarrow ('In2^\Omega \rightarrow 'Out2^\Omega) \rightarrow ('In1 \cup 'In2)^\Omega \rightarrow ('Out1 \cup 'Out2)^\Omega$ | Parallel Composition |
| `spfCompFeed` | | $('In^\Omega \rightarrow 'Out^\Omega) \rightarrow ('In - 'Out)^\Omega \rightarrow 'Out^\Omega$ | Feedback Composition |

Table 7.1: Functions for SPFs

# Chapter 8

# Stream Processing Specification

In this chapter we extend out mathematical model to include two rather interesting aspects of software development, namely underspecification and refinement. From a developers point of view, it is irrelevant, whether a system is underspecified (further refinement steps during the development process can make specifications more precise), or developers allow the implementation to make non-deterministic decisions at runtime.

A single deterministic SPF is not sufficient to describe all possible component behaviors, and instead a set of stream processing function is used to model the component behavior properly [RR11]. The mathematical theory of sets has the phenomenal property that set inclusion corresponds to property implication and thus refinement as development step. Only the signature, i.e. the input and output channels of components are fixed, thus all SPFs in such a set must have the same input and output channels.

## 8.1 Mathematical Definition

We define a stream processing specification as a set of stream processing functions with fixed input and output channels [Rum96].

**Definition 8.1** (SPS)**.** Let $C$ be the set of all possible channels and $I, O \subseteq C$. We define the SPS type $SPS_{I,O}$ with input channels $I$ and output channels $O$ as shown below:

$$SPS_{I,O} := \mathbb{P}(I^{\Omega} \to O^{\Omega})$$

## 8.2 General Composition of SPSs

With our general composition operator for SPFs we can also define the general composition operator for SPSs. It composes every combination of SPFs possible from both input SPSs.

```
definition spsComp::
"('I1Ω → 'O1Ω) set ⇒ ('I2Ω → 'O2Ω) set ⇒
  ((('I1 ∪ 'I2) - 'O1 ∪ 'O2)Ω → ('O1 ∪ 'O2)Ω) set" (infixr "⊗" 70)
where "spsComp F G = {f ⊗ g | f g. f∈F ∧ g∈G }"
```

Refinement of a component in a decomposed structure automatically leads to refinement of the composition [BR07].

This is proven in the following theorem:

```
theorem spscomp_refinement:
fixes F::"('I1Ω → 'O1Ω) set"
  and G::"('I2Ω → 'O2Ω) set"
  and F_ref::"('I1Ω → 'O1Ω) set"
  and G_ref::"('I2Ω → 'O2Ω) set"
assumes "F_ref ⊆ F"
    and "G_ref ⊆ G"
shows "(F_ref ⨂ G_ref) ⊆ (F ⨂ G)"
```

This important property enables independent modification of the modules while preserving properties of the overall system. As long as the modification is a refinement, it does not influence the other components. The resulting component `F_ref` can simply replace `F` in the composed system. Since the result is a refinement, the correctness is still proven.

That way the focus is on the development of each component and not on the integration into the overall system.

Properties of the original system `S` directly hold for the refined version `S'`:

```
theorem assumes "∀f∈S. P f" and "S' ⊆ S"
  shows "∀f'∈S'. P f'"
```

We call a SPS consistent if it is not the empty set. Because the empty set contains no possible behaviour there is no implementation of such a component. Therefore such an SPS can not be used in a real system.

```
definition spsIsConsistent :: "('I1Ω → 'O1Ω) set ⇒ bool" where
"spsIsConsistent sps ≡ (sps ≠ {})"
```

If two SPSs are consistent then the composition of these is also consistent.

```
theorem spscomp_consistent:
 fixes F::"('I1Ω → 'O1Ω) set"
   and G::"('I2Ω → 'O2Ω) set"
 assumes "spsIsConsistent F"
     and "spsIsConsistent G"
   shows "spsIsConsistent (F ⨂ G)"
```

Composing two SPS that fulfill different input output behaviour predicates results in a subset of all possible SPFs that fulfill both behaviour predicates.

```
theorem spscomp_subpred:
 fixes P::"'I1Ω ⇒ 'O1Ω ⇒ bool"
   and H::"'I2Ω ⇒ 'O2Ω ⇒ bool"
 assumes "chDom TYPE ('O1) ∩ chDom TYPE ('O2) = {}"
     and "∀spf∈S1. ∀sb. P sb (spf·sb)"
     and "∀spf∈S2. ∀sb. H sb (spf·sb)"
  shows "S1 ⨂ S2 ⊆
          {g. ∀sb.
               let all = sb ⊎ g·sb in
```

```
            P (all⋆) (all⋆) ∧ H (all⋆) (all⋆)
        }"
```

## 8.3 Special Composition of SPSs

Next we lift the sequential composition (spfCompSeq) to compose two SPSs.

```
definition spsCompSeq  :: "('In^Ω → 'Intern^Ω) set ⇒ ('Intern^Ω → 'Out^Ω) set
    ⇒ ('In^Ω → 'Out^Ω) set" where
"spsCompSeq sps1 sps2 = {spfCompSeq·spf1·spf2 | spf1 spf2.
                                 spf1 ∈ sps1 ∧ spf2 ∈ sps2}"
```

After applying this operator the resulting set contains the sequential composition of every combination of SPFs from both SPSs.

```
theorem spscfcomp_set:
  assumes "spf1 ∈ sps1"
      and "spf2 ∈ sps2"
  shows "spfCompSeq·spf1·spf2 ∈ spsCompSeq sps1 sps2"
```

If we compose two consistent SPSs then the result is again consistent.

```
theorem spscfcomp_consistent:
    assumes "spsIsConsistent sps1"
        and "spsIsConsistent sps2"
      shows "spsIsConsistent (spsCompSeq sps1 sps2)"
```

The sequential composition is monotonic. The sequential composition of two refined components has as an result again a refinement:

```
theorem spscfcomp_mono: assumes "sps1_ref ⊆ sps1"
  and "sps2_ref ⊆ sps2"
  shows "(spsCompSeq sps1_ref sps2_ref) ⊆ (spsCompSeq sps1 sps2)"
```

The parallel and feedback composition is lifted to SPS the same way. Definition and lemmata are shown in the appendix.

## 8.4 SPS Completion

SPS S consists of a set of functions, which each describe deterministic behaviour of a component. Upon a concrete execution, i.e. input stream i the externally visible behaviour is f(i) for an f∈S.

It may happen that for streams $i_1, i_2$ we have $f_1(i_1) = o_1$ and $f_2(i_2) = o_2$, but that no "joint" f∈S exists, where $f(i_1) = o_1$ and $f(i_2) = o_2$. We then speak of an incomplete specification S. From an observational point, S and S∪{f} cannot be distinguished, but when refinement is used to specialize S, this may become a deficit.

We therefore introduce the completion operator spsComplete to include all possible functions of a component such that the black-box behaviour of the component does not change.

```
definition spsComplete ::"('I1^Ω → 'O1^Ω) set ⇒ ('I1^Ω → 'O1^Ω) set"
  where "spsComplete sps = {spf. ∀sb. ∃spf2∈sps. spf·sb = spf2·sb}"
```

By definition, the SPSs behaviour will not be changed.

We give a small example for the completion of two components on the datatype containg just `a` and `b`.

- `spsConst = {[a ↦ a, b ↦ a], [a ↦ b, b ↦ b]}`
- `spsID = {[a ↦ a, b ↦ b], [a ↦ b, b ↦ a]}`

The first component contains two constant functions which have the output a or b regardless of the input. The second component contains the identity function as well as a function that reverses a and b. Therefore `spsConst` and `spsID` are different components. However they can not be distinguished by their black-box behaviour: `spsIO spsConst = {(a,a),(a,b),(b,a),(b,b)} = spsID spsConst`. If we complete both sets then both components are equal: `spsComplete spsConst = spsComplete spsID = {[a ↦ a, b ↦ a], [a ↦ b, b ↦ b], [a ↦ a, b ↦ b], [a ↦ b, b ↦ a]}`.

Completion is often used to show that a completed component `S2` is the extension of another component `S1`. By definition this holds if for every function in `S1` and possible input there is a function in `S2` with the same output behaviour.

```
theorem spscomplete_belowI:
  assumes "⋀spf sb. spf∈S1 ⟹ ∃spf2 ∈ S2. spf·sb = spf2·sb"
  shows "S1 ⊆ spsComplete S2"
```

With this we can show that completion just adds new SPFs to the SPS and does not remove any.

```
theorem spscomplete_below: "sps ⊆ spsComplete sps"
```

After applying the `spsComplete` function the SPS is indeed complete. Applying the function a second time does not change the component anymore. Completion is idempotent.

```
theorem spscomplete_complete [simp]:
 "spsComplete (spsComplete sps) = spsComplete sps"
```

We call a SPS complete if it is the same after completion.

```
definition spsIsComplete :: "('I1^Ω → 'O1^Ω) set ⇒ bool" where
"spsIsComplete sps ≡ (spsComplete sps) = sps"
```

There are certain sets that are not changed by completion. For example the empty set is complete.

```
theorem spscomplete_empty[simp]: "spsIsComplete {}"
```

Completing a set consisting of a single SPF does not change the set.

```
theorem spscomplete_one[simp]: "spsIsComplete {f}"
```

This also holds for the set of all possible functions. Hence, the set of all functions is the same after completion.

```
theorem spscomplete_univ[simp]: "spsIsComplete UNIV"
```

The `spsComplete` function is monotonic. Therefore if a component `sps1` refines a second component `sps2` then this also holds after completion.

```
theorem spscomplete_mono: assumes "sps1 ⊆ sps2"
  shows "spsComplete sps1 ⊆ spsComplete sps2"
```

But completeness is not ensured after refinement

## 8.5 Overview of SPS Functions

In table 8.1 the functions over SPSs are shown. The first rows are general functions over sets. Then the SPSs specific definitions follow.

| Notation | Abbrev | Signature | Description |
|---|---|---|---|
| empty | {} | 'a set | Empty Component |
| UNIV | | 'a set | Greatest Component |
| member | ∈ | 'a ⇒ 'a set ⇒ $\mathbb{B}$ | check if SPF is member |
| union | ∪ | 'a set ⇒ 'a set ⇒ 'a set | Union over Sets |
| inter | ∩ | 'a set ⇒ 'a set ⇒ 'a set | Intersection over Sets |
| image | ` | ('a ⇒ 'b) ⇒ 'a set ⇒ 'b set | Apply function to every element |
| spsIO | | $('I1^{\Omega} \to 'O1^{\Omega})$ set ⇒ $('I1^{\Omega} \times 'O1^{\Omega})$ set | Behaviour Relation |
| spsIOtoSet | | $('I1^{\Omega} \times 'O1^{\Omega})$ set ⇒ $('I1^{\Omega} \times 'O1^{\Omega})$ set | Get complete SPS from I/O behaviour |
| spsComplete | | $('a^{\Omega} \to 'b^{\Omega})$ set ⇒ $('a^{\Omega} \to 'b^{\Omega})$ set | Greatest SPS with same behaviour (8.4) |
| spsComp | ⊗ | $('I1^{\Omega} \to 'O1^{\Omega})$ set ⇒ $('I2^{\Omega} \to 'O2^{\Omega})$ set ⇒ $((('I1 \cup 'I2)-('O1 \cup 'O2))^{\Omega} \to ('O1 \cup 'O2)^{\Omega})$ set | General Composition (8.2) |
| spsCompSeq | | $('In^{\Omega} \to 'Intern^{\Omega})$ set ⇒ $('Intern^{\Omega} \to 'Out^{\Omega})$ set ⇒ $('In^{\Omega} \to 'Out^{\Omega})$ set | Sequential Composition (8.3) |
| spsCompPar | | $('In1^{\Omega} \to 'Out1^{\Omega})$ set ⇒ $('In2^{\Omega} \to 'Out2^{\Omega})$ set ⇒ $(('In1 \cup 'In2)^{\Omega} \to ('Out1 \cup 'Out2)^{\Omega})$ set | Parallel Composition |
| spsCompFeed | | $('In^{\Omega} \to 'Out^{\Omega})$ set ⇒ $(('In - 'Out)^{\Omega} \to 'Out^{\Omega})$ set | Feedback Composition |
| spsIsConsistent | | $('I1^{\Omega} \to 'O1^{\Omega})$ set ⇒ $\mathcal{B}$ | Set is not empty |
| spsIsComplete | | $('I1^{\Omega} \to 'O1^{\Omega})$ set ⇒ $\mathcal{B}$ | Component is complete (8.4) |

Table 8.1: Functions for SPSs

# Chapter 9

# Case Study: Cruise Control

We demonstrate the datatype and function definition from the previous chapters on a case study. Especially interesting is the specification of a single component and the composition of multiple components. All different kinds of composition (parallel, sequential and feedback) are used in this case study.

The case study is part of a cruise control system. The input is the current acceleration. Internally the system adds the acceleration to the last known speed and returns the current speed. The initial speed is set to zero.

We evaluate the stream bundle (SB) and stream-processing function (SPF) structures by proving that the bundle-based specification is equal to the analogue stream-based specification.

CNC

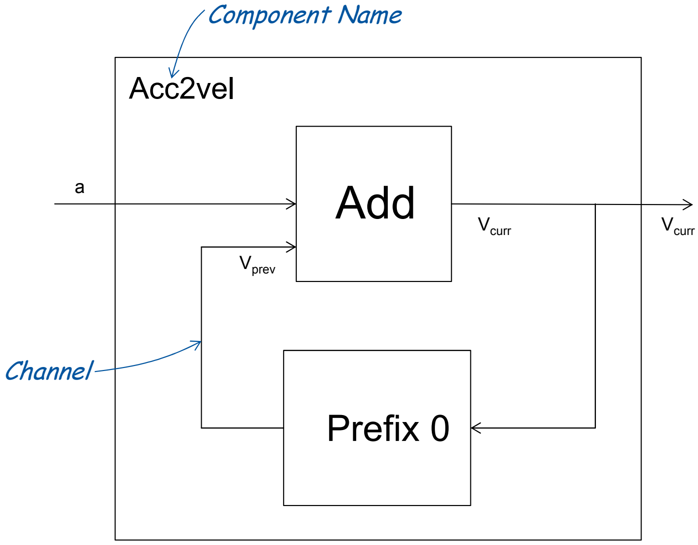



Figure 9.1: Case Study Overview

The component network shown in fig. 9.1 depicts the high level components for the case study. One component initializes the sequence by a 0. The other component performs the addition and has a well-defined interface: input channels a and $v_{prev}$ and output channel $v_{cur}$ denoted by arrows in the figure.

**Channel and Message Datatypes** The case-study consists of three channels. They are named cA cVcurr and cVprev.

```
datatype channel = cA | cVcurr | cVprev |  cempty
```

Furthermore, the channel cempty is added to the datatype, because there must always be a channel on which nothing can be transmitted (see section 6.2).

The messages are all natural numbers. Hence M does not have to be a new datatype, instead it is set to nat.

```
type_synonym M = nat
```

Channel cempty may not contain a message. For every other channel every nat-message can be sent. The definition UNIV is the set containing all nat-values.

```
fun ctype :: "channel ⇒ M set" where
"ctype cempty   = {}" |
"ctype   _      = UNIV"
```

As always, a theorem that confirms the existence of an empty channel has to be provided for the framework theories.

```
theorem ctypeempty_ex: "∃c. ctype c = {}"
```

Now we are going to define the signature of the components. The Add component has the signature $\{cA, cVprev\}^{\Omega} \rightarrow \{cVcurr\}^{\Omega}$. The Prefix0 component has the signature $\{cVcurr\}^{\Omega} \rightarrow \{cVprev\}^{\Omega}$. For each of theses sets we create a new type. Since $\{cVcurr\}^{\Omega}$ is both the output of Add and the input of Prefix0 there are only three definitions.

```
typedef addIn = "{cVprev, cA}"
typedef addOut = "{cVcurr}"         — also prefixIn
typedef prefixOut = "{cVprev}"
```

To use the datatypes to define bundles, they have to be instantiated in the chan class:

```
instantiation addIn::chan
begin
  definition Rep_addIn_def: "Rep = Rep_addIn"
end
```

As mentioned in section 6.2, each of the types need a representation function Rep.

```
instantiation addOut::chan
begin
  definition Rep_addOut_def: "Rep = Rep_addOut"
end
```

By using typedef to define the domain types over channels, a representation function is provided and can be used.

```
instantiation prefixOut::chan
begin
  definition Rep_prefixOut_def: "Rep = Rep_prefixOut"
end
```

**Prefix component** The prefix component is essentially a identity component with an additional initial output. The identity component with the signature `addOut`$^\Omega$ → `prefixOut`$^\Omega$ is definable by renaming the channel of the input SB (`cVcurr`) to the channel the output SB (`cVprev`).

```
definition prefixRename :: "addOutΩ → prefixOutΩ" where
"prefixRename = sbRename_part [Abs cVcurr ↦ Abs cVprev]"
```

Correct behavior is proven in the following theorem, the output stream is equal to the input stream.

```
theorem prefrename_getch:
  "prefixRename·sb ▸ (Abs cVprev) = sb ▸ (Abs cVcurr)"
```

Because one initial output element is needed for the prefix component, the initial output can be represented by a stream bundle element (sbElem). Thus, a lifting function from natural numbers to an output sbElem is defined.

```
lift_definition initOutput:: "nat ⇒ prefixOut√" is
"λinit. Some (λ_. init)"
```

By appending the initial output sbElem to an output SB of the identity component, the complete output of the prefix component can be defined.

```
definition prefixPrefix:: "M ⇒ prefixOutΩ → prefixOutΩ" where
"prefixPrefix init = sbECons (initOutput init)"
```

Therefore, the prefix component is defined by a sequential composition of the identity component `prefixRename` and the appending component `prefixPrefix` with an inital output.

```
definition prefixComp'::"nat ⇒ addOutΩ → prefixOutΩ" where
"prefixComp' init = spfCompSeq· prefixRename· (prefixPrefix init)"
```

The same prefix component can also be defined in a more direct manner by outputting a stream that starts with an initial output and then outputs the input stream from the input SB.

```
lift_definition prefixComp::"nat ⇒ addOutΩ → prefixOutΩ" is
"λinit sb. Abs_sb (λ_. ↑init • sb ▸ (Abs cVcurr))"
```

Both definitions model the same component. This is proven in the following theorem:

```
theorem "prefixComp init = prefixComp' init"
```

In the following, `prefixComp` is used to define the complete system.

**Add component** The add component is defined by using an element-wise add function for streams and applying it to both input streams.

```
lift_definition addComp::"addInΩ → addOutΩ" is
"λsb. Abs_sb (λ_. add·(sb ▸ Abs cA)·(sb ▸ Abs cVprev))"
```

The output on channel `cVcurr` follows directly:

```
theorem addcomp_getch:
  "addComp·sb ▸ (Abs cVcurr) = add·(sb ▸ Abs cA)·(sb ▸ Abs cVprev)"
```

The length of the output is the minimal length of the two input streams.

```
theorem add_len:"#(addComp·sb) = min (#(sb ▸ Abs cA)) (#(sb ▸ Abs cVprev))"
```

Since the lenght over bundles is defined as the minimum, the property can be simplified:

```
theorem "#(addComp·sb) = #sb"
```

**Acc2vel component** The composed components behavior is definable by outputting the addition of the input element and the previous output element (or 0 for the initial input element).

```
definition streamSum::"nat stream → nat stream" where
"streamSum ≡ sscanl (+) 0"
```

Unfolding the definition once leads to the following recursive equation:

```
theorem "streamSum·s = add·s·(↑0 • streamSum·s)"
```

For the composed system unfolding leads to a similar result:

```
theorem comp_unfold: "((addComp ⊗ (prefixComp init))·sb) ▸ Abs cVcurr
    = add·(sb  ▸  Abs cA)·
         (↑init • (addComp ⊗ prefixComp init)·sb  ▸  Abs cVcurr)"
```

While the recursive equations are nearly identical, equality does not directly follow from it since there might be multiple fixpoints which fulfill the recursive equation.

Hence we prove that there is only one fixpoint for the equation. In the lemma `rek2sscanl` the variable `z` is an arbitrary fixpoint. The lemma shows that `z` is the only fixpoint and equivalent to `sscanl`.

```
theorem rek2sscanl:
  assumes "⋀input init. z init·input = add·input·(↑init • z init·input)"
    shows "z init·s = sscanl (+) init·s"
```

Following from this statement, the composition of the `add` and `prefix` component can be evaluated.

```
theorem "(addComp ⊗ (prefixComp init))·sb ▸ Abs cVcurr
      = sscanl (+) init·(sb ▸ Abs cA)"
```

The composition can also be tested over input streams.

```
theorem
 "(addComp ⊗ prefixComp 0)·(Abs_sb (λc. <[1,1,1,0,0,2]>)) ▸ Abs cVcurr
   = <[1,2,3,3,3,5]>"
```

**Non-Deterministic Component** Now we define a non-deterministic component. In this example the component randomly modifies the output. This is used to model impreciseness of the actuator. The actuator is unable to exactly follow the control-command from the `Acc2val` component, instead there exists a delta. This is modeled in the following definition:

```
definition realBehaviour::"nat ⇒ nat set" where
"realBehaviour n ≡ if n<50 then {n} else {n-5 .. n+5}"
```

The actuator can perfectly execute the control command for small values (`n<50`). There is only one reaction: `{n}`. But for greater input, there may exist an error. Here it is a delta of at most `5`, resulting in the possible outputs `{n-5 .. n+5}`.

Now the `realBehaviour` has to be applied to every element in the stream. For this we create a general helper-function, similar to the deterministic `smap`.

```
definition ndetsmap::"('a ⇒ 'b set)
                ⇒ ('a stream → 'b stream) set" where
"ndetsmap T = gfp (λH. {f | f. (f·ε=ε)
        ∧ (∀m s. ∃x g. (f·(↑m•s) = ↑x • g·s) ∧ x∈(T m) ∧ g∈H)})"
```

The component is a set of stream processing functions. Each function returns $\varepsilon$ on the input $\varepsilon$. When the input starts with a message `m` the output one of the possible values described in `T`. The `gfp` operator returns the greatest fixpoint fulfilling the recursive equation.

The two functions are combined to create the final component:

```
definition errorActuator::"(nat stream → nat stream) set" where
"errorActuator = ndetsmap realBehaviour"
```

The component is consistent, there exists a function which is in the description. For example the identity function (`ID`).

```
theorem "ID ∈ errorActuator"
```

The length is not modified by `errorActuator`:

```
theorem error_len:
  assumes "spf ∈ errorActuator"
  shows "#(spf·s) = #s"
```

If the input consists *only* of values less than 50 there is no error. The actuator perfectly follows the commands.

```
theorem assumes "⋀n. n∈sValues·s ⟹ n<50"
            and "spf ∈ errorActuator"
  shows "spf·s = s"
```

If the input is larger than 50, errors can occur. Here an example for the input with an infinite repetition of `n`. The output is non-deterministic. But the values must lie between `{n-5 .. n+5}`.

```
theorem assumes "50 ≤ n"
            and "spf ∈ errorActuator"
          shows "sValues·(spf·(sinftimes (↑n))) ⊆ {n-5 .. n+5}"
```

# References


[Bal06] Clemens Ballarin. "Interpretation of locales in Isabelle: Theories and proof contexts". In: *International Conference on Mathematical Knowledge Management*. Springer. 2006, pp. 31–43.

[Bie97] Armin Biere. "$\mu$cke - Efficient $\mu$-Calculus Model Checking". In: *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*. 1997, pp. 468–471. DOI: 10.1007/3-540-63166-6_50.

[BR07] Manfred Broy and Bernhard Rumpe. "Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung". In: *Informatik-Spektrum* 30.1 (2007), pp. 3–18. ISSN: 0170-6012.

[Bro+92] Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner, and Rainer Weber. *The design of distributed systems: An Introduction to FOCUS*. 1992.

[Bro13] Manfred Broy. *Informatik: Eine grundlegende Einführung Teil I. Problemnahe Programmierung*. Springer-Verlag, 2013.

[BS01] Manfred Broy and Ketil Stølen. *Specification and development of interactive systems: Focus on streams, interfaces, and Refinement*. New York: Springer, 2001. ISBN: 1461300916.

[Bur+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. "Symbolic model checking: $10^{20}$ States and beyond". In: *Information and Computation* 98.2 (1992), pp. 142–170. ISSN: 0890-5401.

[Bür17] Jens Christoph Bürger. *Modular Hierarchical Modelling and Verification of Systems using General Composition Operators*. 2017.

[Cou+12] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed systems - concepts and designs (5. ed.)* International computer science series. Addison-Wesley, 2012. ISBN: 978-0-13-214301-1.

[DGM97] Marco Devillers, W. O. David Griffioen, and Olaf Müller. "Possibly Infinite Sequences in Theorem Provers: A Comparative Study". In: *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97, Murray Hill, NJ, USA, August 19-22, 1997, Proceedings*. Ed. by Elsa L. Gunter and Amy P. Felty. Vol. 1275. Lecture Notes in Computer Science. Springer, 1997, pp. 89–104. ISBN: 3-540-63379-0. DOI: 10.1007/BFb0028381. URL: https://doi.org/10.1007/BFb0028381.

[EKM98] Jacob Elgaard, Nils Klarlund, and Anders Møller. "MONA 1.x: new techniques for WS1S and WS2S". In: *Computer-Aided Verification, (CAV '98)*. Vol. 1427. LNCS. Springer-Verlag, 1998, pp. 516–520.

[GR06] Borislav Gajanovic and Bernhard Rumpe. *Isabelle/HOL-Umsetzung strombasierter Definitionen zur Verifikation von verteilten, asynchron kommunizierenden Systemen*. Tech. rep. Technical Report Informatik-Bericht 2006-03, Braunschweig University of Technology, 2006.

[Hal90] Anthony Hall. "Seven Myths of Formal Methods". In: *IEEE Software* 7.5 (1990), pp. 11–19. DOI: 10.1109/52.57887.

[Hei+15] Robert Heim, Pedram Mir Seyed Nazari, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. "Modeling robot and world interfaces for reusable tasks". In: *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*. IEEE. 2015, pp. 1793–1798.

[HF11] Florian Hölzl and Martin Feilkas. *13 autofocus 3-a scientific tool prototype for model-based development of component-based, reactive, distributed systems*. Model-Based Engineering of Embedded Real-Time Systems, pages 317–322. Springer, 2011.

[Hoa78] C. A. R. Hoare. "Communicating Sequential Processes". In: *Commun. ACM* 21.8 (1978), pp. 666–677.

[HR04] David Harel and Bernhard Rumpe. "Meaningful Modeling: What's the Semantics of "Semantics"?" In: *IEEE Computer* 37.10 (2004), pp. 64–72.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. *MontiArc - Architectural modeling of interactive distributed and cyber-physical systems*. Vol. 2012,3. Technical report / Department of Computer Science, RWTH Aachen. Aachen, Hannover, and Göttingen: RWTH, Technische Informationsbibliothek u. Universitätsbibliothek, and Niedersächische Staats- und Universitätsbibliothek, 2012.

[HSE97] Franz Huber, Bernhard Schätz, and Geralf Einert. "Consistent graphical specification of distributed systems". In: *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods* (1997), pp. 122–141.

[Huf12] Brian Charles Huffman. *HOLCF '11: A definitional domain theory for verifying functional programs*. [Portland, Or.]: Portland State University, 2012.

[Kah74] Gilles Kahn. "The Semantics of a Simple Language for Parallel Programming". In: *Information Processing '74: Proceedings of the IFIP Congress*. Ed. by J. L. Rosenfeld. New York, NY: North-Holland, 1974, pp. 471–475.

[Kau17] Hendrik Kausch. *Spezifikation und Verifikation von Gezeiteten Interaktiven Component-and-Connector Modellen mit dem Theorembeweiser Isabelle*. 2017.

[Kle52] Stephen Cole Kleene. *Introduction to metamathematics*. Vol. v. 1. Bibliotheca mathematica. A series of monographs on pure and applied mathematics. Groningen: Wolters-Noordhoff Pub, 1952. ISBN: 9780720421033.

[Lee09] Edward A. Lee. "Computing needs time". In: *Commun. ACM* 52.5 (2009), pp. 70–79.

[Lee16] Edward A. Lee. "Fundamental Limits of Cyber-Physical Systems Modeling". In: *ACM Transactions on Cyber-Physical Systems* 1.1 (Nov. 2016). URL: http://chess.eecs.berkeley.edu/pubs/1183.html.

[Mao+17] Shahar Maoz, Nitzan Pomerantz, Jan Oliver Ringert, and Rafi Shalom. "Why is My Component and Connector Views Specification Unsatisfiable?" In: *20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17-22, 2017*. 2017, pp. 134–144. DOI: 10.1109/MODELS.2017.26. URL: https://doi.org/10.1109/MODELS.2017.26.

[Mil89] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989. ISBN: 978-0-13-115007-2.

[Mil99] Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999. ISBN: 978-0-521-65869-0.

[Mou+15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. "The Lean Theorem Prover (System Description)". In: *Automated Deduction - CADE-25*. Cham: Springer International Publishing, 2015, pp. 378–388.

[Mül+99] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oscar Slotosch. "HOLCF = HOL + LCF". In: *Journal of Functional Programming* 9.2 (1999), pp. 191–223.

[Mül18] Jan Müllers. *Erweiterung eines Zeit-Sensitiven Frameworks zur Verifikation der Übertragungskorrektheit Fairer und Zustandsbasierten Netzwerkkomponenten*. 2018.

[Nip13] Tobias Nipkow. *Programming and proving in Isabelle/HOL*. 2013.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A proof assistant for Higher-Order Logic*. Vol. 2283. Lecture notes in artificial intelligence. Berlin [etc.]: Springer, 2002. ISBN: 9783540433767.

[Pau90] Lawrence C Paulson. *Logic and computation: interactive proof with Cambridge LCF*. Vol. 2. Cambridge University Press, 1990.

[PB10] Lawrence C. Paulson and Jasmin Christian Blanchette, eds. *Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers*. 2010.

[Pet66] C. A. Petri. *Communication with automata*. 1966.

[Reg94] Franz Regensburger. "HOLCF: eine konservative Erweiterung von HOL um LCF". PhD thesis. Technical University Munich, Germany, 1994.

[Rei12] Wolfgang Reisig. *Petri nets: an introduction*. Vol. 4. Springer Science & Business Media, 2012.

[Rin14] Jan Oliver Ringert. "Analysis and synthesis of interactive component and connector systems". PhD thesis. RWTH Aachen University, Germany, 2014. ISBN: 978-3-8440-3120-1.

[RR11] Jan Oliver Ringert and Bernhard Rumpe. "A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing". In: *Int. J. Software and Informatics* 5.1-2 (2011), pp. 29–53.

[RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20, 2014.: Shaker Verlag, 2014.

[Rum96] Bernhard Rumpe. “Formale Methodik des Entwurfs verteilter objektorientierter Systeme”. PhD thesis. 1996. ISBN: 978-3-89675-149-2.

[SK95] Kenneth Slonneger and Barry L. Kurtz. *Formal syntax and semantics of programming languages - a laboratory based approach*. Addison-Wesley, 1995. ISBN: 978-0-201-65697-8.

[Slo17] Dennis Slotboom. *Ein Verifikationsframework für Zeitsensitive Verteilte Systeme*. 2017.

[Spi08] Maria Spichkova. *Specification and Seamless Verification of Embedded Real-Time Systems: FOCUS on Isabelle*. Saarbrücken: VDM Verlag Dr. Müller Aktiengesellschaft & Co. KG, 2008. ISBN: 978-3836494526.

[Stü16] Sebastian Stüber. *Eine domain-theoretische Formalisierung von strombündelverarbeitenden Funktionen in Isabelle*. 2016.

[Tra09] David Trachtenherz. “Eigenschaftsorientierte Beschreibung der logischen Architektur eingebetteter Systeme”. PhD thesis. Institut für Informatik, Technische Universität München, 2009.

[Wen02] Markus Wenzel. “Isabelle, Isar - a versatile environment for human readable formal proof documents”. PhD thesis. Technical University Munich, Germany, 2002.

[Wia17] Marc Wiartalla. *Compositional Modelling and Verification of Distributed Systems with special Composition Operators*. 2017.

[www13] *The MONA Project*. http://www.brics.dk/mona/. Accessed 09/2013. 2013.

[www18] University of Cambridge and Technische Universität München. *Isabelle*. Accessed 09/2018. 2018. URL: https://isabelle.in.tum.de/.

[Zel17] Oliver Zelmat. *Verifikation von Zeitsensitiven Zustandsbasierten Netzwerkkomponenten*. 2017.

# Glossary

**admissible** A predicate is admissible if it holds for the lub of a chain in whenever it holds for the elements of the chain. 13, 24

**bottom** Least element in a complete partial order. 19, 23

**chain** Totally ordered set with minimal element. 6, 24

**complete partial order (cpo)** A partial order in which every chain has a least upper bound. 7, 18, 24, 28, 65

**continuous** The least upper bound is preserved after application of the function. 9, 19, 23, 33–35, 62

**discrete partial order** A partial order on $=$. 7

**feedback** Special kind of composition. Output channels are used as input of the same component. 62–64, 76

**general** Most general composition, can describe every system. 62, 64

**isar** A proof language in Isabelle. Designed to be similar to handwritten proofs. 20, 21

**lazy natural number (lNat)** Natural numbers extended with an infinity-element. 24, 25

**least fix point (lfp)** Used to define the semantic of recursive definitions. 9, 19, 25

**monotonic** The order is preserved after application of the function. 9

**parallel** Special kind of composition. The two components do not share channels. 62–64, 76

**partial order (po)** A reflexive, transitive and antisymmetric relation. 6, 23, 24, 33

**pointed complete partial order (pcpo)** A complete partial order with a bottom element. 7, 8, 19, 23, 24, 28, 48, 62

**sequential** Special kind of composition. The output of the first component is the input of the second component. 62, 64, 76

**stream bundle (SB)** Combination of multiple streams. 4, 5, 26, 40, 41, 46–60, 76, 78

**stream bundle element (sbElem)** Combination of multiple elements. 78

**stream processing specification (SPS)** Set of stream-processing functions, used to described non-deterministic behaviour. 4, 41, 70–75

**stream-processing function (SPF)** Continuous function from bundle to bundle. 4, 5, 34, 41, 42, 61–66, 68–73, 76

# Appendices

# Appendix A

# Extensions of HOLCF Theories

## A.1 Prelude

```
section ⟨Prelude⟩

theory Prelude
imports HOLCF psl.PSL
begin

default_sort type

(* allows to use lift_definition for continuous functions *)
setup_lifting type_definition_cfun


sledgehammer_params [smt_proofs=false]


lemma trivial[simp]: "(Not ∘ Not) = id"
  by auto

text ⟨Convert a relation to a map (function with ⟨option⟩al result).⟩
definition rel2map ::   "('c * 'm) set ⇒ ('c ⇀ 'm)"
where rel2map_def: "rel2map r ≡ λx. (if x ∈ Domain r then Some (SOME a. (x,a) ∈ r) else None)"


lemma [simp]: "Map.dom (rel2map r) = Domain r"
by (simp add: rel2map_def Map.dom_def)

text ⟨Unwrapping an ⟨'a option⟩ value. Result for ⟨None⟩ is undefined.⟩
definition unsome :: "'a option ⇒ 'a" where
"unsome x = (case x of Some y ⇒ y | None ⇒ undefined)"


lemma [simp]: "unsome (Some x) = x"
by (simp add: unsome_def)

text ⟨For natural numbers j and k with @{term "j ≤ k"}, @{term "k - j"} is natural as well⟩
lemma natl1: "(j::nat) ≤ k⟹∃i. j + i = k"
apply (simp add: atomize_imp)
apply (rule_tac x="j" in spec)
apply (induct_tac k, auto)
by (case_tac "x", auto)


lemma natl2: "(i::nat) + k = k + i"
by auto


primrec lrcdups :: "'a list ⇒ 'a list"
where
  "lrcdups [] = []" |
  "lrcdups (x#xs) =
     (if xs = []
        then [x]
        else
          (if x = List.hd xs
             then lrcdups xs
             else (x#(lrcdups xs))))"


primrec lscanl :: "('b ⇒ 'a ⇒ 'b) ⇒ 'b ⇒ 'a list ⇒ 'b list" where
```

```
"lscanl f e [] = [] " |
"lscanl f e (x#xs) =f e x#(lscanl f (f e x) xs)"

primrec lscanlAg::
"('s ⇒'a ⇒ ('s ×'b)) ⇒
's ⇒ 'a list ⇒ (('s × 'b) list)" where
"lscanlAg f s [] = []" |
"lscanlAg f s (x#xs) =
              (f s x)#(lscanlAg f (fst (f s x)) xs)"


definition stateSemList:: "('state ⇒ 'a ⇒ 'state × 'b) ⇒'state ⇒ 'a list ⇒ 'b list" where
"stateSemList f state xs ≡ map snd (lscanlAg f state xs)"

(* ------------------------------------------------------------------------ *)
section ⟨Some auxiliary HOLCF lemmas⟩
(* ------------------------------------------------------------------------ *)

subsection ⟨cfun⟩

text ⟨Introduction of continuity of ⟨f⟩ using monotonicity and lub on chains:⟩
lemma contI2:
  "⟦monofun (f::'a::cpo ⇒ 'b::cpo);
        (∀Y. chain Y⟶f (⨆i. Y i) ⊑ (⨆i. f (Y i)))⟧⟹cont f"
apply (rule contI)
apply (rule is_lubI)
apply (rule ub_rangeI)
apply (rule monofunE [of f], assumption)
apply (rule is_ub_thelub, assumption)
apply (erule_tac x="Y" in allE, drule mp, assumption)
apply (rule_tac y="⨆i. f (Y i)" in below_trans, assumption)
apply (rule is_lub_thelub)
by (rule ch2ch_monofun [of f], assumption+)


lemma [simp]: "cont (λ f. f x)"
apply (rule contI)
apply (subst lub_fun, assumption)
apply (rule thelubE)
apply (rule ch2ch_fun, assumption)
by (rule refl)


lemma chain_tord: "chain S⟹S k ⊑ S j ∨ S j ⊑ S k"
apply (insert linear [of "j" "k"])
apply (erule disjE)
apply (rule disjI2)
apply (rule chain_mono,simp+)
apply (rule disjI1)
by (rule chain_mono,simp+)


lemma neq_emptyD: "s ≠ {}⟹∃x. x ∈ s"
by auto

(* below lemmata *)
lemma cont_pref_eq1I: assumes "(a ⊑ b)"
  shows "f·a ⊑ f·b"
  by (simp add: assms monofun_cfun_arg)

lemma cont_pref_eq2I:  assumes "(a ⊑ b)"
  shows "f·x·a ⊑ f·x·b"
  by (simp add: assms monofun_cfun_arg)

(* equality lemmata *)
lemma cfun_arg_eqI:  assumes "(a = b)"
  shows "f·a = f·b"
  by (simp add: assms)


(* ------------------------------------------------------------------------ *)
section ⟨More functions⟩
(* ------------------------------------------------------------------------ *)



lemma less_lubI1:
  "⟦chain (Y::nat ⇒ 'a::cpo); X ⊑ (⨆k. Y (k + j))⟧⟹X ⊑ (⨆k. Y k)"
by (subst lub_range_shift [THEN sym, of "Y" "j"], simp+)


lemma less_lubI2:
"⟦chain (Y::nat ⇒ 'a::cpo); chain f; ⋀x. (⨆k. f k·x) = x; ⋀n. f n·x ⊑ (f n·(Lub Y))⟧⟹x ⊑ Lub Y"
by (insert lub_mono [of "λn. f n·x" "λn. f n·(Lub Y)"], simp)


lemma Suc2plus: "Suc n = Suc 0 + n"
by simp


lemma Suc_def2: "Suc i = i + Suc 0"
```

```
by simp

lemma max_in_chainI3: "⟦chain (Y::nat⇒'a::cpo); Y i = Lub Y⟧⟹max_in_chain i Y"
apply (simp add: max_in_chain_def)
apply (rule allI, rule impI)
apply (rule po_eq_conv [THEN iffD2])
apply (rule conjI)
apply (drule sym, simp)
apply (rule chain_mono, assumption+)
by (rule is_ub_thelub)

lemma finite_chainI: "⟦chain Y; max_in_chain i Y⟧⟹finite_chain Y"
by (auto simp add: finite_chain_def)

lemma lub_prod2: "⟦chain (X::nat ⇒ 'a::cpo); chain (Y::nat ⇒ 'b::cpo)⟧⟹
                 (⨆k. (X k,Y k)) = (Lub X, Lub Y)"
by (subst lub_prod, simp+)

lemma lub_range_shift2: "chain Y⟹(⨆i. Y i) = (⨆i. Y (i+j))"
  apply(simp add: lub_def)
  using is_lub_range_shift lub_def by fastforce


lemma l42: "chain S⟹finite_chain S⟹∃t. (⨆ j. S j) = S t"
using lub_eqI lub_finch2 by auto

lemma finite_chain_lub: fixes Y :: "nat ⇒ 'a ::cpo"
  assumes "finite_chain Y" and "chain Y" and "monofun f"
  shows "f (⨆i. Y i) = (⨆i. f (Y i))"
proof -
  obtain nn :: "(nat ⇒ 'a) ⇒ nat" where
    f1: "Lub Y = Y (nn Y)"
    by (meson assms(1) assms(2) l42)
  then have "∀n. f (Y n) ⊑ f (Y (nn Y))"
    by (metis (no_types) assms(2) assms(3) is_ub_thelub monofun_def)
  then show ?thesis
    using f1 by (simp add: lub_chain_maxelem)
qed

(* If you like admissibility proofs you will love this one. Never again "contI" ! *)
(* Dieses Lemma wurde nach langer suche von Sebastian entdeckt. Möge er ewig leben *)
lemma adm2cont:
  fixes f:: "'a::cpo ⇒ 'b::cpo"
  assumes "monofun f" and "⋀k. adm (λY. (f Y)⊑k)"
  shows "cont f"
  apply(rule contI2)
   apply(auto simp add: assms)
proof -
  fix Y:: "nat ⇒ 'a"
  assume "chain Y"
  obtain k where k_def: "k=(⨆i. (f (Y i)))" by simp
      (* komischer zwischenschritt, aber anders schafft sledgi das nicht *)
  have "⋀j. f (Y j) ⊑ (⨆i. (f (Y i)))"
    using ‹chain Y› assms(1) below_lub ch2ch_monofun by blast
  hence "⋀j. f (Y j) ⊑ k" by(simp add: k_def)
  hence "f (⨆j. Y j) ⊑ k"
    by (metis (no_types, lifting) ‹chain Y› adm_def assms(2))
  thus "f (⨆i. Y i) ⊑ (⨆i. f (Y i)) "
    using k_def by blast
qed

text ‹Creating a list from iteration a function ‹f›
  ‹n›-times on a start value ‹s›.›
primrec literate :: "nat ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a list"
where
  literate_0:  "literate 0 f s = []" |
  literate_Suc:"literate (Suc n) f s = s#(literate n f (f s))"


lemma literate_Suc2:
  "set (literate (Suc n) f s) = {s} ∪ set (literate n f (f s))"
by auto


lemma natl3: "{i. x ≤ i ∧ i < Suc n + x} = {x} ∪ {i. Suc x ≤ i ∧ i < Suc n + x}"
by auto


lemma literatel1 [simp]:
  "set (literate n Suc k) = {i. k ≤ i ∧ i < (n + k)}"
apply (rule_tac x="k" in spec)
apply (induct_tac n, simp)
apply (subst literate_Suc2)
apply (rule allI)
apply (erule_tac x="Suc x" in allE)
by (subst natl3, simp)
```

```
lemma card_set_list_le_length: "card (set x) ≤ length x"
apply (induct_tac x, simp+)
by (simp add: card_insert_if)


lemma [simp]: "length (literate n f k) = n"
apply (rule_tac x="k" in spec)
by (induct_tac n, simp+)


lemma [simp]: "map snd (map (Pair k) a) = a"
by (induct_tac a, simp+)


lemma from_set_to_nth: "xa ∈ set x⟹∃k. x!k = xa ∧ k < length x"
apply (simp add: atomize_imp)
apply (induct_tac x, simp+)
apply (rule conjI, rule impI)
apply (rule_tac x="0" in exI, simp)
apply (rule impI, simp)
apply (erule exE)
by (rule_tac x="Suc k" in exI, simp)


lemma filterI4: "⟦⋀x. Q x⟹P x; filter P x = []⟧⟹filter Q x = []"
by (simp add: atomize_imp, induct_tac x, auto)


lemma list_rinduct_lemma: "∀y. length y = k ∧ (P [] ∧ (∀x xs. P xs⟶P (xs @ [x])))⟶P y"
apply (induct_tac k, simp)
apply (rule allI)
apply (rule impI)
apply (erule conjE)+
apply (erule_tac x="butlast y" in allE, auto)
apply (erule_tac x="last y" in allE)
apply (erule_tac x="butlast y" in allE, auto)
by (case_tac "y = []", auto)


(* ------------------------------------------------------------------------ *)
section ‹Some more lemmas about sets›
(* ------------------------------------------------------------------------ *)


lemma finite_subset1: "finite Y⟹(∀X. X ⊆ Y⟶finite X)"
  by (simp add: finite_subset)


lemma ex_new_if_finiteI1:
  "⟦finite Y; ¬ finite X⟧⟹∃a. a ∈ X ∧ a ∉ Y"
apply (rule ccontr, auto)
apply (subgoal_tac "X ⊆ Y")
by (frule_tac Y="Y" in finite_subset1, auto)

text ‹Create a finite set with ‹n› distinct continuously
  numbered entries from set ‹A›.›
primrec
  getinj:: "'a set ⇒ nat ⇒ (nat × 'a) set"
where
  "getinj A 0 = {(0,SOME x. x ∈ A)}" |
  "getinj A (Suc n) = {(Suc n, SOME x. x ∈ A ∧ x ∉ (snd ` (getinj A n)))} ∪ getinj A n"



lemma finite_getinjs[simp]: "finite (getinj A n)"
by (induct_tac n, simp+)

lemma finite_snd_getinjs[simp]: "finite (snd ` (getinj A n))"
by (induct_tac n, simp+)

lemma finite_fst_getinjs[simp]: "finite (fst ` (getinj A n))"
by (induct_tac n, simp+)


lemma getinjs_l1: "∀k. n < k⟶(k, x) ∉ getinj A n"
by (induct_tac n, simp+)


lemma [simp]: "(Suc n,x) ∉ getinj A n"
by (insert getinjs_l1 [of n x A], auto)


lemma card_getinj_lemma[simp]: "¬ finite A⟹card (snd ` (getinj A n)) = card (getinj A n)"
apply (induct_tac n, simp+)
apply (rule someI2_ex)
apply (rule ex_new_if_finiteI1)
by (rule finite_snd_getinjs, simp+)

lemma inj_on_getinj: "¬ finite A⟹inj_on snd (getinj A n)"
by (rule eq_card_imp_inj_on, simp+)


lemma getinj_ex[simp]: "∃a. (n,a) ∈ getinj X n"
by (induct_tac n, simp+)
```

```
lemma getinj_chain:
  "⟦¬ finite A; (j, x) ∈ getinj A j; j ≤ k⟧⟹(j, x) ∈ getinj A k"
apply (simp add: atomize_imp)
apply (induct_tac k, auto)
by (case_tac "j = Suc n", auto)

lemma inter_union_id:"(x ∪ y) ∩ x = x"
  by blast

(* ---------------------------------------------------------------------- *)
section ‹updis›
(* ---------------------------------------------------------------------- *)


abbreviation
  updis :: "'a ⇒ 'a discr u"
    where "updis ≡ (λa. up·(Discr a))"

definition upApply :: "('a ⇒ 'b) ⇒ 'a discr u → 'b discr u" where
"upApply f ≡ Λ a. (if a=⊥ then ⊥ else updis (f (THE b. a = updis b)))"

definition upApply2 :: "('a ⇒ 'b ⇒ 'c) ⇒ 'a discr\<^sub>⊥ → 'b discr\<^sub>⊥ → 'c discr\<^sub>⊥" where
"upApply2 f ≡ Λ a b. (if a=⊥∨b=⊥ then ⊥ else updis (f (THE x. a = updis x) (THE x. b = updis x)))"


(* updis lemma *)
lemma updis_exists: assumes "x≠⊥"
  obtains n where "updis n = x"
  by (metis Discr_undiscr Exh_Up assms)



(* upApply *)
lemma upapply_mono [simp]: "monofun (λ a. (if a=⊥ then ⊥ else updis (f (THE b. a = updis b))))"
apply (rule monofunI, auto)
by (metis (full_types, hide_lams) discrete_cpo upE up_below)

lemma upapply_lub: assumes "chain Y"
  shows "((λ a. (if a=⊥ then ⊥ else updis (f (THE b. a = updis b)))) (⨆i. Y i))
=(⨆i. (λ a. (if a=⊥ then ⊥ else updis (f (THE b. a = updis b)))) (Y i))"
apply (rule finite_chain_lub)
by (simp_all add: assms chfin2finch)

lemma upapply_cont [simp]: "cont (λ a. (if a=⊥ then ⊥ else updis (f (THE b. a = updis b))))"
using chfindom_monofun2cont upapply_mono by blast

lemma upapply_rep_eq [simp]: "upApply f·(updis a) = updis (f a)"
by (simp add: upApply_def)

lemma upapply_insert: "upApply f·a = (if a=⊥ then ⊥ else updis (f (THE b. a = updis b)))"
by(simp add: upApply_def)

lemma upapply_strict [simp]: "upApply f·⊥ = ⊥"
by(simp add: upApply_def)

lemma upapply_nbot [simp]: "x≠⊥⟹upApply f·x≠⊥"
by(simp add: upApply_def)

lemma upapply_up [simp]: assumes "x≠⊥" obtains a where "up·a = upApply f·x"
by(simp add: upApply_def assms)

lemma chain_nbot: assumes "chain Y" and  "(⨆i. Y i) ≠⊥"
  obtains n::nat where "(⋀i. ((Y (i+n)) ≠⊥))"
  by (metis assms(1) assms(2) bottomI le_add2 lub_eq_bottom_iff po_class.chain_mono)

lemma upapply2_mono [simp]:
  "monofun (λ b. (if a=⊥∨b=⊥ then ⊥ else updis (f (THE x. a = updis x) (THE x. b = updis x))))"
apply (rule monofunI, auto)
by (metis discrete_cpo upE up_below)

lemma upapply2_cont [simp]:
  "cont (λb. if a = ⊥ ∨ b = ⊥ then ⊥ else updis (f (THE x. a = updis x) (THE x. b = updis x)))"
by (simp add: chfindom_monofun2cont)

lemma upapply2_mono2 [simp]:
  "monofun (λa. Λ b. if a = ⊥ ∨ b = ⊥ then ⊥ else updis (f (THE x. a = updis x) (THE x. b = updis x)))"
apply (rule monofunI)
apply (subst cfun_belowI, auto)
by (metis discrete_cpo upE up_below)

lemma upapply2_cont2 [simp]:
  "cont (λa. Λ b. if a = ⊥ ∨ b = ⊥ then ⊥ else updis (f (THE x. a = updis x) (THE x. b = updis x)))"
by (simp add: chfindom_monofun2cont)

lemma upapply2_rep_eq [simp]: "upApply2 f·(updis a)·(updis b) = updis (f a b)"
by (simp add: upApply2_def)

lemma upapply2_insert:
  "upApply2 f·a·b = (if a=⊥∨b=⊥ then ⊥ else updis (f (THE x. a = updis x) (THE x. b = updis x)))"
by (simp add: upApply2_def)

lemma upapply2_strict [simp]: "upApply2 f·⊥ = ⊥"
```

```
by(simp add: upApply2_def)

lemma upapply2_nbot [simp]: "x≠⊥⟹y≠⊥⟹upApply2 f·x·y≠⊥"
by(simp add: upApply2_def)

lemma upapply2_up [simp]: assumes "x≠⊥" and "y≠⊥" obtains a where "up·a = upApply2 f·x·y"
by(simp add: upApply2_def assms)

lemma cont2lub_lub_eq: assumes cont: "⋀i. cont (λx. F i x)"
  shows "chain Y⟶ (⨆i. F i (⨆i. Y i)) = (⨆i ia. F i (Y ia))"
proof -
  { assume "∃a. (⨆n. F a (Y n)) ≠ F a (Lub Y)"
    have ?thesis
      by (simp add: cont cont2contlubE) }
  thus ?thesis
    by force
qed

lemma[simp]: "x ⊑ y⟹(Λ ya. f·x·ya) ⊑ (Λ ya. f·y·ya)"
  by (simp add: cont_pref_eq1I eta_cfun)

lemma[simp]:"∀Y. chain Y⟶(Λ y. f·(⨆i. Y i)·y) ⊑ (⨆i. Λ y. f·(Y i)·y)"
  apply(simp add: contlub_cfun_fun contlub_cfun_arg,auto)
  apply(subst contlub_lambda,auto)
  by (simp add: cfun.lub_cfun cont_pref_eq1I)

lemma cont_lam2cont[simp]:"cont (λx. Λ y. f·x·y)"
  by(rule contI2, rule monofunI, simp+)

section ⟨add lemmas to cont2cont⟩

(* The original-Lemma "cont_if" is not general enough *)
declare Cont.cont_if[cont2cont del]
lemma cont_if2 [simp, cont2cont]: "(b⟹cont f)⟹(¬b⟹cont g)⟹cont (λx. if b then f x else g x)"
  by (induct b) simp_all

lemma cont2cont_lambda [cont2cont]:
  assumes f: "⋀y. cont (λx. f x y)"
  shows "cont f"
  by (simp add: f)

lemma comp_cont: (*Not usable for cont2cont*)
  assumes"cont f1"
  and "cont f2"
  shows "cont (f1 ∘ f2)"
  by(simp add: comp_def cont_compose assms)

lemma [cont2cont]:"cont f⟹f∈ cfun"
  by (simp add: cfun_def)

lemma discr_cont: "monofun f⟹cont (λx. g ((f x)::'a:: discrete_cpo))"
  apply(rule Cont.contI2)
  apply(rule monofunI, insert monofunE[of f],auto)
  by (metis is_ub_thelub)

lemma discr_cont2: "cont f⟹cont (λx. g ((f x)::'a:: discrete_cpo))" (*Not cont2cont, problem with domain
     definitions, i.e. lnat*)
  by (simp add: cont2mono discr_cont)

(*monofun f should be enough*)
lemma discr_cont3: "cont h⟹cont f⟹cont (λx. ((h x)) ((f x)::'a:: discrete_cpo))"
  by (simp add: cont2cont_fun cont_apply)

lemma cont_compose_snd [cont2cont]: "cont f⟹cont (λx. f (snd x))"
  by (simp add: cont_compose)

lemma cont_compose_fst [cont2cont]: "cont f⟹cont (λx. f (fst x))"
  by (simp add: cont_compose)


section⟨Monotony and continuity of inverse functions⟩

lemma monofun_inv:
      assumes"surj f"
      and"monofun f"
      and "⋀x y. ¬(x ⊑ y)⟹¬(f x ⊑ f y)" (*Other assumption combinations possible, is this the weakest?*)
      shows"monofun (inv f)"
using assms
proof(subst monofunI;simp_all)
  fix x y::'a
  assume below:"x ⊑ y"
  assume bij: "surj f"
  assume mono: "monofun f"
  from bij obtain a b where x: "x = f a" and y: "y = f b"
    by(fastforce simp: bij_def surj_def)
  show "inv f x ⊑ inv f y"
  proof (cases "a ⊑ b")
    case True
    then show ?thesis
      apply(simp add: x y)
      using assms(3) below bij surj_f_inv_f x y by fastforce
  next
    case False
    then show ?thesis
```

```
      apply(cases "b ⊑ a",auto)
      using below below_antisym mono monofunE x y apply fastforce
      apply(simp add: x y)
      using assms(3) below x y by blast
  qed
qed

lemma cont_inv[cont2cont]:
      assumes"surj f"
      and"cont f" (*Maybe other assumption?*)
      and "⋀x y. ¬(x ⊑ y)⟹¬(f x ⊑ f y)" (*Other assumption combinations possible, is this the weakest?*)
      shows"cont (inv f)"
  apply(rule Cont.contI2)
  apply(rule monofun_inv,simp_all add: assms cont2mono)
  using assms(1) assms(2) assms(3) cont2contlubE surj_f_inv_f
  by fastforce

section ⟨Timing information V3⟩
datatype timeType = TUntimed | TTimed | TTsyn
end
```

## A.2 Set Orderings

```
chapter ‹Set and bool as a pointed cpo.›

theory SetPcpo
imports HOLCF LNat
begin

text ‹PCPO on sets and bools. The ‹⊑› operator of the order is defined as the ‹⊆› operator on sets
  and as ⟷ on booleans.
›

(* ----------------------------------------------------------------------- *)
section ‹Order on sets.›
(* ----------------------------------------------------------------------- *)

text ‹{text "⊑"} operator as the ‹⊆› operator on sets -> partial order.›
instantiation set :: (type) po
begin
  definition less_set_def: "(⊑) = (⊆)"
instance
apply intro_classes
apply (simp add: less_set_def)
apply (simp add: less_set_def)
apply (simp add: less_set_def)
done
end

text ‹The least upper bound on sets corresponds to the ‹Union› operator.›
lemma Union_is_lub: "A <<| ⋃A"
apply (simp add: is_lub_def)
apply (simp add: is_ub_def)
apply (simp add: less_set_def Union_upper)
apply (simp add: Sup_least)
done


lemma lub_eq_Union: "lub = Union"
apply (rule ext)
apply (rule lub_eqI [OF Union_is_lub])
done


instance set :: (type) cpo
apply intro_classes
using Union_is_lub
apply auto
done

text ‹Sets are also pcpo`s, pointed with ‹{}› as minimal element.›
instance set :: (type) pcpo
apply intro_classes
apply (rule_tac x= "{}" in exI)
apply (simp add: less_set_def)
done


lemma UU_eq_empty: "⊥ = {}"
apply (simp add: less_set_def bottomI)
done


lemmas set_cpo_simps = less_set_def lub_eq_Union UU_eq_empty

(* ----------------------------------------------------------------------- *)
section ‹Order on booleans.›
(* ----------------------------------------------------------------------- *)

text ‹If one defines the ‹⊑› operator as the ⟷ operator on booleans,
  one obtains a partial order.›
instantiation bool :: po
begin
  definition less_bool_def: "(⊑) = ⟷"
instance
apply intro_classes
apply (simp add: less_bool_def)
apply (simp add: less_bool_def)
apply (simp add: less_bool_def)
apply (simp add: less_bool_def)
apply auto
done
end


instance bool :: chfin
proof
  fix S:: "nat ⇒ bool"
  assume S: "chain S"
  then have "finite (range S)"
  apply simp
  done
  from S and this
```

```
  have "finite_chain S"
  apply (rule finite_range_imp_finch)
  done
  thus "∃ n. max_in_chain n S"
  apply (unfold finite_chain_def, simp)
  done
qed


instance bool :: cpo ..

text ‹Bools are also pointed with ‹False› as minimal element.›
instance bool :: pcpo
proof
  have "∀y::bool. False ⊑ y"
  unfolding less_bool_def
  apply simp
  done
  thus "∃x::bool. ∀y. x ⊑ y" ..
qed

(* ------------------------------------------------------------------------ *)
section ‹Properties›
(* ------------------------------------------------------------------------ *)

(* ------------------------------------------------------------------------ *)
subsection ‹Admissibility of set predicates›
(* ------------------------------------------------------------------------ *)

text ‹The predicate "λA. ∃x. x ∈ A" is admissible.›
lemma adm_nonempty: "adm (λA. ∃x. x ∈ A)"
apply (rule admI)
apply (simp add: lub_eq_Union)
apply force
done

text ‹The predicate "λA. x ∈ A" is admissible.›
lemma adm_in: "adm (λA. x ∈ A)"
apply (rule admI)
apply (simp add: lub_eq_Union)
done

text ‹The predicate "λA. x ∉ A" is admissible.›
lemma adm_not_in: "adm (λA. x ∉ A)"
apply (rule admI)
apply (simp add: lub_eq_Union)
done

text ‹If for all x the predicate "λA. P A x" is admissible, then so is "λA. ∀x∈A. P A x".›
lemma adm_Ball: "(⋀x. adm (λA. P A x))⟹adm (λA. ∀x∈A. P A x)"
apply (simp add: Ball_def)
apply (simp add: adm_not_in)
done

text ‹The predicate "λA. Bex A P", which means "λA. ∃x. x ∈ A ∧ P x" is admissible.›
lemma adm_Bex: "adm (λA. Bex A P)"
apply (rule admI)
apply (simp add: lub_eq_Union)
done

text ‹The predicate "λA. A ⊆ B" is admissible.›
lemma adm_subset: "adm (λA. A ⊆ B)"
apply (rule admI)
apply (simp add: lub_eq_Union)
apply auto
done

text ‹The predicate "λA. B ⊆ A" is admissible.›
lemma adm_superset: "adm (λA. B ⊆ A)"
apply (rule admI)
apply (simp add: lub_eq_Union)
apply auto
done


lemmas adm_set_lemmas = adm_nonempty adm_in adm_not_in adm_Bex adm_Ball adm_subset adm_superset

(* ------------------------------------------------------------------------ *)
subsection ‹Compactness›
(* ------------------------------------------------------------------------ *)


lemma compact_empty: "compact {}"
apply (fold UU_eq_empty)
apply simp
done


lemma compact_insert: "compact A⟹compact (insert x A)"
apply (simp add: compact_def)
apply (simp add: set_cpo_simps)
apply (simp add: adm_set_lemmas)
done
```

```
lemma finite_imp_compact: "finite A⟹compact A"
apply (induct A set: finite)
apply (rule compact_empty)
apply (erule compact_insert)
done

lemma union_cont:"cont (λS2. union S1 S2)"
  apply(rule contI)
  unfolding  SetPcpo.less_set_def
  unfolding lub_eq_Union
  by (metis (no_types, lifting) UN_simps(3) Union_is_lub empty_not_UNIV lub_eq lub_eqI)



section ⟨setify⟩
definition setify_on::"'m set ⇒ ('m::type ⇒ ('n::type set)) ⇒ ('m ⇒ 'n) set" where
"setify_on Dom ≡ λ f. {g. ∀m∈Dom. g m ∈ (f m)}"

definition setify::"('m::type ⇒ ('n::type set)) ⇒ ('m ⇒ 'n) set" where
"setify ≡ λ f. {g. ∀m. g m ∈ (f m)}"



subsection ⟨setify_on⟩
thm setify_def
lemma setify_on_mono[simp]: "⋀ Dom. monofun (λ f. {g. ∀m∈Dom. g m ∈ (f m)})"
proof (rule monofunI, simp add: less_set_def, rule)
  fix x y::"'m::type ⇒ ('n::type set)"
  fix Dom::"'m set"
  fix xa:: "'m ⇒ 'n"
  assume a1:"x ⊑ y"
  assume a2: "xa ∈ {g. ∀m∈Dom. g m ∈ x m}"
  have f0: "⋀m. x m ⊑ y m"
    by (simp add: a1 fun_belowD)
  have f1: "⋀m. m ∈ Dom⟹xa m ∈ x m"
    using a2 by blast
  have f2: "⋀m. m ∈ Dom⟹xa m ∈ y m"
    by (metis SetPcpo.less_set_def f0 f1 subsetCE)
  show "xa ∈ {g. ∀m∈Dom. g m ∈ y m}"
    using f2 by blast
qed

lemma setify_on_empty:"⋀ Dom. sbe ∈ Dom⟹f sbe = {}⟹setify_on Dom f = {}"
  apply(simp add: setify_on_def)
  by (metis empty_iff)

lemma setify_on_notempty_ex:"setify_on Dom f ≠ {}⟹∃g.(∀m ∈ Dom. g m ∈ (f m))"
  by (metis (no_types, lifting) Collect_empty_eq setify_on_def)

lemma setify_on_notempty:assumes "∀m ∈ Dom. f m ≠ {}" shows" setify_on Dom f ≠ {}"
proof(simp add: setify_on_def)
  have "∀m ∈ Dom. (∃x. x∈((f m)))"
    by (metis all_not_in_conv assms)
  have "∀m ∈ Dom. (λe. SOME x. x∈ (f e)) m ∈ (f m)"
    by (metis assms some_in_eq)
  then show "∃x::'a ⇒ 'b. ∀m::'a ∈ Dom. x m ∈ (f m)"
    by(rule_tac x="(λe. SOME x. x∈ (f e))" in exI, auto)
qed

lemma setify_on_final:assumes "∀m ∈ Dom. f m ≠ {}" and "x ∈ (f m)"
  shows"∃g∈((setify_on Dom f)). g m = x"
proof(simp add: setify_on_def)
  have "∃g.(∀m ∈ Dom. g m ∈ (f m))"
    by(simp add: setify_on_notempty setify_on_notempty_ex assms(1))
  then obtain g where g_def:"(∀m ∈ Dom. g m ∈ (f m))"
    by auto
  have g2_def:"∀n ∈ Dom. (λe. if e = m then x else g e) n ∈ (f n)"
    by (simp add: assms(2) g_def)
  then show "∃g::'a ⇒ 'b. (∀m::'a ∈ Dom. g m ∈ (f m)) ∧ g m = x"
    by(rule_tac x="(λe. if e = m then x else g e)" in exI, auto)
qed



subsection ⟨setify⟩
lemma setify_mono[simp]:"monofun (λf. {g. ∀m. g m ∈ (f m)})"
  apply(rule monofunI)
  by (smt Collect_mono SetPcpo.less_set_def below_fun_def subsetCE)

lemma setify_empty:"f m = {}⟹setify f = {}"
  apply(simp add: setify_def)
  by (metis empty_iff)

lemma setify_notempty:assumes "∀m. f m ≠ {}" shows" setify f ≠ {}"
proof(simp add: setify_def)
  have "∀m. ∃x. x∈((f m))"
    by (metis all_not_in_conv assms)
  have "∀m. (λe. SOME x. x∈ (f e)) m ∈ (f m)"
    by (metis assms some_in_eq)
  then show "∃x::'a ⇒ 'b. ∀m::'a. x m ∈ (f m)"
    by(rule_tac x="(λe. SOME x. x∈ (f e))" in exI, auto)
```

```
qed

lemma setify_notempty_ex:"setify f ≠ {}⟹∃g.(∀m. g m ∈ (f m))"
  by(simp add: setify_def)

lemma setify_final:assumes "∀m. f m ≠ {}" and "x ∈ (f m)" shows"∃g∈((setify f)). g m = x"
proof(simp add: setify_def)
  have "∃g.(∀m. g m ∈ (f m))"
    by(simp add: setify_notempty setify_notempty_ex assms(1))
  then obtain g where g_def:"(∀m. g m ∈ (f m))"
    by auto
  have g2_def:"∀n. (λe. if e = m then x else g e) n ∈ (f n)"
    by (simp add: assms(2) g_def)
  then show "∃g::'a ⇒ 'b. (∀m::'a. g m ∈ (f m)) ∧ g m = x"
    by(rule_tac x="(λe. if e = m then x else g e)" in exI, auto)
qed


inductive setSize_helper :: "'a set ⇒ nat ⇒ bool"
  where
    "setSize_helper {} 0"
  |  "setSize_helper A X ∧ a ∉ A⟹setSize_helper (insert a A) (Suc X)"

definition setSize :: "'a set ⇒ lnat"
  where
  "setSize X ≡ if (finite X) then Fin (THE Y. setSize_helper X Y) else ∞"


lemma setSizeEx: assumes "finite X" shows "∃ Y. setSize_helper X Y"
  apply (rule finite_induct)
  apply (simp add: assms)
  using setSize_helper.intros(1) apply auto[1]
  by (metis setSize_helper.simps)

lemma setSize_remove: "y ∈ F ∧ setSize_helper (F - {y}) A⟶setSize_helper F (Suc A)"
  by (metis Diff_insert_absorb Set.set_insert setSize_helper.intros(2))


lemma setSizeBack_helper:
  assumes "∀(F::'a set) x::'a. (finite F ∧ setSize_helper (insert x F) (Suc A) ∧ x ∉ F)⟶setSize_helper F A"
  shows "∀(F::'a set) x::'a. (finite F ∧ setSize_helper (insert x F) (Suc (Suc A)) ∧ x ∉ F)⟶setSize_helper F
        (Suc A)"
  proof -
  have b0: "⋀A::nat. ∀(F::'a set) x::'a. ((setSize_helper (insert x F) (Suc (Suc A)) ∧ x ∉ F)
   ⟶(∃ y. y ∈ (insert x F) ∧ setSize_helper ((insert x F) - {y}) (Suc A)))"
    by (metis Diff_insert_absorb add_diff_cancel_left' insertI1 insert_not_empty plus_1_eq_Suc
         setSize_helper.simps)
  have b1: "∀(F::'a set) (x::'a) y::'a. ((finite F ∧ setSize_helper (insert x (F - {y})) (Suc A) ∧ x ∉ F)
   ⟶setSize_helper (F - {y}) A)"
   using assms by auto
  have b2: "∀(F::'a set) x::'a. (setSize_helper (insert x F) (Suc (Suc A)) ∧ x ∉ F)
   ⟶((∃ y. (y≠x ∧ y ∈ F ∧ setSize_helper (insert x (F - {y})) (Suc A))) ∨ setSize_helper F (Suc A))"
   by (metis Diff_insert_absorb b0 empty_iff insert_Diff_if insert_iff)
  have b3: "∀(F::'a set) x::'a. (setSize_helper (insert x F) (Suc (Suc A)) ∧ x ∉ F ∧ finite F)
   ⟶((∃ y. (y≠x ∧ y ∈ F ∧ setSize_helper (F - {y}) A)) ∨ setSize_helper F (Suc A))"
   by (meson b1 b2)
  show "∀(F::'a set) x::'a. (finite F ∧ setSize_helper (insert x F) (Suc (Suc A)) ∧ x ∉ F)⟶setSize_helper F (Suc
      A)"
   by (meson b3 setSize_remove)
  qed


lemma setSizeBack: "⋀ F x. (finite F ∧ setSize_helper (insert x F) (Suc A) ∧ x ∉ F)⟹setSize_helper F A"
  apply (induction A)
  apply (metis Suc_inject empty_iff insertI1 insert_eq_iff nat.distinct(1) setSize_helper.simps)
  using setSizeBack_helper by blast


lemma setSizeonlyOne: assumes "finite X" shows "∃! Y. setSize_helper X Y"
  apply (rule finite_induct)
  apply (simp add: assms)
  apply (metis empty_not_insert setSize_helper.simps)
  by (metis insert_not_empty setSizeBack setSize_helper.intros(2) setSize_helper.simps)

lemma setSizeSuc: assumes "finite X" and "z ∉ X" shows "setSize (insert z X) = lnsuc·(setSize X)"
  apply (simp add: setSize_def)
  using assms setSizeonlyOne
  by (metis (mono_tags, lifting) Diff_insert_absorb finite.insertI insertI1 setSize_remove theI_unique)

lemma setSizeEmpty: "setSize {} = Fin 0"
  by (metis finite.emptyI setSize_def setSize_helper.intros(1) setSizeonlyOne theI_unique)

lemma setSizeSingleton: "setSize {x} = lnsuc·(Fin 0)"
  by (simp add: setSizeEmpty setSizeSuc)

lemma setsize_union_helper1:
  assumes "finite F"
      and "x ∉ F"
      and "x ∉ X"
    shows "setSize (X ∪ F) + setSize (X ∩ F) = setSize X + setSize F⟹
       setSize (X ∪ insert x F) + setSize (X ∩ insert x F) = setSize X + setSize (insert x F)"
proof -
  assume a0: "setSize (X ∪ F) + setSize (X ∩ F) = setSize X + setSize F"
```

```
  have b0: "X ∪ insert x F = insert x (X ∪ F)"
    by simp
  have b1: "setSize (X ∪ insert x F) = lnsuc·(setSize (X ∪ F))"
    by (metis Un_iff Un_infinite assms(1) assms(2) assms(3) b0 finite_UnI fold_inf setSizeSuc setSize_def
        sup_commute)
  have b2: "setSize (X ∩ insert x F) = setSize (X ∩ F)"
    by (simp add: assms(3))
  show "setSize (X ∪ insert x F) + setSize (X ∩ insert x F) = setSize X + setSize (insert x F)"
    by (metis (no_types, lifting) a0 ab_semigroup_add_class.add_ac(1) add.commute assms(1) assms(2)
      b1 b2 lnat_plus_suc setSizeSuc)
qed

lemma setsize_union_helper2:
  assumes "finite F"
      and "x ∉ F"
      and "x ∈ X"
    shows "setSize (X ∪ F) + setSize (X ∩ F) = setSize X + setSize F⟹
      setSize (X ∪ insert x F) + setSize (X ∩ insert x F) = setSize X + setSize (insert x F)"
proof -
  assume a0: "setSize (X ∪ F) + setSize (X ∩ F) = setSize X + setSize F"
  have b0: "setSize (X ∪ insert x F) = setSize (X ∪ F)"
    by (metis Un_Diff_cancel assms(3) insert_Diff1)
  have b1: "setSize (X ∩ insert x F) =  lnsuc·(setSize (X ∩ F))"
    by (simp add: assms(1) assms(2) assms(3) setSizeSuc)
  show "setSize (X ∪ insert x F) + setSize (X ∩ insert x F) = setSize X + setSize (insert x F)"
    by (metis a0 ab_semigroup_add_class.add_ac(1) assms(1) assms(2) b0 b1 lnat_plus_suc setSizeSuc)
qed

lemma setsize_union_helper3: assumes "finite X" and "finite Y"
  shows "setSize (X ∪ Y) + setSize (X ∩ Y) = setSize X + setSize Y"
  apply (rule finite_induct)
  apply (simp add: assms)
  apply simp
  by (meson setsize_union_helper1 setsize_union_helper2)

lemma setsize_union_helper4: assumes "infinite X ∨ infinite Y"
  shows "setSize (X ∪ Y) + setSize (X ∩ Y) = setSize X + setSize Y"
proof -
  have b0: "setSize (X ∪ Y) = ∞"
    by (metis (full_types) assms infinite_Un setSize_def)
  have b1: "setSize X = ∞ ∨ setSize Y = ∞"
    by (meson assms setSize_def)
  show ?thesis
    using b0 b1 plus_lnatInf_r by auto
qed

lemma setsize_union: "setSize (X ∪ Y) + setSize (X ∩ Y) = setSize X + setSize Y"
  by (meson setsize_union_helper3 setsize_union_helper4)

lemma setsize_union_disjoint: assumes "X ∩ Y = {}"
  shows "setSize (X ∪ Y) = setSize X + setSize Y"
  by (metis Fin_02bot add.left_neutral assms bot_is_0 lnat_plus_commu setSizeEmpty setsize_union)

lemma setsize_subset_union: assumes "X ⊆ Y"
  shows "setSize (X ∪ Y) = setSize Y"
  by (simp add: assms sup.absorb2)

lemma set_union_ins: "⋀ F G x. setSize (F ∪ G) ≤ setSize (F ∪ (insert x G))"
  by (metis Fin_Suc Fin_leq_Suc_leq  Un_insert_right finite_insert insert_absorb lnat_po_eq_conv
  setSizeSuc setSize_def)

lemma setsize_mono_union_helper1:
  assumes "finite F" and "finite G"
  shows "setSize F ≤ setSize (F ∪ G)"
proof -
  have b0:  "⋀P. P = (λG. setSize F ≤ setSize (F ∪ G))⟹P G"
    by (metis assms(2) finite_induct order_refl set_union_ins sup_bot.right_neutral trans_lnle)
  have b1: "(λG. setSize F ≤ setSize (F ∪ G)) G"
    using b0 by auto
  show "setSize F ≤ setSize (F ∪ G)"
    by (simp add: b1)
qed

lemma setsize_mono_union_helper2:
  assumes "infinite F ∨ infinite G"
  shows "setSize F ≤ setSize (F ∪ G)"
proof -
  have b0: "setSize (F ∪ G) = ∞"
    by (meson assms infinite_Un setSize_def)
  show ?thesis
    by (simp add: b0)
qed

lemma setsize_mono_union: "setSize F ≤ setSize (F ∪ G)"
  by (meson setsize_mono_union_helper1 setsize_mono_union_helper2)


lemma setsize_mono:
  assumes "F ⊆ G"
  shows "setSize F ≤ setSize G"
  by (metis Un_absorb1 assms setsize_mono_union)
```

```
subsection ⟨setflat⟩

definition setflat :: "'a set set → 'a set" where
"setflat = (Λ S. {K | Z K. K∈Z ∧ Z ∈S} )"

lemma setflat_mono: "monofun (λ S. {K | Z K. K∈Z ∧ Z ∈S} )"
  apply(rule monofunI)
  apply auto
  apply (simp add: less_set_def)
  apply (rule subsetI)
  by auto


lemma setflat_cont: "cont (λ S. {K | Z K. K∈Z ∧ Z ∈S} )"
  apply(rule contI2)
  using setflat_mono apply simp
  apply auto
  unfolding SetPcpo.less_set_def
  unfolding lub_eq_Union
  by blast

lemma setflat_insert: "setflat·S = {K | Z K. K∈Z ∧ Z ∈S}"
  unfolding setflat_def
  by (metis (mono_tags, lifting) Abs_cfun_inverse2 setflat_cont)

lemma setflat_empty:"(setflat·S = {})⟷(∀x∈S. x = {})"
  by(simp add: setflat_insert, auto)

lemma setflat_not_empty:"(setflat·S ≠ {})⟷(∃x∈S. x ≠ {})"
  by (simp add: setflat_empty)

lemma setflat_obtain: assumes "f ∈ setflat·S"
  shows "∃ Z ∈ S. f ∈ Z"
proof -
  have "f ∈ {a. ∃A aa. a = aa ∧ aa ∈ A ∧ A ∈ S}"
    by (metis assms setflat_insert)
  then show ?thesis
    by blast
qed

lemma setflat_union: "setflat·S = ⋃S"
  apply (simp add: setflat_insert)
  apply (subst Union_eq)
  by auto

lemma setflatten_mono2: assumes "⋀b. b∈S1⟹ ∃c. c∈S2 ∧ b ⊆ c)"
  shows "setflat·S1 ⊆ setflat· S2"
  by (smt Abs_cfun_inverse2 setflat_def setflat_cont assms mem_Collect_eq subsetCE subsetI)

lemma setfilter_easy: "Set.filter (λf. True) X = X"
  using member_filter by auto

lemma setfilter_cont: "cont (Set.filter P)"
  by (simp add: Prelude.contI2 SetPcpo.less_set_def lub_eq_Union monofun_def subset_eq)

end
```

## A.3 Lazy Naturals

```
section ‹The Datatype of Lazy Natural Numbers›

theory LNat
imports Prelude
begin

(* -------------------------------------------------------------------------- *)
section ‹Type definition and the basics›
(* -------------------------------------------------------------------------- *)

text ‹
  Defined using the 'domain' command. Generates a bottom element (‹⊥›) and an order (‹⊑›),
  which are used to define zero and ‹≤›.
›
domain lnat = lnsuc (lazy lnpred::lnat)

instantiation lnat :: "{ord, zero}"
begin
  definition lnzero_def: "(0::lnat) ≡ ⊥"
  definition lnless_def: "(m::lnat) < n ≡ m ⊑ n ∧ m ≠ n"
  definition lnle_def:   "(m::lnat) ≤ n ≡ m ⊑ n"
instance ..
end

text ‹define @{term lntake} as an abbreviation for @{term lnat_take},
  which is generated by the ‹domain› command›
abbreviation
  lntake :: "nat ⇒ lnat → lnat"
    where "lntake ≡ lnat_take"


lemma lntake_more[simp]:
  "lntake (Suc n)·(lnsuc·k) = lnsuc·(lntake n·k)"
by (induct_tac n, auto)

(* -------------------------------------------------------------------------- *)
section ‹Definitions›
(* -------------------------------------------------------------------------- *)
text ‹‹∞› is the maximum of all @{term lnat}s›
definition Inf'  ::  "lnat"   ("∞") where
"Inf' ≡ fix·lnsuc"


definition Fin  ::  "nat ⇒ lnat" where
"Fin k ≡ lntake k·∞"


definition lnmin ::  "lnat → lnat → lnat" where
"lnmin ≡ fix·(Λ h. strictify·(Λ m. strictify·(Λ n.
                      lnsuc·(h·(lnpred·m)·(lnpred·n)))))"

abbreviation lnatGreater :: "lnat ⇒ lnat ⇒ bool" (infix ">^l" 65) where
"n >^l m ≡  n ≥ lnsuc·m"

abbreviation lnatLess :: "lnat ⇒ lnat ⇒ bool" (infix "<^l" 65) where
"n <^l m ≡  lnsuc·n ≤ m"

instantiation lnat :: plus
begin
  definition plus_lnat:: "lnat ⇒ lnat ⇒ lnat"  where
    "plus_lnat ln1 ln2 ≡ if (ln1 = ∞ ∨ ln2=∞) then ∞ else Fin ((inv Fin) ln1 + (inv Fin) ln2)"

instance
  by(intro_classes)
end

(* -------------------------------------------------------------------------- *)
section ‹Some basic lemmas›
(* -------------------------------------------------------------------------- *)

(* -------------------------------------------------------------------------- *)
subsection
  ‹Brief characterization of
     ‹Fin›, ‹∞›, ‹≤› and ‹<›
›
(* -------------------------------------------------------------------------- *)


lemma less_lnsuc[simp]: "x ≤ lnsuc·x"
apply (subst lnle_def)
by (rule lnat.induct [of _ x], auto)

text ‹‹∞› is a fix point of @{term lnsuc}›
lemma fold_inf[simp]: "lnsuc·∞ = ∞"
by (unfold Inf'_def, subst fix_eq2 [THEN sym], simp+)

text ‹x is smaller then ∞.›
lemma inf_ub[simp]: "x ≤ ∞"
apply (subst lnle_def)
```

```
apply (rule lnat.induct [of _ x], auto)
apply (subst fold_inf [THEN sym])
by (rule monofun_cfun_arg)


lemma Fin_02bot: "Fin 0 = ⊥"
by (simp add: Fin_def)

text ‹(≤) on lnats is antisymmetric›
lemma lnat_po_eq_conv:
  "(x ≤ y ∧ y ≤ x) = ((x::lnat) = y)"
apply (auto simp add: lnle_def)
by (rule po_eq_conv [THEN iffD2], simp)


lemma lnsuc_neq_0[simp]: "lnsuc·x ≠ 0"
by (simp add: lnzero_def)


lemma lnsuc_neq_0_rev[simp]: "0 ≠ lnsuc·x"
by (simp add: lnzero_def)

text ‹0 is not equal ∞.›
lemma Inf'_neq_0[simp]: "0 ≠ ∞"
apply (subst fold_inf [THEN sym])
by (rule notI, simp del: fold_inf)


lemma Inf'_neq_0_rev[simp]: "∞ ≠ 0"
by (rule notI, drule sym, simp)


lemma inject_lnsuc[simp]: "(lnsuc·x = lnsuc·y) = (x = y)"
by (rule lnat.injects)


lemma [simp]: "lntake (Suc k) ∞ = lnsuc·(lntake k·∞)"
apply (subst fold_inf [THEN sym])
by (simp only: lntake_more)


lemma Fin_Suc[simp]: "lnsuc·(Fin k) = Fin (Suc k)"
by (simp add: Fin_def)


lemma Fin_0[simp]: "(Fin k = 0) = (k = 0)"
apply (induct_tac k, auto simp add: lnzero_def)
by (simp add: Fin_def)+


lemma inject_Fin[simp]: "(Fin n = Fin k) = (n = k)"
apply (rule spec [of _ k], induct_tac n, auto)
by (case_tac x, auto simp add: Fin_def)+

text ‹If a lnat cannot be reached by @{term "lnat_take"}, it behaves like ‹∞››
lemma nreach_lnat_lemma:
  "∀x. (∀j. lnat_take j·x ≠ x) ⟶ lnat_take k·x = lnat_take k·∞"
apply (induct_tac k, auto)
apply (rule_tac y=x in lnat.exhaust, auto simp add: lnzero_def)
apply (erule_tac x="lnat" in allE, auto)
by (erule_tac x="Suc j" in allE, auto)

text ‹If a lnat cannot be reached by @{term "lnat_take"}, it
  is ‹∞›.›
lemma nreach_lnat:
  "(∀j. lntake j·x ≠ x) ⟹ x = ∞"
apply (rule lnat.take_lemma)
by (rule nreach_lnat_lemma [rule_format],simp)


lemma nreach_lnat_rev:
  "x ≠ ∞ ⟹ ∃n. lntake n·x = x"
apply (rule ccontr, auto)
by (drule nreach_lnat, simp)


lemma exFin_take:
  "∀x. lntake j·x = x ⟶ (∃k. x = Fin k)"
apply (induct_tac j, auto)
apply (rule_tac x="0" in exI,simp add: Fin_def)
apply (rule_tac y=x in lnat.exhaust, auto)
by (rule_tac x="0" in exI, simp add: Fin_def)

text ‹If a predicate holds for both finite lnats and for ‹∞›,
  it holds for every lnat›
lemma lncases:
  "⋀x P. ⟦x = ∞ ⟹ P; ⋀k. x = Fin k ⟹ P⟧ ⟹ P"
apply (case_tac "x = ∞", auto)
apply (drule nreach_lnat_rev, auto)
by (drule exFin_take [rule_format],auto)

text ‹Only ‹∞› is greater or equal to ‹∞››
lemma inf_less_eq[simp]: "(∞ ≤ x) = (x = ∞)"
apply (auto, rule lnat_po_eq_conv [THEN iffD1])
```

```
by (rule conjI, auto)


lemma bot_is_0: "(⊥::lnat) = 0"
by (simp add: lnzero_def)

text ‹Fin k ≤ 0 holds only for k = 0.›
lemma lnle_Fin_0[simp]: "(Fin k ≤ 0) = (k = 0)"
apply (simp add: lnzero_def lnle_def)
by (subst bot_is_0, simp)

text ‹‹≤› on lnats is antisymmetric›
lemma less2eq: "⟦x ≤ y; y ≤ x⟧⟹(x :: lnat) = y"
by (rule lnat_po_eq_conv [THEN iffD1], simp)


lemma Fin_leq_Suc_leq: "Fin (Suc n) ≤ i⟹Fin n ≤ i"
apply (simp add: lnle_def)
apply (rule below_trans, auto)
apply (simp only: Fin_def)
apply (rule monofun_cfun_fun)
by (rule chainE,simp)

text ‹‹≤› on lnats and on nats›
lemma less2nat_lemma: "∀k. (Fin n ≤ Fin k)⟶(n ≤ k)"
apply (induct_tac n, auto)
apply (case_tac "n=k", simp)
apply (subgoal_tac "Fin k ≤ Fin (Suc k)")
apply (drule less2eq, auto)
apply (subst lnle_def)
apply (rule chainE)
apply (simp add: Fin_def)
apply (erule_tac x="k" in allE,auto)
by (drule Fin_leq_Suc_leq, simp)

text ‹If Fin n ≤ Fin k then n ≤ k.›
lemma less2nat[simp]: "(Fin n ≤ Fin k) = (n ≤ k)"
apply (rule iffI)
apply (rule less2nat_lemma [rule_format], assumption)
apply (simp add: lnle_def)
apply (rule chain_mono)
by (simp add: Fin_def,auto)

text ‹lnsuc x is ∞ iff x is ∞.›
lemma [simp]: "(lnsuc·x = ∞) = (x = ∞)"
apply (rule iffI)
by (rule lnat.injects [THEN iffD1], simp+)

text ‹A finite number is not ∞.›
lemma Fin_neq_inf[simp]: "Fin k ≠ ∞"
apply (induct_tac k, auto)
apply (simp add: Fin_def bot_is_0)
by (simp add: Fin_Suc [THEN sym] del: Fin_Suc)

text ‹If lnsuc x ≤ lnsuc y then x ≤ y.›
lemma lnsuc_lnle_emb[simp]: "(lnsuc·x ≤ lnsuc·y) = (x ≤ y)"
apply (rule_tac x=x in lncases, simp)
by (rule_tac x=y in lncases, auto)


lemma [simp]: "0 ≤ (x::lnat)"
by (simp add: lnzero_def lnle_def)

text ‹If n ≤ 0 then n = 0.›
lemma [simp]: "((n::lnat) ≤ 0) = (n = 0)"
by (rule iffI, rule_tac x=n in lncases, auto)

text ‹If x ⊑ y then x ≤ y.›
lemma lnle_conv[simp]: "((x::lnat) ⊑ y) = (x ≤ y)"
by (subst lnle_def,simp)

text ‹transitivity of ‹≤››
lemma trans_lnle:
  "⟦x ≤ y; y ≤ z⟧⟹(x::lnat) ≤ z"
by (subst lnle_def, rule_tac y = y in below_trans, simp+)

text ‹reflexivity of ‹≤››
lemma refl_lnle[simp]: "(x::lnat) ≤ x"
by (subst lnle_def,rule below_refl)

text ‹0 < ∞.›
lemma Zero_lnless_infty[simp]: "0 < ∞"
by (auto simp add: lnless_def)


lemma gr_0[simp]: "(0 < j) = (∃k. j = lnsuc·k)"
apply (auto simp add: lnless_def)
apply (rule_tac y=j in lnat.exhaust)
by (simp add: lnzero_def, auto)

(*---*)
section ‹Some basic lemmas on ‹<››
(*---*)
```

```
text ⟨0 < lnsuc·k⟩
lemma [simp]: "0 < lnsuc·k"
by (auto simp add: lnless_def)

text ⟨0 < Fin (Suc k)⟩
lemma [simp]: "0 < Fin (Suc k)"
by (simp add: Fin_Suc [THEN sym] del: Fin_Suc)

text ⟨Fin k < ∞⟩
lemma [simp]: "Fin k < ∞"
by (auto simp add: lnless_def)

text ⟨If Fin k < Fin n then k < n.⟩
lemma [simp]: "(Fin k < Fin n) = (k < n)"
by (auto simp add: lnless_def)

lemma trans_lnless:
  "⟦x < y; y < z⟧⟹(x::lnat) < z"
apply (auto simp add: lnless_def)
apply (rule trans_lnle, auto)
by (simp add: lnat_po_eq_conv [THEN iffD1])

text ⟨If lnsuc·n < lnsuc·k then n < k⟩
lemma [simp]: "(lnsuc·n < lnsuc·k) = (n < k)"
by (auto simp add: lnless_def)


lemma [simp]: "¬ lnsuc·k < 0"
by (simp add: lnless_def)

text ⟨∞ is not smaller then anything.⟩
lemma [simp]: "¬ ∞ < i"
by (auto simp add: lnless_def)

(*---*)
section ⟨Relationship between Fin and ⟨∞⟩⟩
(*---*)



lemma ninf2Fin: "x ≠ ∞⟹∃k. x = Fin k"
by (rule_tac x=x in lncases, auto)

lemma assumes "ln = lnsuc·ln"
  shows "ln = ∞"
using assms ninf2Fin by force


lemma infI: "∀k. x ≠ Fin k⟹x = ∞"
by (rule lncases [of x], auto)


lemma below_fin_imp_ninf: "x ⊑ Fin k⟹x ≠ ∞"
by (rule lncases [of "x"], simp_all)

text ⟨⟨∞⟩ is not finite⟩
lemma [simp]: "∞ ≠ Fin k"
by (rule notI, drule sym, simp)

text ⟨⟨∞⟩ is strictly greater than all finite lnats⟩
lemma [simp]: "¬ ∞ ≤ Fin k)"
by (rule notI, auto)


lemma inf_belowI: "∀k. Fin k ⊑ x⟹x = ∞"
proof (rule lncases [of x], simp)
  fix k assume "x = Fin k" and "∀k. Fin k ⊑ x"
  hence "Fin (Suc k) ⊑ Fin k" by simp
  thus ?thesis by simp
qed

(* ---------------------------------------------------------------------- *)
subsection ⟨Induction rules⟩
(* ---------------------------------------------------------------------- *)


lemma lnat_ind: "⋀P x. ⟦adm P; P 0; ⋀l. P l⟹P (lnsuc·l)⟧⟹P x"
apply (rule lnat.induct, simp)
by (simp add: lnzero_def, auto)

(* ---------------------------------------------------------------------- *)
subsection ⟨Basic lemmas on @{term lmin}⟩
(* ---------------------------------------------------------------------- *)

text ⟨lnmin·0·n = 0⟩
lemma strict_lnmin_fst[simp]: "lnmin·0·n = 0"
apply (subst lnmin_def [THEN fix_eq2])
by (simp add: lnzero_def)

text ⟨lnmin·m·0 = 0⟩
lemma strict_lnmin_snd[simp]: "lnmin·m·0 = 0"
apply (subst lnmin_def [THEN fix_eq2], auto)
apply (rule lnat.induct [of _ m], simp)
by (simp add: lnzero_def)+
```

```
lemma lnmin_lnsuc[simp]: "lnmin·(lnsuc·m)·(lnsuc·n) = lnsuc·(lnmin·m·n)"
by (subst lnmin_def [THEN fix_eq2], simp)

text ‹lnmin·∞·n = n›
lemma lnmin_fst_inf[simp]: "lnmin·∞·n = n"
apply (rule lnat_ind [of _ n], auto)
apply (subst fold_inf [THEN sym])
by (simp del: fold_inf)

text ‹lnmin·m·∞ = m›
lemma lnmin_snd_inf[simp]: "lnmin·m·∞ = m"
apply (rule lnat_ind [of _ m], auto)
apply (subst fold_inf [THEN sym])
by (simp del: fold_inf)


lemma [simp]: "Fin 0 = 0"
by simp

text ‹lnmin·(Fin j)·(Fin k) = Fin (min j k)"›
lemma lnmin_fin[simp]: "lnmin·(Fin j)·(Fin k) = Fin (min j k)"
apply (rule_tac x=k in spec)
apply (induct_tac j, auto)
apply (case_tac x, auto)
by (simp add: Fin_def lnzero_def)

lemma lub_mono2: "⟦chain (X::nat⇒lnat); chain (Y::nat⇒lnat); ⋀i. X i ≤ Y i⟧
    ⟹(⨆i. X i) ≤ (⨆i. Y i)"
using lnle_conv lub_mono by blast


lemma inf_chainl2:
  "⟦chain Y; ¬ finite_chain Y⟧⟹∃j. Y k ⊑ Y j ∧ Y k ≠ Y j"
apply (auto simp add: finite_chain_def max_in_chain_def)
apply (erule_tac x="k" in allE, auto)
apply (frule_tac i=k and j=j in chain_mono, assumption)
by (rule_tac x="j" in exI, simp)


lemma max_in_chainI2: "⟦chain Y; ∀i. Y i = k⟧⟹max_in_chain 0 Y"
by (rule max_in_chainI, simp)


lemma finite_chainl1: "⟦chain Y; ¬ finite_chain Y⟧⟹¬ max_in_chain k Y"
apply (rule notI)
by (simp add: finite_chain_def)


lemma inf_chainl3:
  "⟦chain Y; ¬ finite_chain Y⟧⟹∃j. (Fin k) ⊑ Y j ∧ Fin k ≠ Y j"
apply (induct_tac k, simp+)
apply (case_tac "∀i. Y i = ⊥")
apply (frule_tac k="⊥" in max_in_chainI2, assumption)
apply (drule_tac k="0" in finite_chainl1, assumption, clarify)
apply (simp, erule exE)
apply (rule_tac x="i" in exI)
apply (simp add: lnzero_def)
apply (erule exE, erule conjE)
apply (frule_tac k="j" in finite_chainl1, assumption)
apply (simp add: max_in_chain_def)
apply (erule exE, erule conjE)
apply (rule_tac x="ja" in exI)
apply (rule_tac x="Y j" in lncases, simp+)
apply (drule_tac i="j" and j="ja" in chain_mono, assumption, simp+)
apply (rule_tac x="Y ja" in lncases, simp+)
by (drule_tac i="j" and j="ja" in chain_mono, assumption, simp+)

text ‹The least upper bound of an infinite lnat chain is ‹∞››
lemma unique_inf_lub: "⟦chain Y; ¬ finite_chain Y⟧⟹Lub Y = ∞"
apply (rule ccontr, drule ninf2Fin, erule exE)
apply (frule_tac k="k" in inf_chainl3, assumption)
apply (erule exE, simp)
apply (erule conjE)
apply (drule_tac x="j" in is_ub_thelub, simp)
by (rule_tac x="Y j" in lncases, simp+)


lemma compact_Fin: "compact (Fin k)"
apply (rule compactI)
apply (rule admI)
apply (case_tac "finite_chain Y")
apply (simp add: finite_chain_def)
apply (erule exE)
apply (drule lub_finchl [THEN lub_eqI], simp, simp)
apply (frule unique_inf_lub, assumption)
apply (subgoal_tac "range Y <| Fin k")
apply (drule_tac x="Fin k" in is_lub_thelub, simp+)
apply (rule ub_rangeI, simp)
apply (erule_tac x="i" in allE)
by (rule_tac x="Y i" in lncases, simp+)

text ‹If the outputs of a continuous function for finite inputs are
```

```
  bounded, the output for ‹∞› has the same bound›
lemma lnat_adml1[simp]: "adm (λx. f·x ≤ Fin n)"
apply (subst lnle_def)
apply (rule admI)
apply (subst contlub_cfun_arg, assumption)
apply (rule is_lub_thelub, rule chain_monofun, assumption)
by (rule ub_rangeI, simp)

text ‹If a continuous function returns ‹∞› for all finite
  inputs, it also returns ‹∞› for input ‹∞››
lemma lnat_adnl2[simp]: "adm (λx. f·x = ∞)"
apply (rule admI)
apply (subst contlub_cfun_arg, assumption)
apply (rule po_eq_conv [THEN iffD2])
apply (rule conjI)
apply (rule is_lub_thelub, rule chain_monofun, assumption)
apply (rule ub_rangeI, simp)
apply (erule_tac x="SOME x. True" in allE)
apply (drule sym, erule ssubst)
by (rule is_ub_thelub, rule chain_monofun)

text ‹l ≤ Fin k⟹l ≠∞›
lemma notinfI3: "l ≤ Fin k⟹l ≠∞"
by (rule_tac x="l" in lncases, simp+)



definition lnsucu ::"lnat u → lnat u" where
"lnsucu ≡ strictify·(Λ n. up·(fup·lnsuc·n))"


definition upinf  ::"lnat u" (* ("∞\<^isub>u") *) where
"upinf ≡ up·∞"

text ‹lnsucu·⊥ = ⊥›
lemma [simp]: "lnsucu·⊥ = ⊥"
by (simp add: lnsucu_def)

text ‹lnsucu·(up·(Fin n)) = up·(Fin (Suc n))›
lemma [simp]: "lnsucu·(up·(Fin n)) = up·(Fin (Suc n))"
by (simp add: lnsucu_def)

text ‹lnsucu·(upinf) = upinf›
lemma [simp]: "lnsucu·(upinf) = upinf"
by (simp add: lnsucu_def upinf_def)


lemma lnatu_cases:
  "⋀n P. ⟦n = upinf⟹P; ⋀k. n = up·(Fin k)⟹P; n = ⊥⟹P⟧⟹P"
apply (erule upE, auto simp add: upinf_def)
by (rule_tac x="x" in lncases, auto)

text ‹up·(Fin k) ≠ upinf›
lemma [simp]: "up·(Fin k) ≠ upinf"
by (simp add: upinf_def)

text ‹up·(Fin k) ≠⊥›
lemma [simp]: "up·(Fin k) ≠⊥"
by simp

text ‹upinf ≠⊥›
lemma [simp]: "upinf ≠⊥"
by (simp add: upinf_def)

text ‹lnsucu·lu ≠ up·0›
lemma [simp]: "lnsucu·lu ≠ up·0"
apply (rule_tac n="lu" in lnatu_cases)
apply (auto simp add: upinf_def)
by (simp add: lnsucu_def)

text ‹(lnsucu·l = up·(Fin (Suc n))) = (l = up·(Fin n))›
lemma [simp]: "(lnsucu·l = up·(Fin (Suc n))) = (l = up·(Fin n))"
apply (rule_tac n="l" in lnatu_cases)
apply (simp add: upinf_def)
by (auto simp add: lnsucu_def)

text ‹(lnsuc·n = Fin (Suc k)) = (n = Fin k)›
lemma [simp]: "(lnsuc·n = Fin (Suc k)) = (n = Fin k)"
by (simp add: Fin_Suc [THEN sym] del: Fin_Suc)

text ‹(lnsuc·n ≤ Fin (Suc k)) = (n ≤ Fin k)›
lemma [simp]: "(lnsuc·n ≤ Fin (Suc k)) = (n ≤ Fin k)"
by (simp add: Fin_Suc [THEN sym] del: Fin_Suc)

text ‹(lnsuc·n < Fin (Suc k)) = (n < Fin k)›
lemma [simp]: "(lnsuc·n < Fin (Suc k)) = (n < Fin k)"
by (simp add: Fin_Suc [THEN sym] del: Fin_Suc)

text ‹(Fin (Suc n) ≤ lnsuc·l) = (Fin n ≤ l)›
lemma [simp]: "(Fin (Suc n) ≤ lnsuc·l) = (Fin n ≤ l)"
by (simp add: Fin_Suc [THEN sym] del: Fin_Suc)

text ‹(Fin (Suc n) < lnsuc·l) = (Fin n < l)›
lemma [simp]: "(Fin (Suc n) < lnsuc·l) = (Fin n < l)"
```

```
by (simp add: Fin_Suc [THEN sym] del: Fin_Suc)

text ⟨¬ Fin (Suc n) < 0⟩
lemma [simp]: "¬ Fin (Suc n) < 0"
by (simp add: Fin_Suc [THEN sym] del: Fin_Suc)

text ⟨¬ Fin (Suc n) ≤ 0⟩
lemma [simp]: "¬ Fin (Suc n) ≤ 0"
by (simp add: Fin_Suc [THEN sym] del: Fin_Suc)

text ⟨∃x. Fin x < 0⟹False⟩
lemma [simp]: "∃x. Fin x < 0⟹False"
by (simp add: lnless_def)

text ⟨l ≠ 0⟹Fin (Suc 0) ≤ l⟩
lemma neq02Suclnle: "l ≠ 0⟹Fin (Suc 0) ≤ l"
by (rule_tac x="l" in lncases, simp+)

text ⟨(Fin k) < y⟹Fin (Suc k) ≤ y⟩
lemma less2lnleD: "(Fin k) < y⟹Fin (Suc k) ≤ y"
by (rule_tac x="y" in lncases, simp+)

(* ------------------------------------------------------------------------ *)
subsection ⟨Basic lemmas on @{term lmin}⟩
(* ------------------------------------------------------------------------ *)


instantiation lnat :: linorder
begin
  instance
  apply(intro_classes)
  using lnat_po_eq_conv lnle_def lnless_def apply blast
  apply simp
  using trans_lnle apply blast
  using lnat_po_eq_conv apply blast
  by (metis inf_ub less2nat linear ninf2Fin)
end


lemma ln_less[simp]: assumes "ln<∞"
  shows "ln < lnsuc·ln"
proof -
  have "ln ≤ lnsuc·ln" by simp
  obtain n where "Fin n = ln" by (metis assms dual_order.strict_implies_not_eq infI)
  have "Fin n < Fin (Suc n)" by force
  thus ?thesis using ⟨Fin n = ln⟩ by auto
qed

lemma lnle2le: "m < lnsuc·n⟹m ≤ n"
  apply (case_tac "m=∞", auto)
  by (metis Fin_Suc less2lnleD lncases lnsuc_lnle_emb)

lemma le2lnle: "m < ∞⟹lnsuc·m ≤ n⟹m < n"
  by (metis dual_order.strict_iff_order dual_order.trans leD ln_less)

(*few lemmas to simp min*)
text⟨∞ is greater than or equal to any lazy natural number⟩
lemma [simp]: fixes ln :: lnat
  shows "min ∞ ln = ln"
by (simp add: min_def)

lemma [simp]: fixes ln :: lnat
  shows "min ln ∞ = ln"
by (simp add: min_def)


lemma [simp]: fixes ln :: lnat
  shows "min ln 0 = 0"
by (simp add: min_def)

lemma [simp]: fixes ln :: lnat
  shows "min 0 ln = 0"
by (simp add: min_def)

lemma min_rek: assumes  "z = min x (lnsuc·z)"
  shows "z = x"
  apply(rule ccontr, cases "x < z")
   apply (metis assms dual_order.irrefl min_less_iff_conj)
  by (metis assms inf_ub ln_less lnle_def lnless_def min_def)

lemma lnat_well_h1:
  "[| n < Fin m; ⋀k. n = Fin k⟹k < m⟹P |] ==> P"
by (metis less2nat less_le lncases notinfI3)

lemma lnat_well_h2:
  "[| n < ∞; ⋀k. n = Fin k⟹P |] ==> P"
using lncases by auto

lemma lnat_well:
  assumes prem: "⋀n. ∀m::lnat. m < n⟶P m⟹P n" shows "P n"
proof -
  have P_lnat: "⋀k. P (Fin k)"
    apply (rule nat_less_induct)
    apply (rule prem, clarify)
```

```
    apply (erule lnat_well_h1, simp)
    done
  show ?thesis
  proof (induct n)
    next show "adm P" by (metis P_lnat adm_upward inf_ub lnat_well_h2 less_le_trans prem)
    next show "P ⊥" by (metis Fin_02bot P_lnat)
    then show "⋀n. P n⟹P (lnsuc·n)" by (metis Fin_Suc P_lnat lncases)
  qed
qed

instance lnat :: wellorder
proof
  fix P and n
  assume hyp: "(⋀n::lnat. (⋀m::lnat. m < n ==> P m) ==> P n)"
  show "P n" by (blast intro: lnat_well hyp)
qed


(* ----------------------------------------------------------------------- *)
subsection ‹Basic lemmas on @{term lnat_plus}›
(* ----------------------------------------------------------------------- *)


lemma lnat_plus_fin [simp]: "(Fin n) + (Fin m) = Fin (n + m)"
  apply(simp add: plus_lnat_def)
  by (metis UNIV_I f_inv_into_f image_eqI inject_Fin)


lemma plus_lnat0_0:"Fin 0 + Fin 0 = Fin 0"
  apply(simp add: plus_lnat_def)
  apply(simp add: Fin_def inv_def)
  apply(rule_tac someI_ex)
  using Fin_def lnle_Fin_0 by auto


lemma plus_lnat0_r[simp]:"(0::lnat) + n = n"
  apply(simp add: plus_lnat_def)
  by (metis Fin_0 Inf'_neq_0_rev add_cancel_right_left plus_lnat_def lnat_plus_fin ninf2Fin)


lemma plus_lnat0_l:"m + (0::lnat) = m"
  apply(simp add: plus_lnat_def)
  by (metis (mono_tags, lifting) Fin_0 UNIV_I add.right_neutral f_inv_into_f image_eqI plus_lnat_def plus_lnat0_r)

text ‹m + ∞ = ∞›
lemma plus_lnatInf_l[simp]:"m + ∞ = ∞"
  by(simp add: plus_lnat_def)

text ‹∞ + n = ∞›
lemma plus_lnatInf_r:"∞ + n = ∞"
  by(simp add: plus_lnat_def)


lemma lnat_plus_commu:"(ln1::lnat) + ln2 = ln2 + ln1"
  by(simp add: plus_lnat_def)


instance lnat:: semigroup_add
  apply(intro_classes)
  apply(simp add: plus_lnat_def)
  by (smt add.left_commute f_inv_into_f inject_Fin natl2 rangeI)


instance lnat:: ab_semigroup_add
  apply(intro_classes)
  by (simp add: lnat_plus_commu)


instance lnat:: monoid_add
  apply(intro_classes)
  apply (simp)
  by (simp add: plus_lnat0_l)


instantiation lnat :: one
begin
definition one_lnat:: "lnat" where
  "one_lnat = Fin 1"

  instance ..
end


lemma one_def: "1 = lnsuc·0"
   by (metis Fin_02bot Fin_Suc One_nat_def lnzero_def one_lnat_def)

lemma lnat_1_inf [simp]: "1 < ∞"
  unfolding one_lnat_def
  by simp


lemma lnat_plus_suc: "ln1 + 1 = lnsuc·ln1"
  apply(simp add: plus_lnat_def)
  by (metis Fin_Suc Inf'_neq_0_rev One_nat_def Suc_def2 f_inv_into_f fold_inf inf_ub inject_Fin inject_lnsuc
```

```
        less_le lnat_well_h2 one_def one_lnat_def rangeI)


lemma lnat_plus_lnsuc: "ln1 + (lnsuc·ln2) = (lnsuc·ln1) + ln2"
  apply(simp add: plus_lnat_def)
  proof -
    have f1: "⋀f n. f (inv f (f (n::nat)::lnat)) = f n"
      by (simp add: f_inv_into_f)
    have "⋀l. Fin (inv Fin l) = l ∨ ∞ = l"
      by (metis (no_types) f_inv_into_f ninf2Fin rangeI)
    then have f2: "⋀l. inv Fin (lnsuc·l) = Suc (inv Fin l) ∨ ∞ = l"
      using f1 by (metis (no_types) Fin_Suc inject_Fin)
    then have "⋀n l. n + inv Fin (lnsuc·l) = inv Fin l + Suc n ∨ ∞ = l"
      by simp
    then show "ln1 ≠ ∞ ∧ ln2 ≠ ∞ ⟶ inv Fin ln1 + inv Fin (lnsuc·ln2) = inv Fin (lnsuc·ln1) + inv Fin ln2"
      using f2 by (metis (no_types) natl2)
  qed


lemma min_adm[simp]: fixes y::lnat
  shows "adm (λx. min y (g·x) ⊑ h·x)"
proof (rule admI)
  fix Y
  assume Y_ch: "chain Y"  and as: "∀i. min y (g·(Y i)) ⊑ h·(Y i)"
  have h1:"finite_chain Y⟹min y (g·(⨆i. Y i)) ⊑ h·(⨆i. Y i)"
    using Y_ch as l42 by force
  have "¬finite_chain Y⟹min y (g·(⨆i. Y i)) ⊑ h·(⨆i. Y i)"
  proof (cases "g·(⨆i. Y i) ⊑ y")
    case True
    hence "⋀i. g·(Y i) ⊑ y"
      using Y_ch is_ub_thelub monofun_cfun_arg rev_below_trans by blast
    then show ?thesis
      by (metis (no_types, lifting) Y_ch as ch2ch_Rep_cfunR contlub_cfun_arg lnle_conv lub_below_iff lub_mono
           min_absorb2)
  next
    case False
    then show ?thesis
      by (metis Y_ch as below_lub ch2ch_Rep_cfunR contlub_cfun_arg lnle_conv lub_below min.commute min_def)
  qed
  thus "min y (g·(⨆i. Y i)) ⊑ h·(⨆i. Y i)"
    using h1 by blast
qed

lemma min_adm2[simp]: fixes y::lnat
  shows "adm (λx. min (g·x) y ⊑ h·x)"
  apply(subst min.commute)
  using min_adm by blast

lemma lub_sml_eq:"⟦chain (Y::nat⇒lnat); ⋀i. x ≤ Y i⟧⟹x ≤ (⨆i. Y i)"
  using l42 unique_inf_lub by force


lemma min_lub:" chain Y⟹(⨆i::nat. min (x::lnat) (Y i)) = min (x) (⨆i::nat. (Y i))"
  apply (case_tac "x=∞", simp_all)
  apply (case_tac "finite_chain Y")
proof -
  assume a1: "chain Y"
  assume a2: "finite_chain Y"
  then have "monofun (min x)"
    by (metis (mono_tags, lifting) lnle_conv min.idem min.semilattice_order_axioms monofunI
        semilattice_order.mono semilattice_order.orderI)
  then show ?thesis
    using a2 a1 by (metis (no_types) finite_chain_lub)
next
  assume a0:"chain Y"
  assume a1:"¬ finite_chain Y"
  assume a2:"x ≠ ∞"
  have h0:"∀i. ∃j≥i. Y i ⊑ Y j"
  by blast
  then have"(⨆i. min x (Y i)) = x"
  proof -
    have f1: "⋀n. min x (Y n) ⊑ x"
      by (metis (lifting) lnle_def min.bounded_iff order_refl)
    then have f2: "⋀n. min x (Y n) = x ∨ Y n ⊑ x"
      by (metis (lifting) min_def)
    have f3: "∞ \<notsqsubseteq> x"
      by (metis (lifting) a2 inf_less_eq lnle_def)
    have "Lub Y = ∞"
      by (meson a0 a1 unique_inf_lub)
    then obtain nn :: "(nat ⇒ lnat) ⇒ lnat ⇒ nat" where
      f4: "min x (Y (nn Y x)) = x ∨ ∞ ⊑ x"
      using f2 by (metis (no_types) a0 lub_below_iff)
    have "∀f n. ∃na. (f (na::nat)::lnat) \<notsqsubseteq> f n ∨ Lub f = f n"
      by (metis lub_chain_maxelem)
    then show ?thesis
      using f4 f3 f1 by (metis (full_types))
    qed
  then show ?thesis
    by (simp add: a0 a1 unique_inf_lub)
qed


lemma min_lub_rev:"chain Y⟹ min (x) (⨆i::nat. (Y i)) = (⨆i::nat. min (x::lnat) (Y i)) "
  using min_lub by auto
```

```
text‹≤ relation between two chains in a minimum is as well preserved by their lubs.›
lemma lub_min_mono: "⟦chain (X::nat⇒lnat); chain (Y::nat⇒lnat); ⋀i. min x (X i) ≤ Y i⟧
   ⟹min x (⨆i. X i) ≤ (⨆i. Y i)"
  by (metis dual_order.trans is_ub_thelub lnle_def lub_mono2 min_le_iff_disj)

text‹Twisted version of lub_min_mono: ≤ rel. between two chains in minimum is preserved by lubs.›
lemma lub_min_mono2: "⟦chain (X::nat⇒lnat); chain (Y::nat⇒lnat); ⋀i. min (X i) y ≤ Y i⟧
   ⟹min (⨆i. X i) y ≤ (⨆i. Y i)"
  by (metis dual_order.trans is_ub_thelub lnle_def lub_mono2 min_le_iff_disj)


lemma lessequal_addition: assumes "a ≤ b" and "c ≤ d" shows "a + c ≤ b + (d :: lnat)"
proof -
  have "b = ∞⟹a + c ≤ b + d"
    by (simp add: plus_lnatInf_r)
  moreover
  have "d = ∞⟹a + c ≤ b + d"
    by (simp add: plus_lnatInf_r)
  moreover
  have "a = ∞⟹a + c ≤ b + d"
    using assms(1) plus_lnatInf_r by auto
  moreover
  have "c = ∞⟹a + c ≤ b + d"
    using assms(2) plus_lnatInf_r by auto
  moreover
  have "a ≠∞⟹b ≠∞⟹c ≠∞⟹d ≠∞⟹a + c ≤ b + d"
  proof -
    assume "a ≠∞"
    then obtain m where m_def: "Fin m = a"
      using infI by force
    assume "b ≠∞"
    then obtain n where n_def: "Fin n = b"
      using infI by force
    assume "c ≠∞"
    then obtain x where x_def: "Fin x = c"
      using infI by force
    assume "d ≠∞"
    then obtain y where y_def: "Fin y = d"
      using infI by force
    show ?thesis
      using assms m_def n_def x_def y_def by auto
    qed
  then show "a + c ≤ b + d"
    using calculation by blast
qed

lemma lnmin_eqasmthmin: assumes "a = b" and "a ≤ c" shows "a = lnmin·b·c"
proof -
  have "a = ∞⟹a = lnmin·b·c"
    using assms by auto
  moreover
  have "b = ∞⟹a = lnmin·b·c"
    using assms by auto
  moreover
  have "c = ∞⟹a = lnmin·b·c"
    using assms by auto
  moreover
  have "a ≠∞⟹b ≠∞⟹c ≠∞⟹ a = lnmin·b·c"
    by (metis assms less2nat lncases lnmin_fin min.order_iff)

  then show ?thesis
    using calculation by blast
qed

lemma lnmin_asso: "lnmin·x·y = lnmin·y·x"
proof -
  have "x = ∞⟹lnmin·x·y = lnmin·y·x"
    by simp
  moreover
  have "y = ∞⟹lnmin·x·y = lnmin·y·x"
    by simp
  moreover
  have "x ≠∞⟹y ≠∞⟹lnmin·x·y = lnmin·y·x"
    by (metis (full_types) lncases lnmin_fin min.commute)
  then show ?thesis
    using calculation by blast
qed

lemma lnmin_smaller_addition: "lnmin·x·y ≤ x + y"
proof -
  have "x = ∞⟹lnmin·x·y ≤ x + y"
    by (simp add: plus_lnatInf_r)
  moreover
  have "y = ∞⟹lnmin·x·y ≤ x + y"
    by simp
  moreover
  have "x ≠∞⟹y ≠∞⟹lnmin·x·y ≤ x + y"
    by (metis bot_is_0 lessequal_addition linear lnle_def lnmin_asso lnmin_eqasmthmin minimal plus_lnat0_l)
  then show ?thesis
    using calculation by blast
qed

lemma lnat_no_chain: fixes Y:: " nat ⇒ lnat"
```

```
  assumes "range Y = UNIV"
  shows "¬chain Y"
proof (rule ccontr)
  assume "¬¬chain Y"
  obtain i where "Y i = ∞"
    by (metis assms surj_def)
  hence "max_in_chain i Y"
    by (metis ‹¬¬chain Y› inf_less_eq is_ub_thelub lnle_conv max_in_chainI3)
  hence "finite (range Y)"
    using Prelude.finite_chainI ‹¬ ¬ chain Y› finch_imp_finite_range by blast
  thus False
    by (metis (mono_tags, hide_lams) Fin_neq_inf assms ex_new_if_finite  finite_imageI infinite_UNIV_nat
          inject_Fin lncases rangeI)
qed

text ‹If the left summand is smaller then {@term ∞}, then the right summand is unqiuely
        determined by the result of {@term +}›
lemma plus_unique_r:
  fixes "l"
  assumes "m < ∞"
  and "(l::lnat) = m + n"
  and "(l::lnat) = m + p"
shows "n = p"
  using assms apply(induction l, simp_all)
  apply (smt add_left_imp_eq fold_inf inject_Fin less_lnsuc lnat.sel_rews(2) lnat_plus_suc neq_iff notinfI3
        plus_lnat_def triv_admI)
  using assms apply(induction m, simp_all)
  apply (simp_all add: bot_is_0)
  apply (smt add.left_commute lnat_plus_commu plus_lnat0_l triv_admI)
  apply (metis add.left_commute plus_lnat0_l)
  apply (case_tac "l = ∞")
  apply simp_all
proof -
  fix la :: lnat
  assume a1: "m + n ≠ ∞"
  assume a2: "m + p = m + n"
  have f3: "n = 0 + n"
  by auto
  have f4: "∀l la. if l = ∞ ∨ la = ∞ then l + la = ∞ else l + la = Fin (inv Fin l + inv Fin la)"
  using plus_lnat_def by presburger
    have f5: "0 ≠ ∞ ∧ n ≠ ∞"
      using a1 by force
    have f6: "m ≠ ∞ ∧ p ≠ ∞"
      using f4 a2 a1 by metis
    then have f7: "Fin (inv Fin m + inv Fin p) = m + n"
      using f4 a2 by simp
    have "m ≠ ∞ ∧ n ≠ ∞"
      using f4 a1 by fastforce
    then have "inv Fin p = inv Fin n"
      using f7 f4 by simp
    then have "n = 0 + p"
      using f6 f5 f4 f3 by presburger
    then show ?thesis
  by auto
qed

text ‹If the right summand is smaller then {@term ∞}, then the left summand is unqiuely
        determined by the result of {@term +}›
lemma plus_unique_l:
  fixes "l"
  assumes "m < ∞"
  and "(l::lnat) = n + m"
  and "(l::lnat) = p + m"
shows "n = p"
  using assms plus_unique_r
  by (metis lnat_plus_commu)

text‹Declares Fin and {@term ∞} as constructors for lnat. This is useful for patterns that use constructors›
setup ‹Sign.mandatory_path "LNat"›
old_rep_datatype Fin Inf'
  apply (metis ninf2Fin)
  by simp+
setup ‹Sign.parent_path›

(* Allows to directly write "11" instead of "Fin 11" *)
instance lnat::numeral
  by(intro_classes)

lemma lnat_num2fin[simp]: "numeral n = Fin (numeral n)"
  apply(induction n, auto)
  apply (simp add: one_lnat_def)
   apply (metis lnat_plus_fin numeral_Bit0)
  apply(simp add: numeral_Bit1)
  by (metis lnat_plus_fin numeral_One numeral_plus_numeral one_lnat_def)


class len = pcpo +
  fixes len :: "'a::pcpo ⇒ lnat"
  assumes len_mono: "monofun len"
begin
abbreviation len_abbr :: "'a ⇒ lnat" ("#_" [1000] 999) where
"#s ≡ len s"

lemma mono_len: "x ⊑ y⟹#x ≤ #y"
```

```
  using len_mono lnle_def monofun_def by blast

lemma mono_len2: "x ⊑ y⟹#x ⊑ #y"
  using local.len_mono monofunE by blast


end

instantiation prod :: (len, len) len
begin
definition len_prod::"('a::len×'b::len) ⇒ lnat" where
"len_prod tp = min (#(fst tp)) (#(snd tp))"

instance
  apply(intro_classes)
  unfolding len_prod_def monofun_def
  apply (auto simp add: below_prod_def min_def)
  by (meson le_cases len_mono lnle_conv monofun_def trans_lnle)+
end

lemma len_prod_min: "#(a,b) ≤ #a" and  "#(a,b) ≤ #b"
  by (auto simp add: len_prod_def)



instantiation unit::len
begin
definition len_unit::"unit ⇒ lnat" where
"len_unit un = ∞"

instance
  apply(intro_classes)
  apply(rule monofunI)
  by(simp add: len_unit_def)
end


end
```

# Appendix B

# Stream Theories

## B.1 Streams

```
(*:maxLineLen=68:*)
section ‹Lazy Streams›

theory Stream
imports inc.LNat inc.SetPcpo
begin

section ‹The Datatype of Lazy Streams›

default_sort countable

(* deletes the Rule "1 = Suc 0" *)
 declare One_nat_def[simp del]

(* declare [[show_types]] *)
text ‹‹discr u› lifts an arbitrary type ‹'a› to the
  discrete ‹pcpo› and the usual rest operator ‹rt› on streams.›



section‹Streams›


text‹The stream domain in Isabelle is defined with the ‹domain›
command provided by the ‹HOLCF› package. This automatically
instantiates our type as a \gls{pcpo}.›

domain
  'a stream = lscons (lshd::"'a discr u") (lazy srt::"'a stream")
                                        (infixr "&&" 65)


subsection‹Stream Functions›

text‹\Cref{tab:streamfun} gives an overview
about the main functions defined for @{type stream}s. Some of them
are introduced in detail for their later usage in the Isabelle
framework. The type ‹N\<^sub>∞› representing the natural numbers inclusive
‹∞› is defined as \gls{lnat}. An introduction is in \cite{GR07}.›




(* ------------------------------------------------------------------------ *)
section ‹Signatures of Stream Processing Functions›
(* ------------------------------------------------------------------------ *)


type_synonym ('in , 'out) spf = "('in stream → 'out stream)"
type_synonym 'm spfo      = "('m, 'm) spf"


type_synonym ('in , 'out) gspf = "('in stream ⇒ 'out stream)"
type_synonym 'm gspfo      = "('m, 'm) gspf"


type_synonym ('in , 'out) lpf = "('in list ⇒ 'out list)"
```

```
type_synonym 'm lpfo     = "('m, 'm) lpf"

(* ----------------------------------------------------------------------- *)
subsection ‹Some abbreviations›
(* ----------------------------------------------------------------------- *)

text ‹The empty stream is denoted as ‹ϵ›.›
abbreviation
  sbot :: "'a stream"  ("ϵ")
    where "sbot ≡ ⊥"

text ‹@{term stake}: This operator is generated by the ‹domain›
  command. It retrieves the first ‹n› elements of a stream
  (or less, if the stream is shorter).›
abbreviation
  stake :: "nat ⇒ 'a spfo"
    where "stake ≡ stream_take"


(* ----------------------------------------------------------------------- *)
section ‹Common functions on streams›
(* ----------------------------------------------------------------------- *)


definition fup2map :: "('a ⇒ 'b::cpo u) ⇒ ('a ⇀ 'b)" where
"fup2map f a ≡ if (f a = ⊥) then None else Some (SOME x. up·x = f a)"


definition sup'       :: "'a ⇒ 'a stream" ("↑_" [1000] 999) where
"sup' a ≡ updis a && ϵ"


definition sconc      :: "'a stream ⇒ 'a spfo" where
"sconc ≡ fix·(Λ h. (λ s1. Λ s2.
                      if s1 = ϵ then s2 else  (lshd·s1) && (h (srt·s1)·s2)))"

abbreviation sconc_abbr :: "'a stream ⇒ 'a stream ⇒ 'a stream" ("(_ • _)" [66,65] 65)
where "s1 • s2 ≡ sconc s1·s2"

text ‹@{term shd}: Retrieve the first element of a stream.
  For ‹ϵ›, the result is not defined.›
definition shd        :: "'a stream ⇒ 'a" where
"shd s ≡ THE a. lshd·s = updis a"

text ‹@{term slookahd}: Apply function to head of stream.
  If the stream is empty, ‹⊥› is returned.
  This function is especially useful for defining own stream-processing
  functions.›
definition slookahd   :: "'a stream → ('a ⇒ 'b) → ('b::pcpo)" where
"slookahd ≡ Λ s f. if s = ϵ then ⊥ else f (shd s)"


(* ----------------------------------------------------------------------- *)
subsection ‹Conversion of lists to streams and induced order on lists›
(* ----------------------------------------------------------------------- *)



primrec list2s :: "'a list ⇒ 'a stream"
where
  list2s_0:   "list2s [] = ϵ" |
  list2s_Suc: "list2s (a#as) = updis a && (list2s as)"

abbreviation stream_abbrev :: "'a list ⇒ 'a stream" ("<_>" [1000] 999)
where "<l> == list2s l"

text ‹The data type ‹list› is a partial order with the operator
  ‹⊑› derived from streams:›
instantiation list :: (countable) po
begin

  definition sq_le_list:
    "s ⊑ t ≡ (list2s s ⊑ list2s t)"

  (* list2s is a bijection *)
  lemma list2s_inj[simp]: "(list2s l = list2s l') = (l = l')"
  apply (rule iffI)
  apply (simp add: atomize_imp)
  apply (rule_tac x="l'" in spec)
  apply (induct l, simp)
  apply (rule allI)
  apply (induct_tac x, simp+)
  apply (rule allI)
  by (induct_tac x, simp+)

instance
apply (intro_classes)
apply (simp add: sq_le_list)+
apply (rule_tac y="list2s y" in below_trans,assumption+)
apply (simp add: sq_le_list)
apply (rule list2s_inj [THEN iffD1])
by (rule po_eq_conv [THEN iffD2],rule conjI,assumption+)
```

```
end

(* ------------------------------------------------------------------------ *)

(* ------------------------------------------------------------------------ *)

subsubsection‹Length of Streams›

text ‹@{term slen}: Retrieve the length of a stream. It is defined
as the number of its elements or ‹∞› for infinite streams.›

definition slen:: "'a stream → lnat" where
"slen ≡ fix·(Λ h. strictify·(Λ s. lnsuc·(h·(srt·s))))"


text‹\isacommand{theorem} slen\_eq::‹"x ⊑ y⟹#x = #y⟹x = y"››

text‹Abbreviation ‹#› is used for obtaining the length of a
stream.›


instantiation stream :: (countable) len
begin
definition len_stream::"'a stream ⇒ lnat" where
"len_stream s = slen·s"

instance
  apply(intro_classes)
  apply (auto simp add: monofun_def below_prod_def min_def)
  by (metis (mono_tags) cont_pref_eq1l len_stream_def lnle_conv)

end

lemma slen_zero [simp]: "#ϵ = 0"
   apply(simp add:len_stream_def)
  by(subst slen_def [THEN fix_eq2], simp add: lnzero_def)

text ‹@{term sdrop}: Remove the first ‹n› elements
  of the stream.›
definition sdrop      :: "nat ⇒ 'a spfo" where
"sdrop n ≡ Fix.iterate n·srt"


subsubsection‹Stream elements›

text‹The element of a stream at position ‹n› can be accessed by
dropping the first ‹n› elements of the stream. We start counting
positions at 0.›

definition snth  :: "nat ⇒ 'a stream ⇒ 'a" where
"snth k s ≡ shd (sdrop k·s)"




definition sfoot      :: "'a stream ⇒ 'a" where
"sfoot s = snth (THE a. lnsuc·(Fin a) = #s) s"


subsubsection‹Streams Values›

text ‹The values of a stream are a set of messages of type ‹'a› that
occur at any position in the stream.›

definition sValues :: "'a stream → 'a set" where
"sValues ≡ Λ s. {snth n s | n. Fin n < #s}"




text ‹@{term sntimes}: Repeat the given stream ‹n› times.›
text ‹(Only listed as a constant below for reference;
  Use ‹sntimes› with same signature instead).›
(* consts sntimes_    :: "nat ⇒ 'a stream ⇒ 'a stream" *)


definition sinftimes  :: "'a stream ⇒ 'a stream" ("_^∞") where
 "sinftimes ≡ fix·(Λ h. (λs.
                        if s = ϵ then ϵ else (s • (h s))))"

subsubsection‹Applying functions element-wise›


definition smap:: "('a ⇒ 'b) ⇒ 'a stream → 'b stream" where
"smap f ≡ fix·(Λ h s. slookahd·s·(λ a.
                         ↑(f a) • (h·(srt·s))))"

text‹ The ‹n›th element of
the output stream is equal to applying the mapping function to the
```

⟨n⟩th element of the input stream.⟩

```
text⟨\isacommand{theorem} smap\_snth :: ⟨"snth n (smap f·s) = f (snth n s)"⟩⟩
```

```
definition sfilter    :: "'a set ⇒ 'a spfo" where
"sfilter M ≡ fix·(Λ h s. slookahd·s·(λ a.
                          (if (a ∈ M) then ↑a • (h·(srt·s)) else h·(srt·s))))"

text ⟨@{term stakewhile}: Take the first elements of a stream as
  long as the given function evaluates to ⟨true⟩.⟩
definition stakewhile :: "('a ⇒ bool) ⇒ 'a spfo" where
"stakewhile f ≡ fix·(Λ h s. slookahd·s·(λ a. if (f a) then ↑a • h·(srt·s) else ϵ))"

text ⟨@{term sdropwhile}: Drop the first elements of a stream as
  long as the given function evaluates to ⟨true⟩.⟩
definition sdropwhile :: "('a ⇒ bool) ⇒ 'a spfo" where
"sdropwhile f ≡ fix·(Λ h s. slookahd·s·(λ a.
                          if (f a) then h·(srt·s) else s))"
```

```
definition szip       :: "'a stream → 'b stream → ('a × 'b) stream" where
 "szip ≡ fix·(Λ h s1 s2. slookahd·s1·(λ a. slookahd·s2·(λ b.
                    ↑(a,b) • (h·(srt·s1)·(srt·s2)))))"

definition merge:: "('a  ⇒ 'b ⇒ 'c) ⇒ 'a stream → 'b stream → 'c stream" where
"merge f ≡ Λ s1 s2 . smap (λ s3. f (fst s3) (snd s3))·(szip·s1·s2)"

definition sprojfst   :: "(('a × 'b),'a) spf" where
"sprojfst ≡ Λ x. smap fst·x"

definition sprojsnd   :: "(('a × 'b),'b) spf" where
"sprojsnd ≡ Λ x. smap snd·x"
```

```
definition stwbl      :: "('a ⇒ bool) ⇒ 'a spfo" where
"stwbl f ≡ fix·(Λ h s. slookahd·s·(λ a.
                          if (f a) then ↑a •  h·(srt·s) else ↑a))"

text ⟨@{term srtdw}: Rest (⟨srt⟩) of stream after dropwhile.⟩
definition srtdw      :: "('a ⇒ bool) ⇒ 'a spfo" where
"srtdw f ≡ Λ x. srt·(sdropwhile f·x)"

definition srcdups    :: "'a spfo" where
"srcdups ≡ fix·(Λ h s. slookahd·s·(λ a.
                       ↑a •  h·(sdropwhile (λ z. z = a)·(srt·s))))"

(* Takes a nat indicating the number of elements to scan, a reducing
 function, an initial initial element, and an input stream. Returns
a stream consisting of the partial reductions of the input stream.*)
primrec SSCANL::
"nat ⇒ ('o ⇒ 'i ⇒ 'o) ⇒ 'o ⇒ 'i stream ⇒ 'o stream" where
  SSCANL_zero_def: "SSCANL 0 f q s = ϵ" |
  "SSCANL (Suc n) f q s = (if s=ϵ then ϵ
                          else ↑(f q (shd s)) •
                               (SSCANL n f (f q (shd s)) (srt·s)))"

text ⟨@{term sscanl}: Apply a function elementwise to the input
stream. Behaves like ⟨map⟩, but also takes the previously generated
output element as additional input to the function. For the first
computation, an initial value is provided.⟩
definition sscanl :: "('o ⇒ 'i ⇒ 'o) ⇒ 'o ⇒ ('i, 'o) spf" where
"sscanl f q ≡ Λ s. ⨆i. SSCANL i f q s"
```

```
(* scanline Advanced :D  *)
(* or stateful ... *)
(* The user has more control. Instead of the last output ('b) a
state ('s) is used as next input *)
definition sscanlA ::
 "('s ⇒'a ⇒ ('b ×'s)) ⇒ 's  ⇒ 'a stream → 'b stream" where
"sscanlA f s0 ≡ Λ s. sprojfst·(sscanl (λ(_,b). f b) (undefined, s0)·s)"

subsubsection⟨Applying stateful functions element-wise⟩

text⟨One can also apply a state dependent function, like the
transition function of a deterministic automaton, to process the
streams elements. The ⟨(n+1)⟩th output element then depends on the
⟨n⟩th ouput state, because the stateful function may act
differently depending on its state. Hence, we also need an initial
state to start computing the output.⟩
```

```
definition sscanlAg ::
"('s ⇒'a ⇒ ('s ×'b)) ⇒ 's ⇒ 'a stream → ('s×'b) stream" where
"sscanlAg f s0 ≡ Λ s. (sscanl (λ(b,_). f b) (s0, undefined)·s)"



definition %invisible sscanlAfst ::
"('s ⇒'a ⇒ ('s×'b)) ⇒ 's ⇒ 'a stream → 's stream" where
"sscanlAfst f s0 ≡ Λ s. sprojfst·(sscanlAg f s0·s)"

definition sscanlAsnd ::
"('s ⇒'a ⇒ ('s ×'b)) ⇒ 's  ⇒ 'a stream → 'b stream" where
"sscanlAsnd f s0 ≡ Λ s. sprojsnd·(sscanlAg f s0·s)"



text ‹@{term siterate}: Create a stream by repeated application of
  a function to an element. The generated stream starts with ‹a›,
  ‹f(a)›, ‹f(f(a))›, and so on.›
definition siterate   :: "('a ⇒ 'a) ⇒ 'a ⇒ 'a stream" where
"siterate f (a::'a) ≡ ↑a • sscanl (λa (b::'a). f a) a·(SOME x. #x = ∞)"


definition siterateBlock:: "('a stream ⇒ 'a stream) ⇒ 'a stream ⇒ 'a stream" where
"siterateBlock f ≡ fix · (Λ h. (λs. s • (h (f s))))"


(* ---------------------------------------------------------------------- *)

(* ---------------------------------------------------------------------- *)


text ‹(Only listed as a constant below for reference;
  Use ‹list2s› with same signature instead).›
(* consts list2s_   :: "'a list ⇒ 'a stream" *)


definition s2list     :: "'a stream ⇒ 'a list" where
"s2list s ≡ if #s ≠ ∞ then SOME l. list2s l = s else undefined"


definition slpf2spf   ::"('in,'out) lpf ⇒ ('in,'out) spf" where
"slpf2spf f ≡
    if monofun f
      then Λ s. (⊔k. list2s (f (s2list (stake k·s))))
      else undefined"


definition sislivespf ::"('in,'out) spf ⇒ bool" where
"sislivespf f ≡ (∀x. #(f·x) = ∞⟶#x = ∞)"


definition sspf2lpf   ::"('in,'out) spf ⇒ ('in,'out) lpf" where
"sspf2lpf f ≡  if sislivespf f then (λx. s2list (f·(list2s x))) else undefined"

(* ---------------------------------------------------------------------- *)
subsection ‹Syntactic sugar and helpers›
(* ---------------------------------------------------------------------- *)


abbreviation sfilter_abbr :: "'a set ⇒ 'a stream ⇒ 'a stream" ("(_ ⊖ _)" [66,65] 65)
where "F ⊖ s ≡ sfilter F·s"


(* ---------------------------------------------------------------------- *)
subsection ‹Definition of stream manipulating functions›
(* ---------------------------------------------------------------------- *)




(* concatenates a stream to itself n times *)
primrec sntimes :: "nat ⇒ 'a stream ⇒ 'a stream" where
  "sntimes 0 s = ε" |
  "sntimes (Suc n) s = (sconc s)·(sntimes n s)"

(* Abbreviation for sntimes *)
abbreviation sntimes_abbr :: "nat ⇒ 'a stream ⇒ 'a stream" ("_⋆_" [60,80] 90)
where "(n ⋆ s)  == (sntimes n s)"



(* ---------------------------------------------------------------------- *)
section ‹Stream - basics›
(* -------------------------------------␣-------------------------------- *)
```

```
(* ----------------------------------------------------------------------- *)
subsection ‹Fundamental properties of @{term stake}›
(* ----------------------------------------------------------------------- *)


lemmas scases' = stream.exhaust
lemmas sinjects' = stream.injects
lemmas sinverts' = stream.inverts


lemma reach_stream: "(⨆i. stake i·s) = s"
apply (rule stream.take_lemma [OF spec [where x=s]])
apply (induct_tac n, simp, rule allI)
apply (rule_tac y=x in scases', simp)
apply (subst lub_range_shift [where j="Suc 0", THEN sym],simp+)
by (subst contlub_cfun_arg [THEN sym], auto)

(* if two streams xs and ys are identical for any prefix that is a multiple of y long, then the two
   streams are identical for any prefix *)
lemma gstake2stake: assumes "∀i. stake (i*y)·xs = stake (i*y)·ys" and "y≠0"
  shows "∀i. stake i·xs = stake i·ys"
proof
  fix i
  obtain k where "∃l. k = y*l" and "k≥i" by (metis One_nat_def Suc_le_eq assms(2) gr0I mult.commute mult_le_mono2
        nat_mult_1_right)
  thus "stake i·xs = stake i·ys" by (metis assms(1) min_def mult.commute stream.take_take)
qed

(* stake is monotone *)
lemma stake_mono: assumes "i≤j"
  shows "stake i·s ⊑ stake j·s"
by (metis assms min_def stream.take_below stream.take_take)

(* ----------------------------------------------------------------------- *)
subsection ‹Construction by concatenation and more›
(* ----------------------------------------------------------------------- *)

text ‹Basic properties of ‹↑_› constructor›

(* shd composed with ↑ is the identity. *)
lemma [simp]: "shd (↑a) = a"
  by (simp add: shd_def sup'_def)

lemma [simp]: "<[a]> = ↑a"
  by(simp add: sup'_def)

(* the singleton stream is never equal to the empty stream *)
lemma [simp]: "↑a ≠ ε"
by (simp add: sup'_def)

(* the rest of the singleton stream is empty *)
lemma [simp]: "srt·(↑a) = ε"
by (simp add: sup'_def)


lemma reduce_seq: (*never simp*)
  assumes "s1 = s2"
  shows "s • s1 = s • s2"
  by (simp add: assms)

(* the empty stream is the identity element with respect to concatenation *)
lemma sconc_fst_empty[simp]:"ε • s = s"
apply (subst sconc_def [THEN fix_eq2])
by (simp add: cont2cont_LAM)

(* the lazy stream constructor and concatenation are associative *)
lemma sconc_scons': "(updis a && as) • s = updis a && (as • s)"
apply (subst sconc_def [THEN fix_eq2])
by (simp add: cont2cont_LAM)

(* the lazy stream constructor is equivalent to concatenation with a singleton stream *)
lemma lscons_conv: "updis a && s = ↑a • s"
apply (subst sconc_def [THEN fix_eq2])
apply (simp add:  sup'_def)
by (simp add: cont2cont_LAM)

(* concatenation with respect to singleton streams is associative *)
lemma sconc_scons[simp]: "(↑a • as) • s = ↑a • (as • s)"
apply (subst sconc_def [THEN fix_eq2])
by (simp add: sconc_scons' sup'_def cont2cont_LAM)


lemma scases [case_names bottom scons]: "⋀x P. ⟦x = ε⟹P; ⋀a s. x = ↑a • s⟹P⟧⟹P"
apply (rule_tac y=x in scases', simp+)
apply (rule_tac p=u in upE, simp+)
apply (case_tac "xa")
by (auto simp add: sup'_def sconc_scons')

(* Single element streams commute with the stake operation. *)
lemma stake_Suc[simp]: "stake (Suc n)·(↑a • as) = ↑a • stake n·as"
by (simp add: sconc_scons' sup'_def)

(* see also sconc_fst_empty *)
```

```
lemma sconc_snd_empty[simp]: "s • ε = s"
apply (rule stream.take_lemma [OF spec [where x = "s"]])
apply (induct_tac n, simp)
apply (rule allI, simp)
by (rule_tac x=x in scases, simp+)

(* shd is the inverse of prepending a singleton *)
lemma shd1[simp]: "shd (↑a • s) = a"
by (simp add: sconc_scons' shd_def sup'_def)

(* prepending an element a to a stream and extracting it with lshd is equivalent to imposing the
   discrete order on a *)
lemma lshd_updis [simp]: "lshd·(↑a • s) = updis a"
by (metis lscons_conv stream.sel_rews(4))

(* srt is the inverse of appending to a singleton *)
lemma [simp]: "srt·(↑a•as) = as"
by (simp add: sconc_scons' sup'_def)

(* appending to a singleton is monotone *)
lemma [simp]: "↑a ⊑ ↑a • s"
apply (subst sconc_snd_empty [of "↑a", THEN sym])
by (rule monofun_cfun_arg, simp)



(* updis is a bijection *)
lemma updis_eq: "(updis a = updis b) = (a = b)"
by simp

(* the discrete order only considers equal elements to be ordered *)
lemma updis_eq2: "(updis a ⊑ updis b) = (a = b)"
by simp


lemma inject_scons: "↑a • s1 = ↑b • s2⟹a = b ∧ s1 = s2"
apply (subst updis_eq [THEN sym])
apply (rule sinjects' [THEN iffD1], simp)
by (simp add: sconc_scons' sup'_def)

text ‹‹⊑› applied to head and rest›
lemma less_all_sconsD: "↑a • as ⊑ ↑b • bs⟹a = b ∧ as ⊑ bs"
apply (subst updis_eq2 [THEN sym])
apply (rule sinverts' [THEN iffD1], simp)
by (simp add: sconc_scons' sup'_def)

(* appending to a singleton stream can never yield the empty stream *)
lemma [simp]: "ε ≠ ↑a • as"
apply (rule ccontr, simp)
apply (drule po_eq_conv [THEN iffD1])
apply (erule conjE)
by (simp add: sconc_scons' sup'_def)

(* appending to a singleton stream can never yield the empty stream *)
lemma [simp]: "↑a • as ≠ ε"
by (rule notI, drule sym, simp)

text ‹Characterizations of equality with ‹⊑›, head and rest›

(* singleton streams are only in an ordered relation if the two elements are equal *)
lemma [simp]: "(↑a ⊑ ↑b) = (a = b)"
apply (rule iffI)
by (insert less_all_sconsD [of a ε b ε], simp+)

(* length of a stream is smaller than length of this stream concatenated with another stream  *)
lemma [simp]: "#as ⊑ #(as • ys)"
  by (metis minimal monofun_cfun_arg sconc_snd_empty len_stream_def)

(* uparrow is a bijection *)
lemma [simp]: "(↑a = ↑b) = (a = b)"
apply (rule iffI)
by (insert inject_scons [of a ε b ε], simp+)

(* appending a stream x to a singleton stream and producing another singleton stream implies that
   the two singleton streams are equal and x was empty *)
lemma [simp]: "(↑a • x = ↑c) = (a = c ∧ x = ε)"
by (rule iffI, insert inject_scons [of a x c ε], simp+)

(* of course we can also swap the expressions to the left and right of the equality sign *)
lemma [simp]: "(↑c = ↑a • x) = (a = c ∧ x = ε)"
by (rule iffI, insert inject_scons [of c ε a x], simp+)

(* if an appended stream x to a singleton stream is in relation with another singleton stream, this implies that
   a and b are equal and x was empty *)
lemma [simp]: "(↑a • x ⊑ ↑b) = (a = b ∧ x = ε)"
by (rule iffI, insert less_all_sconsD [of a x b ε], simp+)

(* if a singleton stream is the prefix of another stream then the heads of the two streams must match *)
lemma [simp]: "(↑a ⊑ ↑b • x) = (a = b)"
by (rule iffI, insert less_all_sconsD [of a ε b x], simp+)

(* if x isn't empty then concatenating head and rest leaves the stream unchanged *)
lemma surj_scons: "x≠ε⟹↑(shd x) • (srt·x) = x"
by (rule_tac x=x in scases, simp+)
```

```
(* the head of ordered streams are equal *)
lemma below_shd: "x ⊑ y ∧ x ≠ ϵ⟹shd x = shd y"
  by (metis below_bottom_iff less_all_sconsD surj_scons)

(* the head of ordered streams are equal *)
lemma below_shd_alt: "x ⊑ y ∧ x ≠ ϵ⟹shd y = shd x"
  using below_shd by fastforce

text ⟨Characterizations of ⟨⊑⟩ with head and rest⟩

(* any nonempty prefix of a stream y is still a prefix when ignoring the first element *)
lemma less_fst_sconsD: "↑a • as ⊑ y⟹∃ry. y = ↑a • ry ∧ as ⊑ ry"
apply (rule_tac x=y in scases, simp+)
apply (rule_tac x="s" in exI)
by (drule less_all_sconsD, simp)

(* the prefix of any non-empty stream is either empty or shares the same first element *)
lemma less_snd_sconsD:
  "x ⊑ ↑a•as⟹(x = ϵ) ∨ (∃rx. x = ↑a•rx ∧ rx ⊑ as)"
apply (rule_tac x=x in scases, simp+)
apply (rule_tac x="s" in exI)
by (drule less_all_sconsD, simp)

(* semantically equivalent to less_fst_sconsD *)
lemma lessD:
  "x ⊑ y⟹(x = ϵ) ∨ (∃a q w. x = ↑a•q ∧ y = ↑a•w ∧ q ⊑ w)"
apply (rule_tac x=x in scases, simp+)
apply (rule_tac x="a" in exI)
apply (rule_tac x="s" in exI, simp)
by (drule less_fst_sconsD, simp)

(* if ts is a prefix of xs and ts is not bottom, then lshd·ts is equal to lshd·xs *)
lemma lshd_eq: "ts⊑xs⟹ts≠⊥⟹lshd·ts = lshd·xs"
  using lessD by fastforce

(* ------------------------------------------------------------------------ *)
subsection ⟨@{term slen}⟩
(* ------------------------------------------------------------------------ *)

(* the length of the empty stream is zero *)
lemma strict_slen[simp]:"#ϵ = 0"
  by simp

(* prepending a singleton stream increases the length by 1 *)
lemma slen_scons[simp]: "#(↑a•as) = lnsuc·(#as)"
  unfolding len_stream_def
by (subst slen_def [THEN fix_eq2], simp add: lnle_def)

(* the singleton stream has length 1 *)
lemma [simp]: "#(↑a) = Fin (Suc 0)"
apply (subst sconc_snd_empty [of "↑a", THEN sym])
by (subst slen_scons, simp+)


lemma inf_scase:"#s = ∞⟹∃a as. s = ↑a • as ∧ #as = ∞"
by (rule_tac x=s in scases, auto)

(* only the empty stream has length 0 *)
lemma slen_empty_eq[simp]: "(#x = 0) = (x = ϵ)"
by (rule_tac x=x in scases, auto)

text ⟨Appending to an inifite stream does not change its ⟨n⟩th element⟩
lemma sconc_fst_inf_lemma: "∀x. #x=∞⟶stake n·(x•y) = stake n·x"
apply (induct_tac n, auto)
by (rule_tac x=x in scases, auto)


lemma sconc_fst_inf[simp]: "#x=∞⟹x•y = x"
apply (rule stream.take_lemma)
by (rule sconc_fst_inf_lemma [rule_format])


lemma slen_sconc_all_finite:
  "∀x y n. #x = Fin k ∧ #y = Fin n⟶#(x•y) = Fin (k+n)"
apply (induct_tac k, auto)
by (rule_tac x=x in scases, auto)


lemma mono_fst_infD: "⟦#x = ∞; x ⊑ y⟧⟹#(y::'a stream) = ∞ "
  unfolding len_stream_def
apply (drule monofun_cfun_arg [of _ _ slen])
by (rule lnat_po_eq_conv [THEN iffD1], simp)

text ⟨For @{term "s ⊑ t"} with @{term s} and @{term t} of
  equal length, all finite prefixes are identical⟩
lemma stake_eq_slen_eq_and_less:
  "∀s t. #s = #t ∧ s ⊑ t⟶stake n·s = stake n·t"
apply (induct_tac n, auto)
apply (rule_tac x=s in scases, auto)
apply (rule_tac x=t in scases, auto)
by (drule less_all_sconsD, auto)

text ⟨For @{term "s ⊑ t"} with @{term s} and @{term t} of
```

```
  equal length, @{term s} and @{term t} are identical⟩
lemma eq_slen_eq_and_less: "⟦#s = #t; s ⊑ t⟧⟹(s::'a stream) = t"
apply (rule stream.take_lemma)
by (rule stake_eq_slen_eq_and_less [rule_format], rule conjI)


lemma eq_less_and_fst_inf: "⟦s1 ⊑ s2; #s1 = ∞⟧⟹(s1::'a stream) = s2"
apply (rule eq_slen_eq_and_less, simp)
apply (rule sym)
by (rule mono_fst_infD [of "s1" "s2"])

(* if Fin n is smaller than the length of as, then Fin n is also smaller than lnsuc·(#as) *)
lemma [simp]: "Fin n < #as⟹Fin n < lnsuc·(#as)"
  by (smt below_antisym below_trans less_lnsuc lnle_def lnless_def)

text ⟨For infinite streams, ⟨stake n⟩ returns ⟨n⟩ elements⟩
lemma slen_stake_fst_inf[rule_format]:
  "∀x. #x = ∞⟶#(stake n·x) = Fin n"
apply (induct_tac n, auto)
by (rule_tac x=x in scases, auto)

(* mapping a stream to its length is a monotone function *)
lemma mono_slen: "x ⊑ y⟹#x ≤ #y"
  using len_mono lnle_def monofun_def by blast

text ⟨A stream is shorter than ⟨n+1⟩ iff its rest is shorter than ⟨n⟩⟩
lemma slen_rt_ile_eq: "(#x ≤ Fin (Suc n)) = (#(srt·x) ≤ Fin n)"
by (rule_tac x=x in scases, auto)

text ⟨If ⟨#x < #y⟩, this also applies to the streams' rests (for nonempty, finite x)⟩
lemma smono_slen_rt_lemma:
  "#x = Fin k ∧ x ≠ ϵ ∧ #x < #y⟶#(srt·x) < #(srt·y)"
apply (induct_tac k, auto)
apply (rule_tac x=x in scases, auto)
by (rule_tac x=y in scases, auto)

text ⟨If ⟨#x < #y⟩, this also applies to the streams' rests (for finite x)⟩
lemma smono_slen_rt: "⟦x ≠ ϵ; #x < #y⟧⟹#(srt·x) < #(srt·y)"
apply (rule_tac x="#x" in lncases, auto)
by (rule smono_slen_rt_lemma [rule_format], simp)


lemma inf2max: "⟦chain Y; #(Y k) = ∞⟧⟹Y k = (⨆i. Y i)::'a stream)"
apply (subgoal_tac "Y k ⊑ (⨆i. Y i)")
apply (drule eq_less_and_fst_inf, assumption+)
by (rule is_ub_thelub)

text ⟨⟨stake n⟩ returns at most ⟨n⟩ elements⟩
lemma ub_slen_stake[simp]: "#(stake n·x) ≤ Fin n"
apply (rule spec [where x = x])
apply (induct_tac n, auto)
by (rule_tac x=x in scases, auto)

text ⟨⟨stake⟩ always returns finite streams⟩
lemma [simp]: "#(stake n·x) ≠∞"
proof (rule notI)
  assume inf: "#(stake n·x) = ∞"
  have "#(stake n·x) ≤ Fin n" by (rule ub_slen_stake)
  thus False using inf by simp
qed

text ⟨⟨stake⟩ing at least ⟨#x⟩ elements returns ⟨x⟩ again⟩
lemma fin2stake_lemma: "∀x k. #x = Fin k ∧ k≤i⟶stake i·x = x"
apply (induct_tac i, auto)
apply (rule_tac x=x in scases, auto)
by (case_tac "k", auto)

text ⟨⟨stake⟩ing ⟨#x⟩ elements returns ⟨x⟩ again⟩
lemma fin2stake:"#x = Fin n⟹stake n·x = x"
  by (rule fin2stake_lemma [rule_format, of "x" "n" "n"], simp)

text ⟨⟨stake⟩ing only on element from an empty stream is the same as the stream consisting of
      ⟨shd⟩ of the stream⟩
lemma stake2shd:"s≠ϵ⟹stake (Suc 0)·s = ↑(shd s)"
  by(rule scases[of s],simp_all add: Nat.One_nat_def)

lemma stake2shd2:"s≠ϵ⟹stake 1·s = ↑(shd s)"
  by(simp add: Nat.One_nat_def stake2shd)

(* if the stream is not empty, it holds that its length is lnsuc·(#(srt·s)) *)
lemma srt_decrements_length : "s ≠ ϵ⟹#s = lnsuc·(#(srt·s))" by (metis slen_scons surj_scons)

(* the empty stream is the shortest *)
lemma empty_is_shortest : "Fin n < #s⟹s ≠ ϵ" by (metis Fin_0 less_le lnle_Fin_0 strict_slen)

(* if Fin (Suc n) is smaller than length of s, then also Fin n is smaller than length of s *)
lemma convert_inductive_asm : "Fin (Suc n) < #s⟹Fin n < #s" by (metis Fin_leq_Suc_leq less_le not_le)

(* only the empty stream has length zero *)
lemma only_empty_has_length_0 : "#s ≠ 0⟹s ≠ ϵ" by simp


(* ---------------------------------------------------------------------- *)
section ⟨Basic induction rules⟩
```

```
(* ---------------------------------------------------------------------- *)


lemma stakeind:
  "∀x. (P ϵ ∧ (∀a s. P s ⟶ P (↑a • s))) ⟶ P (stake n·x)"
by (induct_tac n, auto, rule_tac x=x in scases, auto)


lemma finind:
  "⟦#x = Fin n; P ϵ; ⋀a s. P s ⟹ P (↑a • s)⟧ ⟹ P x"
apply (drule fin2stake)
apply (drule sym, erule ssubst)
apply (rule stakeind [rule_format])
apply (rule conjI, assumption)
apply (rule allI)+
by (rule impI, simp)


lemma ind:
  "⟦adm P; P ϵ; ⋀a s. P s ⟹ P (↑a • s)⟧ ⟹ P x"
apply (unfold adm_def)
apply (erule_tac x="λi. stake i·x" in allE, auto)
apply (simp add: stakeind)
by (simp add: reach_stream)

lemma finind2: "#s = Fin k ⟹ P ϵ ⟹ (⋀t a. #t < ∞ ⟹ P t ⟹ P (↑a • t)) ⟹ P s"
proof -
  assume "#s = Fin k" and "P ϵ" and "⋀t a. #t < ∞ ⟹ P t ⟹ P (↑a • t)"
  then show "P s"
  proof (induction k arbitrary: s)
    case 0
    then show ?case
      by auto
  next
    case (Suc k)
    then obtain a t where "s = ↑a • t" and "#t = Fin k"
      by (metis Fin_Suc bot_is_0 lnat.con_rews lnat.sel_rews(2) slen_empty_eq srt_decrements_length surj_scons)
    then show ?case
      by (simp add: Suc.IH Suc.prems(2) Suc.prems(3))
  qed
qed

lemma finind3: "#s < ∞ ⟹ P ϵ ⟹ (⋀t a. #t < ∞ ⟹ P t ⟹ P (↑a • t)) ⟹ P s"
  by (metis finind2 less_le ninf2Fin)

(* ---------------------------------------------------------------------- *)
subsection ‹Other properties of @{term stake}›
(* ---------------------------------------------------------------------- *)

text ‹composition of ‹stake››
lemma stakeostake[simp]: "stake k·(stake n·x) = stake (min k n)·x"
apply (rule_tac x="n" in spec)
apply (rule_tac x="k" in spec)
apply (rule ind [of _ x], simp+)
apply (rule allI)+
apply (case_tac "xa", simp+)
by (case_tac "x", simp+)

(* stake always returns a prefix of the input stream *)
lemma ub_stake[simp]: "stake n·x ⊑ x"
by (rule stream.take_below)

(* definition of stake *)
lemma stake_suc: "stake (Suc n)·s = (stake 1·s) • stake n·(srt·s)"
by (metis (no_types, lifting) One_nat_def Rep_cfun_strict1 sconc_snd_empty stake_Suc stream.sel_rews(2)
      stream.take_0 stream.take_strict surj_scons)


(* ---------------------------------------------------------------------- *)
subsection ‹@{term sdrop}›
(* ---------------------------------------------------------------------- *)



(* dropping n·ϵ is the empty stream *)
lemma strict_sdrop[simp]: "sdrop n·ϵ = ϵ"
by (simp add: sdrop_def, induct_tac n, auto)

(* dropping 0·s returns s *)
lemma sdrop_0[simp]: "sdrop 0·s = s"
by (simp add: sdrop_def)

(* dropping an additional element is equivalent to calling srt *)
lemma sdrop_back_rt: "sdrop (Suc n)·s = srt·(sdrop n·s)"
by (simp add: sdrop_def)

(* dropping an additional element is equivalent to sdrop with srt as part of the stream *)
lemma sdrop_forw_rt: "sdrop (Suc n)·s = sdrop n·(srt·s)"
apply (simp add: sdrop_def)
by (subst iterate_Suc2 [THEN sym], simp)

(* dropping n + 1 elements from a non-empty stream is equivalent to dropping n items from the rest *)
lemma sdrop_scons[simp]: "sdrop (Suc n)·(↑a • as) = sdrop n·as"
by (simp add: sdrop_forw_rt)
```

```
(* if dropping n items produces the empty stream then the stream contains n elements or less *)
lemma sdrop_stakel1: "∀s. sdrop n·s = ε⟶stake n·s = s"
apply (induct_tac n, auto)
by (rule_tac x=s in scases, auto)

(* dropping k+x elements is equivalent to dropping x elements first and then k elements *)
lemma sdrop_plus: "sdrop (k+x)·xs = sdrop k·(sdrop x·xs)"
by (simp add: iterate_iterate sdrop_def)


lemma fair_sdrop[rule_format]:
  "∀x. #x = ∞⟶#(sdrop n·x) = ∞"
apply (induct_tac n, simp, clarify)
by (rule_tac x=x in scases, auto)


lemma split_streaml1[simp]:
  "stake n·s • sdrop n·s = s"
apply (rule spec [where x = s])
apply (induct_tac n, auto)
by (rule_tac x=x in scases, auto)


lemma fair_sdrop_rev:
  "#(sdrop k·x) = ∞⟹#x = ∞"
apply (simp add: atomize_imp)
apply (rule_tac x="x" in spec)
apply (induct_tac k, simp)
apply (rule allI, rule impI)
apply (rule_tac x="x" in scases, simp)
by (erule_tac x="s" in allE, simp)

text ⟨construct @{term "sdrop j"} from @{term "sdrop k"} (with @{term "j ≤ k"})⟩
lemma sdropl5:
  "j ≤ k⟹sdrop j·(stake k·x) • sdrop k·x = sdrop j·x"
apply (simp add: atomize_imp)
apply (rule_tac x="j" in spec)
apply (rule_tac x="x" in spec)
apply (induct_tac k, auto)
apply (rule_tac x="x" in scases, auto)
by (case_tac "xa", auto)


lemma sdropl6:
  "#x = Fin k⟹sdrop k·(x • y) = y"
apply (simp add: atomize_imp)
apply (rule_tac x="x" in spec)
apply (rule_tac x="y" in spec)
apply (induct_tac k, auto)
by (rule_tac x="xa" in scases, auto)

(* relation between srt and drop *)
lemma srt_drop : "srt·(sdrop n·s) = sdrop (Suc n)·s" by (simp add: sdrop_back_rt)

(* sdrop n·s should not result in the empty stream *)
lemma drop_not_all : "Fin n < #s⟹sdrop n·s ≠ ε"
proof (induct n)
  show "Fin 0 < #s⟹sdrop 0·s ≠ ε" by auto

  have "⋀ n. Fin n < #s⟹#(sdrop n·s) = lnsuc·(#(srt·(sdrop n·s)))" by (metis not_le sdrop_stakel1
       srt_decrements_length ub_slen_stake)
  hence "⋀ n. Fin n < #s⟹sdrop n·s ≠ ε" using only_empty_has_length_0 by fastforce
  thus "⋀ n. (Fin n < #s⟹sdrop n·s ≠ ε)⟹Fin (Suc n) < #s⟹sdrop (Suc n)·s ≠ ε" by simp
qed


(* ---------------------------------------------------------------------- *)
subsection ⟨@{term snth}⟩
(* ---------------------------------------------------------------------- *)


(* the element k + 1 of the stream s is identical to the element k of the rest of s *)
lemma snth_rt: "snth (Suc k) s = snth k (srt·s)"
apply (simp add: snth_def)
by (subst sdrop_forw_rt,rule refl)

(* semantically equivalent to snth_rt *)
lemma snth_scons[simp]: "snth (Suc k) (↑a • s) = snth k s"
by (simp add: snth_rt)

(* indexing starts at 0, so the 0'th element is equal to the head *)
lemma snth_shd[simp]: "snth 0 s = shd s"
by (simp add: snth_def)


lemma snths_eq_lemma [rule_format]:
  "∀x y. #x = #y ∧ (∀n. Fin n < #x⟶snth n x = snth n y)
         ⟶stake k·x = stake k·y"
apply (induct_tac k, auto)
apply (rule_tac x=x in scases, auto)
apply (rule_tac x=y in scases, auto)
apply (erule_tac x="s" in allE)
```

```
apply (erule_tac x="sa" in allE, auto)
apply (erule_tac x="Suc na" in allE, simp)
by (erule_tac x="0" in allE, auto)


lemma snths_eq:
  "⟦#x = #y; ∀n. Fin n < #x⟶snth n x = snth n y⟧⟹x = y"
apply (rule stream.take_lemma)
by (rule snths_eq_lemma, auto)

(* easy to use rule to show equality on infinite streams *)
(* if two finite streams x, s are identical at every position then x and s are identical *)
lemma sinf_snt2eq: assumes "#s=∞" and "#x=∞" and "⋀i. (snth i s = snth i x)"
  shows "s=x"
by (simp add: assms snths_eq)

lemma snthp_shd: assumes"⋀n.  P(snth n s)"
  shows"P(shd s)"
  by (metis assms snth_shd)

lemma snthp_shd2:  assumes"⋀n.  P(snth n (↑m • s))"
  shows"P(m)"
  by (metis assms shd1 snth_shd)

lemma snthp_snth: assumes"⋀n.  P(snth n (↑m • s))"
      shows"P(snth n(s))"
  by (metis assms snth_scons)

lemma snthp_srt:  assumes"⋀n. P(snth n (s))"
      shows"P(snth n(srt·s))"
  by (metis assms snth_rt)


lemma snth_less: "⟦Fin n < #x; x ⊑ y⟧⟹snth n x = snth n y"
apply (simp add: atomize_imp)
apply (rule_tac x="x" in spec)
apply (rule_tac x="y" in spec)
apply (induct_tac n, auto)
  by (drule lessD, auto)+

(* ----------------------------------------------------------------------- *)
section ⟨Further lemmas⟩
(* ----------------------------------------------------------------------- *)

(* concatenation is associative *)
lemma assoc_sconc[simp]: "(s1•s2)•s3 = s1•s2•s3"
apply (rule_tac x="#s1" in lncases, auto)
by (rule finind [of "s1"], auto)

(* 2 very specific lemmas, used in ⟨stake_add⟩ *)
  lemma stake_conc: "stake i·s • x = stake (Suc i)·s⟹x = stake 1·(sdrop i·s)"
  apply (induction i arbitrary: s)
  apply (simp add: One_nat_def)
  by (smt assoc_sconc inject_scons sdrop_forw_rt stake_Suc stream.take_strict strict_sdrop surj_scons)

  lemma stake_concat:"stake i·s • stake (Suc j)·(sdrop i·s) = stake (Suc i)·s • stake j·(sdrop (Suc i)·s)"
  proof -
    obtain x where x_def: "stake i·s • x = stake (Suc i)·s"
      by (metis (no_types, hide_lams) Suc_n_not_le_n linear min_def split_streaml1 stream.take_take)
    thus ?thesis
      by (smt One_nat_def Rep_cfun_strict1 assoc_sconc sconc_snd_empty sdrop_back_rt stake_Suc stake_conc
           stream.take_0 stream.take_strict strict_sdrop surj_scons)
  qed

(* for arbitrary natural numbers i, j and any streams s the following lemma holds: *)
lemma stake_add: "stake (i+j)·s = (stake i·s) • (stake j·(sdrop i·s))"
apply (induction i arbitrary: j)
apply simp
by (metis add_Suc_shift stake_concat)


lemma inject_sconc: "⟦#x = Fin k; x • y = x • z⟧⟹y = z"
apply (simp add: atomize_imp)
apply (rule_tac x=x in spec)
apply (induct_tac k, auto)
apply (rule_tac x=x in scases, auto)
by (drule inject_scons, auto)

lemma sconc_inj: assumes "#s < ∞"
  shows "inj (Rep_cfun (sconc s))"
  by (meson assms injI inject_sconc lnat_well_h2)

(* x is a prefix of x • y*)
lemma sconc_prefix [simp]: "x ⊑ x • y"
apply (rule_tac x="#x" in lncases, auto)
apply (rule finind [of x], auto)
by (rule monofun_cfun_arg)


lemma slen_sconc_snd_inf: "#y=∞⟹#(x • y) = ∞"
apply (rule_tac x="#x" in lncases, auto)
by (rule finind [of "x"], auto)

(* stake n results in a stream of length n, so sdrop n then results in the empty stream *)
```

```
lemma sdropostake: "sdrop n·(stake n·s) = ϵ"
apply (rule spec [where x = n])
apply (rule ind [of _ s], auto)
by (case_tac x, auto)

(* for all x it holds that if (P ϵ ∧ (∀a s. P s⟶P (s • ↑a))) then it follows that P applied to stake n·x is
     true *)
lemma stakeind2:
  "∀x. (P ϵ ∧ (∀a s. P s⟶P (s • ↑a)))⟶P (stake n·x)"
  apply(induction n)
   apply simp
  apply auto
  apply (subst stake_suc)
  by (metis (no_types, lifting) sconc_snd_empty sdrop_back_rt sdropostake split_streaml1 stake_suc surj_scons)

(* if P is admissible and P holds for the empty stream and ⋀a s. P s implies P (s • ↑a) then P also holds for x *)
lemma ind2: assumes "adm P" and "P ϵ"  and "⋀a s. P s  ⟹P (s • ↑a)"
  shows "P x"
by (metis assms(1) assms(2) assms(3) stakeind2 stream.take_induct)

(* if P holds for bottom and it holds that lscons: "(⋀x xs. x≠⊥⟹ xs⟹P (x&&xs))" and the length of xs is
     finite, then P holds also for xs *)
lemma stream_fin_induct: assumes Bot: "P ⊥" and lscons: "(⋀x xs. x≠⊥⟹ xs⟹P (x&&xs))" and fin: "#xs<∞"
  shows "P xs"
  by (metis finind infl lnless_def sconc_fst_empty sconc_scons' sup'_def up_defined assms)

(* if length of s is finite => P holds for s => P is admissible => P holds for s *)
lemma stream_infs: "(⋀s::'a stream. #s<∞⟹P s)⟹adm P⟹P s"
  by (metis inf_less_eq lel notinfl3 slen_stake_fst_inf stream.take_induct)

lemma slen_stake: "#s ≥ Fin n⟹#(stake n·s) = Fin n"
proof (induction n)
  case 0
  then show ?case
    by simp
next
  case (Suc n)
  assume "#s ≥ Fin (Suc n)"
  then have "#s ≥ Fin n"
    by (simp add: Suc.prems Fin_leq_Suc_leq)
  obtain r where "stake (Suc n)·s = (stake n·s) • r"
    by (metis (no_types) Rep_cfun_strict1 sconc_snd_empty stake_concat stream.take_0)
  then have "r ≠ ϵ"
    by (metis (mono_tags, lifting) Fin_02bot Fin_Suc One_nat_def Suc.prems ‹Fin n ≤ #s› bot_is_0 drop_not_all
        inject_Fin lnle_def lnless_def n_not_Suc_n only_empty_has_length_0 sdropostake slen_scons srt_drop
        stake_Suc stake_conc strictI surj_scons)
  have "#((stake n·s) • r) ≥ Fin (Suc n)"
  proof -
    have f1: "#(stake n·s) = Fin n"
      using Suc.IH ‹Fin n ≤ #s› by fastforce
    have f2: "∀s sa. (sa::'a stream) ⊑ sa • s"
      by simp
    have "∃n. stake n·s • r ≠ stake n·s ∧ Fin n = Fin n"
      using f1 by (metis ‹r ≠ ϵ› inject_sconc sconc_snd_empty)
    then have "#(stake n·s • r) ≠ Fin n"
      by (metis Suc.IH ‹Fin n ≤ #s› ‹r ≠ ϵ› fin2stake sdropl6 sdropostake)
    then show ?thesis
      using f2 f1 by (metis (no_types) less2lnleD lnless_def mono_len lnle_def)
  qed
  then show ?case
    by (metis ‹stake (Suc n)·s = stake n·s • r› dual_order.antisym ub_slen_stake)
qed

(* ----------------------------------------------------------------------- *)
section ‹Additional lemmas for approximation, chains and continuity›
(* ----------------------------------------------------------------------- *)


lemma approxl1:
  "∀s1 s2. s1 ⊑ s2 ∧ #s1 = Fin k⟶stake k·s2 = s1"
apply (induct_tac k, auto)
apply (rule_tac x=s1 in scases, auto)
apply (rule_tac x=s2 in scases, auto)
apply (erule_tac x="s" in allE)
apply (erule_tac x="sa" in allE)
by (drule less_all_sconsD, auto)


lemma approxl2:
  "s1 ⊑ s2⟹(s1 = s2) ∨ (∃n. stake n·s2 = s1 ∧ Fin n = #s1)"
apply (rule_tac x="#s1" in lncases, auto)
apply (rule eq_less_and_fst_inf, assumption+)
by (insert approxl1
  [rule_format, of "s1" "s2"], auto)


lemma inf_chainl1:
  fixes Y::"nat ⇒ 'a stream"
  shows"⟦chain Y; ¬finite_chain Y⟧⟹∃k. #(Y i) = Fin k"
apply (rule ccontr, simp, frule infl)
apply (frule_tac k="i" in inf2max, assumption)
apply (frule_tac i="i" in max_in_chainl3, simp+)
by (simp add: finite_chain_def)
```

```
lemma approxl3: "s1 ⊑ s2⟹∃t. s1•t = s2"
apply (rule_tac x="#s1" in lncases, simp)
apply (drule eq_less_and_fst_inf, simp+)
apply (subst approxl1
  [rule_format, of "s1" "s2", THEN sym], simp+)
by (rule_tac x="sdrop k·s2" in exI, simp)


lemma inf_chainl2:
    fixes Y::"nat ⇒ 'a stream"
    shows "⟦chain Y; ¬ finite_chain Y⟧⟹∃j. Y k ⊑ Y j ∧ #(Y k) < #(Y j)"
apply (auto simp add: finite_chain_def max_in_chain_def)
apply (erule_tac x="k" in allE, auto)
apply (frule_tac i=k and j=j in chain_mono, assumption)
apply (rule_tac x="j" in exI, simp)
apply (auto simp add: lnless_def)
apply (rule mono_slen, assumption)
by (frule eq_slen_eq_and_less, simp+)


lemma inf_chainl3:
    fixes Y::"nat ⇒ 'a stream"
  shows "chain Y ∧ ¬finite_chain Y⟶(∃k. Fin n ≤ #(Y k))"
apply (induct_tac n, auto)
apply (case_tac "Fin n = #(Y k)")
apply (frule_tac k=k in inf_chainl2, auto)
apply (rule_tac x="j" in exI)
apply (drule sym)
apply (rule_tac x="#(Y j)" in lncases, auto)
apply (rule_tac x="k" in exI)
by (rule_tac x="#(Y k)" in lncases, auto)


lemma inf_chainl4:
    fixes Y::"nat ⇒ 'a stream"
  shows "⟦chain Y; ¬finite_chain Y⟧⟹#(lub(range Y)) = ∞"
apply (rule_tac x="#(Lub Y)" in lncases, auto)
apply (frule_tac n = "Suc k" in inf_chainl3
  [rule_format, OF conjI], assumption, erule exE)
apply (subgoal_tac "Y ka ⊑ (Lub Y)")
apply (drule mono_len2, simp)
apply (frule_tac x = "Fin (Suc k)" and
  y = "#(Y ka)" and z = "Fin k" in trans_lnle, simp+)
by (rule is_ub_thelub)


lemma finite_chain_stake:
  "chain Y⟹finite_chain (λi. stake n·(Y i))"
apply (frule ch2ch_Rep_cfunR [of _ "stake n"])
apply (rule ccontr)
apply (frule inf_chainl4 [of "λi. stake n·(Y i)"],assumption)
by (simp add: contlub_cfun_arg [THEN sym])


lemma lub_approx:
  "chain Y⟹∃k. stake n·(lub (range Y)) = stake n·(Y k)"
apply (subst contlub_cfun_arg, assumption)
apply (frule finite_chain_stake [of _ n])
apply (simp add: finite_chain_def, auto)
apply (rule_tac x="i" in exI)
by (rule lub_finch1
  [THEN lub_eqI, of "λi. stake n·(Y i)"], auto)


lemma pr_contI:
  "⟦monofun f; ∀x.∃n. (f x) = f (stake n·x)⟧⟹cont f"
apply (rule contI2, assumption)
apply (rule allI, rule impI)
apply (erule_tac x="lub (range Y)" in allE, erule exE)
apply (frule_tac n = n in lub_approx, erule exE)
apply (subgoal_tac "f (stake n·(Y k)) ⊑ f (Y k)")
apply (subgoal_tac "f (Y k) ⊑ (⨆i. f (Y i))")
apply (drule_tac x="f (stake n·(Y k))" and
  y="f (Y k)" and z = "⨆i. f (Y i)" in below_trans)
apply (rule is_ub_thelub)
apply (rule_tac f=f in ch2ch_monofun, assumption+)
apply (clarsimp)
apply (rule is_ub_thelub)
apply (rule_tac f=f in ch2ch_monofun, assumption+)
by (rule_tac f = f in monofunE, simp+)

text ⟨For continuous functions, each finite prefix of @{term "f·x"} only
  depends on a finite prefix of @{term "x"}⟩
lemma fun_approxl1:
  "∃j. stake k·(f·x) = stake k·(f·(stake j·x))"
apply (subgoal_tac "f·x = (⨆i. f·(stake i·x))")
apply (erule ssubst)
apply (rule lub_approx)
apply (rule chain_monofun)
apply (rule ch2ch_Rep_cfunL)
apply (rule stream.chain_take)
apply (subst contlub_cfun_arg [THEN sym])
apply (rule ch2ch_Rep_cfunL)
```

```
apply (rule stream.chain_take)
apply (subst reach_stream)
by (rule refl)


lemma fun_approxl2: "slen·(f·x) = Fin k⟹∃j. f·x = f·(stake j·x)"
apply (insert fun_approxl1 [of k "f" x], auto)
  apply (rule_tac x="j" in exI)
  unfolding len_stream_def[symmetric]
apply (frule fin2stake [THEN sym], simp)
apply (rule stream.take_lemma, simp)
apply (case_tac "n ≤ k")
apply (simp add: min_def)+
apply (rule po_eq_conv [THEN iffD2])
apply (rule conjI)
apply (rule monofun_cfun_fun)
apply (rule chain_mono)
apply (rule stream.chain_take, simp+)
apply (subgoal_tac "f·(stake j·x) ⊑ f·x")
apply (rule below_trans, auto)
apply (drule sym, drule sym, simp)
by (rule monofun_cfun_arg, simp)

(* if two streams are unequal, it holds for a finite stream a that a • s1 is unequal to a • s2 *)
lemma sconc_neq_h: assumes "s1 ≠ s2"
  shows "#a < ∞⟶a • s1 ≠ a • s2"
  apply(rule ind [of _a ])
    apply(rule admI)
    apply (rule impI)
    apply (metis inf_chainl4 l42 neq_iff)
   apply (simp add: assms)
  by (metis inf_ub inject_scons less_le sconc_scons slen_sconc_snd_inf)

(* if two streams are unequal and a stream a has finite lenght, it holds that a • s1 is unequal to a • s2 *)
lemma sconc_neq: assumes "s1 ≠ s2" and "#a < ∞"
  shows "a • s1 ≠ a • s2"
using assms(1) assms(2) sconc_neq_h by blast

lemma stake_prefix: "#s < ∞⟹t ≠ ε⟹s = t • u⟹∃k. t = stake (Suc k)·s"
proof -
  assume "#s < ∞" and "t ≠ ε" and "s = t • u"
  then obtain k where "t = stake k·s"
    by (metis approxl2 fin2stake inf_less_eq minimal monofun_cfun_arg ninf2Fin not_le sconc_snd_empty)
  then obtain l where "k = Suc l"
    by (metis Rep_cfun_strict1 ‹t ≠ ε› not0_implies_Suc stream.take_0)
  thus ?thesis
    using ‹t = stake k·s› by blast
qed

lemma stake_prefix2: "#s = Fin n⟹s = stake n·(s • t)"
  by (metis approxl1 minimal monofun_cfun_arg sconc_snd_empty)

lemma slen_conc: "#s < ∞⟹t ≠ ε⟹#s ≥ Fin n⟹#(s • t) > Fin n"
  by (metis (no_types, hide_lams) stake_prefix2 infI less_le less_le_trans mono_slen sconc_neq sconc_snd_empty
      stream.take_below)

lemma stake_srt_conc [simp]: "srt·((stake 1·s)•(s)) = s"
  apply (cases s)
  apply simp
  by (metis One_nat_def Rep_cfun_strict1 lscons_conv sconc_snd_empty stake_Suc stream.con_rews(2)
      stream.sel_rews(5) stream.take_0 surj_scons)

(* ----------------------------------------------------------------------- *)
section ‹Lemmas for the remaining definitions›
(* ----------------------------------------------------------------------- *)

(* ----------------------------------------------------------------------- *)
subsection ‹@{term slookahd}›
(* ----------------------------------------------------------------------- *)


lemma cont_slookahd[simp]: "cont (λ s. if s=ε then ⊥ else eq (shd s))"
apply (rule pr_contI)
apply (rule monofunI, auto)
apply (rule_tac x=x in scases, auto)
apply (rule_tac x=y in scases, auto)
apply (drule less_all_sconsD, simp)
apply (rule_tac x=x in scases, auto)
by (rule_tac x="Suc 0" in exI, auto)

(* slookahd applied to the empty stream results in the bottom element for any function eq *)
lemma strict_slookahd[simp]: "slookahd·ε·eq = ⊥"
by (simp add: slookahd_def cont2cont_LAM)

(* if s isn't the empty stream, the function eq will be applied to the head of s *)
lemma slookahd_scons[simp]: "s≠ε⟹slookahd·s·eq = eq (shd s)"
by (simp add: slookahd_def cont2cont_LAM)

(* the constant function that always returns the empty stream unifies the two cases of slookahd *)
lemma strict2_slookahd[simp]: "slookahd·xs·(λy. ε) = ε"
by (cases xs, simp_all)


(* ----------------------------------------------------------------------- *)
```

```
subsection ‹@{term sinftimes}›
(* ------------------------------------------------------------------------ *)


(* repeating the empty stream produces the empty stream again for any n*)
lemma sntimes_eps[simp]: "sntimes n ε = ε"
by (induct_tac n, simp+)

(* after repeating the stream ↑s n-times the head is s *)
  (* n>0 otherwise ⟨0 ⋆ ↑s = ε⟩ *)
lemma shd_sntime [simp]: assumes "n>0" shows "shd (n ⋆ ↑s) = s"
by (metis assms gr0_implies_Suc shd1 sntimes.simps(2))

(* infinitely cycling the empty stream produces the empty stream again *)
lemma strict_icycle[simp]: "sinftimes ε = ε"
by (subst sinftimes_def [THEN fix_eq2], auto)

(* repeating a stream infinitely often is equivalent to repeating it once and then infinitely often *)
lemma sinftimes_unfold: "sinftimes s = s • sinftimes s"
by (subst sinftimes_def [THEN fix_eq2], auto)


lemma slen_sinftimes: "s ≠ ε⟹#(sinftimes s) = ∞"
apply (rule ccontr)
apply (rule_tac x="#(sinftimes s)" in lncases, auto)
apply (rule_tac x="#s" in lncases)
apply (insert sinftimes_unfold [of s], auto)
by (insert slen_sconc_all_finite
  [rule_format, of "s" _ "sinftimes s"], force)

(* lenght of sinftimes of (↑a) is infinity *)
lemma [simp]: "#(sinftimes (↑a)) = ∞"
by (simp add: slen_sinftimes)

(* converting the element x to a singleton stream, repeating the singleton and re-extracting x with
   lshd is equivalent to imposing the discrete order on x *)
lemma lshd_sinf [simp]: "lshd·↑x^∞ = updis x"
by (metis lshd_updis sinftimes_unfold)

(* the infinite repetition of the stream x has the same head as x *)
lemma shd_sinf[simp]: "shd (x^∞) = shd x"
by (metis assoc_sconc shd1 sinftimes_unfold strict_icycle surj_scons)

(* srt has no effect on an infinite constant stream of x *)
lemma srt_sinf [simp]: "srt·↑x^∞ = ((↑x)^∞)"
by (metis lscons_conv sinftimes_unfold stream.sel_rews(5) up_defined)

(* if the stream x contains y elements then the first y elements of the infinite repetition of x will
   be x again *)
lemma stake_y [simp]: assumes "#x = Fin y"
    shows "stake y·(sinftimes x) = x"
by (metis approxl1 assms minimal monofun_cfun_arg sconc_snd_empty sinftimes_unfold)

(* the infinite repetitions of the singleton stream ↑s consists only of the element s *)
lemma snth_sinftimes[simp]: "snth i ((↑s)^∞) = s"
apply (induction i)
apply (simp)
by (simp add: snth_rt)

(* dropping any finite number of elements from an infinite constant stream doesn't affect the stream *)
lemma sdrops_sinf[simp]: "sdrop i·((↑x)^∞) = ((↑x)^∞)"
apply (induction i)
apply(simp)
by (simp add: sdrop_forw_rt)

(* for a finite natural number "i", following relation between sntimes and stake holds:  *)
lemma sntimes_stake: "i ⋆ ↑x = stake i·((↑x)^∞)"
apply(induction i)
apply simp
by (metis sinftimes_unfold sntimes.simps(2) stake_Suc)

(* for every finite number "i" is sntimes ≠ sinftimes. *)
lemma snNEqSinf [simp]: "i ⋆ ↑x ≠ ((↑x)^∞)"
by (metis lshd_sinf sdropostake sdrops_sinf sntimes_stake stream.sel_rews(3) up_defined)

(* for every natural number i, dropping the first (i*y) elements results in the same infinite stream *)
  (* the first i "blocks" of x are dropped *)
lemma sdrop_sinf[simp]: assumes "Fin y = #x"
  shows "sdrop (i * y)·(sinftimes x) = sinftimes x"
apply(induction i)
apply(simp)
by (metis assms mult_Suc sdrop_plus sdropl6 sinftimes_unfold)

(* repeating the empty stream again produces the empty stream *)
lemma sinf_notEps[simp]: assumes "xs ≠ ε" shows "(sinftimes xs) ≠ ε"
using assms slen_sinftimes by fastforce

(* sinftimes has no effect on streams that are already infinite *)
(* removed simp because of lemma stakewhile_sinftimes_lemma*)
lemma sinf_inf: assumes "#s = ∞"
  shows "s^∞ = s"
by (metis assms sconc_fst_inf sinftimes_unfold)
```

```
(* sinftimes is idempotent *)
lemma sinf_dupE [simp]: "(sinftimes s)^∞ = (s^∞)"
using sinf_inf slen_sinftimes by force

(* alternative unfold rule for sntimes, new element is appended on the end *)
lemma sntimes_Suc2: "(Suc i) ⋆ s = (i⋆s) • s"
apply (induction i)
apply simp
by (metis assoc_sconc sntimes.simps(2))

(* Blockwise stake from sinftimes to sntimes. *)
lemma sinf2sntimes: assumes "Fin y = #x"
  shows "stake (i*y)·(x^∞) = i⋆x"
apply(induction i)
apply simp
by (metis assms mult_Suc sdrop_plus sdrop_sinf sntimes.simps(2) stake_add stake_y)

(* for any natural number i, sntimes is a prefix of sinftimes *)
lemma snT_le_sinfT [simp]: "i⋆s ⊑ (s^∞)"
by (metis minimal monofun_cfun_arg ninf2Fin sconc_fst_inf sconc_snd_empty sinf2sntimes sinf_inf sntimes.simps(2)
     sntimes_Suc2 ub_stake)

(* repeating the stream s i times produces a prefix of repeating s i+1 times *)
lemma sntimes_leq: "i⋆s ⊑ (Suc i)⋆s"
by (metis minimal monofun_cfun_arg sconc_snd_empty sntimes_Suc2)

(* the repetitions of a stream constitute a chain *)
lemma sntimes_chain: "chain (λi. i⋆s)"
by (meson po_class.chainI sntimes_leq)

(* xs is an infinite repetition of the finite stream x. Then dropping any fixed number i of repetitions
   of x leaves xs unchanged. *)
lemma sdrop_sinf2: assumes "xs = x•xs" and "#x = Fin y"
  shows "sdrop (y*i)·xs = xs"
apply (induction i)
apply simp
by (metis assms mult_Suc_right sdrop_plus sdropl6)

(* the recursive definition for a stream (xs = x•xs) is identical to the infinite repetition of x at
   every multiple of the length of x *)
lemma stake_eq_sinf: assumes "xs = x•xs" and "#x = Fin y"
  shows "stake (i*y)·xs = stake (i*y)·(sinftimes x)"
proof (induction i)
  case 0 thus ?case by simp
next
  case (Suc i)
  have drop_xs:"sdrop (i*y)·xs = xs" by (metis assms mult.commute sdrop_sinf2)

  have "stake (Suc i * y)·xs  =  stake (i*y)·xs • stake y·(sdrop (i*y)·xs)" by (metis add.commute mult_Suc
       stake_add)
  hence eq1:"stake (Suc i * y)·xs =  stake (i*y)·xs • x"  by (metis approxl1 assms drop_xs minimal
       monofun_cfun_arg sconc_snd_empty)

  have "stake (Suc i * y)·(sinftimes x) = stake (i*y)·(sinftimes x) • stake y·(sdrop (i*y)·(sinftimes x))"
    by (metis add.commute mult_Suc stake_add)
  hence eq2:"stake (Suc i * y)·(sinftimes x) =  stake (i*y)·(sinftimes x) • x" by (simp add: assms(2))

  thus ?case using Suc.IH eq1 by auto
qed

(* when repeating a stream s a different number of times, one of the repetitions will be a prefix of
   the other *)
lemma stake_sntimes2sntimes: assumes "j≤k" and "#s = Fin y"
  shows "stake (j*y)·(k⋆s) = j⋆s"
by (smt assms(1) assms(2) min_def mult_le_mono1 sinf2sntimes stakeostake)

(* for a stream s, a natural y and an arbitrary natural j, apply blockwise stake sntimes. *)
lemma lubStake2sn: assumes "#s = Fin y"
  shows "(⨆ i. stake (y*j)·(i⋆s)) = j⋆s" (is "(⨆i. ?c i) = _")
proof -
  have "max_in_chain j (λi. ?c i)" by (simp add: assms max_in_chainI mult.commute stake_sntimes2sntimes)
  thus ?thesis by (simp add: assms maxinch_is_thelub mult.commute sntimes_chain stake_sntimes2sntimes)
qed

(* building block of the lemma sntimesLub_Fin *)
lemma sntimesChain: assumes "#s = Fin y" and "y ≠ 0"
  shows "∀j. stake (y*j)·(⨆ i. i⋆s) = stake (y*j)· (s^∞)"
by (metis assms(1) contlub_cfun_arg lubStake2sn mult.commute sinf2sntimes sntimes_chain)

(* proof for lemma sntimesLub_Fin *)
lemma sntimesLub_Fin: assumes "#s = Fin y" and "y ≠ 0"
  shows "(⨆ i. i⋆s) = (s^∞)"
proof -
  have  "∀j. stake (j*y)·(⨆ i. i⋆s) = stake (j*y)· (s^∞)" by (metis assms(1) assms(2) mult.commute sntimesChain)
  hence "∀j. stake j·(⨆ i. i⋆s) = stake j· (s^∞)" using assms by (metis gstake2stake)
  thus ?thesis by (simp add:  stream.take_lemma)
qed

(* for any stream s the LUB of sntimes is sinftimes *)
lemma sntimesLub[simp]:"(⨆ i. i⋆s) = (s^∞)"
apply(cases "#s = ∞")
apply (metis inf2max sconc_fst_inf sinf_inf sntimes.simps(2) sntimes_chain)
```

```
by (metis Fin_0 Incases lub_eq_bottom_iff slen_empty_eq sntimesLub_Fin sntimes_chain sntimes_eps strict_icycle)

(* shows that any recursive definition with the following form is equal to sinftimes *)
lemma rek2sinftimes: assumes "xs = x • xs" and "x≠ϵ"
  shows "xs = sinftimes x"
proof (cases "#x = ∞")
  case True thus ?thesis by (metis assms(1) sconc_fst_inf sinftimes_unfold)
next
  case False
  obtain y where y_def: "Fin y = #x ∧ y≠0"  by (metis False Fin_02bot assms(2) infI lnzero_def slen_empty_eq)
  hence "∀i. stake (i*y)·xs = stake (i*y)·(x^∞)" using assms(1) stake_eq_sinf by fastforce
  hence "∀i. stake i·xs = stake i·(x^∞)" using gstake2stake y_def by blast
  thus ?thesis by (simp add: stream.take_lemma)
qed

(* specializes the result from rek2sinftimes to singleton streams *)
lemma s2sinftimes: assumes "xs = ↑x • xs"
  shows "xs = ((↑x)^∞)"
using assms rek2sinftimes by fastforce

(* shows that the infinite repetition of a stream x is the least fixed point of iterating (Λ s. x • s),
   which maps streams to streams *)
lemma fix2sinf[simp]: "fix·(Λ s. x • s) = (x^∞)"
  by (metis eta_cfun fix_eq fix_strict rek2sinftimes sconc_snd_empty strict_icycle)

lemma snth_bool_sinftimes: "snth (Suc n) ((↑bool • ↑(¬ bool))^∞) = (¬ snth n ((↑bool • ↑(¬bool))^∞))"
  apply (induction n)
  apply (metis assoc_sconc shd1 sinftimes_unfold snth_scons snth_shd)
  by (metis (no_types, hide_lams) sconc_fst_empty sconc_scons' sinftimes_unfold snth_scons sup'_def)

lemma sinftimes_srt: "srt·((↑a • ↑b)^∞) = ((↑b • ↑a)^∞)"
  apply (subst sinftimes_unfold, simp)
  by (metis (no_types, lifting) assoc_sconc rek2sinftimes sinftimes_unfold strictI)

lemma sinftimes_snth:"(n mod 2 = 0⟶snth n ((↑a • ↑b)^∞) = a) ∧ (n mod 2 = 1⟶snth n ((↑a • ↑b)^∞) = b)"
proof(induction n arbitrary: a b)
  case 0
  then show ?case
    by simp
next
  case (Suc n)
  moreover obtain m where m_def:"n≠0⟹n = Suc m"
    using not0_implies_Suc by auto
  ultimately show ?case
    apply(cases "n = 0")
    apply(subst sinftimes_unfold, simp)
    apply (metis sconc_scons shd1 sinftimes_unfold snth_scons snth_shd)
    apply auto
    apply(subst sinftimes_unfold, simp)
    apply (simp add: snth_rt)
    apply (metis One_nat_def even_Suc parity_cases sinftimes_srt)
    apply(subst sinftimes_unfold, simp)
    apply (simp add: snth_rt)
    by (metis One_nat_def even_Suc parity_cases sinftimes_srt)
qed


(* ----------------------------------------------------------------------- *)
subsection ‹@{term smap}›
(* ----------------------------------------------------------------------- *)

(* smapping a function to the empty stream gives us the empty stream *)
lemma strict_smap[simp]: "smap f·ϵ = ϵ"
by (subst smap_def [THEN fix_eq2], simp)

(* smap distributes over concatenation *)
lemma smap_scons[simp]: "smap f·(↑a • s) = ↑(f a) • smap f·s"
by (subst smap_def [THEN fix_eq2], simp)

(* if ⋀a as. f·(↑a • as) is equal to ↑(g a) • f·as and f applied to bottom returns the bottom element, then
 f applied to s is the same as applying smap to g·s *)
lemma rek2smap: assumes "⋀a as. f·(↑a • as) = ↑(g a) • f·as"
  and "f·⊥ = ⊥"
  shows "f·s = smap g·s"
apply(rule ind [of _s])
by(simp_all add: assms)

(* mapping f over a singleton stream is equivalent to applying f to the only element in the stream *)
lemma [simp]: "smap f·(↑a) = ↑(f a)"
by (subst smap_def [THEN fix_eq2], simp)

(* smap leaves the length of a stream unchanged *)
lemma slen_smap[simp]: "#(smap f·x) = #x"
  apply (rule ind [of _ x], auto)
  unfolding len_stream_def
  by simp


lemma smap_snth_lemma:
  "Fin n < #s⟹snth n (smap f·s) = f (snth n s)"
apply (simp add: atomize_imp)
apply (rule_tac x="s" in spec)
apply (induct_tac n, simp+)
```

```
by (rule allI, rule_tac x="x" in scases, simp+)+

text⟨Doing smap in two passes, applying h in the first pass and g in the second is
equivalent to applying g ∘ h in a single pass⟩
lemma smaps2smap: "smap g·(smap h·xs) =  smap (λ x. g (h x))·xs"
by (simp add: smap_snth_lemma snths_eq)


lemma sdrop_smap[simp]: "sdrop k·(smap f·s) = smap f·(sdrop k·s)"
apply (rule_tac x="k" in spec)
apply (rule ind [of _ s], simp+)
apply (rule allI)
by (case_tac "x", simp+)


lemma smap_split: "smap f·(a • b) = (smap f·a) • (smap f·b)"
proof (rule lncases [of "#a"], simp)
  fix k assume "#a = Fin k"
  thus ?thesis by (rule finind [of "a"], simp_all)
qed

(* smap distributes over infinite repetition *)
lemma smap2sinf[simp]: "smap f·(x^∞)= ((smap f·x)^∞)"
by (metis (no_types) rek2sinftimes sinftimes_unfold slen_empty_eq slen_smap smap_split strict_icycle)

(* smap and infinity *)
lemma l5: "smap g·((↑x)^∞) = ((↑(g x))^∞)"
  by simp

(* for any nonempty stream it holds that smap f to stream s is ↑(f (shd s)) • smap f·(srt·s) *)
lemma smap_hd_rst : "s ≠ ε⟹smap f·s = ↑(f (shd s)) • smap f·(srt·s)" by (metis smap_scons surj_scons)



lemma smap_inj:"inj f⟹inj (Rep_cfun (smap f))"
  apply(rule injI)
  apply(rule snths_eq,auto)
  apply (metis slen_smap)
  by (metis inj_eq slen_smap smap_snth_lemma)

lemma smapnoteq:assumes "⋀x y. x ≠ y⟹f x ≠ f y"
  shows"x ≠ y⟹smap f·x ≠ smap f·y"
  apply(cases "#x ≠ #y",auto)
proof(metis slen_smap)
  assume a1:"x ≠ y"
  assume a2:"#x = #y"
  assume a3:" smap f·x = smap f·y"
  obtain n where n_def:"Fin n < #x ∧snth n x ≠ snth n y"
    using a1 a2 snths_eq dual_order.strict_implies_order by blast
  then have "snth n (smap f·x) ≠ snth n (smap f·y)"
    apply(subst smap_snth_lemma,simp add: n_def)
    by (simp add: a2 assms smap_snth_lemma)
  thus "False"
    by (simp add: a3)
qed

lemma smapnotbelow:assumes "⋀x y. x ≠ y⟹f x ≠ f y"
  shows"¬x ⊑ y⟹¬smap f·x ⊑ smap f·y"
  apply(cases "#x = #y",auto)
proof-
  assume a1:"¬x ⊑ y"
  assume a2:"#x = #y"
  assume a3:" smap f·x ⊑ smap f·y"
  obtain n where n_def:"Fin n < #x ∧snth n x ≠ snth n y"
    using a1 a2 snths_eq dual_order.strict_implies_order by blast
  then have "snth n (smap f·x) ≠ snth n (smap f·y)"
    apply(subst smap_snth_lemma,simp add: n_def)
    by (simp add: a2 assms smap_snth_lemma)
  thus "False"
    by (metis a2 a3 eq_slen_eq_and_less slen_smap)
next
  assume a1:"¬x ⊑ y"
  assume a2:"#x ≠ #y"
  assume a3:" smap f·x ⊑ smap f·y"
  show False
  proof(cases "#x < #y")
    case True
    obtain n where n_def:"Fin n < #x ∧snth n x ≠ snth n y"
      by (smt a1 a3 approxl2 mono_slen po_eq_conv slen_smap
             slen_stake snth_less snths_eq stream.take_below)
    then have "snth n (smap f·x) ≠ snth n (smap f·y)"
      apply(subst smap_snth_lemma,simp add: n_def)
      by (metis True assms smap_snth_lemma trans_lnless)
    then show ?thesis
      by (metis a3 n_def slen_smap snth_less)
  next
    case False
    then show ?thesis
      by (metis False a2 a3 lnless_def mono_len2 slen_smap)
  qed
qed
```

```
(* ---------------------------------------------------------------------- *)
subsection ‹@{term sprojfst} and @{term sprojsnd}›
(* ---------------------------------------------------------------------- *)


(* sprojfst extracts the first element of the first tuple in any non-empty stream of tuples *)
lemma sprojfst_scons[simp]: "sprojfst·(↑(x, y) • s) = ↑x • sprojfst·s"
by (unfold sprojfst_def, simp)

(* the empty stream is a fixed point of sprojfst *)
lemma strict_sprojfst[simp]: "sprojfst·ϵ = ϵ"
by (unfold sprojfst_def, simp)

(* sprojfst extracts the first element of any singleton tuple-stream *)
lemma [simp]: "sprojfst·(↑(a,b)) = ↑a"
by (simp add: sprojfst_def)


(* sprojsnd extracts the second element of the first tuple in any non-empty stream of tuples *)
lemma sprojsnd_scons[simp]: "sprojsnd·(↑(x,y) • s) = ↑y • sprojsnd·s"
by (unfold sprojsnd_def, simp)

(* the empty stream is a fixed point of sprojsnd *)
lemma strict_sprojsnd[simp]: "sprojsnd·ϵ = ϵ"
by (unfold sprojsnd_def, simp)

(* sprojsnd extracts the second element of any singleton tuple-stream *)
lemma [simp]: "sprojsnd·(↑(a,b)) = ↑b"
  by (simp add: sprojsnd_def)

lemma sprojsnd_shd:
  assumes "s ≠ ϵ"
  shows "shd (sprojsnd·s) = snd (shd s)"
  by (metis assms prod.collapse shd1 sprojsnd_scons surj_scons)

lemma sconc_sprojsnd_shd:
  shows "shd (sprojsnd·(↑a • s)) = snd a"
  by (simp add: sprojsnd_shd)


(* commutativity of sprojsnd and srt *)
lemma rt_Sproj_2_eq: "sprojsnd·(srt·x) = srt·(sprojsnd·x)"
by (rule ind [of _ x], auto)

(* commutativity of sprojsnd and srt *)
lemma rt_Sproj_1_eq: "sprojfst·(srt·x) = srt·(sprojfst·x)"
by (rule ind [of _ x], auto)


(* relation between sprojsnd and sprojfst with respect to the length operator *)
lemma slen_sprojs_eq: "#(sprojsnd·x) = #(sprojfst·x)"
by (rule ind [of _ "x"], auto, simp add: len_stream_def)

(* if sprojfst·x is the empty stream, then x was already empty *)
lemma strict_rev_sprojfst: "sprojfst·x = ϵ⟹x = ϵ"
by (rule ccontr, rule_tac x=x in scases, auto)

(* if sprojsnd·x is the empty stream, then x was already empty *)
lemma strict_rev_sprojsnd: "sprojsnd·x = ϵ⟹x = ϵ"
by (rule ccontr, rule_tac x=x in scases, auto)

(* sprojfst does not change the length of x *)
lemma slen_sprojfst: "#(sprojfst·x) = #x"
by (rule ind [of _ "x"], auto, simp add: len_stream_def)

(* sprojsnd does not change the length of x *)
lemma slen_sprojsnd: "#(sprojsnd·x) = #x"
by (rule ind [of _ "x"], auto, simp add: len_stream_def)

(* updis does not change the length *)
lemma slen_updis_eq: "#s1 = #s2⟹#(updis x1 && s1) = #(updis x2 && s2)"
  by (simp add: lscons_conv)

(* helper lemma for deconstruct_infstream *)
lemma deconstruct_infstream_h:
  assumes "#s = ∞" obtains x xs where "(updis x) && xs = s ∧ #xs = ∞"
  using assms inf_scase lscons_conv by blast

(* deconstruction of infinite streams *)
lemma deconstruct_infstream:
  assumes "#s = ∞" obtains x xs where "(updis x) && xs = s ∧ #xs = ∞ ∧ xs ≠ ϵ"
  by (metis Inf'_neq_0 assms deconstruct_infstream_h slen_empty_eq)


lemma sprojfst_shd[simp]: assumes "s≠ϵ" shows "shd (sprojfst·s) = fst (shd s)"
  by (metis assms prod.collapse shd1 sprojfst_scons surj_scons)

lemma sprojfst_snth[simp]: assumes "Fin n < #s" shows "snth n (sprojfst·s) = fst (snth n s)"
  using assms
```

```
proof(induction n arbitrary: s)
  case 0
  then show ?case
    apply simp
    apply(rule sprojfst_shd)
    by auto
next
  case (Suc n)
  then show ?case
    apply(simp add: snth_rt)
    by (metis not_less rt_Sproj_1_eq slen_rt_ile_eq)
qed

lemma sprojfst_shd2[simp]: "shd (sprojfst·(↑a • s)) = fst (a)"
  by simp

lemma sprojsnd_snth:
  assumes "Fin n < #s"
  shows "snth n (sprojsnd·s) = snd (snth n s)"
  using assms
  apply (induction n arbitrary: s)
  using sprojsnd_shd apply force
  by (metis leD lel rt_Sproj_2_eq slen_rt_ile_eq snth_rt)

lemma sprojsnd_shd2[simp]: "shd (sprojsnd·(↑a • s)) = snd (a)"
  by (simp add: sconc_sprojsnd_shd)

(* ----------------------------------------------------------------------- *)
subsection ‹@{term sfilter}›
(* ----------------------------------------------------------------------- *)


(* note that M is a set, not a predicate *)
lemma strict_sfilter[simp]: "sfilter M·ε = ε"
by (subst sfilter_def [THEN fix_eq2], simp)

(* if the head of a stream is in M, then sfilter will keep the head *)
lemma sfilter_in[simp]:
  "a ∈ M⟹sfilter M·(↑a • s) = ↑a • sfilter M·s"
by (subst sfilter_def [THEN fix_eq2], simp)

(* if the head of a stream isn't in M, then sfilter will discard the head *)
lemma sfilter_nin[simp]:
  "a ∉ M⟹sfilter M·(↑a • s) = sfilter M·s"
by (subst sfilter_def [THEN fix_eq2], simp)

(* if the sole element in a singleton stream is in M then sfilter is a no-op *)
lemma [simp]: "a ∈ M⟹sfilter M·(↑a) = ↑a"
by (subst sfilter_def [THEN fix_eq2], simp)

(* if the sole element in a singleton stream is not in M then sfilter produces the empty stream *)
lemma [simp]: "a ∉ M⟹sfilter M·(↑a) = ε"
by (subst sfilter_def [THEN fix_eq2], simp)



(* filtering all elements that aren't in {a} from a stream consisting only of the element a has no effect *)
lemma sfilter_sinftimes_in[simp]:
  "sfilter {a}·(sinftimes (↑a)) = sinftimes (↑a)"
apply (rule stream.take_lemma)
apply (induct_tac n, auto)
apply (subst sinftimes_unfold, simp)
apply (rule sym)
by (subst sinftimes_unfold, simp)

(* if the element a isn't in the set F then filtering a stream of infinitely many a's using F will
   produce the empty stream *)
lemma sfilter_sinftimes_nin:
  "a ∉ F⟹(F ⊖ (sinftimes (↑a))) = ε"
proof -
  assume a_nin_F: "a ∉ F"
  have "⋀i. (F ⊖ (stake i·(sinftimes (↑a)))) = ε"
  proof (induct_tac i, simp_all)
    fix n assume "F ⊖ (stake n·(sinftimes (↑a))) = ε"
    hence "F ⊖ (stake (Suc n)·(↑a • sinftimes (↑a))) = ε" using a_nin_F by simp
    thus "F ⊖ stake (Suc n)·(sinftimes (↑a)) = ε" by (subst sinftimes_unfold)
  qed
  hence "(F ⊖ (⨆i. stake i·(sinftimes (↑a)))) = ε" by (simp add:contlub_cfun_arg)
  thus ?thesis by (simp add: reach_stream)
qed


lemma slen_sfilter_sdrop_ile:
  "#(sfilter X·(sdrop n·p)) ≤ #(sfilter X·p)"
apply (rule spec [where x = "n"])
apply (rule ind [of _ p], auto, simp add: len_stream_def)
apply (subst lnle_def, simp del: lnle_conv)
apply (case_tac "x", auto)
apply (case_tac "a ∈ X", auto)
apply (erule_tac x="nat" in allE)
by (rule trans_lnle, auto)
```

```
lemma slen_sfilter_sdrop:
  "∀p X. #(sfilter X·p) = ∞⟶#(sfilter X·(sdrop n·p)) = ∞"
apply (induct_tac n, auto)
apply (rule_tac x=p in scases, auto)
by (case_tac "a∈X", auto)

text ‹@{term sfilter} on @{term "stake n"} returns ‹ϵ› if none of the first
  @{term n} elements is included in the filter›
lemma sfilter_empty_snths_nin_lemma:
  "∀p. (∀n. Fin n < #p⟶snth n p ∉ X)⟶sfilter X·(stake k·p) = ϵ"
apply (induct_tac k, auto)
apply (rule_tac x=p in scases, auto)
apply (case_tac "a∈X", auto)
apply (erule_tac x="0" in allE, simp)
apply (case_tac "n", auto)
apply (erule_tac x="s" in allE, auto)
by (erule_tac x="Suc n" in allE, auto)

text ‹@{term sfilter} returns ‹ϵ› if no element is included in the filter›
lemma ex_snth_in_sfilter_nempty:
  "(⋀n. Fin n < #p⟶snth n p ∉ X)⟹sfilter X·p = ϵ"
apply (subgoal_tac "sfilter X·p = (⨆k. sfilter X·(stake k·p))")
apply (erule ssubst)
apply (subst lub_eq_bottom_iff, simp)
apply (subst sfilter_empty_snths_nin_lemma, simp+)
apply (subst contlub_cfun_arg [THEN sym], simp)
by (simp add: reach_stream)


lemma sfilter_snths_in_lemma:
  "∀p. (∀n. Fin n < #p⟶snth n p ∈ X)⟶sfilter X·(stake k·p) = stake k·p"
  apply (induct_tac k, auto)
  apply (rule_tac x=p in scases, auto)
  apply (case_tac "a∈X", auto)
   apply (case_tac "n", auto)
   apply (erule_tac x="s" in allE, auto)
   apply (erule_tac x="Suc n" in allE, auto)
  by (erule_tac x="0" in allE, simp)

lemma sfilter_snths_in_stream_lemma:assumes a1:"⋀ n . Fin n <#p ⟹snth n p ∈ X"
  shows  " p =  sfilter X·(p)"
  apply (subst reach_stream [THEN sym], rule sym)
  apply (subst reach_stream [THEN sym],case_tac "#p=∞")
   apply(smt Inf.INF_cong a1 approxl1 assms monofun_cfun_arg sfilter_snths_in_lemma slen_stake_fst_inf
        stream.take_below)
  by (metis (no_types, hide_lams) assms infI  fin2stake sfilter_snths_in_lemma)


lemma slen_sfilterl1: "#(sfilter S·x) ≤ #x"
apply (rule ind [of _ x], auto, simp add: len_stream_def)
apply (subst lnle_def, simp del: lnle_conv)
apply (case_tac "a ∈ S", auto)
by (rule trans_lnle, auto)


lemma sfilterl4:
  "#(sfilter X·x) = ∞⟹#x = ∞"
by (insert slen_sfilterl1 [of X x], auto)


lemma sfilterl2:
  "∀z. #(sfilter X·s) ≤ #(sfilter X·((stake n·z) • s))"
apply (induct_tac n, auto)
apply (rule_tac x=z in scases, auto)
apply (case_tac "a∈X", auto)
apply (erule_tac x="sa" in allE)
by (drule trans_lnle, auto)

text ‹The filtered result is not changed by concatenating streams which are
  filtered to ‹ϵ››
lemma sfilterl3:
  "∀s. #s = Fin k ∧ sfilter S·s = ϵ⟶
       sfilter S·(s•Z) = sfilter S·Z"
apply (induct_tac k, auto)
apply (rule_tac x=s in scases, auto)
by (case_tac "a ∈ S", auto)


lemma split_sfilter: "sfilter X·x = sfilter X·(stake n·x) • sfilter X·(sdrop n·x)"
apply (rule_tac x=x in spec)
apply (induct_tac n, simp)
apply (rule allI)
apply (rule_tac x=x in scases, simp)
apply (erule_tac x="s" in allE, auto)
by (case_tac "a ∈ X", auto)


lemma int_sfilterl1[simp]: "sfilter S·(sfilter M·s) = sfilter (S ∩ M)·s"
apply (rule ind [of _ s], auto)
apply (case_tac "a ∈ S ∩ M", auto)
by (case_tac "a ∈ M", auto)


lemma add_sfilter:
```

```
  "#x = Fin k⟹sfilter t·(x • y) = sfilter t·x • sfilter t·y"
apply (simp add: atomize_imp)
apply (rule_tac x="y" in spec)
apply (rule_tac x="x" in spec)
apply (induct_tac k, auto)
apply (rule_tac x="x" in scases, auto)
by (case_tac "a ∈ t", auto)


lemma sfilter_smap_nrange:
  "m ∉ range f⟹sfilter {m}·(smap f·x) = ϵ"
apply (rule ex_snth_in_sfilter_nempty [rule_format], simp)
apply (subst smap_snth_lemma, simp+)
apply (rule notI)
apply (drule sym)
by (drule_tac f="f" in range_eqI, simp)

lemma sfilter_lub_inf: assumes "⋀n. ∃i. Fin n ≤ #(A ⊖ (Y i))" and "chain Y"
  shows " #(A ⊖ (⨆i. Y i)) = ∞"
proof(rule ccontr)
  assume as: "#(A ⊖ (⨆i::nat. Y i)) ≠∞"
  obtain n where n_def:"#(A ⊖ (⨆i. Y i)) = Fin n"
    using as lncases by auto
  obtain i where i_def:"Fin (Suc n) ≤  #(A ⊖ (Y i))"
    using assms(1) by blast
  have "#(A ⊖ (Y i)) ≤ #(A ⊖ (⨆i::nat. Y i))"
    using assms(2) cont_pref_eq1I is_ub_thelub mono_slen by blast
  thus False
    using dual_order.trans i_def n_def by fastforce
qed

lemma snth_filter: "Fin n < #s⟹snth n s∈A⟹sfilter A·s ≠⊥"
  apply(induction n arbitrary: s)
  apply auto
  apply (metis lnsuc_neq_0_rev sfilter_in slen_scons strict_slen surj_scons)
  apply(simp add: snth_rt)
  by (metis Fin_02bot Fin_Suc inject_Fin lnzero_def n_not_Suc_n not_le only_empty_has_length_0 sfilter_in
        sfilter_nin slen_rt_ile_eq slen_scons stream.sel_rews(2) strictI surj_scons)


(* ------------------------------------------------------------------------ *)
subsection ‹@{term stakewhile}›
(* ------------------------------------------------------------------------ *)



(* stakewhile f to an empty stream returns the empty stream *)
lemma strict_stakewhile[simp]: "stakewhile f·ϵ = ϵ"
by (subst stakewhile_def [THEN fix_eq2], simp)

(* if the head a passes the predicate f, then the result of stakewhile will start with ↑a *)
lemma stakewhile_t[simp]: "f a⟹stakewhile f·(↑a • s) = ↑a • stakewhile f·s"
by (subst stakewhile_def [THEN fix_eq2], simp)

(* if the head a fails the predicate f, then stakewhile will produce the empty stream *)
lemma stakewhile_f[simp]: "¬f a⟹stakewhile f·(↑a • s) = ϵ"
by (subst stakewhile_def [THEN fix_eq2], simp)

(* if the element a passes the predicate f, then stakewhile applied to ↑a is a no-op *)
lemma [simp]: "f a⟹stakewhile f·(↑a) = ↑a"
by (subst stakewhile_def [THEN fix_eq2], simp)

(* if the element a fails the predicate f, then stakewhile applied to ↑a will produce the empty stream *)
lemma [simp]: "¬f a⟹stakewhile f·(↑a) = ϵ"
by (subst stakewhile_def [THEN fix_eq2], simp)

(* stakewhile can't increase the length of a stream *)
lemma stakewhile_less [simp]: "#(stakewhile f·s)≤#s"
apply(rule ind [of _ s], auto)
apply (metis (mono_tags, lifting) admI inf_chainl4 inf_ub l42)
by (metis bot_is_0 lnle_def lnsuc_lnle_emb minimal slen_empty_eq slen_scons stakewhile_f stakewhile_t)

(* stakewhile doesn't take elements past an element that fails the predicate f *)
lemma stakewhile_slen[simp]: "¬f (snth n s)⟹#(stakewhile f·s)≤Fin n"
apply(induction n arbitrary: s)
apply (metis Fin_02bot lnat_po_eq_conv lnzero_def sdrop_0 slen_empty_eq snth_def stakewhile_f strict_stakewhile
     surj_scons)
by (smt inject_scons slen_rt_ile_eq snth_rt stakewhile_f stakewhile_t stream.take_strict strict_stakewhile
     surj_scons ub_slen_stake)

(* the prefix of the constant stream of x's whose elements aren't equal to x is empty *)
lemma [simp]: "stakewhile (λa. a ≠ x)·↑x^∞ = ϵ"
by (metis (full_types)sinftimes_unfold stakewhile_f)

(* stakewhile produces a prefix of the input *)
lemma stakewhile_below[simp]: "stakewhile f·s ⊑ s"
apply(induction s)
apply(simp+)
by (smt minimal monofun_cfun_arg stakewhile_f stakewhile_t stream.con_rews(2) stream.sel_rews(5) surj_scons)

(* if stakewhile leaves a stream s unchanged, then every element must pass the predicate f *)
lemma stakewhile_id_snth: assumes "stakewhile f·s = s" and "Fin n < #s"
shows "f (snth n s)"
by (metis Fin_leq_Suc_leq assms(1) assms(2) less2eq less2lnleD lnless_def stakewhile_slen)
```

```
(* if stakewhile produces a result of length n or greater, then the nth element in s must pass f *)
lemma stakewhile_snth[simp]: assumes  "Fin n < #(stakewhile f·s)"
shows "f (snth n s)"
by (meson assms not_less stakewhile_slen)

(* if stakewhile changes the stream s, then there must be an element in s that fails the predicate f *)
lemma stakewhile_notin [simp]:
  shows "stakewhile f·s ≠ s⟹#(stakewhile f·s) = Fin n⟹¬ f (snth n s)"
  apply(induction n arbitrary:s)
   apply (metis Fin_02bot lnat.con_rews slen_scons snth_shd stakewhile_t surj_scons)
  by (smt Fin_02bot Fin_Suc approxl2 inject_scons lnat.con_rews lnat_po_eq_conv lnsuc_lnle_emb lnzero_def
        slen_empty_eq slen_rt_ile_eq snth_rt snth_shd stakewhile_below stakewhile_slen stakewhile_t
        stream.take_strict surj_scons ub_slen_stake)


(* ----------------------------------------------------------------------- *)
subsection ‹@{term stwbl}›
(* ----------------------------------------------------------------------- *)



(* stwbl f to an empty stream returns the empty stream *)
lemma strict_stwbl[simp]: "stwbl f·ε = ε"
by (subst stwbl_def [THEN fix_eq2], simp)

(* if the head a passes the predicate f, then the result of stwbl will start with ↑a *)
lemma stwbl_t[simp]: "f a⟹stwbl f·(↑a • s) = ↑a • stwbl f·s"
by (subst stwbl_def [THEN fix_eq2], simp)

(* if the head a fails the predicate f, then stwbl will produce only ↑a *)
lemma stwbl_f[simp]: "¬ f a⟹stwbl f·(↑a • s) = ↑a"
by (subst stwbl_def [THEN fix_eq2], simp)

(* if s is not empty, then stwbl f also does not return the empty stream *)
lemma stwbl_notEps: "s≠ε⟹(stwbl f·s)≠ε"
by (smt lnat.con_rews lnzero_def sconc_snd_empty slen_scons strict_slen stwbl_f stwbl_t surj_scons)

(* if stwbl f applied to s returns the empty stream, then s was empty *)
lemma stwbl_eps: "stwbl f·s = ε⟷s=ε"
using strict_stwbl stwbl_notEps by blast


lemma sfilter_twl1[simp]:
  "sfilter X·(stakewhile (λx. x∉X)·p) = ε"
apply (rule ind [of _ p], auto)
by (case_tac "a∈X", auto)


lemma sfilter_twl2[simp]:
  "sfilter X·(stakewhile (λx. x∈X)·p) = stakewhile (λx. x∈X)·p"
apply (rule ind [of _ p], auto)
by (case_tac "a∈X", auto)

text ‹If @{term "stakewhile (λp. p = t)"} returns an infinite stream, all prefixes
  of the original stream only consist of "@{term t}"s›
lemma stakewhile_sinftimes_lemma:
  "∀z. #(stakewhile (λp. p = t)·z) = ∞⟶stake n·z = stake n·(sinftimes (↑t))"
apply (induct_tac n, auto)
apply (subst sinftimes_unfold, simp)
apply (rule_tac x=z in scases, auto)
by (case_tac "a=t", auto)

text ‹If @{term "stakewhile (λp. p = t)"} returns an infinite stream, the original stream
  is an infinite "@{term t}"-stream›
lemma stakewhile_sinftimesup:
  "#(stakewhile (λp. p = t)·z) = ∞⟹z = sinftimes (↑t)"
apply (rule stream.take_lemma)
by (rule stakewhile_sinftimes_lemma [rule_format])


(* ----------------------------------------------------------------------- *)
subsection ‹@{term sdropwhile}›
(* ----------------------------------------------------------------------- *)



(* sdropwhile f applied to the empty stream returns the empty stream *)
lemma strict_sdropwhile[simp]: "sdropwhile f·ε = ε"
by (subst sdropwhile_def [THEN fix_eq2], simp)

(* if the head a passes the predicate f, then the result of sdropwhile will drop the head *)
lemma sdropwhile_t[simp]: "f a⟹sdropwhile f·(↑a • s) = sdropwhile f·s"
by (subst sdropwhile_def [THEN fix_eq2], simp)

(* if the head a fails the predicate f, then the result of sdropwhile will start with ↑a *)
lemma sdropwhile_f[simp]: "¬f a⟹sdropwhile f·(↑a • s) = ↑a • s"
by (subst sdropwhile_def [THEN fix_eq2], simp)

(* if the only element in a singleton stream passes the predicate f, then sdropwhile will produce
   the empty stream *)
lemma [simp]: "f a⟹sdropwhile f·(↑a) = ε"
by (subst sdropwhile_def [THEN fix_eq2], simp)
```

```
(* if the only element in a singleton stream fails the predicate f, then sdropwhile will be a no-op *)
lemma [simp]: "¬f a⟹sdropwhile f·(↑a) = ↑a"
by (subst sdropwhile_def [THEN fix_eq2], simp)


(* the elements removed by sdropwhile are a subset of the elements removed by sfilter *)
lemma sfilter_dwl1[simp]:
  "sfilter X·(sdropwhile (λx. x∉X)·p) = sfilter X·p"
apply (rule ind [of _ p], auto)
by (case_tac "a∈X", auto)

(* the elements kept by sfilter are a subset of the elements kept by sdropwhile *)
lemma sfilter_dwl2:
  "sfilter T·s ≠ ε⟹sdropwhile (λa. a ∉ T)·s ≠ ε"
apply (rule notI)
apply (erule notE)
apply (subst sfilter_dwl1 [THEN sym])
by simp


lemma stwbl_stakewhile: "stwbl f·s = stakewhile f·s • (stake (Suc 0)·(sdropwhile f·s))"
apply (rule stream.take_lemma)
apply (rule_tac x="s" in spec)
apply (induct_tac n, simp+)
apply (rule allI)
apply (rule_tac x="x" in scases, simp+)
by (case_tac "f a", simp+)


lemma stakewhile_sdropwhilel1:
  "∀x. #(stakewhile f·x) = Fin n⟶sdropwhile f·x = sdrop n·x"
apply (induct_tac n, auto)
apply (rule_tac x=x in scases, auto)
apply (case_tac "f a", auto)
apply (rule_tac x=x in scases, auto)
by (case_tac "f a", auto)


lemma sdropwhile_idem: "sdropwhile f·(sdropwhile f·x) = sdropwhile f·x"
apply (rule ind [of _ x], auto)
by (case_tac "f a", auto)


lemma tdw[simp]: "stakewhile f·(sdropwhile f·s) = ε"
apply (rule ind [of _ s], auto)
by (case_tac "f a", auto)

(* relation between stakewhile and sdropwhile *)
lemma stakewhileDropwhile: "stakewhile f·s • (sdropwhile f·s) = s "
apply(rule ind [of _s])
apply (rule admI)
apply (metis (no_types, lifting) approxl2 inf_chainl4 lub_eql lub_finch2 sconc_fst_inf split_streaml1
     stakewhile_below stakewhile_sdropwhilel1)
apply simp
by (metis assoc_sconc sconc_fst_empty sdropwhile_f sdropwhile_t stakewhile_t tdw)

text ⟨For the head of @{term "sdropwhile f·x"}, @{term f} does not hold⟩
lemma sdropwhile_resup: "sdropwhile f·x = ↑a • s⟹¬ f a"
apply (subgoal_tac "sdropwhile f·(↑a • s) = ↑a • s")
apply (case_tac "f a", auto)
apply rotate_tac
apply (drule cfun_arg_cong [of _ _ "stakewhile f"], simp)
apply (drule sym, simp)
by (rule sdropwhile_idem)


lemma sfilter_srtdwl3[simp]:
  "sfilter X·(srt·(sdropwhile (λx. x∉X)·p)) = srt·(sfilter X·p)"
apply (rule ind [of _ p], auto)
by (case_tac "a∈X", auto)



lemma sfilter_ne_resup: "sfilter T·s ≠ ε⟹shd (sfilter T·s) ∈ T"
apply (subst sfilter_dwl1 [THEN sym])
apply (rule_tac x="sdropwhile (λx. x ∉ T)·s" in scases, auto)
apply (drule sfilter_dwl2, simp)
apply (rule_tac x="s" in scases, auto)
apply (case_tac "aa ∈ T", auto)
apply (drule inject_scons, simp)
by (drule sdropwhile_resup, simp)


lemma sfilter_resl2:
  "sfilter T·s = ↑a • as⟹a ∈ T"
apply (case_tac "sfilter T·s = ε", simp)
by (drule sfilter_ne_resup, simp)


lemma sfilterl7:
  "⟦Fin n < #x; sfilter T·s = x⟧⟹snth n x ∈ T"
apply (simp add: atomize_imp)
```

```
apply (rule_tac x="s" in spec)
apply (rule_tac x="x" in spec)
apply (induct_tac n, auto)
apply (rule sfilter_ne_resup)
apply (rule_tac x="sfilter T·xa" in scases, auto)
apply (rule_tac x="xa" in scases, auto)
apply (simp add: Fin_Suc [THEN sym] del: Fin_Suc)
apply (case_tac "a ∈ T", auto)
apply (case_tac "sfilter T·s = ε",simp+)
apply (simp add: Fin_Suc [THEN sym] del: Fin_Suc)
apply (rule_tac x="sfilter T·s" in scases, auto)
apply (erule_tac x="sa" in allE,simp+)
apply (frule sfilter_resl2)
apply (drule mp)
by (rule_tac x="srt·(sdropwhile (λx. x∉T)·s)" in exI,simp+)


(* ---------------------------------------------------------------------- *)
subsection ⟨@{term srtdw}⟩
(* ---------------------------------------------------------------------- *)



(* srtdw f applied to the empty stream always returns the empty stream *)
lemma [simp]: "srtdw f·ε = ε"
by (simp add: srtdw_def)

(* the rest of any singleton stream is the empty stream, regardless of whether the only element in
   the stream was dropped *)
lemma [simp]: "srtdw f·(↑a) = ε"
apply (simp add: srtdw_def)
by (case_tac "f a", auto)

(* if the head a passes the predicate f, srtdw will drop the head *)
lemma [simp]: "f a⟹srtdw f·(↑a•as) = srtdw f·as"
by (auto simp add: srtdw_def)

(* if the head a fails the predicate f, srtdw will produce the rest of the stream *)
lemma [simp]: "¬ f a⟹srtdw f·(↑a•as) = as"
by (simp add: srtdw_def)

text ⟨@{term "sfilter M"} after @{term "srtdw (λx. x ∉ M)"} almost behaves
  like @{term "sfilter M"} alone⟩
lemma sfilterl8:
  "sfilter M·x ≠ ε⟹
    #(sfilter M·x) = lnsuc·(#(sfilter M·(srtdw (λx. x ∉ M)·x)))"
  apply(induction x rule: ind)
  apply (simp add: len_stream_def)
  apply auto
  by (case_tac "a ∈ M", auto)


lemma sfilter_srtdwl2:
  "#(sfilter X·s) = ∞⟹#(sfilter X·(srtdw (λa. a ∉ X)·s)) = ∞"
apply (case_tac "sfilter X·s = ε", auto)
by (drule sfilterl8, auto)


lemma stwbl_srtdw: "stwbl f·s • srtdw f·s = s"
apply (rule stream.take_lemma)
apply (rule_tac x="s" in spec)
apply (induct_tac n, simp+)
apply (rule allI)
apply (rule_tac x="x" in scases, simp+)
by (case_tac "f a", simp+)

(* the length of srtdw f·x is always smaller than the length of x *)
lemma slen_srtdw: "#(srtdw f·x) ≤ #x"
  apply (induction x rule:ind)
  apply(simp add: len_stream_def)
apply (subst lnle_conv [THEN sym], simp del: lnle_conv, simp)
apply (case_tac "f a", simp+)
by (rule trans_lnle, simp+)

(* stwbl produces a prefix of the input *)
lemma stwbl_below [simp]: "stwbl f·s ⊑ s"
by (metis (no_types) minimal monofun_cfun_arg sconc_snd_empty stwbl_srtdw)

(* relation between srtdw and stwbl *)
lemma srtdw_stwbl [simp]: "srtdw f· (stwbl f·s) = ε" (is "?F s")
proof(rule ind [of _s ])
  show "adm ?F" by simp
  show "?F ε" by simp
  fix a
  fix s
  assume IH: "?F s"
  thus "?F (↑a • s)"
  proof (cases "f a")
    case True thus ?thesis by (simp add: IH)
  next
    case False thus ?thesis by simp
  qed
qed
```

```
(* ------------------------------------------------------------------------ *)
subsection ‹@{term srcdups}›
(* ------------------------------------------------------------------------ *)


(* srcdups applied to the empty stream returns the empty stream *)
lemma strict_srcdups[simp]: "srcdups·ϵ = ϵ"
by (subst srcdups_def [THEN fix_eq2], simp)

(* a singleton stream can't possibly contain duplicates *)
lemma [simp]: "srcdups·(↑a) = ↑a"
by (subst srcdups_def [THEN fix_eq2], simp)

(* if the head a of a stream is followed by a duplicate, only one of the two elements will be kept by srcdups *)
lemma srcdups_eq[simp]: "srcdups·(↑a•↑a•s) = srcdups·(↑a•s)"
apply (subst srcdups_def [THEN fix_eq2], simp)
by (rule sym, subst srcdups_def [THEN fix_eq2], simp)

(* if the head a of a stream is followed by a distinct element, both elements will be keypt by srcdups *)
lemma srcdups_neq[simp]:
  "a≠b⟹srcdups·(↑a • ↑b • s) = ↑a •  srcdups·(↑b • s)"
  by (subst srcdups_def [THEN fix_eq2], simp)

lemma srcdups_slen [simp]: "#(srcdups·s) ≤ #s"
  apply (rule ind [of _ s])
  apply (simp_all)
  apply (metis (mono_tags, lifting) adml inf_chainl4 inf_ub l42)
  apply (rule_tac x=s in scases, simp_all)
  apply (case_tac "a = aa", simp_all)
  using less_lnsuc order.trans by blast

(*Used for ABP-Composition of sender and receiver*)
lemma srcdups_eq2:"a=b⟹srcdups·(↑a•↑b•s) = srcdups·(↑b•s)"
  by simp

lemma srcdups_ex:"∃y. srcdups·(↑a•s) = ↑a•y"
  by(subst srcdups_def [THEN fix_eq2], auto)

lemma srcdups_shd[simp]:"shd (srcdups·(↑a•s)) = a"
  by(subst srcdups_def [THEN fix_eq2],auto)

lemma srcdups_srt:"srt·(srcdups·(↑a•s)) = (srcdups·(sdropwhile (λz. z = a)·s))"
  by(subst srcdups_def [THEN fix_eq2], auto)

lemma srcdups_shd2[simp]:"s≠ϵ⟹shd (srcdups·s) = shd s"
  by(subst srcdups_def [THEN fix_eq2],auto)

lemma srcdups_srt2:"s≠ϵ⟹srt·(srcdups·s) = (srcdups·(sdropwhile (λz. z = shd s)·(srt·s)))"
  by(subst srcdups_def [THEN fix_eq2],auto)

lemma srcdups_imposs_h:"Fin 1 < #(srcdups·s)⟹shd(srcdups·s)≠shd(srt·(srcdups·s))"
  apply (cases "s=ϵ")
  using empty_is_shortest apply fastforce
  apply (subst srcdups_srt2,auto)
  apply (subgoal_tac "srcdups·(sdropwhile (λz. z = shd s)·(srt·s))≠ϵ")
  apply (metis (mono_tags, lifting) sdropwhile_resup srcdups_shd2 strict_srcdups surj_scons)
proof -
  assume a1: "s ≠ ϵ"
  assume a2: "Fin 1 < #(srcdups·s)"
  have f1:"Fin 0 = 0"
    using Fin_02bot bot_is_0 by presburger
  then have "lnsuc·0 = Fin (Suc 0)"
    by (metis Fin_Suc)
  then show "srcdups· (sdropwhile (λa. a = shd s)·(srt·s)) ≠ ϵ"
    using a2 a1 f1
    by (metis Suc_lel less2nat not_le not_one_le_zero srcdups_srt2 srt_decrements_length strict_slen zero_le_one)
qed

lemma srcdups_imposs:"srcdups·(↑a • s) ≠ ↑a • ↑a • y"
  apply(cases "#(srcdups·(↑a •s)) < Fin 1 ")
  apply (metis One_nat_def bot_is_0 lnat.con_rews neq02Suclnle not_less slen_scons)
  apply(insert srcdups_imposs_h[of "↑a• s"])
  by (metis Fin_02bot Fin_Suc One_nat_def lnat.con_rews lnat.sel_rews(2) lscons_conv neq_iff shd1
      slen_scons stream.sel_rews(5) up_defined)

lemma srcdupsimposs: "srcdups·(↑a • s) ≠ ↑a • ↑a • srcdups·s"
  by (simp add: srcdups_imposs)

lemma srcdupsimposs2_h2:"∀x. srcdups·(↑a • s)≠ ↑a • ↑a • x"
  by (simp add: srcdups_imposs)

lemma srcdupsimposs2: "srcdups·(↑a • s) ≠ ↑a • ↑a • s"
  by(simp add: srcdups_imposs)

lemma srcdups_anotb_h:"srcdups·(↑a•↑b) = ↑a • ↑b⟹a ≠ b"
  by (metis sconc_snd_empty srcdups_imposs)

lemma srcdups_anotb:"srcdups·(↑a • ↑b • s) = ↑a • ↑b • s⟹a≠ b"
  using srcdupsimposs2_h2 by auto

lemma srcdups2srcdups: "srcdups·(srcdups·s) = srcdups·s"
proof(induction rule: ind [of _ s])
```

```
    case 1
    then show ?case
      by simp
  next
    case 2
    then show ?case
      by simp
  next
    case (3 a s)
    have f1:"a = shd s⟹s≠ϵ⟹srcdups·(↑a • s) = srcdups·s"
      using srcdups_eq[of "shd s" "srt·s"] surj_scons[of s] by auto
    have f2:"a=shd s⟹s≠ϵ⟹srcdups·(srcdups·(↑a • s)) = srcdups·(↑a • s)"
      using f1 "3.IH" by auto
    moreover have"a≠shd s⟹srcdups·(↑a • s) = ↑a•srcdups·s"
      by (metis srcdups_neq srcdups_shd srcdups_srt strict_sdropwhile surj_scons)
    ultimately have f3:"a≠ shd s⟹srcdups·(srcdups·(↑a • s)) = srcdups·(↑a • s)"
    proof -
      assume a1: "a ≠ shd s"
      then have f2: "srcdups·(↑a • s) = ↑a • srcdups·s"
        by (metis ‹a ≠ shd s⟹srcdups·(↑a • s) = ↑a • srcdups·s›)
      obtain ss :: "'a stream ⇒ 'a ⇒ 'a stream" where
        f3: "∀a s. srcdups·(↑a • s) = ↑a • ss s a"
        by (meson srcdups_ex)
      then have f4: "↑(shd s) • srt·s = s⟶srcdups·(↑a • s) = ↑a • ↑(shd s) • ss (srt·s) (shd s)"
        using f2 by metis
      have f5: "↑(shd s) • srt·s = s⟶srcdups·(↑(shd s) • ss (srt·s) (shd s)) = srcdups·s"
        using f3 by (metis "3.IH")
      { assume "↑a • s ≠ srcdups·(↑a • s)"
        then have "s ≠ ϵ"
          by force
        then have ?thesis
          using f5 f4 f2 a1 by (simp add: surj_scons) }
      then show ?thesis
        by fastforce
    qed
    then show ?case
      using f2 by fastforce
qed

lemma srcdups_prefix_neq:"x ⊑ y⟹srcdups·x ≠x⟹srcdups·y ≠ y"
proof(induction arbitrary: y rule: ind [of _ x])
    case 1
    then show ?case
      by simp
  next
    case 2
    then show ?case
      by simp
  next
    case (3 a s)
    have f1:"a=shd s⟹srcdups·(↑a • s) = srcdups·s"
      by (metis "3.prems"(2) srcdups_eq2 srcdups_shd srcdups_srt strict_sdropwhile strict_srcdups surj_scons)
    have f2:"a≠shd s⟹srcdups·(↑a • s) = ↑a • srcdups·s"
      by (metis srcdups_neq srcdups_shd srcdups_srt strict_sdropwhile surj_scons)
    then have f3:"a≠shd s⟹srcdups·s ≠ s"
      using "3.prems" by auto
    show ?case
      by (smt "3.IH" "3.prems"(1) f1 f2 f3 less_fst_sconsD lscons_conv scases srcdups2srcdups
          srcdups_eq srcdups_ex srcdups_srt srcdupsimposs2 stream.con_rews(2) stream.sel_rews(5)
          sup'_def surj_scons)
qed

lemma srcdups_smap_adm [simp]:
  "adm (λa. srcdups·(smap f·(srcdups·a)) = smap f·(srcdups·a)
    ⟶srcdups·(smap f·a) = smap f·(srcdups·a))"
  apply (rule adm_imp, auto)
  apply(rule adm_upward)
  apply rule+
  using srcdups_prefix_neq
  by (metis monofun_cfun_arg)

lemma srcdups_smap_com_h:"s≠ϵ⟹a ≠ shd s⟹srcdups·(smap f·(srcdups·(↑a • s))) = smap f·(srcdups·(↑a • s))⟹f
      (shd s) ≠ f a"
  apply(cases "shd(srt·s) ≠ shd s")
  apply (insert srcdups_neq[of a "shd s" "srt·s"] surj_scons[of s], simp)
  apply (metis smap_scons srcdups_ex srcdupsimposs2_h2)
  apply (insert srcdups_neq[of a "shd s" "srt·s"] surj_scons[of s], simp)
  apply(insert srcdups_ex[of "shd s" "srt·s"],auto)
  by (simp add: srcdupsimposs2_h2)

lemma srcdups_smap_com:
  shows "srcdups·(smap f·(srcdups·s)) = smap f·(srcdups·s)⟹srcdups·(smap f·s)= smap f·(srcdups·s)"
  proof(induction rule: ind [of _ s])
    case 1
    then show ?case
      by simp
  next
    case 2
    then show ?case
      by simp
  next
    case (3 a s)
    have s_eps: "s = ⊥⟹ srcdups·(↑(f a) • smap f·s) = smap f·(srcdups·(↑a • s))" by simp
    hence f1: "shd s = a⟹?case"
```

```
      by (metis "3.IH" "3.prems" smap_scons srcdups_eq surj_scons)
    have h1: "s≠⊥⟹s = (↑(shd s) • srt·s)" by (simp add: surj_scons)
    have h2: "s≠⊥⟹shd s≠a⟹f (shd s) ≠ f a⟹srcdups·(smap f·(↑a • (↑(shd s) • srt·s))) = smap f·(srcdups·(↑a •
          (↑(shd s) • srt·s)))"
    proof -
      assume a1: "f (shd s) ≠ f a"
      assume a2: "s ≠ ε"
      assume a3: "shd s ≠ a"
      have f4: "s = ↑(shd s) • srt·s"
        using a2 by (metis h1)
      obtain ss :: "'b stream ⇒ 'b ⇒ 'b stream" where
        f5: "∀b s. srcdups·(↑b • s) = ↑b • ss s b"
        by (meson srcdups_ex)
      then have f6: "↑(shd s) • ss (srt·s) (shd s) = srcdups·s"
        using f4 by metis
      have f7: "srcdups· (↑(f a) • ↑(f (shd s)) • smap f· (ss (srt·s) (shd s))) = ↑(f a) • srcdups· (↑(f (shd s))
        • smap f· (ss (srt·s) (shd s)))"
        using a1 by auto
      have f8: "↑a • ↑(shd s) • ss (srt·s) (shd s) = srcdups·(↑a • ↑(shd s) • srt·s)"
        using f5 a3 by (metis (no_types) srcdups_neq)
      then have f9: "↑(f a) • ↑(f (shd s)) • smap f· (ss (srt·s) (shd s)) = smap f·(srcdups·(↑a • s))"
        using f4 by (metis (no_types) smap_scons)
      then obtain ssa :: "'a stream ⇒ 'a ⇒ 'a stream" where
        f10: "↑(f a) • ssa (↑(f (shd s)) • ssa (smap f·(ss (srt·s) (shd s))) (f (shd s))) (f a) = srcdups·(smap
          f·(srcdups·(↑a • s)))"
        by (simp add: "3.prems")
      then have f11: "↑(f a) • ssa (↑(f (shd s)) • ssa (smap f·(ss (srt·s) (shd s))) (f (shd s))) (f a) =
        srcdups· (↑(f a) • ↑(f (shd s)) • smap f· (ss (srt·s) (shd s)))"
        using f9 by presburger
      have "↑(f a) • ssa (↑(f (shd s)) • ssa (smap f·(ss (srt·s) (shd s))) (f (shd s))) (f a) = ↑(f a) • smap
        f·(srcdups·s)"
        using f10 f8 f6 f4 by (metis (no_types) "3.prems" smap_scons)
      then have "srcdups·(smap f·s) = smap f·(srcdups·s)"
        using f11 f7 f6 by (metis (no_types) "3.IH" inject_scons smap_scons)
      then have "srcdups· (smap f·(↑a • ↑(shd s) • srt·s)) = smap f· (↑a • ↑(shd s) • ss (srt·s) (shd s))"
        using f6 f4 a1 by (metis (no_types) smap_scons srcdups_neq)
      then show ?thesis
        using f8 by presburger
    qed
    have f2: "s≠⊥⟹shd s≠a⟹f (shd s) ≠ f a⟹?case"
      using h1 h2 by auto
    have f3: "s≠⊥⟹shd s≠a⟹f (shd s) = f a⟹?case"
      by (simp add: "3.prems" srcdups_smap_com_h)
    then show ?case using f1 f2 by fastforce
  qed

lemma srcdups_nbot: "s≠⊥⟹srcdups·s ≠ ⊥"
  by (metis lscons_conv srcdups_ex stream.con_rews(2) surj_scons)

lemma srcdups_fin: assumes "#(srcdups·s)<∞" and "#s=∞"
 obtains x where "srcdups·x = srcdups·s" and "x⊑s" and "#x < ∞"
proof -
  obtain n where "srcdups·(stake n·s) = srcdups·s"
    by (metis assms(1) fun_approxl2 len_stream_def lnat_well_h2)
  thus ?thesis
    using lnless_def that by auto
qed

lemma srcdups_step: "srcdups·(↑a • s) = ↑a • srcdups·(sdropwhile (λx. x=a)·s)"
  apply(rule ind [of _ s], simp_all)
  by (metis lscons_conv srcdups_ex srcdups_srt stream.sel_rews(5) up_defined)

lemma snprefix: "¬x⊑y⟹lshd·x=lshd·y⟹¬(srt·x)⊑(srt·y)"
  apply auto
  by (metis lshd_updis monofun_cfun_arg stream.sel_rews(2) stream.sel_rews(3) sup'_def surj_scons)

lemma srcdups_consec_noteq: "Fin (Suc n) < #(srcdups·xs)⟹snth n (srcdups·xs) ≠ snth (Suc n) (srcdups·xs)"
proof
  fix n :: nat
  assume "Fin (Suc n) < #(srcdups·xs)" and "snth n (srcdups·xs) = snth (Suc n) (srcdups·xs)"
  then obtain a s where "sdrop n·(srcdups·xs) = ↑a • ↑a • s"
    by (metis convert_inductive_asm drop_not_all sdrop_back_rt snth_def surj_scons)
  then have p: "srcdups·(sdrop n·(srcdups·xs)) ≠ sdrop n·(srcdups·xs)"
    by (simp add: srcdupsimposs2_h2)
  have not_p: "srcdups·(sdrop n·(srcdups·xs)) = sdrop n·(srcdups·xs)"
  proof -
    have "srcdups·(sdrop n·(srcdups·xs)) = sdrop n·(srcdups·(srcdups·xs))"
    proof (induction n)
      case 0
      then show ?case
        by simp
    next
      case (Suc n)
      then show ?case
        by (metis sdrop_back_rt srcdups2srcdups srcdups_srt2 stream.sel_rews(2))
    qed
    thus "srcdups·(sdrop n·(srcdups·xs)) = sdrop n·(srcdups·xs)"
      by (simp add: srcdups2srcdups)
  qed
  thus "False"
    using p by auto
qed

lemma bool_stream_snth:
```

```
  fixes s t :: "bool stream" and n :: nat and a :: bool
  shows "s = ↑a • t⟶Fin n < #(srcdups·s)⟶snth n (srcdups·s) = (even n = a)"
proof (induction n)
  case 0
  then show ?case
    by simp
next
  case (Suc n)
  then show ?case
    using convert_inductive_asm even_Suc srcdups_consec_noteq by blast
qed

lemma srcdups_snth_stake_fin: "⋀s n. #s = Fin k⟹k > (Suc n)⟹snth n s ≠ snth (Suc n) s⟹srcdups·(stake (Suc
    n)·s) ≠ srcdups·s"
proof (induction k rule: less_induct)
  case (less k)
  then show ?case
  proof (cases "k ≤ Suc 0")
    case True
    then show ?thesis
      using less.prems(2) by linarith
  next
    case False
    then obtain a b t where "s = ↑a • ↑b • t"
      by (metis drop_not_all lel le_SucI less.prems(1) less2nat_lemma sdrop_0 sdrop_back_rt surj_scons)
    obtain l where "k = Suc l"
      using Suc_less_eq2 less.prems(2) by blast
    then have "l < k"
      by simp
    moreover have "#(↑b • t) = Fin l"
      using ⟨k = Suc l⟩ ⟨s = ↑a • ↑b • t⟩ less.prems(1) by auto
    then show ?thesis
    proof (cases n)
      case 0
      then have "srcdups·(stake (Suc n)·s) = ↑a"
        by (simp add: ⟨s = ↑a • ↑b • t⟩)
      moreover have "srcdups·s ≠ ↑a"
      proof -
        have "snth 0 s ≠ snth (Suc 0) s"
          using "0" less.prems(3) by blast
        then have "a ≠ b"
          by (simp add: ⟨s = ↑a • ↑b • t⟩)
        then have "srcdups·s = ↑a • srcdups·(↑b • t)"
          using ⟨s = ↑a • ↑b • t⟩ srcdups_neq by blast
        then show ?thesis
          using srcdups_nbot by force
      qed
      ultimately show ?thesis
        by auto
    next
      case (Suc m)
      have "Suc m < l"
        using Suc ⟨k = Suc l⟩ less.prems(2) by blast
      moreover have "snth m (↑b • t) ≠ snth (Suc m) (↑b • t)"
      proof -
        have "↑b • t = srt·s"
          by (simp add: ⟨s = ↑a • ↑b • t⟩)
        then show ?thesis
          by (metis (no_types) Suc less.prems(3) sdrop_forw_rt snth_def)
      qed
      ultimately have "srcdups·(stake (Suc m)·(↑b • t)) ≠ srcdups·(↑b • t)"
        using ⟨#(↑b • t) = Fin l⟩ ⟨l < k⟩ less.IH by blast
      then have "srcdups·(↑b • (stake m·t)) ≠ srcdups·(↑b • t)"
        by simp
      then have "srcdups·s ≠ ↑a • (srcdups·(↑b • (stake m·t)))"
        by (metis (no_types, lifting) ⟨s = ↑a • ↑b • t⟩ inject_scons srcdups2srcdups srcdups_eq srcdups_neq
            srcdups_step)
      then show ?thesis
      proof -
        { assume "a ≠ b"
          then have "srcdups·(↑a • ↑b • stake m·t) ≠ srcdups·s"
            using ⟨srcdups·s ≠ ↑a • srcdups·(↑b • stake m·t)⟩ by auto
          then have ?thesis
            using Suc ⟨s = ↑a • ↑b • t⟩ by force }
        then show ?thesis
          using Suc ⟨s = ↑a • ↑b • t⟩ ⟨srcdups·(↑b • stake m·t) ≠ srcdups·(↑b • t)⟩ by fastforce
      qed
    qed
  qed
qed

lemma srcdups_end_neq: "#s < ∞⟹a ≠ b⟹srcdups·(s • ↑a • ↑b) = srcdups·(s • ↑a) • ↑b"
proof (rule finind3 [of s], simp+)
  assume "#s < ∞" and "a ≠ b"
  then show "srcdups·(↑a • ↑b) = ↑a • ↑b"
    by (metis lscons_conv srcdups_neq srcdups_step strict_sdropwhile strict_srcdups sup'_def)
next
  fix t :: "'a stream" and c :: 'a
  assume "#t < ∞" and "srcdups·(t • ↑a • ↑b) = srcdups·(t • ↑a) • ↑b"
  then have "srcdups·(↑c • (t • ↑a)) • ↑b = srcdups·(↑c • (t • ↑a • ↑b))"
  proof (cases "t = ε")
    case True
    then show ?thesis
      by (metis (no_types, lifting) ⟨srcdups·(t • ↑a • ↑b) = srcdups·(t • ↑a) • ↑b⟩ assoc_sconc inject_scons
```

```
          sconc_snd_empty srcdups_eq srcdups_neq)
  next
    case False
    then have not_empty: "t ≠ ε"
      by simp
    then show ?thesis
    proof (cases "c = shd t")
      case True
      then show ?thesis
        by (metis False ⟨srcdups·(t • ↑a • ↑b) = srcdups·(t • ↑a) • ↑b⟩ sconc_scons srcdups_eq surj_scons)
    next
      case False
      then have "srcdups·(↑c • (t • ↑a • ↑b)) = ↑c • srcdups·(t • ↑a • ↑b)"
      proof -
        have "↑(shd t) • srt·t = t"
          using not_empty surj_scons by blast
        then show ?thesis
          by (metis (no_types) False assoc_sconc srcdups_neq)
      qed
      then show ?thesis
      proof -
        have "↑(shd t) • srt·t = t"
          by (meson not_empty surj_scons)
        then show ?thesis
          by (metis (no_types) False ⟨srcdups·(t • ↑a • ↑b) = srcdups·(t • ↑a) • ↑b⟩ assoc_sconc srcdups_neq)
      qed
    qed
  qed
  thus "srcdups·((↑c • t) • ↑a • ↑b) = srcdups·((↑c • t) • ↑a) • ↑b"
    by simp
qed

lemma srcdups_end_eq: "srcdups·(s • ↑a • ↑a) = srcdups·(s • ↑a)"
proof (cases "#s < ∞")
  case True
  then show ?thesis
  proof (rule finind3, simp)
    show "srcdups·(↑a • ↑a) = ↑a"
      by (simp add: srcdups_step)
  next
    fix t :: "'a stream" and b :: 'a
    assume "#t < ∞" and "srcdups·(t • ↑a • ↑a) = srcdups·(t • ↑a)"
    then show "srcdups·((↑b • t) • ↑a • ↑a) = srcdups·((↑b • t) • ↑a)"
    proof (cases "t = ε")
      case True
      then show ?thesis
        by (metis sconc_snd_empty srcdups_eq srcdups_neq)
    next
      case False
      then have 1: "t ≠ ε"
        by simp
      then show ?thesis
      proof(cases "shd t = b")
        case True
        then show ?thesis
          by (metis False ⟨srcdups·(t • ↑a • ↑a) = srcdups·(t • ↑a)⟩ assoc_sconc srcdups_eq surj_scons)
      next
        case False
        have "t ≠ ε"
          by (simp add: "1")
        then have "srcdups·(↑b • t • ↑a • ↑a) = ↑b • srcdups·(t • ↑a • ↑a)"
          by (metis (no_types, lifting) False assoc_sconc srcdups_neq surj_scons)
        then show ?thesis
          by (metis (no_types, lifting) "1" False ⟨srcdups·(t • ↑a • ↑a) = srcdups·(t • ↑a)⟩ sconc_scons
              srcdups_neq surj_scons)
      qed
    qed
  qed
next
  case False
  then show ?thesis
    by (simp add: less_le)
qed

lemma srcdups_sntimes: "n > 0⟹srcdups·(sntimes n (↑a)) = ↑a"
proof (induction n)
  case 0
  then show ?case
    by simp
next
  case (Suc n)
  then show ?case
  proof (cases "n > 0")
    case True
    then show ?thesis
      by (metis Suc.IH gr0_implies_Suc sntimes.simps(2) srcdups_eq)
  next
    case False
    then show ?thesis
      by auto
  qed
qed

lemma srcdups_sntimes_prefix: "n > 0⟹srcdups·((sntimes n (↑a)) • s) = ↑a • srcdups·(sdropwhile (λx. x=a)·s)"
```

```
proof (induction n)
  case 0
  then show ?case
    by simp
next
  case (Suc n)
  then show ?case
  proof (cases "n > 1")
    case True
    then obtain m where "n = Suc m" and "m > 0"
      by (metis One_nat_def Suc_lessE)
    then have "srcdups·((sntimes (Suc n) (↑a)) • s) = srcdups·((sntimes n (↑a)) • s)"
      by (simp add: ⟨n = Suc m⟩)
    then show ?thesis
      using Suc.IH ⟨n = Suc m⟩ zero_less_Suc by auto
  next
    case False
    have "srcdups·(↑a • s) = ↑a • srcdups·(sdropwhile (λx. x=a)·s)"
      using srcdups_step by blast
    moreover have "sntimes 1 (↑a) = ↑a"
      by (simp add: One_nat_def)
    ultimately show ?thesis
      by (metis (no_types, lifting) False One_nat_def Suc_lessI assoc_sconc neq0_conv sntimes.simps(2) srcdups_eq)
  qed
qed

(* ---------------------------------------------------------------------- *)
subsection ⟨@{term sscanl}⟩
(* ---------------------------------------------------------------------- *)

(* SSCANL with the empty stream results in the empty stream *)
lemma SSCANL_empty[simp]: "SSCANL n f q ε = ε"
by (induct_tac n, auto)


lemma mono_SSCANL:
  "∀ x y q. x ⊑ y⟶SSCANL n f q x ⊑ SSCANL n f q y"
apply (induct_tac n, auto)
apply (drule lessD, erule disjE, simp)
apply (erule exE)+
apply (erule conjE)+
by (simp, rule monofun_cfun_arg, simp)


lemma contlub_SSCANL:
  "∀f q s. SSCANL n f q s = SSCANL n f q (stake n·s)"
apply (induct_tac n, auto)
apply (rule_tac x=s in scases)
apply auto
apply (rule_tac x=s in scases)
by auto


lemma chain_SSCANL: "chain SSCANL"
apply (rule chainI)
apply (subst fun_below_iff)+
apply (induct_tac i, auto)
apply (rule monofun_cfun_arg)
apply (erule_tac x="x" in allE)
apply (erule_tac x="x xa (shd xb)" in allE)
by (erule_tac x="srt·xb" in allE, auto)


lemma cont_lub_SSCANL: "cont (λs. ⨆i. SSCANL i f q s)"
apply (rule cont2cont_lub)
apply (rule ch2ch_fun)
apply (rule chainI)
apply (rule fun_belowD [of _ _ "q"])
apply (rule fun_belowD [of _ _ "f"])
apply (rule chainE)
apply (rule chain_SSCANL)
apply (rule pr_contI)
apply (rule monofunI)
apply (rule mono_SSCANL [rule_format], assumption)
apply (rule allI)
apply (rule_tac x="i" in exI)
by (rule contlub_SSCANL [rule_format])


(* sscanl applied to the empty stream returns the empty stream *)
lemma sscanl_empty[simp]: "sscanl f q·ε = ε"
apply (simp add: sscanl_def)
apply (subst beta_cfun, rule cont_lub_SSCANL)
by (subst is_lub_const
  [THEN lub_eql, of "ε", THEN sym], simp)

(* scanning ↑a•s using q as the initial element is equivalent to computing ↑(f q a) and appending the
   result of scanning s with (f q a) as the initial element *)
lemma sscanl_scons[simp]:
  "sscanl f q·(↑a•s) = ↑(f q a) • sscanl f (f q a)·s"
apply (simp add: sscanl_def)
apply (subst beta_cfun, rule cont_lub_SSCANL)+
apply (subst contlub_cfun_arg)
apply (rule ch2ch_fun, rule ch2ch_fun)
```

```
apply (rule chainI)
apply (rule fun_belowD [of _ _ "f"])
apply (rule chain_SSCANL [THEN chainE])
apply (subst lub_range_shift [where j="Suc 0", THEN sym])
apply (rule ch2ch_fun, rule ch2ch_fun)
apply (rule chainI)
apply (rule fun_belowD [of _ _ "f"])
by (rule chain_SSCANL [THEN chainE], simp)

(* scanning a singleton stream is equivalent to computing ↑(f a b) *)
lemma sscanl_one[simp]: "sscanl f a·(↑b) = ↑(f a b)"
by (insert sscanl_scons [of f a b ε], auto)


lemma fair2_sscanl: "#x ≤ #(sscanl f a·x)"
apply (rule spec [where x = a])
  apply (rule ind [of _ x], auto)
  apply(simp add: len_stream_def)
by (subst lnle_def, simp del: lnle_conv)

(* The first element of the result of sscanl_h is (f q (shd s)) *)
lemma sscanl_shd: "s≠ε⟹shd (sscanl f q·s) = (f q (shd s))"
by (metis shd1 sscanl_scons surj_scons)

(* dropping the first element of the result of sscanl is equivalent to beginning the scan with
   (f a (shd s)) as the initial element and proceeding with the rest of the input *)
lemma sscanl_srt: "srt·(sscanl f a·s) = sscanl f (f a (shd s)) ·(srt·s) "
by (metis (no_types, lifting) sconc_fst_empty sconc_scons' sscanl_empty sscanl_scons stream.sel_rews(2)
      stream.sel_rews(5) sup'_def surj_scons up_defined)

(* the n + 1'st element produced by sscanl is the result of mering the n + 1'st item of s with the n'th
   element produced by sscanl *)
lemma sscanl_snth:  "Fin (Suc n) < #s⟹snth (Suc n) (sscanl f a·s) = f (snth n (sscanl f a·s)) (snth (Suc n) s)"
apply(induction n arbitrary:  a s)
apply (smt Fin_02bot Fin_leq_Suc_leq less2lnleD less_lnsuc lnat_po_eq_conv lnless_def lnzero_def shd1
      slen_empty_eq slen_rt_ile_eq snth_scons snth_shd sscanl_scons surj_scons)
by (smt Fin_Suc lnat_po_eq_conv lnle_def lnless_def lnsuc_lnle_emb lnzero_def minimal slen_scons snth_scons
      sscanl_scons strict_slen surj_scons)

(* the result of sscanl has the same length as the input stream x *)
lemma fair_sscanl[simp]: "#(sscanl f a·x) = #x"
apply (rule spec [where x = a])
by (rule ind [of _ x], auto, simp add: len_stream_def)

(* Verification of sscanl with sscanl_nth *)

primrec sscanl_nth :: "nat ⇒ ('a ⇒ 'a  ⇒ 'a) ⇒ 'a  ⇒ 'a stream ⇒ 'a" where
"sscanl_nth 0 f q s =  f q (shd s)" |
"sscanl_nth (Suc n) f q s = sscanl_nth n f (f q (shd s)) (srt·s)"

(* Nth element of sscanl is equal to sscanl_nth *)
lemma sscanl2sscanl_nth:
  "Fin n<#s⟹snth n (sscanl f q·s) = sscanl_nth n f q s"
proof (induction n arbitrary: q s, auto)
  fix q :: "'a" and s :: "'a stream" and k :: "lnat"
  assume a1: "#s = lnsuc·k"
  hence h1: "s ≠ ε"
    using lnsuc_neq_0_rev strict_slen by auto
  thus "shd (sscanl f q·s) = f q (shd s)"
    by (simp add: sscanl_shd)
next
  fix n :: "nat" and  q :: "'a" and s :: "'a stream"
  assume a2: "⋀q s. Fin n < #s⟹snth n (sscanl f q·s) = sscanl_nth n f q s"
  assume a3: "Fin (Suc n) < #s"
  thus "snth (Suc n) (sscanl f q·s) = sscanl_nth n f (f q (shd s)) (srt·s)"
    by (metis a2 a3 lel not_less slen_rt_ile_eq snth_rt sscanl_srt)
qed

lemma sscan_ntimes_loop:
  assumes "⋀r. sscanl f state·(s • r) = out • (sscanl f state·r)"
  shows "sscanl f state·(sntimes n s) = sntimes n out"
  using assms
  by (induction n, simp_all)

lemma sscanl_inftimes_loop:
  assumes "⋀r. sscanl f state·(s • r) = out • (sscanl f state·r)"
  shows "sscanl f state·(sinftimes s) = sinftimes out"
  using assms
  by (metis rek2sinftimes sinftimes_unfold slen_empty_eq sscanl_empty fair_sscanl strict_icycle)

(* ---------------------------------------------------------------------- *)
subsection ⟨@{term szip}⟩
(* ---------------------------------------------------------------------- *)



(* szip applied to ε·s returns the empty stream *)
lemma strict_szip_fst[simp]: "szip·ε·s = ε"
by (subst szip_def [THEN fix_eq2],simp)

(* szip applied to s·ε returns the empty stream *)
lemma strict_szip_snd[simp]: "szip·s·ε = ε"
by (subst szip_def [THEN fix_eq2], simp)
```

```
(* unfolding szip *)
lemma szip_scons[simp]: "szip·(↑a•s1)·(↑b•s2) = ↑(a,b) • (szip·s1·s2)"
by (subst szip_def [THEN fix_eq2], simp)

(* rules for szip *)
lemma [simp]: "szip·(↑a)·(↑b • y) = ↑(a,b)"
by (subst szip_def [THEN fix_eq2], simp)

(* rules for szip *)
lemma [simp]: "szip·(↑a • x)·(↑b) = ↑(a,b)"
by (subst szip_def [THEN fix_eq2], simp)

(* rules for szip *)
lemma [simp]: "szip·(↑a)·(↑b) = ↑(a,b)"
by (subst szip_def [THEN fix_eq2], simp)


lemma strict_rev_szip: "szip·x·y = ϵ⟹x = ϵ ∨ y = ϵ"
apply (rule_tac x=x in scases, auto)
by (rule_tac x=y in scases, auto)


lemma sprojfst_szipl1[rule_format]:
  "∀x. #x = ∞⟶sprojfst·(szip·i·x) = i"
apply (rule ind [of _ i], auto)
by (rule_tac x=x in scases, auto)

(* analogous for sprojsnd *)
lemma sprojsnd_szipl1[rule_format]:
  "∀x. #x = ∞⟶sprojsnd·(szip·x·i) = i"
apply (rule ind [of _ i], auto)
by (rule_tac x=x in scases, auto)

(* zipping the infinite constant streams ↑x∞ and ↑y∞ is equivalent to infinitely repeating the tuple
   ↑(x, y) *)
lemma szip2sinftimes[simp]: "szip·((↑x)^∞)·((↑y)^∞) = ((↑(x, y))^∞) "
by (metis s2sinftimes sinftimes_unfold szip_scons)

(* the length of szip·as·bs is the minimum of the lengths of as and bs *)
lemma szip_len [simp]: "#(szip·as·bs) = min (#as) (#bs)"
apply(induction as arbitrary: bs)
apply(rule adml)
apply auto[1]
apply (metis inf_chainl4 inf_less_eq lub_eql lub_finch2 min_def slen_sprojsnd sprojsnd_szipl1)
apply simp
apply(case_tac "bs=ϵ")
apply auto[1]
proof -
  fix u :: "'a discr u"
  fix as:: "'a stream"
  fix bs:: "'b stream"
  assume as1: "u ≠ ⊥" and
    as2: "(⋀bs::'b stream. #(szip·as·bs) = min (#as) (#bs))" and as3: "bs ≠ ϵ"
  obtain a where a_def: "updis a = u" by (metis (mono_tags)as1 lshd_updis stream.sel_rews(4) stream.sel_rews(5)
       surj_scons)
  obtain b bs2 where b_def: "bs = ↑b • bs2" by (metis as3 surj_scons)
  hence "#(szip·(↑a • as)·(↑b • bs2)) = min (#(↑a • as)) (#(↑b • bs2))" by (simp add: as2 min_def)
  thus "#(szip·(u && as)·bs) = min (#(u && as)) (#bs)" by (metis a_def b_def lscons_conv)
qed

lemma szip_nth: "Fin n < #s1⟹Fin n < #s2⟹snth n (szip·s1·s2) = (snth n s1, snth n s2)"
  apply(induction n arbitrary: s1 s2)
  apply auto
   apply (metis lnsuc_neq_0 only_empty_has_length_0 shd1 surj_scons szip_scons)
  apply(simp add: snth_rt)
  by (smt empty_is_shortest leD not_le only_empty_has_length_0 only_empty_has_length_0 only_empty_has_length_0
       slen_rt_ile_eq slen_rt_ile_eq snth_rt snth_scons stream.sel_rews(2) stream.sel_rews(2) strict_slen
       strict_slen strict_slen strict_szip_snd surj_scons surj_scons szip_scons)

lemma szip_sdrop: "sdrop n·(szip·s·t) = szip·(sdrop n·s)·(sdrop n·t)"
  apply(induction n arbitrary: s t, simp)
  by (metis (no_types, lifting) sdrop_forw_rt sdrop_scons stream.sel_rews(2) strict_szip_fst
           strict_szip_snd surj_scons szip_scons)
(* ---------------------------------------------------------------------- *)
subsection ‹@{term sscanlA}›
(* ---------------------------------------------------------------------- *)

lemma sscanla_cont: "cont (λs. sprojfst·(sscanl (λ(_,b). f b) (undefined, s0)·s))"
  by simp

lemma sscanla_len [simp]: "#(sscanlA f s0·s) = #s"
  by (simp add: sscanlA_def slen_sprojfst)

lemma sscanla_bot [simp]: "sscanlA f s0·⊥ = ⊥"
  by (simp add: sscanlA_def)

lemma sscanla_step [simp]: "sscanlA f s0·(↑a • as) = ↑(fst (f s0 a)) • sscanlA f (snd (f s0 a))·as"
  apply(simp add: sscanlA_def sprojfst_def)
proof -
  have "(case f s0 a of (a, x) ⇒ f x) = (case (undefined::'a, snd (f s0 a)) of (a, x) ⇒ f x)"
by (metis (no_types) old.prod.case prod.collapse)
  then have "↑(shd as) • srt·as = as⟶sscanl (λ(a, y). f y) (f s0 a)·as = sscanl (λ(a, y). f y) (undefined, snd
       (f s0 a))· (↑(shd as) • srt·as)"
    by (metis (no_types) sscanl_scons)
```

```
  then show "↑(fst (f s0 a)) • smap fst· (sscanl (λ(a, y). f y) (f s0 a)· as) = ↑(fst (f s0 a)) • smap fst·
      (sscanl (λ(a, y). f y) (undefined, snd (f s0 a))· as)"
    using surj_scons by force
qed

lemma sscanla_one [simp]: "sscanlA f b·(↑x) = ↑(fst (f b x))"
  apply(simp add: sscanlA_def)
  by (metis prod.collapse sconc_snd_empty sprojfst_scons strict_sprojfst)

lemma sscanla_shd: "s≠ε⟹shd (sscanlA f q·s) = fst (f q (shd s))"
  by (metis shd1 sscanla_step surj_scons)

lemma sscanla_srt: "srt·(sscanlA f a·s) = sscanlA f (snd (f a (shd s)))·(srt·s)"
  by (metis (no_types, lifting) sconc_fst_empty sconc_scons' sscanla_bot sscanla_step stream.sel_rews(2)
      stream.sel_rews(5) sup'_def surj_scons up_defined)

lemma sscanla_ntimes_loop:
  assumes "⋀r. sscanlA f state·(s • r) = out • (sscanlA f state·r)"
  shows "sscanlA f state·(sntimes n s) = sntimes n out"
  using assms
  by (induction n, simp_all)

lemma sscanla_inftimes_loop:
  assumes "⋀r. sscanlA f state·(s • r) = out • (sscanlA f state·r)"
  shows "sscanlA f state·(sinftimes s) = sinftimes out"
  using assms
  by (metis rek2sinftimes sinftimes_unfold slen_empty_eq sscanla_bot sscanla_len strict_icycle)

(* ----------------------------------------------------------------------- *)
subsection ‹@{term sscanlAg}›
(* ----------------------------------------------------------------------- *)

lemma sscanlag_cont: "cont (λs. (sscanl (λ(b,_). f b) (s0, undefined)·s))"
  by simp

lemma sscanlag_len [simp]: "#(sscanlAg f s0·s) = #s"
  by(simp add: sscanlAg_def slen_sprojsnd)

lemma sscanlag_bot [simp]: "sscanlAg f s0·⊥ = ⊥"
  by (simp add: sscanlAg_def)


lemma sscanlag_one [simp]: "sscanlAg f b·(↑x) = ↑(fst(f b x), snd (f b x))"
  by(simp add: sscanlAg_def)

lemma sscanlag_step [simp]: "sscanlAg f s0·(↑a • as) = ↑((f s0 a)) • sscanlAg f (fst (f s0 a))·as"
  apply(simp add: sscanlAg_def sprojfst_def)
  by (smt case_prod_conv prod.collapse sscanl_empty sscanl_scons surj_scons)

lemma sscanlag_shd: "s≠ε⟹shd (sscanlAg f q·s) = (f q (shd s))"
  by (metis shd1 sscanlag_step surj_scons)

lemma sscanlag_srt: "srt·(sscanlAg f a·s) = sscanlAg f (fst (f a (shd s)))·(srt·s)"
  by (metis (no_types, lifting) sconc_fst_empty sconc_scons' sscanlag_bot sscanlag_step stream.sel_rews(2)
      stream.sel_rews(5) sup'_def surj_scons up_defined)

lemma snth_sscanlAg:
  assumes "Fin (Suc j)<#i"
    shows "snth (Suc j) (sscanlAg f s0·i) = f (fst (snth j (sscanlAg f s0·i))) (snth (Suc j) i)"
  apply(simp add: sscanlAg_def)
  by (metis (mono_tags, lifting) assms case_prod_conv prod.exhaust_sel sscanl_snth)

lemma sscanlag_ntimes_loop:
  assumes "⋀r. sscanlAg f state·(s • r) = out • (sscanlAg f state·r)"
  shows "sscanlAg f state·(sntimes n s) = sntimes n out"
  using assms
  by (induction n, simp_all)

lemma sscanlag_inftimes_loop:
  assumes "⋀r. sscanlAg f state·(s • r) = out • (sscanlAg f state·r)"
  shows "sscanlAg f state·(sinftimes s) = sinftimes out"
  using assms
  by (metis rek2sinftimes sinftimes_unfold slen_empty_eq sscanlag_bot sscanlag_len strict_icycle)

(* ----------------------------------------------------------------------- *)
subsection ‹@{term sscanlAfst}›
(* ----------------------------------------------------------------------- *)

lemma sscanlafst_cont: "cont (λs. sprojfst·(sscanlAg (λ(_,b). f b) (undefined, s0)·s))"
  by simp

lemma sscanlafst_len [simp]: "#(sscanlAfst f s0·s) = #s"
  by (simp add: sscanlAfst_def slen_sprojfst)

lemma sscanlafst_bot [simp]: "sscanlAfst f s0·⊥ = ⊥"
  by (simp add: sscanlAfst_def)

lemma sscanlafst_step [simp]: "sscanlAfst f s0·(↑a • as) = ↑(fst (f s0 a)) • sscanlAfst f (fst (f s0 a))·as"
  apply (simp add: sscanlAfst_def sprojfst_def sscanlAg_def)
  by (smt approxl2 case_prod_conv eta_cfun fair_sscanl fin2stake prod.collapse sconc_prefix sconc_snd_empty
      slen_scons slen_smap smap_hd_rst sprojfst_def sscanl_empty sscanl_shd sscanl_srt strict_sprojfst)

lemma sscanlafst_one [simp]: "sscanlAfst f b·(↑x) = ↑(fst (f b x))"
  apply(simp add: sscanlAfst_def)
```

```
  by (metis prod.collapse sconc_snd_empty sprojfst_scons strict_sprojfst)

lemma sscanlafst_shd: "s≠ϵ⟹shd (sscanlAfst f q·s) = fst (f q (shd s))"
  by (metis shd1 sscanlafst_step surj_scons)

lemma sscanlafst_srt: "srt·(sscanlAfst f a·s) = sscanlAfst f (fst (f a (shd s)))·(srt·s)"
  by (metis (no_types, lifting) sconc_fst_empty sconc_scons' sscanlafst_bot sscanlafst_step stream.sel_rews(2)
      stream.sel_rews(5) sup'_def surj_scons up_defined)

lemma sscanlafst_snth: assumes "Fin (Suc n) < #s" and "s2 = fst (shd (sscanlAg f state·s))"
  shows "snth (Suc n) (sscanlAfst f state·s) = snth n (sscanlAfst f s2·(srt·s))"
  apply(simp add: assms sscanlAfst_def)
  apply(subst sprojfst_snth)
  apply(simp add: assms)
  apply (meson assms(1) leD lel slen_rt_ile_eq)
  apply(subst snth_sscanlAg)
  apply(simp add: assms)
  by (metis (no_types, lifting) assms(1) empty_is_shortest shd1 snth_scons snth_sscanlAg sscanlag_step surj_scons)

lemma sscanlafst_ntimes_loop:
  assumes "⋀r. sscanlAfst f state·(s ● r) = out ● (sscanlAfst f state·r)"
  shows "sscanlAfst f state·(sntimes n s) = sntimes n out"
  using assms
  by (induction n, simp_all)

lemma sscanlafst_inftimes_loop:
  assumes "⋀r. sscanlAfst f state·(s ● r) = out ● (sscanlAfst f state·r)"
  shows "sscanlAfst f state·(sinftimes s) = sinftimes out"
  using assms
  by (metis rek2sinftimes sinftimes_unfold slen_empty_eq sscanlafst_bot sscanlafst_len strict_icycle)

(* ---------------------------------------------------------------------- *)
subsection ‹@{term sscanlAsnd}›
(* ---------------------------------------------------------------------- *)

lemma sscanlasnd_cont: "cont (λs. sprojsnd·(sscanl (λ(b,_). f b) (s0, undefined)·s))"
  by simp

lemma sscanlasnd_len [simp]: "#(sscanlAsnd f s0·s) = #s"
  by(simp add: sscanlAsnd_def slen_sprojsnd)

lemma sscanlasnd_bot [simp]: "sscanlAsnd f s0·⊥ = ⊥"
  by (simp add: sscanlAsnd_def)

lemma sscanlasnd_step [simp]: "sscanlAsnd f s0·(↑a ● as) = ↑(snd (f s0 a)) ● sscanlAsnd f (fst (f s0 a))·as"
  apply(simp add: sscanlAsnd_def sprojsnd_def sscanlAg_def)
proof -
  have "(case f s0 a of (x, a) ⇒ f x) = (case (fst (f s0 a), undefined::'a) of (x, a) ⇒ f x)"
    by (metis (no_types) old.prod.case prod.collapse)
  then have "↑(shd as) ● srt·as = as⟶⊬sscanl (λ(b, uu). f b) (f s0 a)·as)= (sscanl (λ(b, uu). f b) (fst (f s0
      a), undefined)·as)"
    by (metis (no_types) sscanl_scons)
  then show "↑(snd (f s0 a)) ● smap snd·(sscanl (λ(b, uu). f b) (f s0 a)·as) = ↑(snd (f s0 a)) ● smap snd·(sscanl
      (λ(b, uu). f b) (fst (f s0 a), undefined)·as)"
    using surj_scons by force
qed

lemma sscanlasnd_one [simp]: "sscanlAsnd f b·(↑x) = ↑(snd (f b x))"
  apply(simp add: sscanlAsnd_def)
  by (metis eq_snd_iff sconc_snd_empty sprojsnd_scons strict_sprojsnd)

lemma sscanlasnd_shd: "s≠ϵ⟹shd (sscanlAsnd f q·s) = snd (f q (shd s))"
  by (metis shd1 sscanlasnd_step surj_scons)

lemma sscanlasnd_srt: "srt·(sscanlAsnd f a·s) = sscanlAsnd f (fst (f a (shd s)))·(srt·s)"
  by (metis (no_types, lifting) sconc_fst_empty sconc_scons' sscanlasnd_bot sscanlasnd_step stream.sel_rews(2)
      stream.sel_rews(5) sup'_def surj_scons up_defined)

lemma sscanlasnd_snth: assumes "Fin (Suc n)<#s" and "s2 = fst(shd(sscanlAg f state·s))"
  shows "snth (Suc n) (sscanlAsnd f state·s) = snth n (sscanlAsnd f s2·(srt·s))"
  apply(simp add: assms sscanlAsnd_def)
  apply(subst sprojsnd_snth)
   apply(simp add: assms)
  by (metis assms(1) empty_is_shortest eta_cfun rt_Sproj_2_eq smap_snth_lemma snth_rt sprojsnd_def sscanlag_len
      sscanlag_shd sscanlag_srt)

lemma sscanlasnd_snth2:
  assumes "Fin (Suc n) < #(↑a ● s)"
  shows "snth (Suc n) (sscanlAsnd f state·(↑a ● s)) = snth n (sscanlAsnd f (fst (f state a))·s)"
  using assms
  apply (induction s arbitrary: a rule: ind)
  by simp_all

lemma sscanlasnd_ntimes_loop:
  assumes "⋀r. sscanlAsnd f state·(s ● r) = out ● (sscanlAsnd f state·r)"
  shows "sscanlAsnd f state·(sntimes n s) = sntimes n out"
  using assms
  by (induction n, simp_all)

lemma sscanlasnd_inftimes_loop:
  assumes "⋀r. sscanlAsnd f state·(s ● r) = out ● (sscanlAsnd f state·r)"
  shows "sscanlAsnd f state·(sinftimes s) = sinftimes out"
  using assms
  by (metis rek2sinftimes sinftimes_unfold slen_empty_eq sscanlasnd_bot sscanlasnd_len strict_icycle)
```

```
lemma sscanlasnd2sinftimes:
  assumes "#s=∞"
    and "sscanlAsnd f state·s = out • (sscanlAsnd f state·s)"
    and "out ≠ ε"
  shows "sscanlAsnd f state·s = sinftimes out"
  using assms rek2sinftimes by auto

lemma sscanl2smap:
  assumes "⋀e. fst(f s e) = s"
  and "g = (λa. snd(f s a))"
shows"sscanlAsnd f s = smap g"
  apply(rule cfun_eqI)
  by(induct_tac x rule: ind,simp_all add: assms)

(* ----------------------------------------------------------------------- *)
subsection ‹@{term merge}›
(* ----------------------------------------------------------------------- *)

(* unfolding of merge function *)
lemma merge_unfold: "merge f·(↑x • xs)·(↑y• ys) = ↑(f x y) • merge f·xs·ys"
  by(simp add: merge_def)

(* relation between merge and snth *)
lemma merge_snth[simp]: "Fin n <#xs⟹Fin n < #ys⟹snth n (merge f·xs·ys) = f (snth n xs) (snth n ys)"
  apply(induction n arbitrary:xs ys)
   apply (metis Fin_02bot merge_unfold lnless_def lnzero_def shd1 slen_empty_eq snth_shd surj_scons)
  by (smt Fin_Suc Fin_leq_Suc_leq Suc_eq_plus1_left merge_unfold inject_lnsuc less2eq less2lnleD lnle_conv
       lnless_def lnsuc_lnle_emb sconc_snd_empty sdropostake shd1 slen_scons snth_rt snth_scons split_streaml1
       stream.take_strict surj_scons ub_slen_stake)

(* merge applied to f·ε·ys return the empty stream *)
lemma merge_eps1[simp]: "merge f·ε·ys = ε"
  by(simp add: merge_def)

(* merge applied to f·xs·ε also returns the empty stream *)
lemma merge_eps2[simp]: "merge f·xs·ε = ε"
  by(simp add: merge_def)

(* relation between srt and merge *)
lemma [simp]: "srt·(merge f·(↑a • as)·(↑b • bs)) = merge f·as·bs"
  by (simp add: merge_unfold)

(* the length of merge f·as·bs is the minimum of the lengths of as and bs *)
lemma merge_len [simp]: "#(merge f·as·bs) = min (#as) (#bs)"
by(simp add: merge_def)

(* the merge function is commutative *)
lemma merge_commutative: assumes "⋀ a b. f a b = f b a"
  shows "merge f·as·bs = merge f·bs·as"
  apply(rule snths_eq)
   apply (simp add: min.commute)
  by (simp add: assms)


(* ----------------------------------------------------------------------- *)
subsection ‹@{term siterate}›
(* ----------------------------------------------------------------------- *)


lemma siterate_inv_lemma:
  "∀x z a. #z = #x
    ⟶stake n·(sscanl (λa b. f a) a·x) =
        stake n·(sscanl (λa b. f a) a·z)"
apply (induct_tac n, auto)
apply (rule_tac x=x in scases, auto)
by (rule_tac x=z in scases, auto)


lemma siterate_def2:
  "#x = ∞⟹siterate f a = ↑a • sscanl (λa b. f a) a·x"
apply (subst siterate_def)
apply (rule someI2_ex)
apply (rule_tac x="sinftimes (↑(SOME a. True))" in exI, simp)
apply (rule cfun_arg_cong)
apply (rule stream.take_lemma)
by (rule siterate_inv_lemma [rule_format], simp)


lemma siterate_scons: "siterate f a = ↑a • siterate f (f a)"
apply (rule stream.take_lemma [OF spec [where x="a"]])
apply (induct_tac n, auto)
apply (insert siterate_def2 [of _ f], atomize)
apply (erule_tac x="sinftimes (↑x)" in allE, auto)
by (subst sinftimes_unfold, simp)

(* to define the nth element of siterate we define a helper function ‹niterate› *)
(* ‹iterate› cannot be used, because the function is only about CPO's, maybe some
of those lemmata about niterate could be in Prelude, but not all of them *)
primrec niterate :: "nat ⇒ ('a::type ⇒ 'a) ⇒ ('a ⇒ 'a)" where
    "niterate 0 = (λ F x. x)"
  | "niterate (Suc n) = (λ F x. F (niterate n F x))"

(* niterate applied to the successor of n is the same as applying niterate to n F *)
```

```
lemma niterate_Suc2: "niterate (Suc n) F x = niterate n F (F x)"
apply(induction n)
by(simp_all)

(* relation between iterate and niterate *)
lemma niter2iter: "iterate g·h·x = niterate g (Rep_cfun h) x"
apply (induction g)
by(simp_all)

(* iterate and the empty stream *)
lemma iterate_eps [simp]: assumes "g ϵ = ϵ"
  shows "(iterate i·(Λ h. (λs. s • h (g s)))·⊥) ϵ = ϵ"
using assms by (induction i, auto)

(* fix and the empty stream *)
lemma fix_eps [simp]: assumes "g ϵ = ϵ"
  shows "(μ h. (λs. s • h (g s))) ϵ = ϵ"
proof -
  have al: "max_in_chain 0 (λi. (iterate i·(Λ h. (λs. s • h (g s)))·⊥) ϵ )" by (simp add: max_in_chainI assms)
  hence "(⨆i. (iterate i·(Λ h. (λs. s • h (g s)))·⊥) ϵ) = ϵ"  using assms by auto
  hence " (⨆ i. iterate i·(Λ h. (λs. s • h (g s)))·⊥) ϵ = ϵ" by (simp add: lub_fun)
  thus ?thesis using fix_def2 by (metis (no_types, lifting) lub_eq)
qed

(* beginning the iteration of the function h with the element (h x) is equivalent to beginning the
   iteration with x and dropping the head of the iteration *)
lemma siterate_sdrop: "siterate h (h x) = sdrop 1·(siterate h x)"
by (metis One_nat_def sdrop_0 sdrop_scons siterate_scons)

(* iterating the function h infinitely often after having already iterated i times is equivalent to
   beginning the iteration with x and dropping i elements from the resulting stream *)
lemma siterate_drop2iter: "siterate h (niterate i h x) = sdrop i·(siterate h x)"
apply (induction i)
apply (simp add: One_nat_def)
by (simp add: sdrop_back_rt siterate_sdrop One_nat_def)

(* the head of iterating the function g on x doesn't have any applications of g *)
lemma shd_siter[simp]: "shd (siterate g x) = x"
by (simp add: siterate_def)

(* dropping i elements from the infinite iteration of the function g on x and then extracting the head
   is equivalent to computing the i'th iteration via niterate *)
lemma shd_siters: "shd (sdrop i·(siterate g x)) = niterate i g x"
by (metis shd_siter siterate_drop2iter)


lemma snth_siterate_Suc: "snth k (siterate Suc j) = k + j"
apply (rule_tac x="j" in spec)
apply (induct_tac k, simp)
apply (rule allI)
by (subst siterate_scons, simp)+

(* applying snth to k and siterate Suc 0 returns k *)
lemma snth_siterate_Suc_0[simp]: "snth k (siterate Suc 0) = k"
by (simp add: snth_siterate_Suc)

(* relation between sdrop and siterate *)
lemma sdrop_siterate:
  "sdrop k·(siterate Suc j) = siterate Suc (j + k)"
apply (rule_tac x="j" in spec)
apply (induct_tac k, simp+)
apply (rule allI)
by (subst siterate_scons, simp)


lemma [simp]: "#(siterate f k) = ∞"
apply (rule infI)
apply (rule allI)
apply (rule_tac x="k" in spec)
apply (induct_tac k, simp+)
by (subst siterate_scons, simp)+

(* the i'th element of the infinite stream of iterating the function g on x can alternatively be found
   with (niterate i g x) *)
lemma snth_siter: "snth i (siterate g x) = niterate i g x"
by (simp add: shd_siters snth_def)

(* dropping j elements from the stream x and then extracting the i'th element is equivalent to extracting
   the i+j'th element directly *)
lemma snth_sdrop: "snth i (sdrop j·x) = snth (i+j) x"
by (simp add: sdrop_plus snth_def)

(* extracting the i+1'st element from the stream of iterating the function g on x is equivalent to extracting
   the i'th element and then applying g one more time *)
lemma snth_snth_siter: "snth (Suc i) (siterate g x) = g (snth i (siterate g x))"
by (simp add: snth_siter)

(* dropping the first element from the chain of iterates is equivalent to shifting the chain by applying g *)
lemma sdrop_siter:  "sdrop 1·(siterate g x) = smap g·(siterate g x)"
apply(rule sinf_snt2eq)
apply (simp add: fair_sdrop)
apply simp
by (simp add: smap_snth_lemma snth_sdrop snth_snth_siter One_nat_def)
```

```
(* if the functions g and h commute then g also commutes with any number of iterations of h *)
lemma iterate_insert: assumes "∀z. h (g z) = g (h z)"
  shows "niterate i h (g x) = g (niterate i h x)"
using assms by (induction i, auto)

(* lifts iterate_insert from particular iterations to streams of iterations *)
lemma siterate_smap:  assumes "∀z. g (h z) = h (g z)"
  shows "smap g·(siterate h x) = siterate h (g x)"
apply (rule sinf_snt2eq, auto)
by (simp add: assms smap_snth_lemma snth_siter iterate_insert)

(* iterating the function g on x is equivalent to the stream produced by concatenating ↑x and the
   iteration of g on x shifted by another application of g *)
lemma siterate_unfold: "siterate g x = ↑x • smap g·(siterate g x)"
by (metis siterate_scons siterate_smap)

(* iterating the identity function produces an infinite constant stream of the element x *)
lemma siter2sinf: "siterate id x = sinftimes (↑x)"
by (metis id_apply s2sinftimes siterate_scons)

(* dropping i and iterating the identity function returns siterate id x *)
lemma "sdrop i·(siterate id x) = siterate id x"
  by (smt sdrops_sinf siter2sinf)

(* if g acts as the identity for the element x then iterating g on x produces an infinite constant
   stream of x *)
lemma siter2sinf2: assumes "g x = x"
  shows "siterate g x = sinftimes (↑x)"
by (smt assms s2sinftimes siterate_scons)

(* shows the equivalence of an alternative recursive definition of iteration *)
lemma rek2niter: assumes "xs = ↑x • (smap g·xs)"
  shows "snth i xs = niterate i g x"
proof (induction i)
  case 0 thus ?case by (metis assms niterate.simps(1) shd1 snth_shd)
next
  case (Suc i)
  have "#xs = ∞" by (metis Inf'_def assms below_refl fix_least_below inf_less_eq lnle_def slen_scons slen_smap)
  thus ?case by (metis Fin_neq_inf Suc assms inf_ub lnle_def lnless_def niterate.simps(2) smap_snth_lemma
        snth_scons)
qed

(* important *)
(* recursively mapping the function g over the rest of xs is equivalent to the stream of iterations of g on x *)
lemma rek2siter: assumes "xs = ↑x • (smap g·xs)"
  shows "xs = siterate g x"
apply (rule sinf_snt2eq, auto)
apply (metis Inf'_def assms fix_least inf_less_eq lnle_conv slen_scons slen_smap)
by (metis assms rek2niter snth_siter)

(* shows that siterate produces the least fixed point of the alternative recursive definition *)
lemma fixrek2siter: "fix·(Λ s . (↑x • smap g·s)) =  siterate g x"
by (metis (no_types) cfcomp1 cfcomp2 fix_eq rek2siter)

(* dropping elements from a stream of iterations is equivalent to adding iterations to every element *)
lemma sdrop2smap: "sdrop i·(siterate g x) = smap (niterate i g)·(siterate g x)"
by (simp add: iterate_insert siterate_drop2iter siterate_smap)


(* ---------------------------------------------------------------------- *)
section ⟨Adm simp rules⟩
(* ---------------------------------------------------------------------- *)


lemma adm_subsetEq [simp]: "adm (λs. g·s ⊆ h·s)"
by (metis (full_types) SetPcpo.less_set_def adm_below cont_Rep_cfun2)


lemma adm_subsetEq_rc [simp]: "adm (λs. g·s ⊆ cs)"
by (metis (no_types, lifting) adm_def chain_monofun contlub_cfun_arg lub_below set_cpo_simps(1))


lemma adm_subsetEq_lc [simp]: "adm (λs. cs ⊆ h·s)"
by (simp add: adm_subst adm_superset)


lemma adm_subsetNEq_rc [simp]: "adm (λs. ¬ g·s ⊆ cs)"
  apply(rule admI)
  apply(rule+)
  by (metis SetPcpo.less_set_def is_ub_thelub monofun_cfun_arg subset_eq)


lemma sValues_adm2[simp]: "adm (λa. sValues·(g·a) ⊆ sValues·a)"
apply(rule admI)
by (smt SetPcpo.less_set_def ch2ch_Rep_cfunR contlub_cfun_arg is_ub_thelub lub_below subset_iff)

(* admissibility of finstream *)
lemma adm_finstream [simp]: "adm (λs::'a stream. #s<∞⟶P s)"
apply(rule admI)
  apply auto
  using inf_chainl4 len_stream_def lub_eqI lub_finch2 by fastforce

(* admissibility of fin below *)
lemma adm_fin_below: "adm (λx::'a stream . ¬ Fin n ⊑ # x)"
```

```
  apply(rule admI)
  apply auto
  by (metis inf_chainl3 finite_chain_def maxinch_is_thelub)

(* admissibility of fin below for special case of ⊑ *)
lemma adm_fin_below2: "adm (λx::'a stream . ¬ Fin n ⊑ # x)"
by(simp only: lnle_def adm_fin_below)


(* ----------------------------------------------------------------------- *)
section ‹New @{term sfilter} lemmata and @{term sfoot}›
(* ----------------------------------------------------------------------- *)

(* ----------------------------------------------------------------------- *)
subsection ‹New @{term sfilter} lemmata›
(* ----------------------------------------------------------------------- *)

text‹Appending the singleton stream ↑a increases the length of the stream y by one›
lemma slen_lnsuc:
  shows "#(y • ↑a) = lnsuc·(#y)"
  apply(induction y)
    apply (smt admI fold_inf inf_chainl4 lub_eqI lub_finch2 sconc_fst_inf)
   apply (metis sconc_fst_empty sconc_snd_empty slen_scons)
  by (metis (no_types, lifting) assoc_sconc slen_scons stream.con_rews(2) stream.sel_rews(5) surj_scons)


(* if filtering the stream s2 with the set A produces infinitely many elements then prepending any
   finite stream s1 to s2 will still produce infinitely many elements *)
lemma sfilter_conc2[simp]: assumes "#(sfilter A·s2) = ∞" and "#s1 < ∞"
  shows "#(sfilter A·(s1•s2)) = ∞"
proof -
  have "(sfilter A·(s1•s2)) = ((sfilter A·s1) • (sfilter A·s2))"
    using add_sfilter assms(2) lnless_def ninf2Fin by fastforce
  thus ?thesis by (simp add: assms(1) slen_sconc_snd_inf)
qed

(* if the stream z is a prefix of another non-empty stream (y•↑a) but isn't equal to it, then z is
   also a prefix of y *)
lemma below_conc: assumes "z ⊑ (y•↑a)" and "z≠(y•↑a)"
  shows "z⊑y"
proof(cases "#y = ∞")
  case True thus ?thesis using assms(1) sconc_fst_inf by auto
next
  case False
  obtain n_y where "Fin n_y= #y" using False ninf2Fin by fastforce
  have "#z ≤ #(y•↑a)" using assms(1) mono_slen by blast
  have "#z ≠ #(y•↑a)" using assms(1) assms(2) eq_slen_eq_and_less by blast
  hence "#z < #(y•↑a)" using ‹#z ≤ #(y • ↑a)› lnle_def lnless_def by blast
  obtain n_z where nz_def: "Fin n_z = #z" using approxl2 assms(1) assms(2) by blast
  have "#y < #(y • ↑a)"
    by (metis ‹Fin n_y = #y› len_stream_def eq_slen_eq_and_less inject_sconc lnless_def minimal monofun_cfun_arg
        sconc_snd_empty stream.con_rews(2) sup'_def up_defined)
  have "#y < ∞" by (simp add: False lnless_def)
  hence "Fin n_z ≤ #y" by (metis Fin_Suc ‹#z < #(y • ↑a)› less2lnleD lnsuc_lnle_emb nz_def slen_lnsuc)
  have "⋀s. stake n_y·(y • s) = y" by (simp add: ‹Fin n_y = #y› approxl1)
  hence "stake n_z·y = stake n_z·(y•↑a)" by (metis ‹Fin n_y = #y› ‹Fin n_z ≤ #y› less2nat min_def stakeostake)
  thus ?thesis by (metis ‹Fin n_z = #z› approxl1 assms(1) stream.take_below)
qed

(* for any set A and singleton stream ↑a the following predicate over streams is admissible *)
lemma sfilter_conc_adm: "adm (λb. #b<∞⟶#(A ⊖ b) < #(A ⊖ b • ↑a))" (is "adm ?F")
by (metis (mono_tags, lifting) admI inf_chainl4 lnless_def lub_eqI lub_finch2)

(* the element a is kept when filtering with A, so (x • ↑a) produces a larger result than just x,
   provided that x is finite *)
lemma sfilter_conc: assumes "a∈A"
  shows "#x<∞⟹#(A ⊖ x) < #(A ⊖ (x • ↑a))" (is "_⟹?F x")
  proof(induction x)
    show "adm (λb. #b < ∞⟶#(A ⊖ b) < #(A ⊖ b • ↑a))" using sfilter_conc_adm by blast
   show "?F ε" using assms(1) lnless_def by auto
  next
  fix u :: "'a discr u"
  fix s :: "'a stream"
  assume "u≠⊥" and "#s<∞⟹?F s" and "#(u && s) < ∞"
  obtain ua where "(updis ua) = u" by (metis ‹u ≠ ⊥› discr.exhaust upE)
  hence "u && s = ↑ua •s" using lscons_conv by blast
  thus "?F (u&&s)"
  by (smt ‹#(u && s) < ∞› ‹#s < ∞⟹#(A ⊖ s) < #(A ⊖ s • ↑a)› assoc_sconc fold_inf lnat.sel_rews(2) lnless_def
       monofun_cfun_arg sfilter_in sfilter_nin slen_scons)
qed

(* for any finite stream s and set A, if filtering s with A doesn't produce the empty stream, then
   filtering and infinite repetition are associative *)
lemma sfilter_sinf [simp]: assumes "#s<∞" and "(A ⊖ s) ≠ ε"
  shows "A ⊖ (s^∞) = ((A ⊖ s)^∞)"
by (metis add_sfilter assms(1) assms(2) infI lnless_def rek2sinftimes sinftimes_unfold)

(* if filtering the stream s with the set A produces infinitely many elements, then filtering the
   rest of s with A also produces infinitely many elements *)
lemma sfilter_srt_sinf [simp]: assumes "#(A ⊖ s) = ∞"
  shows  "#(A ⊖ (srt·s)) = ∞"
by (smt assms inf_scase inject_scons sfilter_in sfilter_nin stream.sel_rews(2) surj_scons)

(* additional snth--lemma *)
```

```
lemma sfilter_ntimes [simp]: "#({True} ⊖ ((sntimes n (↑False))•ora2)) = #({True} ⊖ ora2)"
  apply(induction n)
    by auto

(* snth to sntimes *)
lemma snth2sntimes: "(⋀i. i<n⟹snth i s = False)⟹Fin n < #s⟹(sntimes n (↑False)) ⊑ s"
proof(induction n arbitrary: s)
  case 0
  then show ?case by auto
next
  case (Suc n)
    have "shd s = False"
      using Suc.prems snth_shd by blast
     hence "s = ↑False • (srt·s)"
       using Suc.prems(2) empty_is_shortest surj_scons by force
    have "(⋀i. i < n⟹snth i (srt·s) = False)"
      using Suc.prems snth_rt by auto
     hence "n⋆↑False ⊑ (srt·s)"
       by (meson Suc.IH Suc.prems(2) linorder_not_le slen_rt_ile_eq)
     hence "↑False • (n⋆↑False) ⊑ ↑False • (srt·s)"
       by (simp add: monofun_cfun_arg)
     then show ?case
       using ⟨s = ↑False • srt·s⟩ by auto
qed

(* length of sntimes *)
lemma sntimes_len [simp]: "#(n⋆↑a) = Fin n"
  apply(induction n)
  by auto

(* if length of a stream is Fin n, then snth (n+m) (xs•ys) is equal to snth m ys *)
lemma snth_scons2: assumes "#xs = Fin n"
  shows "snth (n+m) (xs•ys) = snth m ys"
  apply(simp add: snth_def)
  by (simp add: add.commute assms sdrop_plus sdropl6)

(* if ({True} ⊖ s) is unequal to bottom, then (sntimes n (↑False)) • ↑True is a prefix of s *)
lemma sbool_ntimes_f: assumes "({True} ⊖ s) ≠⊥"
  obtains n where "(sntimes n (↑False)) • ↑True ⊑ s"
proof -
  have h1: "∃i. snth i s = True"
    by (metis assms ex_snth_in_sfilter_nempty singletonD)
  obtain n where "Fin n < #s" and n_def: "n = Least (λi. snth i s = True)"
    by (smt assms ex_snth_in_sfilter_nempty less2nat not_less not_less_Least not_less_iff_gr_or_eq singletonD
         trans_lnless)
  have "snth n s = True"
    using LeastI h1 n_def by force
  have "⋀i. i<n⟹snth i s ≠ True" using not_less_Least n_def by fastforce
  hence"⋀i. i<n⟹snth i s = False" by auto
  hence "(sntimes n (↑False)) ⊑ s"
    using ⟨Fin n < #s⟩ snth2sntimes by blast
  obtain xs where xs_def: "s = (sntimes n (↑False)) • xs"
    using ⟨n⋆↑False ⊑ s⟩ approxl3 by auto
  hence "xs≠⊥"
    by (metis assms sfilter_ntimes slen_empty_eq strict_sfilter)
  have "shd xs = snth n s"
    by (simp add: sdropl6 snth_def xs_def)
  hence "shd xs = True"
    using ⟨snth n s = True⟩ by auto
  thus ?thesis
    by (metis (full_types)⟨xs ≠ ε⟩ lscons_conv minimal monofun_cfun_arg sup'_def surj_scons that xs_def)
  qed

(* ------------------------------------------------------------------------ *)
section ⟨@{term sfoot}⟩
(* ------------------------------------------------------------------------ *)

(* appending the singleton stream ↑a to a finite stream s causes sfoot to extract a again *)
lemma sfoot1[simp]: assumes "xs = s•(↑a)" and "#xs < ∞"
   shows "sfoot xs = a"
proof -
  have "xs ≠ ε" using assms(1) strictI by force
  obtain n where n_def: "Fin n = #xs" by (metis assms(2) lncases lnless_def)
  hence "n>0" using assms(2) gr0I using Fin_02bot ⟨xs ≠ ε⟩ lnzero_def slen_empty_eq by fastforce
  hence "(THE n'. lnsuc·(Fin n') = #xs) = n-1" by (metis (mono_tags, lifting) Fin_Suc Suc_diff_1 n_def inject_Fin
       inject_lnsuc the_equality)
  have "snth (n-1) xs = a" by (metis Fin_0 Fin_Suc Suc_diff_1 assms(1) n_def bot_is_0 inject_lnsuc lnat.con_rews
       lscons_conv neq0_conv sdropl6 shd1 slen_lnsuc snth_def sup'_def)
  thus ?thesis by (metis ⟨(THE n'. lnsuc·(Fin n') = #xs) = n - 1⟩ sfoot_def)
qed

(* sfoot extracts the element a from any finite stream ending with ↑a *)
lemma sfoot12 [simp]: assumes "#s<∞"
  shows "sfoot (s•↑a) = a"
by (metis assms fold_inf inject_lnsuc lnless_def monofun_cfun_arg sfoot1 slen_lnsuc)

(* sfoot extracts a from the singleton stream ↑a *)
lemma sfoot_one [simp]: "sfoot (↑a) = a"
by (metis Inf'_neq_0 inf_ub lnle_def lnless_def sconc_fst_empty sfoot12 strict_slen)

(* concatenating finite streams produces another finite stream *)
lemma sconc_slen [simp]: assumes "#s<∞" and "#xs<∞"
  shows "#(s•xs) < ∞"
by (metis Fin_neq_inf assms(1) assms(2) infI inf_ub lnle_def lnless_def slen_sconc_all_finite)
```

```
lemma sconc_slen2: "⋀(s1::'a stream) s2::'a stream. #(s1 • s2) = #s1 + #s2"
proof -
  fix s1::"'a stream"
  fix s2::"'a stream"

  have "#s1 = ∞⟹#(s1 • s2) = #s1 + #s2"
    by (simp add: plus_lnatInf_r)
  moreover
  have "#s2 = ∞⟹#(s1 • s2) = #s1 + #s2"
    by (simp add: slen_sconc_snd_inf)
  moreover
  have "#s1 ≠∞⟹#s2 ≠∞⟹#(s1 • s2) = #s1 + #s2"
  proof-
    assume "#s1 ≠∞"
    then obtain l_s1 where l_s1_def: "Fin l_s1 = #s1"
      using infI by metis
    assume " #s2 ≠∞"
    then obtain l_s2 where l_s2_def: "Fin l_s2 =  #s2"
      using infI by metis
    show ?thesis
      using l_s1_def l_s2_def by (metis lnat_plus_fin slen_sconc_all_finite)
  qed

  then show "#(s1 • s2) = #s1 + #s2"
    using calculation by linarith
qed

(* if the foot of a non-empty stream xs is a, then xs consists of another stream s (possibly empty)
   concatenated with ↑a *)
lemma sfoot2 [simp]: assumes "sfoot xs = a" and "xs≠ϵ"
  shows "∃s. xs = s • ↑a"
proof (cases "#xs = ∞")
  case True thus ?thesis by (metis sconc_fst_inf)
next
  case False
  obtain n where "#xs = Fin n" using False lncases by auto
  hence "(THE n'. lnsuc·(Fin n') = #xs) = n-1"
    by (smt Fin_02bot Fin_Suc Suc_diff_1 assms(2) bot_is_0 inject_Fin inject_lnsuc neq0_conv slen_empty_eq
         the_equality)
  have "stake (n-1)·xs • ↑(snth (n-1) xs) = xs"
    by (smt Fin_0 Fin_Suc Inf'_def Suc_diff_1 ⟨#xs = Fin n⟩ assms(2) fin2stake fix_least_below lnle_def neq0_conv
         notinfI3 sconc_snd_empty sdrop_back_rt sdropostake slen_empty_eq snth_def split_streamI1 surj_scons
         ub_slen_stake)
  thus ?thesis by (metis ⟨(THE n'. lnsuc·(Fin n') = #xs) = n - 1⟩ assms(1) sfoot_def)
qed

(* when sfoot is applied to the concatenation of two finite streams s and xs, and xs is not empty,
   then sfoot will produce the foot of xs *)
lemma sfoot_conc [simp]: assumes "#s<∞" and "#xs<∞" and "xs≠ϵ"
  shows "sfoot (s•xs) = sfoot xs"
by (metis (no_types, hide_lams) assms(1) assms(2) assms(3) assoc_sconc sconc_slen sfoot1 sfoot2)

(* if the finite stream s contains more than one element then the foot of s will be the foot of the
   rest of s *)
lemma sfoot_sdrop: assumes "Fin 1<#s" and "#s<∞"
  shows "sfoot (srt·s) = sfoot s"
proof -
  obtain s' where "s = s' • ↑(sfoot s)" by (metis assms(1) below_antisym bot_is_0 lnless_def minimal sfoot2
        strict_slen)
  hence "s' ≠ ϵ"
    by (metis Fin_02bot Fin_Suc One_nat_def assms(1) lnless_def lnzero_def slen_lnsuc strict_slen)
  hence "srt·s = srt·s' • ↑(sfoot s)"
    by (smt ⟨s = s' • ↑(sfoot s)⟩ assoc_sconc inject_scons sconc_snd_empty strictI surj_scons)
  thus ?thesis
    by (metis ⟨s = s' • ↑(sfoot s)⟩ ⟨s' ≠ ϵ⟩ assms(2) inf_ub lnle_conv lnless_def sconc_snd_empty sfoot1
         slen_sconc_snd_inf strictI surj_scons)
qed

(* if length of xs is finite, then it holds that sfoot (↑a • ↑b • xs) = sfoot (↑b • xs) *)
lemma [simp]: assumes "#xs < ∞"
  shows "sfoot (↑a • ↑b • xs) = sfoot (↑b • xs)"
using assms lnless_def by auto

(* the foot of any stream s is the nth element of s for some natural number n *)
lemma sfoot_exists [simp]:"∃n. snth n s = sfoot s"
by (metis sfoot_def)

(* if the stream s contains n+1 elements then the foot of s will be found at index n *)
lemma  sfoot_exists2:
  shows "Fin (Suc n) = #s⟹snth n s = sfoot s"
  apply(induction n arbitrary: s)
   apply (metis (mono_tags, lifting) Fin_02bot Fin_Suc Zero_lnless_infty inject_lnsuc lnzero_def sconc_fst_empty
        sconc_snd_empty sfoot12 slen_empty_eq slen_scons snth_shd surj_scons)
  by (smt Fin_Suc Fin_neq_inf fold_inf inf_ub inject_lnsuc lnat.con_rews lnle_conv lnless_def lnzero_def
       sconc_snd_empty sfoot_conc slen_scons snth_rt strict_slen surj_scons)


lemma add_sfilter2: assumes "#x < ∞"
  shows "sfilter A·(x • y) = sfilter A·x • sfilter A·y"
by (metis (no_types) add_sfilter assms lncases lnless_def)

(* if lenght of s is Fin (Suc n), then it holds that (stake n·s) • ↑(sfoot s) = s *)
lemma sfood_id: assumes"#s = Fin (Suc n)"
```

```
  shows "(stake n·s) • ↑(sfoot s) = s"
  using assms apply(induction n arbitrary: s)
   apply simp
   apply (metis Fin_02bot Fin_Suc lnat.sel_rews(2) lnsuc_neq_0_rev lnzero_def lscons_conv sfoot_exists2
         slen_scons snth_shd strict_slen sup'_def surj_scons)
  apply (subst stake_suc)
  apply simp
  by (smt Fin_02bot Fin_Suc One_nat_def Rep_cfun_strict1 Zero_not_Suc lel lnat.sel_rews(2) lnle_Fin_0 lnzero_def
        notinfl3 sconc_snd_empty sfoot_sdrop slen_rt_ile_eq slen_scons stake_Suc stream.take_0 strict_slen
        surj_scons)

lemma footind: "#s = Fin k⟹P ε⟹(⋀t a. #t < ∞⟹P t⟹P (t • ↑a))⟹P s"
proof -
  assume "#s = Fin k" and "P ε" and "⋀t a. #t < ∞⟹P t⟹P (t • ↑a)"
  then show "P s"
  proof (induction k arbitrary: s rule: less_induct)
    case (less k)
    then have IH: "⋀t. #t < Fin k⟹P ε⟹(⋀u b. #u < ∞⟹P u⟹P (u • ↑b))⟹P t"
      by (meson lnat_well_h1)
    then show ?case
    proof (cases "k = 0")
      case True
      then show ?thesis
        using less.prems(1) less.prems(2) by force
    next
      case False
      then obtain u b where "s = u • ↑b"
        using less.prems(1) sfoot2 by force
      then have "#u < Fin k"
        by (metis fold_inf lel less.prems(1) ln_less lnsuc_lnle_emb notinfl3 slen_lnsuc)
      then show ?thesis
        by (metis IH ⟨s = u • ↑b⟩ inf_ub less.prems(2) less.prems(3) less_le not_less)
    qed
  qed
qed

lemma footind2: "#s < ∞⟹P ε⟹(⋀t a. #t < ∞⟹P t⟹P (t • ↑a))⟹P s"
  by (metis footind infI less_le)

lemma srcdups_sfoot:
  "s ≠ ε⟹#s < ∞⟹sfoot (srcdups·s) = sfoot s"
proof (rule footind2, simp+)
  fix t :: "'a stream" and a :: 'a
  assume "#t < ∞" and "sfoot (srcdups·t) = sfoot t"
  show "sfoot (srcdups·(t • ↑a)) = a"
  proof (rule footind2 [of t], auto, simp add: ⟨#t < ∞⟩)
    fix t :: "'a stream" and b :: 'a
    assume "#t < ∞" and "sfoot (srcdups·(t • ↑a)) = a"
    then show "sfoot (srcdups·(t • ↑b • ↑a)) = a"
    proof -
      have "⋀l. l < ∞ ∨ l = ∞"
        by (meson inf_less_eq le_less_linear)
      then show ?thesis
        by (metis (no_types) Fin_Suc ⟨#t < ∞⟩ ⟨sfoot (srcdups·(t • ↑a)) = a⟩ lnat_well_h2 notinfl3 sfoot12
              slen_lnsuc srcdups_end_eq srcdups_end_neq srcdups_slen)
    qed
  qed
qed

lemma sfoot_end:
  fixes s and a
  assumes "#s < ∞" and "s ≠ ε"
  shows "∃t n. s = t • (sntimes n (↑(sfoot s))) ∧ (t = ε ∨ sfoot s ≠ sfoot t)"
  apply (rule footind2)
  apply (simp add: assms(1))
   apply (metis sconc_fst_empty sntimes.simps(1))
proof -
  fix t :: "'a stream" and a :: 'a
  assume "#t < ∞" and "∃ta n. t = ta • (sntimes n (↑(sfoot t))) ∧ (ta = ε ∨ sfoot t ≠ sfoot ta)"
  then obtain u n where "t = u • (sntimes n (↑(sfoot t)))" and "u = ε ∨ sfoot t ≠ sfoot u"
    by blast
  then have expr: "t • ↑a = u • (sntimes n (↑(sfoot t))) • ↑a"
    by (metis assoc_sconc)
  then show "∃ta n. ((t • ↑a) =(ta • (n⋆(↑(sfoot (t • ↑a)))))) ∧ ((ta = ε) ∨ (sfoot (t • ↑a) ≠ sfoot ta))"
  proof (cases "a = sfoot t")
    case True
    then have "t • ↑a = u • (sntimes (Suc n) (↑(sfoot t)))"
      by (metis expr sntimes_Suc2)
    then show ?thesis
      by (metis True ⟨#t < ∞⟩ ⟨u = ε ∨ sfoot t ≠ sfoot u⟩ sfoot12)
  next
    case False
    then have "t • ↑a = (u • (sntimes n (↑(sfoot t)))) • (sntimes (Suc 0) (↑(sfoot (t • ↑a))))"
      using ⟨#t < ∞⟩ ⟨t = u • (n⋆↑(sfoot t))⟩ by auto
    then show ?thesis
      by (metis False ⟨#t < ∞⟩ ⟨t = u • (n⋆↑(sfoot t))⟩ sfoot12)
  qed
qed

lemma srcdups_split_fin: "#s = Fin k⟹Suc n < k⟹snth n s ≠ snth (Suc n) s⟹srcdups·s = srcdups·(stake (Suc
    n)·s) • (srcdups·(sdrop (Suc n)·s))"
proof (induction k arbitrary: n s)
  case 0
  then show ?case
```

```
    by auto
next
  case (Suc k)
  then obtain a t where "s = ↑a • t"
    by (metis Fin_0 Suc_neq_Zero strict_slen surj_scons)
  assume "snth n s ≠ snth (Suc n) s"
  then have "Suc n < Suc k"
    using Suc.prems(2) by blast
  then show ?case
  proof (cases "k > 1")
    case True
    then obtain a b t where "s = ↑a • ↑b • t"
    proof -
      assume a1: "⋀a b t. s = ↑a • ↑b • t⟹thesis"
      have "#(ϵ::'a stream) = ⊥"
        using bot_is_0 by auto
      then show ?thesis
        using a1 by (metis Fin_Suc Suc.prems(1) True inject_lnsuc less2nat_lemma lnat.con_rews lnle_def minimal
            not_le srt_decrements_length surj_scons)
    qed
    then have "t ≠ ϵ"
      using Suc.prems(1) True by force
    then show ?thesis
    proof (cases "n = 0")
      case True
      then have "a ≠ b"
        using Suc.prems(3) ⟨s = ↑a • ↑b • t⟩ by auto
      then show ?thesis
        using True ⟨s = ↑a • ↑b • t⟩ by auto
    next
      case False
      then obtain m where "n = Suc m"
        using not0_implies_Suc by blast
      have "#(↑b • t) = Fin k"
        using Suc.prems(1) ⟨s = ↑a • ↑b • t⟩ by auto
      moreover have "snth n s = snth m (↑b • t)"
        by (simp add: ⟨n = Suc m⟩ ⟨s = ↑a • ↑b • t⟩)
      ultimately have "srcdups·(↑b • t) = srcdups·(stake n·(↑b • t)) • (srcdups·(sdrop n·(↑b • t)))"
        by (metis Suc.IH Suc.prems(2) Suc.prems(3) Suc_less_SucD ⟨n = Suc m⟩ ⟨s = ↑a • ↑b • t⟩ snth_scons)
      then show ?thesis
      proof (cases "srcdups·s = srcdups·(↑b • t)")
        case True
        then show ?thesis
        proof -
          have "a = b"
            by (metis (no_types) True ⟨s = ↑a • ↑b • t⟩ srcdups_shd)
          then show ?thesis
            by (simp add: ⟨n = Suc m⟩ ⟨s = ↑a • ↑b • t⟩ ⟨srcdups·(↑b • t) = srcdups· (stake n· (↑b • t)) •
                srcdups· (sdrop n·(↑b • t))⟩)
        qed
      next
        case False
        then show ?thesis
        proof -
          have "a ≠ b"
            using False ⟨s = ↑a • ↑b • t⟩ srcdups_eq2 by blast
          then show ?thesis
            by (simp add: ⟨n = Suc m⟩ ⟨s = ↑a • ↑b • t⟩ ⟨srcdups·(↑b • t) = srcdups· (stake n· (↑b • t)) •
                srcdups· (sdrop n·(↑b • t))⟩)
        qed
      qed
    qed
  next
    case False
    then have "k ≤ 1"
      by auto
    then show ?thesis
    proof (cases "k = 0")
      case True
      then show ?thesis
        using Suc.prems(2) by blast
    next
      case False
      then have "k = 1"
        using ⟨k ≤ 1⟩ by auto
      then obtain a b where "s = ↑a • ↑b"
        by (metis One_nat_def Rep_cfun_strict1 Suc.prems(1) ⟨⋀thesis. (⋀a t. s = ↑a • t⟹thesis)⟹thesis⟩
            sconc_snd_empty sfood_id stake_Suc stream.take_0)
      then have "a ≠ b"
        using Suc.prems(2) Suc.prems(3) ⟨k = 1⟩ by auto
      then have "srcdups·s = ↑a • ↑b"
        by (metis ⟨s = ↑a • ↑b⟩ lscons_conv srcdups_neq srcdups_step strict_sdropwhile strict_srcdups sup'_def)
      then show ?thesis
        using Suc.prems(2) ⟨k = 1⟩ ⟨s = ↑a • ↑b⟩ by auto
    qed
  qed
qed

lemma srcdups_split_inf: "#s = ∞⟹snth n s ≠ snth (Suc n) s⟹srcdups·s = srcdups·(stake (Suc n)·s) •
    (srcdups·(sdrop (Suc n)·s))"
proof (induction n arbitrary: s)
  case 0
  then obtain a b t where "s = ↑a • ↑b • t" and "a ≠ b"
    by (metis inf_scase shd1 snth_scons snth_shd)
```

```
  then show ?case
    by auto
next
  case (Suc n)
  obtain a t where "s = ↑a • t"
    using Suc.prems(1) inf_scase by blast
  then have "#t = ∞"
    using Suc.prems(1) by auto
  moreover have "snth n t ≠ snth (Suc n) t"
    using Suc.prems(2) ⟨s = ↑a • t⟩ by auto
  ultimately have "srcdups·t = srcdups·(stake (Suc n)·t) • (srcdups·(sdrop (Suc n)·t))"
    using Suc.IH by blast
  then show ?case
  proof (cases "a = shd t")
    case True
    then have "srcdups·s = srcdups·t"
      by (metis Inf'_neq_0 ⟨#t = ∞⟩ ⟨s = ↑a • t⟩ slen_empty_eq srcdups_eq surj_scons)
    then show ?thesis
      using True ⟨#t = ∞⟩ ⟨s = ↑a • t⟩ ⟨srcdups·t = srcdups· (stake (Suc n)·t) • srcdups·(sdrop (Suc n)·t)⟩
            inf_scase by fastforce
  next
    case False
    then have "srcdups·s = ↑a • srcdups·t"
      by (metis Inf'_neq_0 ⟨#t = ∞⟩ ⟨s = ↑a • t⟩ slen_empty_eq srcdups_neq surj_scons)
    then show ?thesis
      using False ⟨#t = ∞⟩ ⟨s = ↑a • t⟩ ⟨srcdups·t = srcdups· (stake (Suc n)·t) • srcdups·(sdrop (Suc n)·t)⟩
            inf_scase by fastforce
  qed
qed

lemma srcdups_split2: "Fin (Suc n) < #s⟹snth n s ≠ snth (Suc n) s⟹srcdups·s = srcdups·(stake (Suc n)·s) •
    (srcdups·(sdrop (Suc n)·s))"
  by (metis less2nat ninf2Fin not_le srcdups_split_fin srcdups_split_inf)

(* ----------------------------------------------------------------------- *)
subsection ⟨@{term sValues}⟩
(* ----------------------------------------------------------------------- *)

(* sValues equality rule *)
lemma sValues_eq: "{z. ∃n. Fin n < #s ∧ z = snth n s} = {snth n s |n. Fin n < #s}"
by auto

(* another sValues equality rule *)
lemma sValues_eq2: "{snth n s |n. Fin n < #s} = {z. ∃n. Fin n < #s ∧ z = snth n s}"
by auto

lemma slen_snth_prefix: "#s > Fin n⟹snth n s = snth n (s • t)"
  by (simp add: monofun_cfun_arg snth_less)

lemma srcdups_sconc:
  "#xs < ∞⟹xs ≠ ε⟹
    srcdups·(xs • ys) = (srcdups·xs) • (srcdups·(sdropwhile (λx. x=sfoot xs)·ys))"
proof -
  assume "#xs < ∞" and "xs ≠ ε"
  then obtain t n where "xs = t • (sntimes n (↑(sfoot xs)))" and "t = ε ∨ sfoot t ≠ sfoot xs"
    using sfoot_end by fastforce
  then show ?thesis
  proof (cases "t = ε")
    case True
    then have "xs • ys = (sntimes n (↑(sfoot xs))) • ys"
      using ⟨xs = t • (n⋆↑(sfoot xs))⟩ by auto
    then have "srcdups·(xs • ys) = (↑(sfoot xs)) • srcdups·(sdropwhile (λx. x=sfoot xs)·ys)"
      by (metis Fin_02bot True ⟨xs = t •( n⋆↑(sfoot xs))⟩ ⟨xs ≠ ε⟩ lnzero_def neq0_conv sconc_fst_empty
            slen_empty_eq sntimes_len srcdups_sntimes_prefix)
    then show ?thesis
      by (metis Fin_02bot True ⟨xs = t • (n⋆↑(sfoot xs))⟩ ⟨xs ≠ ε⟩ lnzero_def neq0_conv sconc_fst_empty
            slen_empty_eq sntimes_len srcdups_sntimes)
  next
    case False
    then obtain k where "#t = Fin (Suc k)"
      by (metis Fin_02bot ⟨t = ε ∨ sfoot t ≠ sfoot xs⟩ ⟨xs = t • (n⋆↑(sfoot xs))⟩ bot_is_0 ninf2Fin
            not0_implies_Suc sconc_fst_inf slen_empty_eq)
    then show ?thesis
    proof (cases "ys = ε")
      case True
      then show ?thesis
        by simp
    next
      case False
      have "snth k (xs • ys) ≠ snth (Suc k) (xs • ys)"
      proof
        assume "snth k (xs • ys) = snth (Suc k) (xs • ys)"
        then have "snth k xs = snth (Suc k) xs"
          by (metis Fin_02bot Fin_Suc len_stream_def Suc_neq_Zero ⟨#t = Fin (Suc k)⟩ ⟨#xs < ∞⟩ ⟨t = ε ∨ sfoot t ≠
              sfoot xs⟩ ⟨xs = t • (n⋆↑(sfoot xs))⟩ stake_prefix slen_snth_prefix inject_Fin le2lnle lel less_le
              lnless_def lnzero_def monofun_cfun_arg notinfI3 sfoot_exists2 stream.take_below strict_slen)
        then show "False"
          by (metis Fin_02bot Fin_Suc Suc_neq_Zero ⟨#t = Fin (Suc k)⟩ ⟨t = ε ∨ sfoot t ≠ sfoot xs⟩ ⟨xs = t •
              (n⋆↑(sfoot xs))⟩ slen_snth_prefix inject_Fin lel ln_less lnzero_def neq0_conv notinfI3
              sconc_snd_empty sdropl6 sfoot_exists2 shd_sntime slen_empty_eq snth_def sntimes_len)
      qed
      moreover have "Fin (Suc k) < #(xs • ys)"
        by (metis False ⟨#t = Fin (Suc k)⟩ ⟨#xs < ∞⟩ ⟨xs = t • (n⋆↑(sfoot xs))⟩ minimal mono_slen
            monofun_cfun_arg sconc_snd_empty slen_conc)
```

```
      ultimately have "srcdups·(xs • ys) = srcdups·(stake (Suc k)·(xs • ys)) • srcdups·(sdrop (Suc k)·(xs • ys))"
        using srcdups_split2 by blast
      then have "srcdups·(xs • ys) = srcdups·t • srcdups·((sntimes n (↑(sfoot xs))) • ys)"
        by (metis ⟨#t = Fin (Suc k)⟩ ⟨xs = t • (n⋆↑(sfoot xs))⟩ assoc_sconc stake_prefix2 sdropl6)
      then have "srcdups·(xs • ys) = srcdups·t • ↑(sfoot xs) • srcdups·(sdropwhile (λx. x=sfoot xs)·ys)"
        by (metis ⟨t = ε ∨ sfoot t ≠ sfoot xs⟩ ⟨xs = t • (n⋆↑(sfoot xs))⟩ ⟨xs ≠ ε⟩ neq0_conv sconc_snd_empty
            sntimes.simps(1) srcdups_sntimes_prefix)
      moreover have "srcdups·xs = srcdups·t • ↑(sfoot xs)"
      proof -
        have "srcdups·xs = srcdups·t • srcdups·(sntimes n (↑(sfoot xs)))"
          by (metis ⟨#t = Fin (Suc k)⟩ ⟨snth k (xs • ys) ≠ snth (Suc k) (xs • ys)⟩ ⟨xs = t • (n⋆↑(sfoot xs))⟩
              convert_inductive_asm slen_snth_prefix stake_prefix2 not_less notinfl3 sconc_snd_empty sdropl6
              slen_conc srcdups_split2 strict_srcdups ub_slen_stake)
        then show ?thesis
          by (metis ⟨t = ε ∨ sfoot t ≠ sfoot xs⟩ ⟨xs = t • (n⋆↑(sfoot xs))⟩ ⟨xs ≠ ε⟩ neq0_conv sconc_snd_empty
              sntimes.simps(1) srcdups_sntimes)
      qed
      ultimately show ?thesis
        by simp
    qed
  qed
qed


lemma sValues_mono: "monofun (λs. {snth n s |n. Fin n < #s})"
apply (simp add: sValues_eq2)
apply (rule monofunI)
apply (rule_tac x="#x" in lncases)
apply (drule eq_less_and_fst_inf, simp+)
apply (simp add: atomize_imp)
apply (rule_tac x="y" in spec)
apply (rule_tac x="x" in spec)
apply (induct_tac k, simp+)
apply (auto simp add: less_set_def)
apply (drule lessD, auto)
apply (erule_tac x="q" in allE)
apply (erule_tac x="w" in allE, auto)
apply (case_tac "na", auto)
apply (rule_tac x="0" in exI, auto)
apply (frule_tac mono_len2, simp)
apply (rule_tac x="Suc nat" in exI, auto)
apply (rule_tac x="#w" in lncases, auto)
by (rule snth_less, auto)


lemma inf_chainl3rf: fixes Y::"nat ⇒ 'a stream"
  shows "⟦chain Y; ¬finite_chain Y⟧⟹∃k. Fin n ≤ #(Y k)"
by (rule inf_chainl3 [rule_format], auto)


lemma sValues_cont: "cont (λs. {snth n s | n. Fin n < #s})"
apply (rule contI2)
apply (rule sValues_mono)
apply (simp add: sValues_eq2)
apply (rule allI, rule impI)
apply (simp add: less_set_def)
apply (auto simp add: lub_eq_Union)
apply (case_tac "finite_chain Y")
apply (subst lub_finch2 [THEN lub_eqI], simp)
apply (rule_tac x="LEAST i. max_in_chain i Y" in exI)
apply (rule_tac x="n" in exI, simp)
apply (subst lub_finch2 [THEN lub_eqI, THEN sym], simp+)
apply (frule_tac n="Suc n" in inf_chainl3rf, simp+)
apply (erule exE)
apply (rule_tac x="k" in exI)
apply (rule_tac x="n" in exI)
apply (rule conjI)
apply (rule_tac x="#(Y k)" in lncases, simp+)
apply (rule sym)
apply (rule snth_less)
apply (rule_tac x="#(Y k)" in lncases, simp+)
by (rule is_ub_thelub)


lemma sValues_def2: "sValues·s = {snth n s | n. Fin n < #s}"
apply (subst sValues_def)
apply (subst beta_cfun)
by (rule sValues_cont, simp)

(* continuity of sValues *)
lemma sValues_cont2: "∀Y. chain Y⟶sValues·(⨆ i. Y i) = (⨆ i. sValues·(Y i))"
by (simp add: contlub_cfun_arg)

lemma srcdups_bool_prefix:
  fixes xs :: "bool stream" and ys :: "bool stream"
  assumes "lshd·(srcdups·xs) = lshd·(srcdups·ys)" and "#(srcdups·xs) ≤ #(srcdups·ys)"
  shows "(srcdups·xs) ⊑ (srcdups·ys)"
proof (rule scases [of xs])
  assume "xs = ε"
  then have "srcdups·xs = ε"
    by simp
  thus "srcdups·xs ⊑ srcdups·ys"
    by simp
next
```

```
  fix a :: bool and s :: "bool stream"
  assume "xs = ↑a • s"
  have "lshd·(srcdups·ys) = updis a"
    by (metis ⟨xs = ↑a • s⟩ assms(1) lshd_updis srcdups_step)
  then have "lshd·ys = updis a"
    by (metis lshd_updis srcdups_shd2 stream.sel_rews(3) strict_srcdups surj_scons up_defined)
  then have "∀n. Fin n < #(srcdups·ys)⟶snth n (srcdups·ys) = (even n = a)"
    by (metis (no_types, lifting) bool_stream_snth lshd_updis stream.sel_rews(3) sup'_def surj_scons)
  then have ys_expr: "∀n. Fin n < #(srcdups·xs)⟶snth n (srcdups·ys) = (even n = a)"
    using assms(2) less_le_trans by blast
  then have first_n_eq: "∀n. Fin n < #(srcdups·xs)⟶snth n (srcdups·xs) = snth n (srcdups·ys)"
    using ⟨xs = ↑a • s⟩ bool_stream_snth by blast
  then show "srcdups·xs ⊑ srcdups·ys"
  proof (cases "#(srcdups·xs) < ∞")
    case False
    then have "#(srcdups·xs) = #(srcdups·ys)"
      using assms(2) less_le by fastforce
    then have "∀k. snth k (srcdups·xs) = snth k (srcdups·ys)"
      by (metis False Fin_neq_inf first_n_eq inf_ub order.not_eq_order_implies_strict)
    then have "srcdups·xs = srcdups·ys"
      by (simp add: ⟨∀k. snth k (srcdups·xs) = snth k (srcdups·ys)⟩ ⟨#(srcdups·xs) = #(srcdups·ys)⟩ snths_eq)
    thus "srcdups·xs ⊑ srcdups·ys"
      by simp
  next
    case True
    then obtain k where "#(srcdups·xs) = Fin k"
      using lnat_well_h2 by blast
    then have eq_len: "#(srcdups·xs) = #(stake k·(srcdups·ys))"
      using assms(2) slen_stake by force
    have "∀n. Fin n < #(srcdups·xs)⟶snth n (srcdups·xs) = snth n (stake k·(srcdups·ys))"
      by (metis eq_len first_n_eq snth_less stream.take_below)
    then have "srcdups·xs = stake k·(srcdups·ys)"
      by (simp add: ⟨∀n. Fin n < #(srcdups·xs)⟶snth n (srcdups·xs) = snth n (stake k·(srcdups·ys))⟩ eq_len
          snths_eq)
    thus ?thesis
      by simp
  qed
qed

lemma srcdups_snth_stake_inf: "#s = ∞⟹snth n s ≠ snth (Suc n) s⟹srcdups·(stake (Suc n)·s) ≠ srcdups·s"
proof
  assume "#s = ∞" and "snth n s ≠ snth (Suc n) s" and "srcdups·(stake (Suc n)·s) = srcdups·s"
  have "#(stake (Suc (Suc n))·s) = Fin (Suc (Suc n))"
    by (simp add: ⟨#s = ∞⟩ slen_stake_fst_inf)
  moreover have "Suc (Suc n) > Suc n"
    by simp
  moreover have "snth n (stake (Suc (Suc n))·s) ≠ snth (Suc n) (stake (Suc (Suc n))·s)"
    by (metis Fin_leq_Suc_leq Suc_n_not_le_n ⟨snth n s ≠ snth (Suc n) s⟩ calculation(1) less2nat_lemma not_le
        snth_less stream.take_below)
  ultimately have "srcdups·(stake (Suc n)·(stake (Suc (Suc n))·s)) ≠ srcdups·(stake (Suc (Suc n))·s)"
    using srcdups_snth_stake_fin by blast
  then have "srcdups·(stake (Suc n)·s) ≠ srcdups·(stake (Suc (Suc n))·s)"
    by (simp add: min_def)
  moreover have "srcdups·(stake (Suc (Suc n))·s) = srcdups·s"
  proof (rule ccontr)
    assume "srcdups·(stake (Suc (Suc n))·s) ≠ srcdups·s"
    moreover have "srcdups·(stake (Suc n)·s) ⊑ srcdups·(stake (Suc (Suc n))·s)"
      by (simp add: less_imp_le_nat monofun_cfun_arg stake_mono)
    ultimately have "srcdups·(stake (Suc n)·s) ≠ srcdups·s"
      by (metis below_antisym monofun_cfun_arg stream.take_below)
    then show "False"
      by (simp add: ⟨srcdups·(stake (Suc n)·s) = srcdups·s⟩)
  qed
  ultimately show "False"
    by (simp add: ⟨srcdups·(stake (Suc n)·s) = srcdups·s⟩)
qed



(* sValues applied to the empty stream returns the empty set *)
lemma [simp]: "sValues·ϵ = {}"
by (auto simp add: sValues_def2 lnless_def)

(* the head of any stream is always an element of the domain *)
lemma sValues2un[simp]: "sValues·(↑z • s) = {z} ∪ sValues·s"
apply (auto simp add: sValues_def2)
apply (case_tac "n", auto)
apply (rule_tac x="0" in exI, auto)
by (rule_tac x="Suc n" in exI, auto)

lemma srcdups_dom_h: assumes "sValues·(srcdups·s) = sValues·s"
  shows "sValues·(srcdups·(↑a • s)) = insert a (sValues·s)"
proof (cases "shd s = a")
  case True
  have "srcdups·(↑a • ↑a • srt·s) = srcdups·(↑a • srt·s)"
    using srcdups_eq by blast
  hence "a ∈ sValues·(srcdups·(↑a • srt·s))"
    by (simp add: srcdups_step)
  then show ?thesis
    by (metis True Un_insert_left assms insert_absorb2 sValues2un srcdups_eq srcdups_step strict_sdropwhile
        sup_bot.left_neutral surj_scons)
next
  case False
```

```
    then show ?thesis
      by (metis (no_types, lifting) assms insert_is_Un sValues2un srcdups_neq srcdups_step strict_sdropwhile
            surj_scons)
qed

lemma srcdups_dom [simp]: "sValues·(srcdups·xs) = sValues·xs"
apply(rule ind, simp_all)
  by (simp add: srcdups_dom_h)

(* only the empty stream has no elements in its domain *)
lemma strict_sValues_rev: "sValues·s = {}⟹s = ε"
apply (auto simp add: sValues_def2)
apply (rule_tac x="s" in scases, auto)
by (metis Fin_02bot gr_0 lnzero_def)

lemma sValues_notempty:"s≠ε⟷sValues·s≠{}"
  using strict_sValues_rev by auto

(* the infinite repetition of a only has a in its domain *)
(*with new lemmata not necessary:
 apply (subst sinftimes_unfold, simp)*)
(*apply (induct_tac n, auto)
apply (subst sinftimes_unfold, simp)
apply (rule_tac x="0" in exI)
by (subst sinftimes_unfold, simp)*)
lemma [simp]: "sValues·(sinftimes (↑a)) = {a}"
by (auto simp add: sValues_def2)

(* any singleton stream of z only has z in its domain *)
lemma [simp]: "sValues·(↑z) = {z}"
by (auto simp add: sValues_def2)

(* if an element z is in the domain of a stream s, then z is the n'th element of s for some n *)
lemma sValues2snth: "z ∈ sValues·s⟹∃n. snth n s = z"
by (auto simp add: sValues_def2)

(* if the natural number n is less than the length of the stream s, then snth n s is in the domain of s *)
lemma snth2sValues: "Fin n < #s⟹snth n s ∈ sValues·s"
  by (auto simp add: sValues_def2)

lemma smap_well:"sValues·x⊆range f⟹ ∃s. smap f·s = x"
  apply(rule_tac x = "smap (inv f)·x" in exI)
  by (simp add: snths_eq smap_snth_lemma f_inv_into_f snth2sValues subset_eq)

lemma smap_inv_id[simp]: "sValues·s ⊆ range F⟹smap (F ∘ (inv F))·s = s"
  apply (induction s  rule: ind )
  by(simp_all add: f_inv_into_f)

lemma smap_inv_eq[simp]:"inj F⟹smap (inv F ∘ F)·x = x"
  by (metis inv_o_cancel smap_inv_id subset_UNIV surj_id surj_iff)

(* checking if the domain of a stream x isn't a subset of another set M is an admissible predicate *)
lemma [simp]: "adm (λx. ¬ sValues·x ⊆ M)"
apply (rule admI)
apply (rule notI)
apply (frule_tac x="0" in is_ub_thelub)
apply (frule_tac f="sValues" in monofun_cfun_arg)
by (erule_tac x="0" in allE, auto simp add: less_set_def)


lemma sfilter_sValuesl3:
  "sValues·s ⊆ X⟶sfilter X·s = s"
apply (rule impI)
apply (rule stream.take_lemma)
apply (simp add: atomize_imp)
apply (rule_tac x="s" in spec)
apply (rule_tac x="X" in spec)
apply (induct_tac n, simp+)
apply (rule allI)+
apply (rule impI)
by (rule_tac x="xa" in scases, simp+)


lemma sfilter_sValuesl4 [simp]:
  "sfilter (sValues·s)·s = s"
by (rule sfilter_sValuesl3 [rule_format, of "s" "sValues·s"], simp)

lemma sfilter_fin: assumes "#(A ⊖ s) < ∞"
  shows "∃n. (A ⊖ (sdrop n·s)) = ⊥"
  apply(rule ccontr)
  apply auto
  by (metis len_stream_def assms fun_approxl2 lnat_well_h2 sconc_neq_h sconc_snd_empty split_sfilter)

lemma s_one_dom_inf: assumes "sValues·s = {x}" and "#s = ∞"
    shows "s = ((↑x)^∞)"
  by (metis Fin_02bot Fin_Suc Suc_n_not_le_n assms(1) assms(2) bot_is_0 inject_Fin less_or_eq_imp_le
        sinftimes_unfold singleton_iff slen_scons slen_sinftimes snth2sValues snth_sinftimes snths_eq
        strict_icycle strict_slen)

lemma sfilter_bot_dom: "(A ⊖ s) = ⊥⟹sValues·s ⊆ UNIV - A"
  apply(induction s rule: ind)
    apply auto
  by (metis DiffD2 inject_scons rev_subsetD sfilter_in sfilter_nin strictI)
```

```
lemma sValues_sconc2un:
  "#x = Fin k⟹sValues·(x • y) = sValues·x ∪ sValues·y"
apply (simp add: atomize_imp)
apply (rule_tac x="x" in spec)
apply (induct_tac k, simp+)
apply (rule allI, rule impI)
by (rule_tac x="x" in scases, simp+)

(* sValues applied to s1•s2 is a subset of the union of sValues s1 and sValues s2 *)
lemma sconc_sValues: "sValues·(s1•s2) ⊆ sValues·s1 ∪ sValues·s2"
by (metis SetPcpo.less_set_def below_refl lncases sconc_fst_inf sValues_sconc2un sup.coboundedI1)

(* relation between sValues and sfoot *)
lemma sfoot_dom: assumes "#s = Fin (Suc n)" and "sValues·s⊆A"
  shows "sfoot s∈A"
by (metis Suc_n_not_le_n assms(1) assms(2) contra_subsetD leI less2nat_lemma sfoot_exists2 snth2sValues)

(* stakewhile doesn't include the element a that failed the predicate f in the result *)
lemma stakewhile_dom[simp]:assumes "¬f a"
  shows "a∉sValues·(stakewhile f·s)"
by (smt assms below_antisym lnle_conv lnless_def mem_Collect_eq sValues_def2 snth_less stakewhile_below
      stakewhile_slen)

lemma srcdups_sconc_duplicates:
  assumes "#xs < ∞" and "xs ≠ ε" and "srcdups·xs = srcdups·(xs • ys)"
  shows "sValues·ys ⊆ {sfoot xs}"
proof -
  have "srcdups·(xs • ys) = (srcdups·xs) • (srcdups·(sdropwhile (λx. x=sfoot xs)·ys))"
    using assms(1) assms(2) srcdups_sconc by blast
  then have "srcdups·xs = srcdups·xs • (srcdups·(sdropwhile (λx. x=sfoot xs)·ys))"
    using assms(3) by presburger
  moreover have "#(srcdups·xs) < ∞"
    by (meson assms(1) leD leI srcdups_slen trans_lnle)
  ultimately have "srcdups·(sdropwhile (λx. x=sfoot xs)·ys) = ε"
    using sconc_neq_h by fastforce
  then have "sdropwhile (λx. x=sfoot xs)·ys = ε"
    using srcdups_nbot by blast
  then show ?thesis
    by (metis (full_types) insertI1 sconc_snd_empty stakewhileDropwhile stakewhile_dom subsetI)
qed

(* if stakewhile changes the stream s, which is a prefix of the stream s', then stakewhile of s and s'
   produce the same result *)
lemma stakewhile_finite_below:
shows "stakewhile f·s ≠ s⟹s⊑s'⟹stakewhile f·s = stakewhile f·s'"
apply(induction s)
apply simp+
by (smt approxl1 len_stream_def approxl2 lnless_def monofun_cfun_arg rev_below_trans snth_less stakewhile_below
      stakewhile_notin stakewhile_snth)

(* if there is an element in the stream s that fails the predicate f, then stakewhile will change s *)
lemma stakewhile_noteq[simp]: assumes "¬f (snth n s)" and "Fin n < #s"
  shows "stakewhile f·s ≠ s"
proof (rule ccontr)
  assume "¬ stakewhile f·s ≠ s"
  hence "sValues·(stakewhile f·s) = sValues·s" by simp
  hence "(snth n s)∈sValues·(stakewhile f·s)" by (simp add: assms(2) snth2sValues)
  thus False by (simp add: assms(1))
qed

(* if there's an element a in the domain of s which fails the predicate f, then stwbl will produce a
   finite result *)
lemma stwbl_fin [simp]: assumes "a∈sValues·s" and "¬ f a"
  shows "#(stwbl f·s) < ∞"
by (metis assms(1) assms(2) inf_ub lnle_conv lnless_def notinfI3 sconc_slen sValues2snth stakewhile_slen
      stwbl_stakewhile ub_slen_stake)

(* stwbl keeps at least all the elements that stakewhile keeps *)
lemma stakewhile_stwbl [simp]: "stakewhile f·(stwbl f·s) = stakewhile f·s"
proof -
  have "⋀s sa. (s::'a stream) ⊑ s • sa"
    by simp
  then have "stakewhile f·(stwbl f·s) = stwbl f·s⟶stakewhile f·(stwbl f·s) = stakewhile f·s"
    by (metis (no_types) below_antisym monofun_cfun_arg stwbl_below stwbl_stakewhile)
  then show ?thesis
    using stakewhile_finite_below stwbl_below by blast
qed

(* sValues applied to sntimes n s is a subset of sValues applied to s *)
lemma sntimes_sValues1[simp]: "sValues·(sntimes n s) ⊆ sValues·s"
proof (induction n)
  case 0 thus ?case by simp
next
  case (Suc n) thus ?case using sconc_sValues sntimes.simps(2) sup.orderE by auto
qed

(* if filtering everything except z from the stream x doesn't produce the empty stream, then z must
   be an element of the domain of x *)
lemma sfilter2dom:
  "sfilter {z}·x ≠ ε⟹z ∈ sValues·x"
apply (subgoal_tac "∃k. snth k x = z ∧ Fin k < #x", erule exE)
apply (erule conjE)
apply (drule sym, simp)
```

```
apply (rule snth2sValues, simp)
apply (rule ccontr, simp)
by (insert ex_snth_in_sfilter_nempty [of x "{z}"], auto)

text ‹For injective functions @{term f} with @{term "f(y) = x"}, @{term x} can only
  be contained in @{term "smap f·s"} if the original stream contained @{term y}›
lemma sValues_smapI1: "⟦x ∈ sValues·(smap f·s); inj f; f y = x⟧⟹y ∈ sValues·s"
  by (smt mem_Collect_eq sValues_def2 slen_smap smap_snth_lemma the_inv_f_f)
    (*
apply (auto simp add: sValues_def2)
apply (rule_tac x="n" in exI, simp)
apply (simp add: smap_snth_lemma)
by (simp add: inj_on_def) *)

(* appending another stream xs can't shrink the domain of a stream x *)
lemma sValues_sconc[simp]: "sValues·x ⊆ sValues·(x • xs)"
by (metis minimal monofun_cfun_arg sconc_snd_empty set_cpo_simps(1))

(* repeating a stream doesn't add elements to the domain *)
lemma sinftimes_sValues[simp]: "sValues·(sinftimes s) ⊆ sValues·s"
by (smt chain_monofun contlub_cfun_arg lub_below set_cpo_simps(1) sntimesLub sntimes_chain sntimes_sValues1)

(* repeating a stream doesn't remove elements from the domain either *)
lemma sinf_sValues [simp]: "sValues·(s^∞) = sValues·s"
by (metis antisym_conv sValues_sconc sinftimes_sValues sinftimes_unfold)

(* sfilter doesn't add elements to the domain *)
lemma sbfilter_sbdom[simp]: "sValues·(sfilter A·s) ⊆ sValues·s"
apply(rule ind [of _s], auto)
by (metis (mono_tags, lifting) UnE contra_subsetD sValues2un sfilter_in sfilter_nin singletonD)

(* smap can only produce elements in the range of the mapped function f *)
lemma smap_sValues_range [simp]: "sValues·(smap f·s) ⊆ range f"
by (smt mem_Collect_eq range_eqI sValues_def2 slen_smap smap_snth_lemma subsetI)

(* every element produced by (smap f) is in the image of the function f *)
lemma smap_sValues: "sValues·(smap f·s) =  f ` sValues·s"
apply(rule)
apply (smt image_eqI mem_Collect_eq sValues_def2 slen_smap smap_snth_lemma subsetI)
by (smt image_subset_iff mem_Collect_eq sValues_def2 slen_smap smap_snth_lemma)

(* lemmas for SB *)
(* if the stream a is a prefix of the stream b then a's domain is a subset of b's *)
lemma sValues_prefix [simp]: "a ⊑ b⟹sValues·a ⊆ sValues·b"
by (metis SetPcpo.less_set_def monofun_cfun_arg)

(* the lub of a chain of streams contains any elements contained in any stream in the chain *)
lemma sValues_chain2lub: "chain S⟹sValues·(S i) ⊆ sValues·(⨆ j. S j)"
using sValues_prefix is_ub_thelub by auto

(* if every element in a chain S is a prefix of s then also the least upper bound in the chain S if prefix of s *)
lemma lubChainpre: "chain S⟹S i ⟹∀i. S i ⊑ s⟹(⨆ j. S j) ⊑ s"
by (simp add: lub_below)

(* if every element in a chain S is a prefix of s, then the domain of S i is a subset of the domain of s *)
lemma sValues_chainprefix: "chain S⟹∀i. S i ⊑ s⟹∀i. sValues·(S i) ⊆ sValues·s"
by simp

(* if every element in a chain S is a prefix of s, then the domain of the lub is a subset of the domain of s *)
lemma sValues_chainlub: "chain S⟹ ∀i. S i ⊑ s⟹sValues·(⨆ j. S j) ⊆ sValues·s"
using sValues_prefix lub_below by blast

(* streams appearing later in the chain S contain the elements of preceding streams *)
lemma sValues_chain_below: "chain S⟹i ≤ j⟹sValues·(S i) ⊆ sValues·(S j)"
by (simp add: po_class.chain_mono)

(* for two elements i, j with i ≤ j in a chain S it holds that the domain of S i is a subset of the domain of S j
    *)
lemma sValues_lub2union: "chain S⟹finite_chain S⟹sValues·(⨆ j. S j) ⊆ (⋃i. sValues·(S i))"
using l42 by fastforce

(* important *)
(* the lub doesn't have any elements that don't appear somewhere in the chain *)
lemma sValues_lub: "chain S⟹sValues·(⨆ j. S j) = (⋃i. sValues·(S i))"
apply (simp add: contlub_cfun_arg)
by (simp add: lub_eq_Union)

lemma sscanlasnd_smap_state_loop:
  assumes"⋀e. e ∈ sValues·s⟹fst (f state e) = state"
shows "sscanlAsnd f state·(s) = smap (λe. snd (f state e))·s"
  using assms
  apply (induction s rule: ind)
  apply (rule adm_imp)
  apply (rule admI)
  apply (meson sValues_chain2lub set_rev_mp)
  by simp_all

(* core lemma for exchanging transition function (general lemma) *)
lemma sscanla_exchange_f:
  assumes "⋀e state. P e⟹F1 state e = F2 state e"
    and "∀x ∈ sValues·s. P x"
  shows "sscanlAsnd F1 state·s = sscanlAsnd F2 state·s"
  using assms
  apply (induction s arbitrary: state rule: ind)
```

```
  apply (rule adm_imp, simp)+
  apply (rule admI)
  apply (meson sValues_chain2lub subsetCE)
  by simp_all

lemma l44: assumes "chain S" and "∀i. sValues·(S i) ⊆ B"
  shows "sValues·(⊔ j. S j) ⊆ B"
by (metis (mono_tags, lifting) UN_E assms sValues_lub subsetCE subsetI)

(* helper lemma *)
lemma l6: "chain S⟹∀i. sValues·(S i) ⊆ B⟹sValues·(⊔ j. S (j + (SOME k. A))) ⊆ B"
by (simp add: l44 lub_range_shift)

(* dropping elements can't increase the domain *)
lemma sdrop_sValues[simp]: "sValues·(sdrop n·s)⊆sValues·s"
by (metis Un_upper2 approxl2 sValues_prefix sValues_sconc2un sdrop_0 sdropostake split_streaml1 stream.take_below)

(* if none of the elements in the domain of the stream s are in the set A, then filtering s with A
   produces the empty stream *)
lemma sfilter_sValues_eps: "sValues·s ∩ A = {}⟹(A ⊖ s) = ε"
by (meson disjoint_iff_not_equal ex_snth_in_sfilter_nempty snth2sValues)

(* if x in sValues·(A⊖s) then x is in A *)
lemma sValues_sfilter1: assumes "x∈sValues·(A⊖s)"
shows "x∈A"
by (smt assms mem_Collect_eq sValues_def2 sfilterl7)

(* if u is not bottom then sValues·s⊆sValues·(u && s) *)
lemma sValues_subset: assumes "u≠⊥"
shows "sValues·s⊆sValues·(u && s)"
by (metis Un_upper2 assms sValues2un stream.con_rews(2) stream.sel_rews(5) surj_scons)

(* if u is not bottom then sValues·(A⊖s)⊆sValues·(A ⊖ (u && s)) *)
lemma sValues_sfilter_subset: assumes "u≠⊥"
shows "sValues·(A⊖s)⊆sValues·(A ⊖ (u && s))"
by (smt Un_upper2 assms eq_iff sValues2un sfilter_in sfilter_nin stream.con_rews(2) stream.sel_rews(5) surj_scons)

(* if x in A then x∈sValues·s implies x∈(sValues·(A ⊖ s)) *)
lemma sValues_sfilter2: assumes "x∈A"
shows "x∈sValues·s⟹x∈(sValues·(A ⊖ s))"
apply(induction s)
apply(rule admI)
apply rule
  apply (metis (mono_tags, lifting) UN_iff ch2ch_Rep_cfunR contlub_cfun_arg sValues_lub)
apply simp
by (smt UnE assms empty_iff insert_iff sconc_sValues sValues2un sValues_sconc sValues_sfilter_subset sfilter_in
     stream.con_rews(2) stream.sel_rews(5) subsetCE surj_scons)

(* sValues applied to A⊖s returns the intersection of sValues applied to s and A *)
lemma sValues_sfilter[simp]: "sValues·(A⊖s) = sValues·s ∩ A"
apply rule
apply (meson IntI sbfilter_sbdom sValues_sfilter1 subset_iff)
apply rule
  by (simp add: sValues_sfilter2)

(* if sfilter of A·s is s then sValues·s is a subset of A *)
lemma sfilterEq2sValues_h: "sfilter A·s = s⟶sValues·s ⊆ A"
  apply(rule ind [of _s])
    apply (smt admI inf.orderI sValues_sfilter)
   apply(simp)
  apply(rule)
  by (metis inf.orderI sValues_sfilter)

(* sfilter of A·s is s implies that sValues·s is a subset of A *)
lemma sfilterEq2sValues: "sfilter A·s = s⟹sValues·s ⊆ A"
  by (simp add: sfilterEq2sValues_h)

(* if ∀a∈sValues·s. f a then stwbl applied to f·s returns s *)
lemma stwbl_id_help:
  shows "(∀a∈sValues·s. f a)⟶stwbl f·s = s"
  apply (rule ind [of _s])
    apply(rule adm_imp)
     apply(rule admI, rule+)
     using sValues_chain2lub apply blast
    apply(rule admI)
    apply (metis cont2contlubE cont_Rep_cfun2 lub_eq)
   using strict_stwbl apply blast
  apply rule+
  by simp

(* ⋀ a. a∈sValues·s⟹f a implies that stwbl applied to f·s is s *)
lemma stwbl_id [simp]: "(⋀ a. a∈sValues·s⟹f a)⟹stwbl f·s = s"
by (simp add: stwbl_id_help)

(* if a in sValues s and ¬f a then it holds that ∃x. (stwbl f·s) = stakewhile f·s • ↑x *)
lemma stwbl2stakewhile: assumes "a∈sValues·s" and "¬f a"
  shows "∃x. (stwbl f·s) = stakewhile f·s • ↑x"
proof -
  have "#(stwbl f·s) < ∞" using assms(1) assms(2) snth2sValues stwbl_fin by blast
  hence "stwbl f·s ≠ ε" by (metis assms(1) assms(2) stakewhile_dom strict_stakewhile stwbl_notEps)
  thus ?thesis
    by (smt Fin_02bot approxl2 assms(1) assms(2) bottomI lnle_def lnzero_def mem_Collect_eq sconc_snd_empty
         sValues_def2 sdrop_0 slen_empty_eq slen_rt_ile_eq split_streaml1 stakewhile_below stakewhile_noteq
```

```
        stakewhile_sdropwhilel1 stwbl_notEps stwbl_stakewhile surj_scons tdw ub_slen_stake)
qed

(* if a in sValues s and ¬f a it holds that ¬ f (sfoot (stwbl f·s)) *)
lemma stwbl_sfoot: assumes "a∈sValues·s" and "¬f a"
  shows "¬ f (sfoot (stwbl f·s))"
proof(rule ccontr)
  assume "¬ ¬ f (sfoot (stwbl f·s))"
  hence "f (sfoot (stwbl f·s))" by blast
  obtain x where x_def:"(stwbl f·s) = stakewhile f·s • ↑x"
    using assms(1) assms(2) stwbl2stakewhile by blast
  hence "sfoot (stwbl f·s) = x"
    using assms(1) assms(2) sfoot1 stwbl_fin by blast
  have "stakewhile f·s • ↑x ⊑ s" by (metis stwbl_below x_def)
  have "f x"
    using ⟨f (sfoot (stwbl f·s))⟩ ⟨sfoot (stwbl f·s) = x⟩ by blast
  thus False
    by (metis approxl2 assms(1) assms(2) inject_sconc sconc_snd_empty sdropwhile_resup stakewhileDropwhile
        stakewhile_below stakewhile_dom stakewhile_stwbl x_def)
(*    by (smt Fin_02bot ⟨sfoot (stwbl f·s) = x⟩ approxl2 assms(1) assms(2) assoc_sconc bottomI lnle_def
     lnzero_def sconc_fst_empty sconc_snd_empty sdrop_0 sdropwhile_t sfoot1 slen_empty_eq slen_rt_ile_eq
     split_streaml1 stakewhile_below stakewhile_dom stakewhile_sdropwhilel1 stakewhile_stwbl stream.take_strict
     strict_stakewhile stwbl_fin stwbl_notEps stwbl_stakewhile surj_scons tdw ub_slen_stake) *)
qed

(* stwbl applied to f and stwbl f·s returns stwbl f·s *)
lemma stwbl2stbl[simp]: "stwbl f·(stwbl f·s) = stwbl f·s"
  apply(rule ind [of _s])
    apply simp_all
  by (metis sconc_snd_empty stwbl_f stwbl_t)

(* (λx. b ∉ sValues·x) is admissible *)
lemma adm_nsValues [simp]:  "adm (λx. b ∉ sValues·x)"
proof (rule admI)
  fix Y
  assume as1: "chain Y" and as2: "∀i. b∉sValues·(Y i)"
  thus "b∉sValues·(⨆i. Y i)"
  proof (cases "finite_chain Y")
    case True thus ?thesis using as1 as2 l42 by fastforce
  next
    case False
    hence "#(⨆i. Y i) = ∞" using as1 inf_chainl4 by blast
    hence "⋀n. snth n (⨆i. Y i) ≠ b"
    proof -
      fix n
      obtain j where "Fin n < # (Y j)"  by (metis False inf_chainl2 as1 inf_chainl3rf less_le)
      hence "snth n (Y j) ≠b" using as2 snth2sValues by blast
      thus "snth n (⨆i. Y i) ≠ b" using ⟨Fin n < #(Y j)⟩ as1 is_ub_thelub snth_less by blast
    qed
    thus ?thesis using sValues2snth by blast
  qed
qed

(* strdw_filter helper lemma *)
lemma strdw_filter_h: "b∈sValues·s⟶lnsuc·(#({b} ⊖ srtdw (λa. a ≠ b)·s)) = #({b} ⊖ s)"
proof(rule ind [of _s])
  have "adm (λa. lnsuc·(#({b} ⊖ srtdw (λa. a ≠ b)·a)) = #({b} ⊖ a))" by (simp add: len_stream_def)
  thus "adm (λa. b ∈ sValues·a⟶lnsuc·(#({b} ⊖ srtdw (λa. a ≠ b)·a)) = #({b} ⊖ a))" by simp
  show "b ∈ sValues·ϵ⟶lnsuc·(#({b} ⊖ srtdw (λa. a ≠ b)·ϵ)) = #({b} ⊖ ϵ)" by simp
  fix a
  fix s
  assume IH: " b ∈ sValues·s⟶lnsuc·(#({b} ⊖ srtdw (λa. a ≠ b)·s)) = #({b} ⊖ s)"
  show " b ∈ sValues·(↑a • s)⟶lnsuc·(#({b} ⊖ srtdw (λa. a ≠ b)·(↑a • s))) = #({b} ⊖ ↑a • s)"
  proof (cases "a=b")
    case True thus ?thesis by simp
  next
    case False
    hence f1:"#({b} ⊖ ↑a • s) = #({b} ⊖ s)" using sfilter_nin singletonD by auto
    hence f2:"#({b} ⊖ srtdw (λa. a ≠ b)·(↑a • s)) = #({b} ⊖ srtdw (λa. a ≠ b)·(s))" using False by auto
    hence "b ∈ sValues·(↑a • s)⟹b∈sValues·s" using False by auto
    thus ?thesis using IH f2 local.f1 by auto
  qed
qed

(* strdw filter lemma *)
lemma strdw_filter: "b∈sValues·s⟹lnsuc·(#({b} ⊖ srtdw (λa. a ≠ b)·s)) = #({b} ⊖ s)"
by(simp add: strdw_filter_h)

(* length of stwbl filter *)
lemma stwbl_filterlen[simp]: "b∈sValues·ts⟶#({b} ⊖ stwbl (λa. a ≠ b)·ts) = Fin 1"
  apply(rule ind [of _ ts])
    apply(rule adm_imp)
     apply simp
    apply (simp add: len_stream_def)
   apply simp
  apply auto
  by (metis (mono_tags, lifting) Fin_02bot Fin_Suc One_nat_def lnzero_def sconc_snd_empty sfilter_in sfilter_nin
      singletonD singletonI slen_scons strict_sfilter strict_slen stwbl_f stwbl_t)

(* srtdw concatenation *)
lemma srtdw_conc: "b∈sValues·ts ⟹(srtdw (λa. a ≠ b)·(ts • as)) = srtdw (λa. a ≠ b)·(ts) • as"
  apply(induction ts arbitrary: as)
    apply (rule adm_imp)
     apply auto
```

```
    apply(rule admI)
    apply rule+
    apply (metis (no_types, lifting) approxl3 assoc_sconc is_ub_thelub)
  proof -
    fix u ts as
    assume as1: "u ≠ ⊥" and as2: "(⋀as. b ∈ sValues·ts⟹srtdw (λa. a ≠ b)·(ts • as) = srtdw (λa. a ≠ b)·ts • as)"
        and as3: "b ∈ sValues·(u && ts)"
    obtain a where a_def: "updis a = u" by (metis Exh_Up as1 discr.exhaust)
    have "a≠b⟹b∈sValues·ts" by (metis UnE a_def as3 lscons_conv sValues2un singletonD)
    hence "a≠b⟹srtdw (λa. a ≠ b)·(↑a• (ts • as)) = srtdw (λa. a ≠ b)·(↑a• ts) • as " using as2 a_def by auto
    thus "srtdw (λa. a ≠ b)·((u && ts) • as) = srtdw (λa. a ≠ b)·(u && ts) • as "
      by (smt a_def inject_scons lscons_conv sconc_scons stwbl_f stwbl_srtdw)
  qed

(* stwbl concatenation *)
lemma stwbl_conc[simp]: "b∈sValues·ts⟹
    (stwbl (λa. a ≠ b)·(stwbl (λa. a ≠ b)·ts • xs)) =
    (stwbl (λa. a ≠ b)·(ts))"
  apply(induction ts)
    apply(rule adm_imp)
     apply simp
    apply(rule admI)
    apply (metis (no_types, lifting) ch2ch_Rep_cfunR contlub_cfun_arg inf_chainl4 lub_eql lub_finch2
          sconc_fst_inf stwbl2stbl)
   apply simp
  by (smt UnE assoc_sconc sValues2un singletonD stream.con_rews(2) stream.sel_rews(5) stwbl_f stwbl_t surj_scons)


(* ----------------------------------------------------------------------- *)
section ‹@{term siterateBlock}›
(* ----------------------------------------------------------------------- *)

(* block-iterating the function f on the stream x is equivalent to the stream produced by concatenating x
   and the iteration of f on x shifted by another application of f *)
lemma siterateBlock_unfold: "siterateBlock f x = x • siterateBlock f (f x)"
by(subst siterateBlock_def [THEN fix_eq2], auto)

(* if g doesn't change the length of the input, then iterating g doesn't either *)
lemma niterate_len[simp]: assumes "∀z. #z = #(g z)"
  shows "#((niterate i g) x) = #x"
using assms by(induction i, auto)

(* dropping i blocks from siterateBlock g x is equivalent to beginning siterateBlock after i iterations
   of g have already been applied *)
lemma siterateBlock_sdrop2: assumes "#x = Fin y" and "∀z. #z = #(g z)"
  shows "sdrop (y*i)·(siterateBlock g x) = siterateBlock g ((niterate i g) x)"
apply (induction i, auto)
by (metis assms(1) assms(2) niterate_len sdrop_plus sdropl6 siterateBlock_unfold)

(* the y*i'th element of siterateBlock is the same as the head of the i'th iteration *)
lemma siterateBlock_snth: assumes "#x = Fin y" and "∀z. #z = #(g z)" and "#x > Fin 0"
  shows "snth (y*i) (siterateBlock g x) = shd ((niterate i g) x)"
proof -
  have eq1: "sdrop (y*i)·(siterateBlock g x) = siterateBlock g ((niterate i g) x)" using assms(1) assms(2)
        siterateBlock_sdrop2 by blast

  have "#((niterate i g) x) > 0" by (metis Fin_02bot assms(2) assms(3) lnzero_def niterate_len)
  hence "shd (siterateBlock g ((niterate i g) x)) = shd (((niterate i g) x))" by (metis Fin_0 minimal
        monofun_cfun_arg sconc_snd_empty siterateBlock_unfold snth_less snth_shd)
  thus ?thesis by (simp add: eq1 snth_def)
qed

(* dropping a single block from siterateBlock is equivalent to beginning the iteration with (g x) *)
lemma siterateBlock_sdrop: assumes "#x = Fin y"
  shows "sdrop y·(siterateBlock g x) = siterateBlock g (g x)"
by (metis assms sdropl6 siterateBlock_unfold)

(* block-iterating the function g on the empty stream produces the empty stream again *)
lemma siterateBlock_eps[simp]: assumes "g ε = ε"
  shows "siterateBlock g ε = ε"
by(simp add: siterateBlock_def assms)

(* block-iterating the identity on the element x is equivalent to infinitely repeating x *)
lemma siterateBlock2sinf: "siterateBlock id x = sinftimes x"
by (metis id_apply rek2sinftimes siterateBlock_eps siterateBlock_unfold strict_icycle)

(* siterateBlock doesn't affect infinite streams *)
lemma siterBlock_inf [simp]: assumes "#s = ∞"
  shows "siterateBlock f s = s"
by (metis assms sconc_fst_inf siterateBlock_unfold)

(* the first element of siterateBlock doesn't have any applications of g *)
lemma siterateBlock_shd [simp]: "shd (siterateBlock g (↑x)) = x"
by (metis shd1 siterateBlock_unfold)

(* helper lemma for siterateBlock2siter *)
lemma siterateBlock2niter: "snth i (siterateBlock (λs. (smap g·s)) (↑x)) = niterate i g x" (is "snth i (?B) = ?N
    i")
proof -
  have f1: "#(↑x) = Fin 1" by simp
  have "∀z. #z = #((λs. (smap g·s)) z)" by simp
  hence f2: " snth (i) (siterateBlock (λs. (smap g·s)) (↑x)) = shd (niterate i (λs. (smap g·s)) (↑x))"
  by (metis Fin_0 Fin_Suc One_nat_def f1 lnat.con_rews lnless_def lnzero_def minimal nat_mult_1
        siterateBlock_snth)
```

```
  have "shd (niterate i (λs. (smap g·s)) (↑x)) = niterate i g x"
  proof (induction i)
    case 0 thus ?case by simp
  next
    case (Suc i) thus ?case
    by (smt Fin_0 f1 inject_scons neq0_conv niterate.simps(2) niterate_len slen_smap smap_scons strict_slen
        surj_scons zero_less_one)
  qed
  thus ?thesis by (simp add: f2)
qed

(* siterateBlock creates an infinitely long stream *)
lemma siterateBlock_len [simp]: "#(siterateBlock (λs. (smap g·s)) (↑x)) = ∞"
apply (rule infI)
apply (rule allI)
apply (rule_tac x="x" in spec)
apply (induct_tac k, simp+)
apply (metis bot_is_0 lnat.con_rews siterateBlock_unfold slen_scons strict_slen)
by (metis Fin_Suc lnat.sel_rews(2) sconc_snd_empty siterateBlock_unfold slen_scons smap_scons strict_smap)

(* iterating the identity function commutes with any function f *)
lemma siterateBlock_smap: "siterateBlock id (smap f·x) =  smap f·(siterateBlock id x)"
by (simp add: siterateBlock2sinf)

(* converting x to a singleton stream and applying siterateBlock using smap g is equivalent to
   iterating using g directly on x *)
lemma siterateBlock2siter [simp]: "siterateBlock (λs. (smap g·s)) (↑x) = siterate g x"
apply (rule sinf_snt2eq, auto)
by (simp add: siterateBlock2niter snth_siter)


(* ---------------------------------------------------------------------- *)
subsection ‹@{term sislivespf}›
(* ---------------------------------------------------------------------- *)


(* if length of f·x is infinite then also length of x is infinite, and then sislivespf f holds *)
lemma sislivespfI:
  "(⋀x. #(f·x) = ∞⟹#x = ∞)⟹sislivespf f"
by (simp add: sislivespf_def)

(* if length of x is finite then also length of f·x is finite, and then it holds that sislivespf f *)
lemma sislivespfI2:
  "(⋀k. ∀x. #x = Fin k⟶#(f·x) ≠∞)⟹sislivespf f"
apply (rule sislivespfI)
by (rule_tac x="#x" in lncases, simp+)

(* if sislivespf f holds and length of x is finite, then also length of f·x is finite *)
lemma sislivespfD1:
  "⟦sislivespf f; #x = Fin k⟧⟹#(f·x) ≠∞"
apply (rule notI)
by (simp add: sislivespf_def)

(* if sislivespf f holds and f·x has infinite length, then x has infinite length *)
lemma sislivespfD2:
  "⟦sislivespf f; #(f·x) = ∞⟧⟹#x = ∞"
by (simp add: sislivespf_def)


(* ---------------------------------------------------------------------- *)
section ‹Lemmas on lists and streams›
(* ---------------------------------------------------------------------- *)

(* ---------------------------------------------------------------------- *)
subsection ‹@{term list2s}›
(* ---------------------------------------------------------------------- *)

(* consing onto a list is equivalent to prepending an element to a stream *)
lemma [simp]: "list2s (a#as) = ↑a • list2s as"
by (simp add: lscons_conv)

declare list2s_Suc [simp del]

(* infinite lists don't exist *)
lemma [simp]: "#(list2s x) ≠∞"
by (induct x, simp+)


lemma s2list_ex:
  "#s = Fin k⟹∃l. list2s l = s"
apply (simp add: atomize_imp)
apply (rule_tac x="s" in spec)
apply (induct_tac k, simp+)
apply (rule_tac x="[]" in exI, simp+)
apply (rule allI, rule impI)
apply (rule_tac x="x" in scases, simp+)
apply (erule_tac x="s" in allE)
apply (drule mp)
apply (simp add: Fin_Suc [THEN sym] del: Fin_Suc)
apply (erule exE)
by (rule_tac x="a # l" in exI, simp)

(* the empty stream corresponds to the empty list *)
```

```
lemma [simp]: "s2list ε = []"
apply (simp add: s2list_def)
apply (rule someI2_ex)
apply (rule_tac x="[]" in exI, simp)
apply (simp add: atomize_imp)
by (induct_tac x, simp+)

(* the singleton stream corresponds to the singleton list *)
lemma [simp]: "s2list (↑a) = [a]"
apply (simp add: s2list_def)
apply (rule someI2_ex)
apply (rule_tac x="[a]" in exI, simp)
apply (simp add: atomize_imp)
apply (induct_tac x, auto)
by (case_tac "list", simp+)

(* the empty list is the bottom element for lists *)
lemma [simp]: "[] ⊑ l"
by (simp add: sq_le_list)


lemma list2s_emb: "⟦#s ≠∞; #s' ≠∞⟧⟹(s2list s ⊑ s2list s') = (s ⊑ s')"
apply (simp add: s2list_def)
apply (rule someI2_ex)
apply (rule_tac x="#s'" in lncases, simp)
apply (frule s2list_ex, simp)
apply (rule someI2_ex)
apply (rule_tac x="#s" in lncases, simp)
apply (frule s2list_ex, simp)
apply (rule iffI)
apply (drule sym, drule sym, simp)
apply (simp add: sq_le_list)
by (simp add: sq_le_list)


lemma list2s_mono: "l ⊑ l'⟹list2s l ⊑ list2s l'"
by (simp add: sq_le_list)


lemma monofun_lcons: "monofun (λl. a # l)"
apply (rule monofunI)
apply (simp add: atomize_imp)
apply (rule_tac x="a" in spec)
apply (induct_tac x, simp+)
apply (rule allI)
apply (simp add: sq_le_list)
apply (rule allI)
apply (rule impI)
apply (simp add: sq_le_list)
by (rule monofun_cfun_arg, simp)


lemma s2list2lcons: "#s ≠∞⟹s2list (↑a • s) = a # (s2list s)"
apply (rule_tac x="#s" in lncases, simp+)
apply (simp add: atomize_imp)
apply (rule_tac x="s" in spec)
apply (rule_tac x="a" in spec)
apply (induct_tac k, simp+)
apply (rule allI,rule allI, rule impI)
apply (rule_tac x="xa" in scases, simp+)
apply (simp add: s2list_def)
apply (rule someI2_ex)
apply (frule s2list_ex, simp)
apply (rule someI2_ex)
apply (frule s2list_ex)
apply (erule exE)
apply (rule_tac x="x#a#l" in exI, simp+)
by (rule list2s_inj [THEN iffD1], simp)


lemma [simp]: "s2list (list2s l) = l"
apply (induct_tac l, simp+)
by (subst s2list2lcons, simp+)


lemma slistl5[simp]: "list2s (l @ [m]) = list2s l • ↑m"
by (induct_tac l, simp+)


(* ------------------------------------------------------------------- *)
subsection ⟨List- and stream-processing functions⟩
(* ------------------------------------------------------------------- *)

(* concatenating streams corresponds to concatenating lists *)
lemma listConcat: "<l1> • <l2> = <(l1 @ l2)>"
apply(induction l1)
by auto

(* smap for streams is equivalent to map for lists *)
lemma smap2map: "smap g·(<ls>) = <(map g ls)>"
apply(induction ls)
by auto

(* the notion of length is the same for streams as for lists *)
```

```
lemma list2streamFin: "#(<ls>) = Fin (length ls)"
apply(induction ls)
by auto


lemma mono_slpf2spf:
  "monofun f⟹monofun (λs. list2s (f (s2list (stake k·s))))"
apply (rule monofunI)
apply (simp add: atomize_imp)
apply (rule_tac x="y" in spec)
apply (rule_tac x="x" in spec)
apply (induct_tac k, simp+)
apply (rule impI)
apply (drule mp, assumption)
apply (rule allI)+
apply (rule impI)
apply (drule lessD, simp)
apply (erule disjE, simp)
apply (rule list2s_mono)
apply (rule_tac f="f" in monofunE,simp+)
apply (erule exE)+
apply (erule conjE)
apply (erule exE,simp)
apply (erule conjE)
apply (rule list2s_mono)
apply (rule_tac f="f" in monofunE,simp+)
apply (rule_tac x="xa" in scases,simp)
apply (subst list2s_emb,simp+)
apply (rule monofun_cfun_arg)+
by simp


lemma chain_slpf2spf:
  "monofun f⟹list2s (f (s2list (stake i·x))) ⊑ list2s (f (s2list (stake (Suc i)·x)))"
apply (rule list2s_mono)
apply (rule_tac f="f" in monofunE,simp+)
apply (subst list2s_emb,simp+)
by (rule chainE, simp)


lemma slpf2spfI_contI:
  "monofun f⟹
  cont (λs. (⨆k. list2s (f (s2list (stake k·s)))))"
apply (rule cont2cont_lub)
apply (rule chainI)
apply (rule chain_slpf2spf, simp)
apply (rule pr_contI)
apply (rule mono_slpf2spf, assumption)
apply (rule allI)
by (rule_tac x="k" in exI, simp)


lemma slpf2spf_cont:
  "monofun f⟹
     (Λ s. (⨆k. list2s (f (s2list (stake k·s)))))·s = (⨆k. list2s (f (s2list (stake k·s))))"
apply (subst beta_cfun)
by (rule slpf2spfI_contI, assumption, simp)


lemma slpf2spf_def2:
  "monofun f⟹slpf2spf f·x = (⨆k. list2s (f (s2list (stake k·x))))"
apply (simp add: slpf2spf_def)
by (rule slpf2spf_cont)


lemma sislivespf_slpf2spf:
  "monofun f⟹sislivespf (slpf2spf f)"
apply (rule sislivespfI)
apply (rule_tac x="#x" in lncases, assumption)
apply (simp add: slpf2spf_def2)
apply (subgoal_tac
  "finite_chain (λk. list2s (f (s2list (stake k·x))))")
apply (simp add: finite_chain_def)
apply (erule conjE, erule exE)
apply (frule lub_finch1, simp+)
apply (frule lub_eqI, simp)
apply (simp add: finite_chain_def, rule conjI)
apply (rule chainI)
apply (rule chain_slpf2spf, assumption)
apply (rule_tac x="k" in exI)
apply (simp add: max_in_chain_def)
apply (rule allI, rule impI)
apply (subgoal_tac "stake j·x = stake k·x", simp)
apply (subst fin2stake [THEN sym], simp+)
by (simp add: min_def)


lemma sspf2lpf_mono:
  "sislivespf f⟹monofun (sspf2lpf f)"
apply (rule monofunI)
apply (simp add: sspf2lpf_def)
apply (subst list2s_emb)
apply (rule notI, frule sislivespfD2, simp+)+
apply (rule monofun_cfun_arg)
```

```
by (simp add: sq_le_list)


lemma monofun_spf_ubl[simp]:
  "#((f·x)::'a stream) = ∞⟹f·(x • y) = f·x"
apply (rule sym)
apply (rule eq_less_and_fst_inf [of "f·x"])
by (rule monofun_cfun_arg, auto)


lemma inj_sfilter_smap_siteratel1:
  "inj f⟹sfilter {f j}·(smap f·(siterate Suc (Suc (k + j)))) = ε"
apply (rule ex_snth_in_sfilter_nempty [rule_format])
apply (simp add: atomize_imp)
apply (rule impI)
apply (subst smap_snth_lemma, simp+)
apply (simp add: snth_siterate_Suc)
apply (rule notI)
by (frule_tac x="Suc (n+(k+j))" and y="j" in injD, simp+)

(* an element m can't appear infinitely often in a stream produced by mapping an injective function f
   over the natural numbers *)
lemma inj_sfilter_smap_siteratel2[simp]:
  "inj f⟹#(sfilter {m}·(smap f·(siterate Suc j))) ≠∞"
apply (case_tac "m∈range f")
apply (rule_tac b="m" and f="f" in rangeE, simp+)
apply (rule notI)
apply (drule_tac n="Suc x" in slen_sfilter_sdrop [rule_format], simp)
apply (simp add: sdrop_siterate)
apply (simp add: inj_sfilter_smap_siteratel1)
by (simp add: sfilter_smap_nrange)


(* ----------------------------------------------------------------------- *)
subsection ⟨compact lemmas⟩
(* ----------------------------------------------------------------------- *)

(* finite chains have lub  *)
lemma finChainapprox:fixes Y::"nat ⇒ 'a stream"
  assumes "chain Y" and "# (⨆i. Y i) =Fin  k"
  shows "∃i. Y i = (⨆i. Y i)"
  using assms(1) assms(2) inf_chainl4 lub_eqI lub_finch2 by fastforce

(* finite streams are compact *)
lemma finCompact: assumes "#(s::'a stream) = Fin k"
  shows "compact s"
  proof (rule compactI2)
  fix Y assume as1: "chain Y" and as2: "s ⊑ (⨆i. Y i)"
  show "∃i. s ⊑ Y i" by (metis approxl2 as1 as2 assms finChainapprox lub_approx stream.take_below)
qed

(* the empty stream is compact *)
lemma "compact ε"
  by simp

(* ↑x is compact *)
lemma "compact (↑x)"
  by (simp add: sup'_def)

(* not so compact stuff *)
lemma nCompact: assumes "chain Y" and "∀i. (Y i ⊑ x)" and "∀i.  (Y i ≠ x)" and "x ⊑ (⨆i. Y i)"
  shows "¬(compact x)"
  by (meson assms below_antisym compactD2)

(* infinite streams are not compact *)
lemma infNCompact: assumes "#(s::'a stream) = ∞"
  shows"¬ (compact s)"
  proof (rule nCompact)
     show "chain (λi. stake i·s)" by simp
    show "∀i. stake i·s ⊑ s" by simp
   show "∀i. stake i·s ≠ s" by (metis Inf'_neq_0 assms fair_sdrop sdropostake strict_slen)
  show "s ⊑ (⨆ i. stake i·s)" by (simp add: reach_stream)
qed

(* sinftimes (↑x) is not compact *)
lemma "¬ (compact (sinftimes (↑x)))"
  by (simp add: infNCompact slen_sinftimes)

(* add function *)
definition add:: "nat stream → nat stream → nat stream" where
"add ≡ Λ s1 s2 . smap (λ s3. (fst s3) + (snd s3))·(szip·s1·s2)"

(* add is continuous *)
lemma "cont (λ s1 s2 . smap (λ s3. (fst s3) + (snd s3))·(szip·s1·s2))"
by simp

(* add returns the same result as merge plus *)
lemma "add = merge plus"
by(simp add: add_def merge_def)

(* unfolding rule for add *)
lemma add_unfold: "add·(↑x • xs)·(↑y• ys) = ↑(x+y) • add·xs·ys"
  by(simp add: add_def)
```

```
(* relation between snth and add *)
lemma add_snth: "Fin n <#xs⟹Fin n < #ys⟹snth n (add·xs·ys) = snth n xs + snth n ys"
  by (simp add: add_def smap_snth_lemma szip_nth)

(* add applied to the empty stream always returns the empty stream *)
lemma add_eps1[simp]: "add·ϵ·ys = ϵ"
  by(simp add: add_def)

(* add applied to the empty stream always returns the empty stream *)
lemma add_eps2[simp]: "add·xs·ϵ = ϵ"
  by(simp add: add_def)

(* relation between srt and add *)
lemma [simp]: "srt·(add·(↑a • as)·(↑b • bs)) = add·as·bs"
  by (simp add: add_unfold)

(* helper lemma for commutativity of add *)
lemma add_commu_helper: assumes "⋀y. add·x·y = add·y·x"
  shows "add·(↑a • x)·y = add·y·(↑a • x)"
  apply(cases "y = ϵ")
   apply auto[1]
  by (metis (no_types, lifting) Groups.add_ac(2) assms add_unfold surj_scons)

(* the add function is commutative *)
lemma add_commutative: "add·x·y = add·y·x"
  apply(induction x arbitrary: y)
    apply(simp_all)
  by (metis add_commu_helper stream.con_rews(2) stream.sel_rews(5) surj_scons)

(* relation between add, lnsuc and srt *)
lemma add_len: assumes "xs≠⊥" and "u≠⊥"
  shows "#(add·xs·(u && ys)) = lnsuc·(#(add·(srt·xs)·ys))"
  by (metis (no_types, lifting) add_unfold assms(1) assms(2) slen_scons stream.con_rews(2) stream.sel_rews(5)
       surj_scons)

(* helper lemma for add2smapsu *)
lemma add2smapsuc_helper:" Suc = (λz. z+1)"
by auto

(* relation between add and smap applied to (Suc)·sc *)
lemma inf_srcdups_stake_snth_sdrop:
  assumes "#s = ∞" and "srcdups·s = srcdups·(stake k·s)"
  shows "snth n (sdrop k·s) = snth k s"
proof (induction n)
  case 0
  then show ?case
    by (simp add: snth_def)
next
  case (Suc n)
  then have "snth (Suc n) (sdrop k·s) ≠ snth k s⟹False"
  proof -
    assume "snth (Suc n) (sdrop k·s) ≠ snth k s"
    have "#s = ∞"
      by (simp add: assms(1))
    moreover have "snth (n + k) s ≠ snth (n + k + 1) s"
      by (metis Suc.IH Suc.prems Suc_eq_plus1 ‹snth (Suc n) (sdrop k·s) ≠ snth k s›
           semiring_normalization_rules(23) snth_sdrop)
    ultimately have "srcdups·(stake (n + k + 1)·s) ≠ srcdups·s"
      by (metis add2smapsuc_helper srcdups_snth_stake_inf)
    moreover have "srcdups·(stake (n + k + 1)·s) = srcdups·(stake k·s)"
    proof -
      have "srcdups·(stake k·s) ⊑ srcdups·(stake (n + k + 1)·s)"
      proof -
        have "k + (1 + n) = n + k + 1"
          by simp
        then show ?thesis
          by (metis (no_types) minimal monofun_cfun_arg sconc_snd_empty stake_add)
      qed
      then show ?thesis
        by (metis assms(2) below_antisym monofun_cfun_arg stream.take_below)
    qed
    ultimately show "False"
      by (simp add: assms(2))
  qed
  then show ?case
    by blast
qed

lemma srcdups_split:
  assumes "#(srcdups·s) < ∞" and "#s = ∞"
  obtains n where "s = (stake n·s) • ((↑(snth n s))^∞)"
proof -
  obtain k where "srcdups·s = srcdups·(stake k·s)"
    by (metis len_stream_def assms(1) fun_approxl2 lnat_well_h2)
  then have "sdrop k·s ≠ srt·(sdrop k·s)⟹False"
  proof -
    assume "sdrop k·s ≠ srt·(sdrop k·s)"
    moreover have "#(sdrop k·s) = #(srt·(sdrop k·s))"
      by (metis assms(2) fair_sdrop sdrop_back_rt)
    ultimately obtain n where "snth n (sdrop k·s) ≠ snth n (srt·(sdrop k·s))"
      using snths_eq by blast
    moreover have "snth n (srt·(sdrop k·s)) = snth k s"
      by (metis ‹srcdups·s = srcdups·(stake k·s)› assms(2) inf_srcdups_stake_snth_sdrop snth_rt)
    then show "False"
```

```
      by (metis (no_types) ⟨snth n (srt·(sdrop k·s)) = snth k s⟩ ⟨srcdups·s = srcdups·(stake k·s)⟩ assms(2)
            calculation inf_srcdups_stake_snth_sdrop)
  qed
  then have "sdrop k·s = ↑(snth k s) • (sdrop k·s)"
    by (metis Inf'_neq_0 assms(2) fair_sdrop snth_def strict_slen surj_scons)
  then have "sdrop k·s = ((↑(snth k s))^∞)"
    using s2sinftimes by blast
  then have "s = (stake k·s) • ((↑(snth k s))^∞)"
    by (metis split_streaml1)
  thus ?thesis
    by (metis that)
qed

lemma add2smapsuc:"add·((↑1)^∞)·s=smap (Suc)·s"
  by (metis add_eps2 add_unfold plus_1_eq_Suc rek2smap sinftimes_unfold)

(* relation between add and smap *)
lemma add2smap_f: "add·((↑x)^∞) = smap (λz. z+x)"
apply(rule cfun_eqI)
by (metis (no_types, lifting) add_commutative add_eps2 add_unfold rek2smap sinftimes_unfold)

(* relation between shd and updis *)
lemma shd_updis:"shd (u && s) = (THE a. updis a= u)"
by(simp add: shd_def the_equality, metis)

(* smap applied to the identity + stream returns the stream *)
lemma smap_id:"smap (id)·s = s"
apply(induction s, auto)
apply(simp add: smap_hd_rst)
using stream.con_rews(2) surj_scons by fastforce

(* smap applied to the identity + stream returns the identity + stream *)
lemma smapid2ID_h:"smap id· s = ID· s"
apply(simp add: ID_def)
by(rule snths_eq, auto, simp add: smap_id)

(* smap applied to the identity returns the identity *)
lemma smapid2ID:"smap id= ID"
by(rule cfun_eqI, simp add: smapid2ID_h)

(* add applied to ↑0∞·s returns the identity + stream *)
lemma add2ID_h:"add·((↑0)^∞)·s=ID·s"
proof(induction s rule: ind)
  case 1
  then show ?case
    by simp
next
case 2
  then show ?case
    by simp
next
  case (3 a s)
  then show ?case
    by (metis ID1 add_unfold semiring_normalization_rules(5) sinftimes_unfold)
qed

(* add applied to ↑0∞ returns the identity *)
lemma add2ID:"add·↑0^∞ = ID"
by (simp add: add2ID_h cfun_eqI)

(* ------------------------------------------------------------------------ *)
subsection ⟨Reachability⟩
(* ------------------------------------------------------------------------ *)

definition freach_h :: "('s ⇒ 'i ⇒ ('s × 'o)) ⇒ 's ⇒ 'i set ⇒ 's set  ⇒ 's set" where
  "freach_h f initial domain states
     = states ∪ {fst (f s elem) | elem s. s ∈ (states ∪ {initial}) ∧ elem ∈ domain}"

definition freach :: "('s ⇒ 'i ⇒ ('s × 'o)) ⇒ 's ⇒ 'i set ⇒ 's set" where
  "freach f initial domain = {initial} ∪ fix·(Λ S. freach_h f initial domain S)"

lemma freach_h_mono: "monofun (λS. freach_h f initial domain S)"
  apply (rule monofunI)
  by (auto simp add: less_set_def freach_h_def)

lemma freach_h_cont: "cont (λS. freach_h f initial domain S)"
  apply (rule contI2)
  apply (simp add: freach_h_mono)
  by (auto simp add: chain_def less_set_def lub_eq_Union freach_h_def)

lemma freach_insert:
  "freach f initial domain = {initial} ∪ freach f initial domain
     ∪ {fst (f s elem) | elem s. s ∈ (freach f initial domain ∪ {initial}) ∧ elem ∈ domain}"
  apply (subst freach_def)
  by (smt Abs_cfun_inverse2 Collect_cong UnCI UnE Un_def fix_eq freach_def freach_h_cont
      freach_h_def mem_Collect_eq)

lemma freach_empty [simp]: "freach f i {} = {i}"
  apply (simp add: freach_def)
  apply (subst fix_strict)
  apply (simp add: freach_h_cont freach_h_def)
  by (simp add: UU_eq_empty)
```

```
lemma freach_mono: "monofun (freach f i)"
  apply (rule monofunI)
  apply (simp add: less_set_def)
  apply (simp add: freach_def)
  apply (rule parallel_fix_ind , auto)
  apply (rule admI)
  apply (subgoal_tac "fst (⨆i. Y i) = (⋃i. fst (Y i))", simp)
  apply (subgoal_tac "snd (⨆i. Y i) = (⋃i. snd (Y i))", simp)
  apply blast
  apply (metis (mono_tags, lifting) lub_prod set_cpo_simps(2) snd_conv)
  apply (simp add: lub_prod set_cpo_simps(2))
  by (auto simp add: freach_h_cont freach_h_def)

lemma freach_initial_in:
  "i ∈ freach f i domain"
  by (simp add: freach_def)

lemma freach_suc_in:
  assumes "a ∈ freach f i domain"
  shows "∀e ∈ domain. fst (f a e) ∈ freach f i domain"
  using assms freach_insert by fastforce

lemma freach_ext_dom:
  assumes "b ∈ freach f i (sValues·(srt·s))"
  shows "b ∈ freach f i (sValues·s)"
  using assms
  apply (subgoal_tac "freach f i (sValues·(srt·s)) ⊑ freach f i (sValues·s)")
  apply (metis SetPcpo.less_set_def contra_subsetD)
  apply (subgoal_tac "(sValues·(srt·s)) ⊑ (sValues·s)")
  apply (rule monofunE [of "freach f i"], auto)
  apply (simp add: freach_mono)
  by (metis (full_types) SetPcpo.less_set_def sdrop_0 sdrop_forw_rt sdrop_sValues)

lemma freach_ext_dom2:
  assumes "s ≠ ϵ"
  shows "freach f i (sValues·(srt·s)) ⊑ freach f i (sValues·s)"
  using assms
  by (simp add: SetPcpo.less_set_def freach_ext_dom subset_iff)

lemma freach_step_ext:
  assumes "s ≠ ϵ"
  shows "freach f (fst (f i (shd s))) (sValues·s) ⊑ freach f i (sValues·s)"
  using assms
  apply (simp add: SetPcpo.less_set_def)
  apply (subst freach_def)
  apply (rule fix_ind)
  apply (rule admI)
  apply (simp add: SUP_least lub_eq_Union)
  apply (metis UU_eq_empty freach_initial_in freach_suc_in sfilter_ne_resup sfilter_sValuesl4
         singletonD subsetI sup_bot.right_neutral)
  apply (auto simp add: freach_h_cont freach_h_def)
  apply (simp add: freach_suc_in)
  by (meson contra_subsetD freach_suc_in)

lemma freach_step:
  assumes "s ≠ ϵ"
  shows "freach f (fst (f i (shd s))) (sValues·(srt·s)) ⊑ freach f i (sValues·s)"
  using assms
  by (meson freach_ext_dom2 freach_step_ext rev_below_trans)

lemma f2snth_sscanlasnd_freach:
  assumes "Fin j < #s"
    and "⋀b e. e ∈ sValues·s⟹b ∈ freach f i (sValues·s)⟹P e (snd (f b e))"
  shows "P (snth j s) (snth j (sscanlAsnd f i·s))"
  using assms
  proof (induction j arbitrary: s f i)
    case 0
    then show ?case
      by (metis Fin_02bot freach_initial_in lnless_def lnzero_def slen_empty_eq snth2sValues
          snth_shd sscanlasnd_shd)
    next
    case (Suc j)
    then show ?case
      apply (simp add: snth_rt sscanlasnd_srt)
      apply (rule Suc.IH)
      apply (meson not_le slen_rt_ile_eq)
      apply (rule Suc.prems)
      apply (metis (no_types, hide_lams) contra_subsetD empty_iff lscons_conv sValues_subset
             sfilterEq2sValues sfilter_ne_resup snth2sValues stream.con_rews(2) surj_scons)
      by (metis (no_types, lifting) SetPcpo.less_set_def contra_subsetD empty_is_shortest
          freach_step)
  qed

lemma freach_initial_transfer:
  assumes "P i"
    and "∀e ∈ sValues·s. ∀b. P b⟶P (fst (f b e))"
  shows "∀b ∈ freach f i (sValues·s). P b"
  using assms
  apply (subst freach_def)
  apply (rule fix_ind)
  apply simp_all
  apply (rule admI)
  apply (simp add: lub_eq_Union)
  apply (simp add: UU_eq_empty)
```

```
  by (auto simp add: freach_h_cont freach_h_def)

hide_const %invisible slen
end
```

# Appendix C

# Stream Bundle Theories

## C.1 Datatype

```
(*:maxLineLen=68:*)
theory Datatypes

imports inc.Prelude

begin

default_sort %invisible type

section ‹System specific Datatypes›




subsection ‹Channel Datatype›

text ‹The channel datatype is fixed for every system. The
temperature alarm system \cref{fig:sensor} would have the channel
type ‹cempty | cTemp | cAlarm›. This datatype contains every used
channel and at least one dummy "channel" for defining components with no
input or no output channels. The ‹cempty› element in the channel
datatype is a technical work-around since there are no empty types in Isabelle.
Thus, even the type of an empty channel set has to contain an element.›


datatype channel = DummyChannel


hide_const DummyChannel

subsection ‹Message Datatype›


text‹Analogous to the channel datatype, the message datatype
contains the messages that channels can transmit. Hence, every kind
of message has to be described here. The messages for our sensor
system would be defined as ‹\<I> int | \<B> bool›. This message type
contains all messages transmittable in a system.›

datatype M = DummyMessage



hide_const DummyMessage



instance M :: countable
  apply(intro_classes)
  by(countable_datatype)


definition %invisible ctype :: "channel ⇒ M set" where
```

```
"ctype c ≡ {}" (* Should be invisible to the user, would only confuse  *)


text⟨Such a mapping is described by the @{const ctype} function. Only
messages included in the @{const ctype} are allowed to be transmitted
on the respective channel. For the sensor system, channel ⟨c1⟩ would
be allowed to transmit all ⟨\<I> int⟩ and ⟨c2⟩ all ⟨\<B> bool⟩ messages.
The ⟨cempty⟩ channel can never transmit any message, hence,
@{const ctype} of ⟨cempty⟩ would be empty.⟩


theorem ctypeempty_ex: "∃c. ctype c = {}"
  by (simp add: ctype_def)


hide_fact %invisible ctype_def


end
```

## C.2 Channel

```
(*:maxLineLen=68:*)
theory Channel

imports HOLCF user.Datatypes
begin


subsection⟨Domain Classes⟩




paragraph ⟨Preliminaries for Domain Classes \\⟩


definition cEmpty :: "channel set" where
"cEmpty = {c. ctype c = {}}"


lemma cempty_exists: "cEmpty ≠ {}"
  by(simp add: cEmpty_def ctypeempty_ex)

paragraph⟨Classes \\⟩


class rep =
  fixes Rep :: "'a ⇒ channel"
begin
  abbreviation "Abs ≡ inv Rep"
end


class chan = rep +
  assumes chan_botsingle:
      "range Rep ⊆ cEmpty ∨
       range Rep ∩ cEmpty = {}"
  assumes chan_inj[simp]:"inj Rep"
begin
  theorem abs_rep_id[simp]:"Abs (Rep c) = c"
    by simp
end


paragraph ⟨Class Functions \\⟩

text⟨We will now define a function for types of @{class chan}. It
returns the Domain of the type. As a result of our class assumptions
and of interpreting empty channels as non existing, our domain is
empty, if and only if the input type contains channel(s) from
@{const cEmpty}. A type can be defined as the input of a function by
using ⟨itself⟩ type in the signature. Then, input
⟨chDom TYPE ('cs)⟩ results in the domain of ⟨'cs⟩.⟩

definition chDom::"'cs::chan itself ⇒ channel set" where
"chDom a ≡ range (Rep::'cs ⇒ channel) - cEmpty"


abbreviation chDomEmpty ::"'cs::chan itself ⇒ bool" where
```

```
"chDomEmpty cs ≡ chDom cs = {}"

lemma inchdom[simp]:"¬chDomEmpty TYPE('cs)
                     ⟹Rep (c::'cs::chan) ∈ chDom TYPE('cs)"
  apply(simp add: chDom_def)
  using chan_botsingle by blast


paragraph⟨Class somechan \\⟩

text⟨Types of ⟨somechan⟩ can transmit at least one message
on every channel.⟩

class somechan = rep +
  assumes chan_notempty: "(range Rep) ∩ cEmpty = {}"
      and chan_inj[simp]:"inj Rep"
begin end


subclass (in somechan) chan
  apply(standard)
  by (simp_all add: local.chan_notempty)

lemma somechannotempty[simp]:"¬chDomEmpty TYPE('c::somechan)"
  using chDom_def somechan_class.chan_notempty by fastforce

lemma somechandom:"chDom(TYPE('c::somechan))
                    = range(Rep::'c⇒channel)"
  by(simp add: chDom_def somechan_class.chan_notempty Diff_triv)


paragraph⟨Class emptychan \\⟩

text⟨Types of ⟨emptychan⟩ can not transmit any message on any
channel.⟩

class emptychan = rep +
  assumes chan_empty:"(range Rep) ⊆ cEmpty"
      and chan_inj[simp]:"inj Rep"
begin end


subclass (in emptychan) chan
  apply(standard)
  by (simp_all add: local.chan_empty)


theorem emptychanempty[simp]:"chDomEmpty TYPE('cs::emptychan)"
  by (simp add: chDom_def emptychan_class.chan_empty)


lemma emptychan_type[simp]: "ctype (Rep (c::('cs::emptychan))) = {}"
  using chan_empty cEmpty_def by auto


subsubsection %invisible⟨ rep abs chan lemmata ⟩
default_sort %invisible chan

lemma repinrange[simp]:"Rep (c::'c) = x
                        ⟹x∈ range(Rep::'c ⇒ channel)"
  by blast

lemma chan_eq[simp]:"Rep (c::'c) = x⟹x∈ range(Rep::'d⇒channel)
                     ⟹Rep((Abs::channel ⇒ 'd)(Rep c)) = x"
  by (simp add: f_inv_into_f)

lemma cempty_rule[simp]:assumes"chDomEmpty(TYPE('c))"
  shows"Rep (c::'c) ∈ cEmpty"
  using assms chan_botsingle chDom_def by blast

lemma cnotempty_rule[simp]:assumes"¬chDomEmpty(TYPE('c))"
  shows"Rep (c::'c) ∉ cEmpty"
  using assms chan_botsingle chDom_def by blast

lemma cnotempty_cdom[simp]:assumes"¬chDomEmpty(TYPE('c))"
  shows"Rep (c::'c) ∈ chDom(TYPE('c))"
  using assms by (simp add: chDom_def)

lemma cdom_notempty[simp]:assumes"c ∈chDom TYPE('c)"
  shows" c ∉ cEmpty"
  using assms by (simp add: chDom_def)

lemma notcdom_empty[simp]:assumes"Rep (c::'c) ∉chDom TYPE('c)"
  shows" Rep c ∈ cEmpty"
  using assms by (simp add: chDom_def)

lemma chdom_in: fixes c::"'cs::chan"
  assumes "chDom TYPE('cs) ≠ {}"
```

```
    shows "Rep c ∈ chDom TYPE('cs)"
  by (metis Diff_eq_empty_iff Diff_triv assms chDom_def
      chan_botsingle rangeI)

lemma abs_reduction[simp]:
  fixes c::"'cs1::chan"
  assumes"Rep c∈chDom TYPE('cs1)"
  and "Rep c ∈ chDom TYPE('cs2)"
  shows "Rep ((Abs::channel⇒ 'cs2) (Rep c)) = Rep c"
  by (metis DiffD1 assms(2) chDom_def chan_eq)

lemma abs_fail:
  fixes c::"'cs1"
  assumes"Rep c∈chDom TYPE('cs1)"
  and "Rep ((Abs::channel⇒'cs2) (Rep c)) ≠ Rep c"
  shows "Rep c ∉ chDom TYPE('cs2)"
  using assms(1) assms(2) by auto

lemma dom_ref:
  fixes c::"'cs1"
  assumes"Rep c∈chDom TYPE('cs1)"
  and "Rep ((Abs::channel⇒'cs2) (Rep c)) = Rep c"
  shows "Rep c ∈ chDom TYPE('cs2)"
  using assms(1) assms(2) chDom_def by fastforce

lemma rep_reduction:
  assumes " c ∈ chDom TYPE('cs2)"
  shows "Rep ((Abs::channel⇒ 'cs2) c) = c"
  by (metis DiffD1 assms chDom_def f_inv_into_f)

lemma rep_reduction2[simp]:
  assumes "Rep c ∈ chDom TYPE('c)"
  shows"Abs (Rep ((Abs::channel ⇒ 'c) (Rep c))) = Abs (Rep c)"
  using assms rep_reduction by force

lemma abs_eql:
  fixes c::"channel"
  and c1::"'cs1"
  and c2::"'cs2"
  assumes "Rep c1 = Rep c2"
  and "Rep c1 = c"
  shows "Abs c = c1"
  and "Abs c = c2"
  using assms(2) apply auto[1]
  using assms(1) assms(2) by auto

lemma cdomemty_type[simp]:
 "chDomEmpty TYPE('cs)⟹ctype (Rep (c::'cs)) = {}"
  by(simp add: chDom_def cEmpty_def subset_eq)

declare %invisible[[show_types]]
declare %invisible[[show_consts]]

subsection ⟨Interconnecting Domain Types⟩



text⟨Furthermore, the type-system of Isabelle has no dependent
types which would allow types to be based on their value
\cite{Moura.2015}. This also effects this framework, because a type
⟨'cs1 ∪ 'cs2⟩ is always different from type ⟨'cs2 ∪ 'cs1⟩, without
assuming anything about the definition of ⟨∪⟩. This also makes
evaluating types harder. Even type ⟨'cs ∪ 'cs⟩ is not
directly reducible to type ⟨'cs⟩ by evaluating ⟨∪⟩. Of course the
same holds for the ⟨-⟩ type.⟩

subsubsection⟨Union Type⟩


typedef ('cs1,'cs2) union (infixr "∪" 20) =
        "if chDomEmpty TYPE ('cs1) ∧ chDomEmpty TYPE ('cs2)
            then cEmpty
            else chDom TYPE('cs1) ∪ chDom TYPE('cs2)"
  apply(auto)
  using chDom_def by blast

text⟨Because channels in @{const cEmpty} are interpreted as no real
channels, the union of two empty domains is defined as the
channel set @{const cEmpty}. The next step is to instantiate the
union of two members of class @{class chan} as a member of class
@{class chan}. This is rather easy, because either the union results
in @{const cEmpty}, so there are no channels where a message can
be transmitted, or it results in the union of the domains without
channels from @{const cEmpty}. Hence, the representation function
@{const Rep} is defined as the representation function @{const Rep_union}
generated from the ⟨typedef⟩-keyword. The output type union
type of two input @{class chan} types is always a member of
@{class chan} as shown in following instantiation.⟩

instantiation union :: (chan, chan) chan
begin
  definition "Rep == Rep_union"
instance
  apply intro_classes
```

```
  apply auto
  apply (metis Rep_union Rep_union_def Un_iff cdom_notempty)
  by (simp add: Channel.Rep_union_def Rep_union_inject inj_on_def)
end

lemma union_range_empty:"chDomEmpty TYPE ('cs1)
                       ∧ chDomEmpty TYPE ('cs2)⟹
                    range (Rep_union::'cs1 ∪ 'cs2 ⇒ channel) =
                    cEmpty"
  by (metis (mono_tags, lifting) type_definition.Rep_range
      type_definition_union)

lemma union_range_union:"¬(chDomEmpty TYPE ('cs1)
                       ∧ chDomEmpty TYPE ('cs2))⟹
                   range (Rep_union::'cs1 ∪ 'cs2 ⇒ channel) =
                   chDom TYPE ('cs1) ∪ chDom TYPE('cs2)"
  by (smt type_definition.Rep_range type_definition_union)


theorem chdom_union[simp]:"chDom TYPE('cs1 ∪ 'cs2) =
                           chDom TYPE ('cs1) ∪ chDom TYPE('cs2)"
  apply(subst chDom_def)
  apply(simp_all add: Rep_union_def)
  using chDom_def union_range_empty union_range_union by auto

subsubsection⟨Minus Type⟩


typedef ('cs1,'cs2) minus (infixr "-" 20) =
        "if chDom TYPE('cs1) ⊆ chDom TYPE('cs2)
              then cEmpty
              else chDom TYPE('cs1) - chDom TYPE('cs2)"
apply(cases "range Rep ⊆ range Rep", auto)
  using cempty_exists by blast+


instantiation minus :: (chan, chan) chan
begin
  definition "Rep == Rep_minus"
instance
  apply intro_classes
  apply auto
  apply (metis Diff_iff Rep_minus Rep_minus_def cdom_notempty)
  by (simp add: Channel.Rep_minus_def Rep_minus_inject inj_on_def)
end


lemma minus_range_empty:"chDom TYPE('cs1) ⊆ chDom TYPE('cs2)⟹
                 range (Rep_minus::'cs1 - 'cs2 ⇒ channel) = cEmpty"
  by (metis (mono_tags, lifting) type_definition.Rep_range
      type_definition_minus)

lemma minus_range_minus:"¬(chDom TYPE('cs1) ⊆ chDom TYPE('cs2))⟹
                   range (Rep_minus::'cs1 - 'cs2 ⇒ channel) =
                   chDom TYPE('cs1) - chDom TYPE('cs2)"
  by (metis (mono_tags, lifting) type_definition.Rep_range
      type_definition_minus)


theorem chdom_minus[simp]:"chDom TYPE('cs1 - 'cs2) =
                           chDom TYPE ('cs1) - chDom TYPE('cs2)"
  apply(subst chDom_def)
  apply(simp_all add: Rep_minus_def)
  using Diff_Int_distrib2 minus_range_empty minus_range_minus
  by auto

text⟨If we subtract domain ⟨'cs2⟩ from domain ⟨'cs1⟩ the resulting domain
should contain no channels from ⟨'cs2⟩.We also verify this correctness
property.⟩

theorem [simp]:"chDom TYPE('cs1 - 'cs2) ∩ chDom TYPE ('cs2) = {}"
  by auto


end
```

## C.3 SBelem Data Type

```
(*:maxLineLen=68:*)
theory sbElem
  imports Channel
begin


declare %invisible[[show_types]]
declare %invisible[[show_consts]]
```

```
default_sort %invisible chan

section ⟨Stream Bundle Elements⟩


fun sbElem_well :: "('cs ⇒ M) option ⇒ bool" where
"sbElem_well    None    = chDomEmpty TYPE('cs)" |
"sbElem_well (Some sbe) = (∀c. sbe c ∈ ctype(Rep c))"
  (* cbot ist leer, daher wird das nie wahr sein für das leere Bündel.
     Also geht für "cbot" nur "None" *)


typedef 'cs sbElem ("(_^√)"[1000] 999) =
        "{f::('cs ⇒ M) option. sbElem_well f}"
proof(cases "chDomEmpty(TYPE('cs))")
  case True
  then show ?thesis
    apply(rule_tac x=None in exI)
    by (simp add: chDom_def)
next
  case False
  then have "∀c∈(range (Rep::'cs⇒channel)). ctype c ≠ {}"
    using cEmpty_def chDom_def chan_botsingle by blast
  then have "sbElem_well
             (Some(λ(c::'cs). (SOME m. m ∈ ctype (Rep c))))"
    apply(simp add: sbElem_well.cases,auto)
    by (simp add: some_in_eq)
  then show ?thesis
    by blast
qed

text⟨The suffix ⟨(_^√)⟩ abbreviates ⟨'cs sbElem⟩ to ⟨'cs^√⟩.⟩



instantiation  sbElem::(chan) discrete_cpo
begin
  definition  below_sbElem::"'cs^√⇒ 'cs^√⇒ bool" where
  "below_sbElem sbe1 sbe2 ≡ sbe1 = sbe2"

  instance
    by(standard, simp add: below_sbElem_def)
end


lemma sbe_eqI:"Rep_sbElem sbe1 = Rep_sbElem sbe2⟹sbe1 = sbe2"
  by (simp add: Rep_sbElem_inject)

lemma sbelemwell2fwell[simp]:"Rep_sbElem sbe = f⟹sbElem_well f"
  using Rep_sbElem by auto


subsection⟨Properties⟩


lemma sbtypeempty_sbewell:"chDomEmpty TYPE ('cs)
                           ⟹sbElem_well (None::('cs ⇒ M) option)"
  by(simp add: chDom_def)

lemma sbtypeempty_notsbewell:"chDomEmpty TYPE ('cs)
                             ⟹¬sbElem_well (Some (f::'cs ⇒ M))"
  by(simp add: chDom_def)

lemma sbe_emptyiff: fixes sbe :: "'cs^√"
  shows "Rep_sbElem sbe = None⟷chDomEmpty TYPE('cs)"
  apply auto
  using sbelemwell2fwell apply force
  using sbElem_well.elims(2) sbelemwell2fwell sbtypeempty_notsbewell by blast



theorem sbtypeepmpty_sbenone[simp]:
  fixes sbe::"'cs^√"
  assumes "chDomEmpty TYPE ('cs)"
  shows "sbe = Abs_sbElem None"
  using assms
  apply(simp add: chDom_def)
  apply(rule sbe_eqI)
  by (metis Diff_eq_empty_iff not_Some_eq Rep_sbElem mem_Collect_eq
      chDom_def sbtypeepmpty_notsbewell)


theorem sbtypefull_none[simp]:
  fixes sbe::"'cs^√"
  assumes "¬chDomEmpty TYPE ('cs)"
  shows "Rep_sbElem sbe ≠ None"
  using sbElem_well.simps(1) assms sbelemwell2fwell by blast
```

```
theorem sbtypenotempty_somesbe:
  assumes "¬chDomEmpty TYPE ('cs)"
  shows "∃f::'cs ⇒ M. sbElem_well (Some f)"
  using assms sbElem_well.simps(1) sbelemwell2fwell by blast


 (*Not in pdf at the moment, because ugly*)

setup_lifting %invisible type_definition_sbElem
subsection ⟨sbElem functions⟩


(*works if sbe ≠ None* and 'e ⊆ 'c *)
definition sbegetch::"'e ⇒ 'c^√⇒ M"where
"sbegetch c = (λ sbe. ((the (Rep_sbElem sbe)) (Abs (Rep c))))"


lemma sbtypenotempty_fex[simp]:
"¬(chDomEmpty TYPE ('cs))⟹∃f. Rep_sbElem (sbe::'cs^√) = (Some f)"
  apply(rule_tac x="(λ(c::'c). (THE m. m= sbegetch c sbe))" in exI)
  by(simp add: sbegetch_def)


definition sbeConvert::"'c^√⇒ 'd^√"where
"sbeConvert = (λsbe. Abs_sbElem(Some (λc. sbegetch c sbe)))"

lemma chDomEmpty2chDomEmpty:"chDomEmpty TYPE ('c)⟹
  Rep (c::'c) ∈ range(Rep::'d⇒ channel)⟹chDomEmpty TYPE ('d)"
  apply(simp add: chDom_def cEmpty_def,auto)
  by (metis (mono_tags, lifting) Int_Collect cEmpty_def
      chan_botsingle insert_not_empty le_iff_inf mk_disjoint_insert
      repinrange)

lemma sbgetch_ctype:
  assumes "Rep (c::'e) ∈ range(Rep::'d ⇒ channel)"
  and "¬chDomEmpty(TYPE('d))"
  shows "sbegetch c (sbe2::'d^√) ∈ ctype ((Rep::'e ⇒ channel) c)"
  using assms apply(simp add: sbegetch_def)
  by (metis (no_types, hide_lams) assms(1) assms(2) f_inv_into_f
      option.sel sbElem_well.simps(2) sbegetch_def sbelemwell2fwell
      sbtypenotempty_fex)

lemma sberestrict_getch:
  assumes"Rep (c::'c) ∈ range(Rep::'d ⇒ channel)"
  and "¬(chDomEmpty TYPE('c))"
  and "range(Rep::'d ⇒ channel) ⊆ range(Rep::'c ⇒ channel)"
  shows "sbegetch c ((sbeConvert::'c^√⇒ 'd^√) sbe) = sbegetch c sbe"
  using assms
  apply(simp add: sbeConvert_def)
  apply(simp add: sbegetch_def)
  apply(subst Abs_sbElem_inverse)
  apply (smt Rep_sbElem chDom_def f_inv_into_f mem_Collect_eq
         option.sel rangeI sbElem_well.elims(1) sbElem_well.simps(2)
         subset_iff)
  by simp


definition sbeUnion::"'c^√⇒ 'd^√⇒ 'e^√"where
"sbeUnion = (λsbe1 sbe2. Abs_sbElem (Some(λ c.
if (Rep c ∈ (range (Rep ::'c ⇒ channel)))
    then sbegetch c sbe1
    else  sbegetch c sbe2)))"

lemma sbeunion_getchfst:
  assumes "Rep (c::'c) ∈ range(Rep::'e ⇒ channel)"
  and "¬(chDomEmpty TYPE('c))"
  and "range(Rep::'e ⇒ channel) ⊆
       range(Rep::'c ⇒ channel) ∪ range(Rep::'d ⇒ channel)"
  shows "sbegetch c ((sbeUnion::'c^√⇒ 'd^√⇒ 'e^√) sbe1 sbe2)
       = sbegetch c sbe1"
  apply(simp add: sbeUnion_def sbegetch_def)
  apply(subst Abs_sbElem_inverse)
  apply (auto simp add: chDom_def assms)
  using assms(2) sbgetch_ctype apply force
  apply (smt assms(2) sbElem_well.simps(2) Un_iff assms(1) assms(3)
         chDomEmpty2chDomEmpty chan_eq repinrange sbgetch_ctype
         subset_eq)
  by(simp add: sbegetch_def assms)


lemma sbeunion_getchsnd:
  assumes "Rep (c::'d) ∈ range(Rep::'e ⇒ channel)"
  and "Rep c ∉ range(Rep::'c ⇒ channel)"
  and "¬(chDomEmpty TYPE('d))"
  and "range(Rep::'e ⇒ channel) ⊆
       range(Rep::'c ⇒ channel) ∪ range(Rep::'d ⇒ channel)"
shows"sbegetch c ((sbeUnion::'c^√⇒ 'd^√⇒ 'e^√) sbe1 sbe2) =
      sbegetch c sbe2"
  apply(simp add: sbeUnion_def sbegetch_def)
  apply(subst Abs_sbElem_inverse)
```

```
  apply (auto simp add: chDom_def assms)
  apply (metis assms(1) assms(3) chDomEmpty2chDomEmpty chan_eq
         rangeI sbgetch_ctype)
  apply (smt assms sbElem_well.simps(2) Un_iff assms(1) assms(3)
         chDomEmpty2chDomEmpty chan_eq repinrange sbgetch_ctype
         subset_eq)
  by(simp add: sbegetch_def assms)


end
```

## C.4 SB Data Type

```
(*:maxLineLen=68:*)
theory SB
  imports stream.Stream sbElem
begin


declare %invisible[[show_types]]
declare %invisible[[show_consts]]

default_sort %invisible chan

section ⟨Stream Bundles Datatype⟩



definition sb_well :: "('c::chan ⇒ M stream) ⇒ bool" where
"sb_well f ≡ ∀c. sValues·(f c) ⊆ ctype (Rep c)"



lemma sbwellI:
  assumes"⋀c. sValues·(f c) ⊆ ctype (Rep c)"
  shows"sb_well f"
  by (simp add: assms sb_well_def)

lemma sbwellD: assumes "sb_well sb" and "ctype (Rep c) ⊆ A"
    shows "sValues·(sb c) ⊆ A"
  using assms(1) assms(2) sb_well_def by blast

lemma sbwell_ex:"sb_well (λc. ε)"
  by(simp add: sb_well_def)

lemma sbwell_adm: "adm sb_well"
  unfolding sb_well_def
  apply(rule adm_all, rule admI)
  by (simp add: ch2ch_fun l44 lub_fun)



pcpodef 'c::chan sb("(_^Ω)"[1000] 999)
        = "{f::('c::chan ⇒ M stream). sb_well f}"
  by (auto simp add: sbwell_ex sbwell_adm lambda_strict[symmetric])

(* TODO: Remove Warning *)
setup_lifting %invisible type_definition_sb

paragraph ⟨SB Type Properties \\⟩

text⟨The ⟨⊥⟩ element of our \gls{sb} type is a
mapping to empty streams.⟩

theorem bot_sb: "⊥ = Abs_sb (λc. ε)"
  by (simp add: Abs_sb_strict lambda_strict)

lemma rep_sb_well[simp]:"sb_well(Rep_sb sb)"
  using Rep_sb by auto

lemma abs_rep_sb_sb[simp]:"Abs_sb(Rep_sb sb) = sb"
  using Rep_sb_inverse by auto

lemma sbrep_cont[simp, cont2cont]: "cont Rep_sb"
  using cont_Rep_sb cont_id by blast
(*
*)
lemma sb_abs_cont2cont [cont2cont]:
  assumes "cont h"
  and "⋀x. sb_well (h x)"
  shows "cont (λx. Abs_sb (h x))"
  by (simp add: assms(1) assms(2) cont_Abs_sb)

lemma comp_abs_cont[cont2cont]:
  assumes"⋀x. sb_well (f2 x)"
  and "cont f2"
  shows "cont (Abs_sb ∘ f2)"
  apply(rule Cont.contI,simp)
  using assms cont_Abs_sb cont_def by force
```

```
lemma sb_rep_eql:assumes"⋀c. (Rep_sb sb1) c = (Rep_sb sb2) c"
  shows "sb1 = sb2"
  by(simp add: po_eq_conv below_sb_def fun_belowI assms)

text‹In case of an empty domain, no stream should be in a \gls{sb}.
Hence, every \gls{sb} with an empty domain should be ‹⊥›. This is
proven in the following theorem.›

theorem sbtypeepmpty_sbbot[simp]:
  fixes sb::"'cs^Ω"
  assumes "chDomEmpty TYPE ('cs)"
  shows "sb = ⊥"
  unfolding bot_sb
  using assms
  apply(simp add: chDom_def cEmpty_def)
  apply(rule sb_rep_eqI)
  apply(subst Abs_sb_inverse)
  apply (simp add: sbwell_ex ,auto)
  apply(insert sb_well_def[of "Rep_sb sb"],auto)
  using strict_sValues_rev by fastforce

lemma sbwell2fwell[simp]:"Rep_sb sb = f⟹sb_well f"
  using Rep_sb by auto

section ‹Functions for Stream Bundles›


subsubsection ‹Converter from sbElem to SB›


text‹First we construct a converter from @{type sbElem}s to
\Gls{sb}. This is rather straight forward, since we either have a
function from channels to messages, which we can easily convert to a
function from channels to streams. This consists only of streams
with the exact message from the @{type sbElem}. In the case of an
empty domain, we map @{const None} to the ‹⊥› element of \Gls{sb}.›

lift_definition sbe2sb::"'c^√⇒ 'c^Ω" is
"λ sbe. case (Rep_sbElem sbe) of Some f ⇒ λc. ↑(f c)
                               | None  ⇒ ⊥ "
  apply(rule sbwellI, auto)
  apply(case_tac "Rep_sbElem sbElem = None")
  apply auto
  apply(subgoal_tac "sbElem_well (Some y)",simp)
  by(simp only: sbelemwell2fwell)

text‹Through the usage of keyword ‹lift_definition› instead of
‹definition› we automatically have to proof that the output is
indeed a \gls{sb}.›

subsubsection ‹Extracting a single stream›


text‹The direct access to a stream on a specific channel is one of
the most important functions in the framework and also
often used for verifying properties. Intuitively, the signature of
such a function should be ‹'cs ⇒ 'cs^Ω → M stream ›, but we use a
slightly more general signature. Two domain types could contain
exactly the same channels, but we could not obtain the streams of a
\gls{sb} with the intuitive signature, when the type of the \gls{sb}
is different (see \cref{sec:interdom}). To avoid this, we can use the
@{const Rep} and @{const Abs} functions of our domain types to
convert between the them by representation and abstraction via the
global channel type. This also facilitates later function
definitions and reduces the total framework size by using
abbreviations of one general function that only restrict the
signature.›

lift_definition sbGetCh :: "'cs1 ⇒ 'cs2^Ω → M stream" is
"λc sb. if Rep c∈chDom TYPE('cs2)
           then Rep_sb sb (Abs(Rep c))
           else ε"
  by(intro cont2cont,simp add: cont2cont_fun)

text‹Our general signature allows the input of any channel from the
@{type channel} type. If the channel is in the domain of the input
\gls{sb}, we obtain the corresponding channel by converting the
channel to an element of our domain type with the nesting of ‹Abs›
and ‹Rep›. Is the channel not in the domain, the empty stream ‹ε›
is returned. The continuity of this function is also immediately
proven.›

lemmas sbgetch_insert = sbGetCh.rep_eq



abbreviation sbgetch_magic_abbr :: "'cs1^Ω⇒ 'cs2 ⇒ M stream"
(infix " \<^enum>\<^sub>⋆ " 65) where "sb \<^enum>\<^sub>⋆ c ≡ sbGetCh c·sb"
```

```
abbreviation sbgetch_abbr :: "'cs^Ω ⇒ 'cs ⇒ M stream"
(infix " \<^enum> " 65) where "sb \<^enum> c ≡ sbGetCh c·sb"


definition sbHdElemWell::"'c^Ω ⇒ bool" where
"sbHdElemWell  ≡ λ sb. (∀c. sb  \<^enum>  c ≠ ε)"


abbreviation sbIsLeast::"'cs^Ω ⇒ bool" where
"sbIsLeast sb ≡ ¬sbHdElemWell sb"


(* paragraph ⟨sbGetCh Properties \\⟩ *)


theorem sbgetch_insert2:"sb \<^enum> c = (Rep_sb sb) c"
  apply(simp add: sbgetch_insert)
  by (metis (full_types)Rep_sb_strict app_strict cnotempty_cdom
      sbtypeepmpty_sbbot)

lemma sbgetch_empty[simp]: fixes sb::"'cs^Ω"
    assumes "Rep c ∉ chDom TYPE('cs)"
    shows "sb \<^enum>\<^sub>* c = ε"
  by(simp add: sbgetch_insert assms)

lemma sbhdelemchain[simp]:
  "sbHdElemWell x⟹ x ⊑ y⟹sbHdElemWell y"
  apply(simp add: sbHdElemWell_def sbgetch_insert2)
  by (metis below_antisym below_sb_def fun_belowD minimal)

lemma sbgetch_ctypewell[simp]:"sValues·(sb \<^enum>\<^sub>* c) ⊆ ctype (Rep c)"
  apply(simp add: sbgetch_insert)
  by (metis DiffD1 chDom_def f_inv_into_f sb_well_def sbwell2fwell)

lemma sbmap_well:assumes"⋀s. sValues·(f s) ⊆ sValues·s"
                 shows"sb_well (λc. f (b \<^enum>\<^sub>* c))"
  apply(rule sbwellI)
  using assms sbgetch_ctypewell by fastforce

lemma sbgetch_ctype_notempty:"sb  \<^enum>\<^sub>*  c ≠ ε⟹ctype (Rep c) ≠ {}"
proof-
  assume a1: "sb  \<^enum>\<^sub>*  c ≠ ε"
  then have "∃e. e∈ sValues·(sb  \<^enum>\<^sub>*  c)"
    by (simp add: sValues_notempty strict_sValues_rev neq_emptyD)
  then show "ctype (Rep c) ≠ {}"
    using sbgetch_ctypewell by blast
qed

lemma sbhdelemnotempty:
  "sbHdElemWell (sb::'cs^Ω)⟹ ¬ chDomEmpty TYPE('cs)"
  by(auto simp add: sbHdElemWell_def chDom_def cEmpty_def)

lemma sbgetch_empty2: fixes sb::"'cs^Ω"
  assumes "chDomEmpty (TYPE ('cs))"
  shows "sb \<^enum>\<^sub>* c = ε"
  by(simp add: sbgetch_insert assms)

lemma sbempt2least: fixes sb::"'cs^Ω"
  assumes "chDomEmpty (TYPE ('cs))"
  shows  "sbIsLeast sb"
  unfolding sbHdElemWell_def
  apply simp
  by(rule exI [where x="undefined"], simp add: assms)

text⟨If a \gls{sb} ⟨sb1⟩ is prefix of another \gls{sb} ⟨sb2⟩, the
order also holds for each streams on every channel.⟩

theorem sbgetch_sbelow[simp]:"sb1 ⊑ sb2⟹sb1 \<^enum> c ⊑ sb2 \<^enum> c"
  by (simp add: mono_slen monofun_cfun_arg)

lemma sbgetch_below_slen[simp]:
  "sb1 ⊑ sb2⟹#(sb1 \<^enum>\<^sub>* c) ≤ #(sb2 \<^enum>\<^sub>* c)"
  by (simp add: mono_slen monofun_cfun_arg)

lemma sbgetch_bot[simp]:"⊥ \<^enum>\<^sub>* c = ε"
  apply(simp add: sbGetCh.rep_eq bot_sb)
  by (metis Rep_sb_strict app_strict bot_sb)


theorem sb_belowI:
  fixes  sb1 sb2::"'cs^Ω"
  assumes "⋀ c. Rep c∈chDom TYPE('cs)⟹ sb1 \<^enum> c ⊑ sb2 \<^enum> c"
  shows  "sb1 ⊑ sb2"
  apply(subst below_sb_def)
  apply(rule fun_belowI)
  by (metis (full_types) assms  po_eq_conv sbGetCh.rep_eq
      sbgetch_insert2)
```

```
theorem sb_eql:
  fixes    sb1 sb2::"'cs^Ω"
  assumes "⋀c. Rep c∈chDom TYPE('cs)⟹sb1 \<^enum> c = sb2 \<^enum> c"
  shows   "sb1 = sb2"
  apply(cases "chDom TYPE('cs) ≠ {}")
  apply(metis Diff_eq_empty_iff Diff_triv assms chDom_def
        chan_botsingle rangeI sb_rep_eqI sbgetch_insert2)
  by (metis (full_types) sbtypeepmpty_sbbot)


lemma sb_empty_eq[simp]: fixes sb1 sb2::"'cs^Ω"
  assumes  "chDomEmpty TYPE('cs)"
  shows "sb1 = sb2"
  by(rule sb_eqI, simp add: assms)

lemma slen_empty_eq:  assumes"chDomEmpty(TYPE('c))"
  shows " #(sb \<^enum> (c::'c)) =0"
  using assms chDom_def cEmpty_def sbgetch_ctype_notempty
  by fastforce

text⟨Lastly, the conversion from a @{type sbElem} to a \gls{sb}
should never result in a \gls{sb} which maps its domain to ⟨ϵ⟩.⟩

theorem sbgetch_sbe2sb_nempty:
  fixes    sbe::"'cs^√"
  assumes "¬chDomEmpty TYPE('cs)"
  shows   "sbe2sb sbe  \<^enum>  c ≠ ϵ"
  apply (simp add: sbe2sb_def)
  apply (simp split: option.split)
  apply (rule conjI)
  apply (rule impl)
  using assms chDom_def sbElem_well.simps(1) sbelemwell2fwell
  apply blast
  by (metis (no_types) option.simps(5) sbe2sb.abs_eq sbe2sb.rep_eq
      sbgetch_insert2 sconc_snd_empty srcdups_step srcdupsimposs
      strict_sdropwhile)

lemma botsbleast[simp]:"sbIsLeast ⊥"
  by(simp add: sbHdElemWell_def)

lemma sbleast_mono[simp]:"x ⊑ y⟹¬sbIsLeast x⟹¬ sbIsLeast y"
  by simp

lemma sbnleast_mex:"¬sbIsLeast x⟹x \<^enum> c ≠ ϵ"
  by(simp add: sbHdElemWell_def)

lemma sbnleast_mexs[simp]:"¬sbIsLeast x⟹∃a s. x \<^enum> c = ↑a • s"
  using sbnleast_mex scases by blast

lemma sbnleast_hdctype[simp]:
"¬sbIsLeast x⟹∀c. shd (x \<^enum> c) ∈ ctype (Rep c)"
  apply auto
  apply(subgoal_tac "sValues·(x \<^enum> c)⊆ ctype(Rep c) ")
  apply (metis sbnleast_mex sfilter_ne_resup sfilter_sValuesl3)
  by simp

lemma sbgetchid[simp]:"Abs_sb (( \<^enum> ) (x)) = x"
  by(simp add: sbgetch_insert2)


paragraph⟨Bundle Equality \\⟩



definition sbEQ::"'cs1^Ω⇒ 'cs2^Ω⇒ bool" where
"sbEQ sb1 sb2 ≡ chDom TYPE('cs1) = chDom TYPE('cs2) ∧
                (∀c. sb1 \<^enum> c = sb2 \<^enum>\<^sub>⋆ c)"

text⟨The operator checks the domain equality of both bundles and
then the equality of its streams. For easier use, an infix
abbreviation ⟨\<triangleq>⟩ is defined.⟩

abbreviation sbeq_abbr :: "'cs1^Ω  ⇒  'cs2^Ω⇒  bool"
(infixr "\<triangleq>" 70) where "sb1 \<triangleq> sb2 ≡ sbEQ sb1 sb2"

lemma sbeq_getch: assumes "sb1 \<triangleq> sb2"
  shows "sb1 \<^enum>\<^sub>⋆ c = sb2 \<^enum>\<^sub>⋆ c"
  apply(auto simp add: sbGetCh.rep_eq)
  using assms apply(auto simp add: sbEQ_def)
  by (metis(mono_tags) rep_reduction sbgetch_insert)

subsubsection ⟨Concatenation⟩


lemma sbconc_well[simp]:"sb_well (λc. (sb1 \<^enum> c) • (sb2 \<^enum> c))"
  apply(rule sbwellI)
  by (metis (no_types, hide_lams) Un_subset_iff dual_order.trans
      sbgetch_ctypewell sconc_sValues)
```

```
lift_definition sbConc:: "'cs^Ω ⇒ 'cs^Ω → 'cs^Ω" is
"λsb1 sb2. Abs_sb(λc. (sb1 \<^enum> c) • (sb2 \<^enum> c))"
  by(intro cont2cont, simp)

lemmas sbconc_insert = sbConc.rep_eq



abbreviation sbConc_abbr :: "'cs^Ω  ⇒  'cs^Ω ⇒  'cs^Ω"
(infixr "•^Ω" 70) where "sb1 •^Ω sb2 ≡ sbConc sb1·sb2"


theorem sbconc_getch [simp]:
  shows "(sb1 •^Ω sb2)  \<^enum> c = (sb1 \<^enum> c) • (sb2 \<^enum> c)"
  unfolding sbgetch_insert2 sbconc_insert
  apply(subst Abs_sb_inverse)
  apply simp
  apply(rule sbwellI)
  apply (metis (no_types, hide_lams) Un_subset_iff dual_order.trans
         sbgetch_ctypewell sbgetch_insert2 sconc_sValues)
  ..

text⟨It follows, that concatenating a \gls{sb} with the ⟨⊥⟩ bundle
in any order, results in the same \gls{sb}.⟩

theorem sbconc_bot_r[simp]: "sb •^Ω ⊥ = sb"
  by(rule sb_eqI, simp)

theorem sbconc_bot_l[simp]: "⊥ •^Ω sb = sb"
  by(rule sb_eqI, simp)

subsubsection ⟨Length of SBs⟩

text⟨We define the length of a \gls{sb} as
follows:
  \<^item> A \gls{sb} with an empty domain is infinitely long
  \<^item> A \gls{sb} with an non-empty domain is as long as its shortest
    stream

The definition for the empty domain was designed with the timed case in mind.
This definition can be used to define causality.⟩

definition sbLen::"'cs^Ω ⇒ lnat"where
"sbLen sb ≡ if chDomEmpty TYPE('cs) then ∞
                else LEAST n . n∈{#(sb \<^enum> c) | c. True}"

text⟨Our @{const sbLen} function works exactly as described. It
returns ⟨∞⟩, if the domain is empty. Else it chooses the minimal
length of all the bundles streams.⟩



lemma sblen_empty'[simp]:
  fixes    sb::"'cs^Ω"
  assumes "chDomEmpty TYPE('cs)"
  shows    "sbLen sb = ∞"
  by(simp add: sbLen_def assms slen_empty_eq)

lemma sblenleq': assumes "¬ chDomEmpty TYPE('a)" and
 "∃c::'a. #(sb\<^enum>c) ≤ k"
  shows "sbLen sb ≤ k"
  apply(simp add: sbLen_def assms)
  apply(subgoal_tac "⋀c::'a. Rep c ∉ cEmpty")
  apply auto
  apply (metis (mono_tags, lifting) Least_le assms(2)
         dual_order.trans)
  using assms(1) by simp

lemma sblengeq':
  fixes sb::"'cs^Ω"
    assumes "⋀c. (Rep c)∈chDom TYPE('cs)⟹k≤ #(sb\<^enum>c)"
  shows "k ≤ sbLen sb"
  apply(cases  "chDomEmpty(TYPE('cs))",simp add: assms)
  apply(subgoal_tac "⋀c.  k≤ #(sb\<^enum>c)")
  apply(simp add: sbLen_def)
  using LeastI2_wellorder_ex inf_ub insert_iff mem_Collect_eq
        sbLen_def assms apply smt
  by (simp add: assms)

lemma sblen_mono:"monofun sbLen"
  apply(rule monofunI,simp)
  apply(cases "chDomEmpty TYPE('a)",simp)
  apply(rule sblengeq')
  apply(rule sblenleq')
  using sbgetch_below_slen by auto

instantiation sb :: (chan) len
begin
definition len_sb::"'cs^Ω ⇒ lnat" where
"len_sb = sbLen"
```

```
instance
  apply(intro_classes)
  by (simp add: len_sb_def sblen_mono)

end



hide_const %invisible sbLen

lemma sblen_empty[simp]:
  fixes   sb::"'cs^Ω"
  assumes "chDomEmpty TYPE('cs)"
  shows   "#sb = ∞"
  by(simp add: len_sb_def assms)

lemma sblenleq:
  assumes "¬ chDomEmpty TYPE('a)"
      and "∃c::'a. #(sb\<^enum>c) ≤ k"
    shows "#sb ≤ k"
  by(simp add: len_sb_def assms sblenleq')

lemma sblengeq: assumes "⋀c. k≤ #(sb\<^enum>c)"
  shows "k ≤ #sb"
  by(simp add: len_sb_def assms sblengeq')

lemma sblen_min_len [simp]:
  assumes"¬chDomEmpty(TYPE('c))"
  shows"# (sb :: 'c^Ω) ≤ #(sb \<^enum> c)"
  apply(simp add: len_sb_def sbLen_def assms)
  by (metis (mono_tags, lifting) Least_le)

lemma sblen_min_len2: fixes sb::"'cs^Ω"
  assumes "(Rep c)∈chDom TYPE('cs)"
  shows "#sb ≤ # (sb\<^enum>c)"
  apply(rule sblen_min_len)
  using assms by blast



theorem sblen_sbconc: "#sb1 + #sb2 ≤ #(sb1 •^Ω sb2)"
  apply(cases  "chDomEmpty(TYPE('a))",simp)
  apply(rule sblengeq)
  by (metis lessequal_addition sbconc_getch sblen_min_len
      sconc_slen2)


lemma sblen_monosimp[simp]:"x ⊑ y⟹# x ≤ # y"
  by (simp add: mono_len)

text⟨This rule captures all necessary assumptions to obtain the
exact length of a \gls{sb} with a non-empty domain:
  \<^item> All streams must be at least equally long to the length of the
    \gls{sb}
  \<^item> There exists a stream with length equal to the length of the
    \gls{sb}⟩

theorem sblen_rule:
  fixes sb::"'cs^Ω"
  assumes "¬chDomEmpty TYPE('cs)"
      and "⋀c. k ≤ #(sb \<^enum> c)"
      and "∃c. #(sb \<^enum> c) = k"
  shows   "#sb = k"
  by (metis assms(1) assms(2) assms(3) dual_order.antisym
      sblen_min_len sblengeq)



theorem sblen_sbeql:
  fixes sb1 sb2::"'cs^Ω"
  assumes "sb1⊑sb2" and "#sb1 =∞"
  shows   "sb1 = sb2"
  apply(cases "chDomEmpty TYPE('cs)")
  apply (metis (full_types)sbtypeepmpty_sbbot)
using assms proof(simp add: len_sb_def sbLen_def)
  assume a1: "sb1 ⊑ sb2"
  assume a2: "(LEAST n::lnat. ∃c::'cs. n = #(sb1  \<^enum>  c)) = ∞"
  assume a3: "chDom TYPE('a) ≠ {}"
  then have "⋀c. #(sb1 \<^enum> c) = ∞"
    by (metis (mono_tags, lifting) Least_le a2 inf_less_eq)
  moreover have "⋀c. #(sb2 \<^enum> c) = ∞"
    using a1 calculation cont_pref_eq1I mono_fst_infD by blast
  then show ?thesis
    apply(subst sb_eqI[of sb1 sb2],auto)
    by (simp add: a1 calculation cont_pref_eq1I eq_less_and_fst_inf)
qed

lemma sblen_leadm:
   fixes sb::"'cs^Ω"
   shows "adm (λsb. k ≤ #sb)"
  apply(rule admI)
```

```
  using is_ub_thelub mono_len order_trans by blast

lemma sblen2slen_h:
  fixes "c1"
  assumes"¬chDomEmpty(TYPE('c))"
  and "∀c2. #((sb :: 'c^Ω) \<^enum> c1) ≤ #(sb \<^enum> c2)"
  shows "#((sb :: 'c^Ω) \<^enum> c1) = #sb"
  apply(simp add: sbLen_def len_sb_def)
  apply(subst Least_equality)
  apply(simp_all add: assms)
   apply auto[1]
  using assms(2) by auto

lemma sb_minstream_exists:
  assumes "¬chDomEmpty(TYPE('c))"
  shows "∃c1. ∀c2. #((sb :: 'c^Ω) \<^enum> c1) ≤ #(sb \<^enum> c2)"
  using  assms
proof -
  { fix cc :: "'c ⇒ 'c"
    have ff1: "∀s c l. l ≤ #(s \<^enum> (c::'c)) ∨ ¬ l ≤ #s"
      by (meson assms sblen_min_len trans_lnle)
    { assume "∃c l. ¬ l ≤ #(sb \<^enum> c)"
      then have "¬∞≤ #sb"
        using ff1 by (meson inf_ub trans_lnle)
      then have "∃c. #(sb \<^enum> c) ≤ #(sb \<^enum> cc c)"
        using ff1 by (metis less_le_not_le ln_less
                Orderings.linorder_class.linear lnle2le sblengeq) }
    then have "∃c. #(sb \<^enum> c) ≤ #(sb \<^enum> cc c)"
      by meson }
  then show ?thesis
    by metis
qed



theorem sblen2slen:
  assumes "¬chDomEmpty TYPE('cs)"
  shows "∃c. #(sb :: 'cs^Ω) = #(sb \<^enum> c)"
proof -
  obtain min_c where "∀c2. #((sb :: 'cs^Ω) \<^enum> min_c) ≤ #(sb \<^enum> c2)"
    using sb_minstream_exists assms by blast
  then have "#(sb :: 'cs^Ω) = #(sb \<^enum> min_c)" using sblen2slen_h
    using assms by fastforce
  then show ?thesis
    by auto
qed

lemma sbconc_chan_len:"#(sb1 •^Ω sb2 \<^enum> c) = #(sb1 \<^enum> c)+ #(sb2 \<^enum> c)"
  by (simp add: sconc_slen2)

lemma sblen_sbconc_eq:
  assumes "⋀c.#(sb1 \<^enum> c) = k"
  shows "(#(sb1 •^Ω sb2)) = (#sb2) + k"
  apply(cases "chDomEmpty(TYPE('a))",simp)
  apply (simp add: plus_lnatInf_r)
  apply(subgoal_tac "#sb1 = k")
  apply(rule sblen_rule,simp)
  apply (metis add.commute dual_order.trans sblen_min_len
     sblen_sbconc)
  apply (metis assms lnat_plus_commu sbconc_chan_len sblen2slen)
  by(rule sblen_rule,simp_all add: assms)

lemma sblen_sbconc_rule:
  assumes "⋀c.#(sb1 \<^enum> c) ≥ k"
  shows "(#(sb1 •^Ω sb2)) ≥ (#sb2) + k"
  by (metis (full_types) add.commute assms dual_order.trans
      lessequal_addition order_refl sblen_sbconc sblengeq)



theorem sbelen_one[simp]:
  fixes   sbe::"'cs^√"
  assumes "¬chDomEmpty TYPE('cs)"
  shows   " #(sbe2sb sbe) = 1"
proof-
  have "⋀c. #(sbe2sb (sbe::'cs^√) \<^enum> (c :: 'cs )) = 1"
    apply(simp add: sbe2sb_def)
    apply(subgoal_tac "Rep_sbElem sbe ≠ None")
    apply auto
    apply(simp add: sbgetch_insert2)
    apply(subst Abs_sb_inverse,auto)
    apply (metis (full_types) option.simps(5) sbe2sb.rep_eq
       sbwell2fwell)
    apply (simp add: one_lnat_def)
    by(simp add: assms)
  then show ?thesis
    apply(subst sblen_rule)
    by(simp_all add: assms)
qed

lemma sbe2slen_1:  assumes"¬chDomEmpty(TYPE('a))"
```

```
  shows  "⋀c::'a. #(sbe2sb sbe  \<^enum>  c) = (1::lnat)"
    apply(simp add: sbe2sb_def)
    apply(subgoal_tac "Rep_sbElem sbe ≠ None")
    apply auto
    apply(simp add: sbgetch_insert2)
    apply(subst Abs_sb_inverse,auto)
    apply (metis (full_types) option.simps(5) sbe2sb.rep_eq
           sbwell2fwell)
    apply (simp add: one_lnat_def)
    by(simp add: assms)

lemma sbnleast_len[simp]:"¬sbIsLeast x⟹#x ≠ 0"
  apply(rule ccontr,auto)
  apply(simp add: sbHdElemWell_def)
  apply(cases "chDomEmpty TYPE('a)",simp)
  by (metis Stream.slen_empty_eq sblen2slen)

lemma sblen_eql2:
  fixes sb1 sb2::"'cs^Ω"
  assumes "sb1 ⊑ sb2"
  and "⋀c. Rep c∈chDom TYPE('cs)⟹#(sb1 \<^enum> c) = #(sb2 \<^enum> c)"
  shows "sb1 = sb2"
  by (simp add: assms(1) assms(2) eq_slen_eq_and_less
      monofun_cfun_arg sb_eql)

lemma sbnleast_dom[simp]:
  "¬sbIsLeast (x::'cs^Ω)⟹¬chDomEmpty TYPE('cs)"
  using sbhdelemnotempty by blast

lemma sbleast2sblenempty[simp]:
"sbIsLeast (x::'cs^Ω)⟹chDomEmpty TYPE('cs) ∨ # x = 0"
  apply(simp only: sbLen_def len_sb_def sbHdElemWell_def,auto)
  by (metis (mono_tags, lifting) LeastI_ex Least_le gr_0 leD lnle2le
      neqE strict_slen)

subsubsection ⟨Dropping Elements⟩

text⟨Through dropping a number of \gls{sb} elements, it is possible
to access any element in the \gls{sb} or to get a later part.
Dropping the first ⟨n⟩ Elements of a \gls{sb} means dropping the
first ⟨n⟩ elements of every stream in the \gls{sb}.⟩

lemma sbdrop_well[simp]:"sb_well (λc. sdrop n·(b \<^enum>\<^sub>⋆ c))"
  apply(rule sbwellI)
  by (meson dual_order.trans sbgetch_ctypewell sdrop_sValues)

lift_definition sbDrop::"nat ⇒ 'cs^Ω → 'cs^Ω"is
"λ n sb. Abs_sb (λc. sdrop n·(sb \<^enum> c))"
  apply(intro cont2cont)
  by(simp add: sValues_def)

lemmas sbdrop_insert = sbDrop.rep_eq



abbreviation sbRt :: "'cs^Ω → 'cs^Ω"  where
"sbRt ≡ sbDrop 1"

lemma sbdrop_bot[simp]:"sbDrop n·⊥ = ⊥"
  apply(simp add: sbdrop_insert)
  by (simp add: bot_sb)

lemma sbdrop_eq[simp]:"sbDrop 0·sb = sb"
  by(simp add: sbdrop_insert sbgetch_insert2)

subsubsection ⟨Taking Elements⟩

text⟨Through taking the first ⟨n⟩ elements of a \gls{sb}, it is
possible to reduce any \gls{sb} to a finite part of itself. The
output is always a prefix of the input.⟩

lemma sbtake_well[simp]:"sb_well (λc. stake n·(sb  \<^enum>\<^sub>⋆  c))"
  by(simp add: sbmap_well)

lift_definition sbTake::"nat ⇒ 'cs^Ω →  'cs^Ω"is
"λ n sb. Abs_sb (λc. stake n·(sb \<^enum> c))"
  by(intro cont2cont, simp)

lemmas sbtake_insert = sbTake.rep_eq



abbreviation sbHd :: "'cs^Ω → 'cs^Ω"  where
"sbHd ≡ sbTake 1"



theorem sbtake_getch[simp]:"sbTake n·sb \<^enum> c = stake n·(sb \<^enum> c)"
  apply(simp add: sbgetch_insert sbTake.rep_eq)
  apply(subst Abs_sb_inverse,auto simp add: sb_well_def)
  by (metis sValues_sconc sbgetch_ctypewell sbgetch_insert2
      split_streamI1 subsetD)
```

```
theorem sbtake_below[simp]: "sbTake i·sb ⊑ sb"
  by (simp add: sb_belowI)

lemma sbTakezero[simp]:"sbTake 0·sb = ⊥"
  by(rule sb_eqI,simp)

lemma sbtake_idem[simp]:
  assumes "n ≥ i"
  shows"sbTake n·(sbTake i·sb) = (sbTake i·sb)"
  by (simp add: sb_eqI assms min_absorb2)

lemma sbmap_stake_eq:"
  Abs_sb (λc::'a. stake n·(sb \<^enum> c)) \<^enum> c = stake n·(sb \<^enum> c)"
  apply(simp add: sbgetch_insert2)
  apply(subst Abs_sb_inverse)
  apply simp
  apply(rule sbwellI)
  apply (metis sbgetch_insert2 sbgetch_ctypewell dual_order.trans
         sValues_sconc split_streamI1)
  by simp

lemma sbtake_max_len [simp]:"#(sbTake n·sb  \<^enum>  c) ≤ Fin n"
  by simp

lemma abs_sb_eta:
  assumes "sb_well  (λc::'cs. f·(sb \<^enum> c))"
  and "¬chDomEmpty TYPE('cs)"
  shows "(Abs_sb (λc::'cs. f·(sb  \<^enum>  c))  \<^enum>  c) = f·(sb  \<^enum>  c)"
  by (metis Abs_sb_inverse assms(1) mem_Collect_eq sbgetch_insert2)

lemma sbconc_sconc:
  assumes  "sb_well  (λc::'cs. f·(sb \<^enum> c))"
  and  "sb_well  (λc. g·( sb \<^enum> c))"
  and "¬chDomEmpty TYPE('cs)"
  shows "Abs_sb (λc. f·(sb  \<^enum>  c)) •^Ω Abs_sb (λc. g·(sb  \<^enum>  c)) =
        Abs_sb (λc. f·(sb  \<^enum>  c) • g·(sb  \<^enum>  c))"
  by (simp add: assms abs_sb_eta sbconc_insert)

text⟨Concatenating the first ⟨n⟩ elements of a \gls{sb} to the
\gls{sb} without the first ⟨n⟩ elements results in the same
\gls{sb}.⟩

theorem sbconctakedrop[simp]:"sbConc (sbTake n·sb)·(sbDrop n·sb) = sb"
  apply(cases "chDomEmpty TYPE('a)")
  apply (metis (full_types) sbtypeepmpty_sbbot)
  apply (simp add: sbtake_insert sbdrop_insert)
  by (subst sbconc_sconc,simp_all)

lemma sbcons [simp]:"sbConc (sbHd·sb)·(sbRt·sb) = sb"
  by simp


lemma sbtakesuc:"sbTake (Suc n)·sb = sbHd·sb •^Ω sbTake n·(sbRt·sb)"
  apply(rule sb_eqI,auto)
  apply(case_tac "sb  \<^enum>  c = ε",simp)
  apply (metis sbconc_bot_l sbconc_bot_r sbconc_getch
         sbconctakedrop sbdrop_bot sbgetch_bot sbtake_getch)
proof -
  fix c :: 'a
  assume a1: "sb \<^enum> c ≠ ε"
  have "sb = sbHd·sb •^Ω sbDrop (Suc 0)·sb"
    by auto
  then show "stake (Suc n)·(sb \<^enum> c) = stake (Suc 0)· (sb \<^enum> c) •
                                     stake n·(sbDrop (Suc 0)·sb \<^enum> c)"
    using a1 by (metis One_nat_def sbconc_getch sbtake_getch
                 stake2shd stake_Suc)
qed


lemma sbtake_len:
  assumes "¬chDomEmpty TYPE('b)"
    and "Fin i ≤ #(sb::'b^Ω)"
  shows "#(sbTake i·sb) = Fin i"
  using assms
  apply (induction i)
  apply (simp add: sbLen_def len_sb_def)
   apply (metis (mono_tags, lifting) LeastI)
  apply(rule sblen_rule, auto simp add: assms)
   apply (auto simp add: sbLen_def len_sb_def)
proof -
  fix ia :: nat
  assume "Fin (Suc ia) ≤ (if chDomEmpty (TYPE('b)::'b itself) then ∞ else LEAST n. n ∈ {#(sb \<^enum> c) |c.
       True})"
  then have "Fin (Suc ia) ≤ #sb"
    by (simp add: len_sb_def sbLen_def)
  then show "∃b. #(stake (Suc ia)·(sb \<^enum> b)) = Fin (Suc ia)"
    by (metis (no_types) assms(1) sblen2slen slen_stake)
next
  fix ia :: nat and c :: 'b
  assume "Fin (Suc ia) ≤ (if chDomEmpty (TYPE('b)::'b itself) then ∞ else LEAST n. n ∈ {#(sb \<^enum> c) |c.
       True})"
  then have "Fin (Suc ia) ≤ #sb"
```

```
    by (simp add: len_sb_def sbLen_def)
  then show "Fin (Suc ia) ≤ #(stake (Suc ia)·(sb \<^enum> c))"
    by (metis (no_types) assms(1) refl_lnle sblen_min_len slen_stake trans_lnle)
qed


subsubsection ⟨Converter from SB to sbElem⟩


text⟨Converting a \gls{sb} to a @{type sbElem} is rather complex.
The main goal is to obtain the first slice of a \gls{sb} as a
@{type sbElem}. This is not possible, if there is an empty stream in
the bundles domain. Hence, the @{type sbElem} can only be obtained,
if the domain is:
  \<^item> empty, then the head element is @{const None}
  \<^item> non-empty and contains no empty stream, the head element is some
    function that maps to the head of the corresponding bundle
    streams

For defining the sbHdElem function we use a helper that has always a
defined output. For this, the output is extended by ⟨⊥⟩. In the case
of an non-empty domain and an empty stream in the bundle, ⟨⊥⟩ is
returned. In \cref{subsub:sbhdelemc} we will also show the helpers
continuity for finite bundles.⟩

lemma sbhdelem_mono:
"monofun (λsb::'c^Ω.
            if chDomEmpty TYPE('c)
               then Iup (Abs_sbElem None)
               else if sbIsLeast sb
                      then ⊥
                      else Iup (Abs_sbElem
                             (Some (λc::'c. shd (sb  \<^enum>\<^sub>*  c)))))"
  apply(rule monofunI)
  apply(cases "chDomEmpty TYPE('c)")
  apply auto
  by (metis below_shd_alt monofun_cfun_arg sbnleast_mex)


definition sbHdElem_h::"'cs^Ω ⇒ ('cs^√) u"where
"sbHdElem_h sb =
      (if chDomEmpty TYPE('cs)
          then Iup(Abs_sbElem None)
          else if sbIsLeast sb
                  then ⊥
                  else Iup(Abs_sbElem (Some (λc. shd((sb) \<^enum> c)))))"

text⟨The final @{type sbElem} obtaining function then uses
@{const sbHdElem_h} to obtain only the @{type sbElem} outputs, if
the helper returns ⟨⊥⟩ the output is @{const undefined}.⟩

definition sbHdElem::"'c^Ω ⇒ 'c^√"where
"sbHdElem sb = (case (sbHdElem_h sb) of
                  Iup sbElem ⇒ sbElem     |
                  _          ⇒ undefined)"


text⟨ The @{const sbHdElem} function checks if the output of
@{const sbHdElem_h} is a @{type sbElem}. And then returns it. If the
helper returns ⟨⊥⟩ our converter maps to ⟨undefined⟩ as mentioned
above.⟩

abbreviation sbHdElem_abbr :: "'c^Ω ⇒ 'c^√" ( "\<lfloor>_" 70) where
"\<lfloor>sb ≡ sbHdElem sb"


paragraph ⟨sbHdElem Properties \\⟩

text⟨Our ⟨sbHdElem⟩ operator maps each \gls{sb} to a
corresponding @{type sbElem} exactly as intended. If the domain of
the \gls{sb} is empty, it results in the @{const None}
@{type sbElem} and if the input bundle contains no empty stream, the
resulting @{type sbElem} maps to the head of the corresponding
streams.⟩

theorem sbhdelem_none[simp]:
  fixes   sb::"'cs^Ω"
  assumes "chDomEmpty TYPE('cs)"
  shows   "sbHdElem sb = Abs_sbElem None"
  by(simp add: sbHdElem_def assms sbHdElem_h_def)



theorem sbhdelem_some:
  fixes   sb::"'cs^Ω"
  assumes "sbHdElemWell sb"
  shows   "sbHdElem sb = Abs_sbElem(Some(λc. shd(sb \<^enum> c)))"
  using assms
  by(simp add: sbHdElem_def sbHdElem_h_def)


theorem sbhdelem_mono_eq[simp]:
```

```
  fixes    sb1::"'cs^Ω"
  assumes "sbHdElemWell sb1"
  and     "sb1 ⊑ sb2"
  shows   "sbHdElem sb1 = sbHdElem sb2"
  apply(cases "chDomEmpty TYPE('cs)",simp)
  apply(simp_all add: sbhdelem_some assms)
  apply(subst sbhdelem_some)
  using assms sbleast_mono apply blast
  by (metis below_shd_alt monofun_cfun_arg assms sbnleast_mex)


theorem sbhdelem_mono_empty[simp]:
  fixes    sb1::"'cs^Ω"
  assumes "chDomEmpty TYPE('cs)"
  shows   "sbHdElem sb1 = sbHdElem sb2"
  by (simp add: assms)

subsubsection ⟨Concatenating sbElems with SBs⟩


text⟨Given a @{type sbElem} and a \gls{sb}, we can append the
@{type sbElem}to the \gls{sb}. Of course we also have to consider the
domain when appending the bundle:
  \<^item> If the domain is empty, the output \gls{sb} is ⟨⊥⟩
  \<^item> If the domain is not empty, the output \gls{sb} has the input
    @{type sbElem} as its first element.


Using only this operator allows us to construct all \glspl{sb} where
every stream has the same length. But since there is no restriction
for the input bundle, we can map to any \gls{sb} with a length
greater 0.⟩

definition sbECons::"'cs^√⇒ 'cs^Ω→ 'cs^Ω" where
"sbECons sbe = sbConc (sbe2sb sbe)"



abbreviation sbECons_abbr::"'cs^√⇒ 'cs^Ω⇒ 'cs^Ω"(infixr "•^√" 100)
where "sbe •^√ sb ≡ sbECons sbe·sb"

text⟨The concatenation results in ⟨⊥⟩ when the domain is empty.⟩

theorem sbtypeempty_sbecons_bot:
  fixes    sbe::"'cs^√"
  assumes "chDomEmpty TYPE ('cs)"
  shows   "sbe •^√ sb = ⊥"
  by (simp add: assms)

lemma sb_empty_unfold: fixes sb::"'cs^Ω"
  assumes "chDomEmpty TYPE('cs)"
  shows "sb = (Abs_sbElem None) •^√ sb"
  by(rule sb_empty_eq, simp add: assms)

lemma exchange_bot_sbecons:
"chDomEmpty TYPE ('cs)⟹P sb⟹P((sbe::'cs^√) •^√ sb)"
  by (metis (full_types) sbtypeepmpty_sbbot)



theorem sbrt_sbecons: "sbRt·(sbe •^√ sb) = sb"
  apply (cases "chDomEmpty(TYPE('a))", simp)
  apply (simp add: sbDrop.rep_eq)
  apply (simp add: sbECons_def)
  apply (subst sdropl6)
  apply (subgoal_tac "⋀c. ∃m. sbe2sb sbe  \<^enum>  c = ↑m")
  apply (metis Fin_0 Fin_Suc lnzero_def lscons_conv slen_scons
         strict_slen sup'_def)
  apply (simp add: sbgetch_insert2 sbe2sb.rep_eq chDom_def)
  apply (metis Diff_eq_empty_iff chDom_def option.simps(5)
         sbtypenotempty_fex)
  by (simp add: sb_rep_eqI sbgetch_insert2 Rep_sb_inverse)

lemma sbhdelem_h_sbe:" sbHdElem_h (sbe •^√ sb) = up·sbe"
  apply (cases "chDomEmpty(TYPE('a))")
  apply (simp_all add: sbHdElem_def sbHdElem_h_def)+
  apply (simp_all add: up_def)
  apply (metis sbtypeepmpty_sbenone)
  apply (simp add: sbECons_def,auto)
  apply (subgoal_tac "∀c::'a. sbe2sb sbe  \<^enum>  c ≠ ε")
  apply (simp add: sbe2sb_def)
  apply (simp split: option.split)
  apply (rule conjI)
  apply (metis Abs_sb_strict option.simps(4) sbgetch_bot)
  apply (metis (no_types, lifting) option.simps(5) sbHdElemWell_def
         sbconc_bot_r sbconc_getch strictI)
  using sbgetch_sbe2sb_nempty apply auto[1]
  apply(simp only: sbHdElemWell_def sbe2sb_def)
  apply (simp split: option.split,auto)
  apply (metis emptyE option.discI sbtypenotempty_fex)
  apply (subgoal_tac
```

```
        "∀c::'a. Abs_sb (λc::'a. ↑(x2 c))  \<^enum>  c = ↑(x2 c)")
  apply (simp add: Abs_sbElem_inverse)
  apply (metis Rep_sbElem_inverse)
  by (metis option.simps(5) sbe2sb.abs_eq sbe2sb.rep_eq
      sbgetch_insert2)


lemma sbhdelem_sbecons: "sbHdElem (sbe  •^√ sb) = sbe"
  by(simp add: sbHdElem_def sbhdelem_h_sbe up_def)

theorem sbh_sbecons: "sbHd·(sbe  •^√ sb) = sbe2sb sbe"
  apply(rule sb_eqI, auto simp add: sbECons_def)
  apply(auto simp add: sbe2sb.rep_eq sbgetch_insert2)
  apply(cases "Rep_sbElem sbe = None")
  apply auto
  using sbtypefull_none by blast

text⟨Constructing a \gls{sb} with @{const sbECons} increases its
length by exactly 1. This also holds for empty domains, because we
interpret the length of those \Gls{sb} as ⟨∞⟩.⟩

theorem sbecons_len:
  shows "#(sbe •^√ sb) = lnsuc·(# sb)"
  apply(cases "chDomEmpty(TYPE('a))")
  apply(simp)
  apply(rule sblen_rule,simp)
  apply(simp add: sbECons_def sbgetch_insert2 sbconc_insert)
  apply(subst Abs_sb_inverse)
  apply simp
  apply(insert sbconc_well[of "sbe2sb sbe" sb],simp add:
        sbgetch_insert2)
  apply(subst sconc_slen2)
  apply(subgoal_tac "#(Rep_sb (sbe2sb sbe) c) = 1",auto)
  apply (metis equals0D lessequal_addition lnat_plus_commu
         lnat_plus_suc sbelen_one sbgetch_insert2 sblen_min_len)
   apply (metis emptyE sbe2slen_1 sbgetch_insert2)
  by (metis all_not_in_conv lnat_plus_commu lnat_plus_suc sbECons_def sbconc_chan_len sbe2slen_1 sblen2slen)


lemma sbHdElem:
"# (sb::'cs^Ω) ≠ (0::lnat)⟹sbe2sb (sbHdElem sb) = sbHd·sb"
  apply (case_tac "chDomEmpty (TYPE ('cs))")
  apply (metis (full_types) sbtypeepmpty_sbbot)
  apply (rule sb_rep_eqI)
  apply (simp add: sbHdElem_def sbHdElem_h_def)
  apply rule+
  using sbleast2sblenempty apply blast
  apply(simp add:sbtake_insert stake2shd sbe2sb.abs_eq
        sbe2sb.rep_eq Abs_sbElem_inverse Abs_sb_inverse sb_well_def)
  by (metis (no_types) sbgetch_insert2 sbmap_stake_eq sbnleast_mex
      stake2shd)

(*sb_ind*)

lemma sbtake_chain:"chain (λi::nat. sbTake i·x)"
  apply (rule chainI)
  apply(simp add: below_sb_def)
  apply(rule fun_belowI)
  apply(simp add: sbtake_insert)
  by (metis (no_types) Suc_leD le_refl sbgetch_insert2
      sbmap_stake_eq stake_mono)

lemma sblen_sbtake:
"¬chDomEmpty TYPE ('c)⟹# (sbTake n·(x :: 'c^Ω)) ≤ Fin (n)"
proof-
assume a0:"¬chDomEmpty TYPE ('c)"
  have h0:"⋀c. # (sbTake n·x) ≤ #((sbTake n·x) \<^enum> (c::'c))"
    by(rule sblen_min_len, simp add: a0)
  have h1:"⋀c. #((sbTake n·x) \<^enum> (c::'c)) ≤ Fin (n)"
   by simp
  then show ?thesis
    using dual_order.trans h0 by blast
qed

lemma sbtake_lub:"(⨆i::nat. sbTake i·x) = x"
  apply(rule sb_eqI)
  apply(subst contlub_cfun_arg)
  apply(simp add: sbtake_chain)
  by(simp add: sbtake_insert sbmap_stake_eq reach_stream)

lemma sbECons_sbLen:"# (sb::'cs^Ω) ≠ (0::lnat)⟹
 ¬ chDomEmpty TYPE('cs)⟹∃ sbe sb'. sb = sbe •^√ sb'"
  by (metis sbECons_def sbHdElem sbcons)


lemma sbecons_sbhdelemwell: "sbHdElemWell (sbe2sb sbe)⟹
                             sbHdElemWell (sbe •^√ sb)"
  by (metis monofun_cfun_arg sbECons_def sbTakezero sbconc_bot_r
      sbleast_mono sbtake_below)

paragraph ⟨SB induction and case rules \\⟩
```

```
text‹This framework also offers proof methods using the
@{type sbElem} constructor, that offer an easy proof process when
applied correctly. The first method is a case distinction for
\glspl{sb}. It differentiates between the short \glspl{sb} where an
empty stream exists and all other \glspl{sb}. The configuration of
the lemma splits the goal into the cases ‹least› and ‹sbeCons›. It
also causes the automatic usage of this case tactic for variables of
type \gls{sb}.›


theorem sb_cases [case_names least sbeCons, cases type: sb]:
  assumes "sbIsLeast (sb'::'cs^Ω)⟹P"
  and     "⋀sbe sb. sb' = sbe •^√ sb⟹¬chDomEmpty TYPE ('cs)
           ⟹P"
  shows   "P"
  by (meson assms sbECons_sbLen sbnleast_dom sbnleast_len)

lemma sb_cases2 [case_names least sbeCons]:
  assumes "# (sb'::'cs^Ω) = 0⟹P"
  and     "⋀sbe sb. sb' = sbe •^√ sb⟹P"
  shows   "P"
  by (metis (full_types) assms(1) assms(2) sbECons_def sbHdElem sbconctakedrop)

lemma sb_finind1:
    fixes x::"'cs^Ω"
    shows "# x = Fin k
          ⟹(⋀sb. sbIsLeast sb⟹P sb)
          ⟹(⋀sbe sb. P sb
          ⟹¬chDomEmpty TYPE ('cs)⟹P (sbe •^√ sb))
          ⟹P x"
  apply(induction k  arbitrary:x)
  using sbnleast_len apply fastforce
  by (metis Fin_Suc inject_lnsuc sb_cases sbecons_len)

lemma sb_finind:
  fixes x::"'cs^Ω"
  assumes "# x < ∞"
  and "⋀sb. sbIsLeast sb⟹P sb"
  and "⋀sbe sb. P sb⟹¬chDomEmpty TYPE ('cs)⟹P (sbe •^√ sb)"
  shows "P x"
  by (metis assms(1) assms(2) assms(3) lnat_well_h2 sb_finind1)

lemma sbtakeind1:
  fixes x::"'cs^Ω"
  shows "∀x. (( ∀(sb::'cs^Ω) . sbIsLeast sb⟶P sb) ∧
        (∀ (sbe::'cs^√) sb::'cs^Ω. P sb ⟶¬chDomEmpty TYPE ('cs)
      ⟶P (sbe •^√ sb))) ∧
        ( ¬chDomEmpty TYPE ('cs)⟶# x ≤ Fin n)⟶P (x)"
  by (metis (no_types, lifting) inf_ub less2eq
      order.not_eq_order_implies_strict sb_cases sb_finind
      sb_finind1)

lemma sbtakeind:
  fixes x::"'cs^Ω"
  shows "∀x. (( ∀(sb::'cs^Ω) . sbIsLeast sb⟶P sb) ∧
         (∀ (sbe::'cs^√) sb::'cs^Ω. P sb ⟶¬chDomEmpty TYPE ('cs)
         ⟶P (sbe •^√ sb)))
         ⟶P (sbTake  n·x)"
  apply rule+
  apply(subst sbtakeind1, simp_all)
  using sblen_sbtake sbtakeind1 by auto

text‹The second showcased proof method is the induction for
\glspl{sb}. Beside the admissibility of the predicate, the
inductions subgoals are also divided into the cases ‹least› and
‹sbeCons›.›

theorem sb_ind[case_names adm least sbeCons, induct type: sb]:
  fixes    x::"'cs^Ω"
  assumes "adm P"
  and     "⋀sb. sbIsLeast sb⟹P sb"
  and     "⋀sbe sb. P sb⟹¬chDomEmpty TYPE ('cs)
           ⟹P (sbe •^√ sb)"
  shows   "P x"
  using assms(1) assms(2) assms(3)
  apply(unfold adm_def)
  apply(erule_tac x="λi. sbTake i·x" in allE,auto)
  apply(simp add: sbtake_chain)
  apply(simp add: sbtakeind)
  by(simp add: sbtake_lub)

text‹Here we show a small example proof for our \gls{sb} cases rule.
First the ‹ISAR› proof is started by applying the proof tactic
to the theorem. This automatically generates the proof structure
with the two cases and their variables. These two generated cases
match with our theorem assumptions from ‹sb_cases›. Our theorems
statement then follows then directly by proving both generated
cases.›

theorem sbecons_eq:
  assumes "# sb ≠ 0"
```

```
  shows "sbHdElem sb •^√ sbRt·sb = sb"
proof %visible(cases sb)
  assume "sbIsLeast sb"
  thus "sbHdElem sb •^√ sbRt·sb = sb"
    using assms by(simp only: assms sbECons_def sbHdElem sbcons)
next
  fix sbe and sb'
  assume "sb = sbe •^√ sb'"
  thus "(sbHdElem sb) •^√ sbRt·sb = sb"
    using assms by(simp only: assms sbhdelem_sbecons sbrt_sbecons)
qed

text⟨The first subgoals assumption after applying the case tactic
      is ⟨sbIsLeast sb⟩ and proving this case and the ⟨sbeCons⟩
      case is often simpler than proving the theorem without
      case distinction.⟩

text⟨The second subgoals assumes ⟨sb = sbe •^√ sb'⟩. This allows splitting the \gls{sb} in
     two parts, where the first part is a @{type sbElem}. This
     helps if a function works element wise on its input.⟩


text⟨The next theorem is an example for the induction rule. Similar
to the cases rule there are automatically generated cases that
correspond to the assumptions of ⟨sb_ind⟩. Our theorem is
proven after showing the three generated goals.⟩

theorem shows "sbTake n·sb ⊑ sb "
proof %visible(induction sb)
  case adm
  then show ?case
    by simp
next
  case (least sb)
  then show "sbIsLeast sb⟹sbTake n·sb ⊑ sb"
    by simp
next
  case (sbeCons sbe sb)
  then show "sbTake n·sb ⊑ sb⟹sbTake n·(sbe •^√ sb) ⊑ sbe •^√ sb"
    by simp
qed

subsubsection ⟨Converting Domains of SBs⟩


text⟨Two \glspl{sb} with a different type are not comparable, since
only \glspl{sb} with the same type have an order. This holds even if
the domain of both types is the same. To make them comparable we
introduce a type caster that converts the type of a \gls{sb}. This
casting makes two \Gls{sb} of different type comparable.
Since it does change the type, it can also restrict or expand the
domain of a \gls{sb}. Newly added channels map to ⟨ϵ⟩.⟩

lemma sbtypecast_well[simp]:"sb_well (λc. sb \<^enum>\<^sub>⋆ c)"
  by(rule sbwelll, simp)

lift_definition sbTypeCast::"'cs1^Ω→ 'cs2^Ω"is
"(λ sb. Abs_sb (λc.  sb \<^enum>\<^sub>⋆ c ))"
  by(intro cont2cont, simp)



lemmas sbtypecast_insert = sbTypeCast.rep_eq

abbreviation sbTypeCast_abbr :: "'cs1^Ω⇒ 'cs2^Ω"
 ( "_⋆" 200)  where "sb⋆ ≡ sbTypeCast·sb"



abbreviation sbrestrict_abbr_fst :: "('cs1 ∪ 'cs2)^Ω⇒ 'cs1^Ω"
 ( "_⋆\<^sub>1" 200) where "sb⋆\<^sub>1 ≡ sbTypeCast·sb"



abbreviation sbrestrict_abbr_snd :: "('cs1∪'cs2)^Ω⇒ 'cs2^Ω"
 ( "_⋆\<^sub>2" 200) where "sb⋆\<^sub>2 ≡ sbTypeCast·sb"



abbreviation sbrestrict_abbr_commu :: "('cs1 ∪'cs2)^Ω⇒ ('cs2 ∪ 'cs1)^Ω"
 ( "_⋆\<^sub>⇌" 200) where "sb⋆\<^sub>⇌ ≡ sbTypeCast·sb"

abbreviation sbrestrict_abbr_minus :: "'cs1^Ω⇒ ('cs1-'cs2)^Ω"
 ( "_⋆\<^sub>-" 200) where "sb⋆\<^sub>- ≡ sbTypeCast·sb"


abbreviation sbTypeCast_abbr_fixed :: "'cs1^Ω⇒ 'cs3 itself ⇒ 'cs3^Ω"
 ( "_¦_" 201) where "sb ¦ _ ≡ sbTypeCast·sb"

text⟨A \gls{sb} with domain ⟨('cs1 ∪ 'cs2) -'cs3⟩ can be restricted
to domain ⟨('cs1 - 'cs3)⟩ by using ⟨sb ¦ TYPE ('cs1 - 'cs3)⟩.⟩
```

```
lemma sbtypecast_rep[simp]: "Rep_sb(sb★) = (λc. sb \<^enum>\<^sub>★ c)"
  by (simp add: Abs_sb_inverse sbtypecast_insert)



lemma sbconv_eq[simp]:"sb★ = sb"
  apply(rule sb_eqI)
  by (metis (no_types) Abs_sb_inverse mem_Collect_eq
        sbtypecast_insert sbtypecast_well sbgetch_insert2)



theorem sbtypecast_getch [simp]: "sb★ \<^enum> c = sb \<^enum>\<^sub>★ c"
  by (simp add: sbgetch_insert2)

lemma sbtypecast_len:
  assumes "chDom TYPE('cs2) ⊆ chDom TYPE('cs1)"
  shows "#sb ≤ #((sbTypeCast_abbr :: 'cs1^Ω ⇒ 'cs2^Ω) sb)"
proof (cases "chDomEmpty TYPE('cs2)")
  case True
  then show ?thesis
    by simp
next
  case cs2_typenempty: False
  obtain least_ch where least_ch1_def: "#sb = #(sb \<^enum> least_ch)"
    by (metis cs2_typenempty assms empty_subsetI equalityI
        order.trans sblen2slen)
  have cs2_sbtypecast_getch: "⋀c::'cs2. sb★ \<^enum> c = sb \<^enum>\<^sub>★ c"
    by simp
  thus ?thesis
  proof (cases "Rep least_ch ∈ chDom TYPE('cs2)")
    case least_ch_in_cs2: True
    then show ?thesis
      by (metis Un_iff assms chdom_in cs2_sbtypecast_getch
          cs2_typenempty least_ch1_def order_refl sbgetch_empty2
          sbgetch_insert sbgetch_insert2 sblen_min_len sblengeq
          strict_slen sup.absorb_iff1)
  next
    case least_ch_nin_cs2: False
    then show ?thesis
      by (metis Un_iff assms chdom_in cs2_sbtypecast_getch
          cs2_typenempty least_ch1_def refl_lnle sbgetch_empty2
          sbgetch_insert sbgetch_insert2 sblen_min_len sblengeq
          strict_slen sup.absorb_iff1)
  qed
qed

lemma sb_star21:
  fixes sb::"'cs0^Ω"
  assumes "chDom TYPE('cs2) ⊆ chDom TYPE('cs1)"
    shows "sb ¦ TYPE('cs1) ¦ TYPE ('cs2) = sb ¦ TYPE('cs2)"
  apply(rule sb_eqI, auto)
  apply(auto simp add: sbGetCh.rep_eq)
  using assms by blast

subsubsection ⟨Union of SBs⟩



definition sbUnion::"'cs1^Ω → 'cs2^Ω → ('cs1 ∪ 'cs2)^Ω" where
"sbUnion ≡ Λ sb1 sb2. Abs_sb (λ c.
                 if Rep c ∈ chDom TYPE('cs1)
                 then sb1 \<^enum>\<^sub>★ c
                 else sb2 \<^enum>\<^sub>★ c)"

lemma sbunion_sbwell[simp]: "sb_well ((λ (c::'e).
                 if (Rep c ∈ chDom TYPE('c)) then
                 (sb1::'c^Ω) \<^enum>\<^sub>★ c else  (sb2::'d^Ω) \<^enum>\<^sub>★ c))"
  apply(rule sbwellI)
  by simp

lemma sbunion_insert:"(sbUnion·(sb1::'c^Ω)·sb2) = Abs_sb (λ c. if
                 (Rep c ∈ chDom TYPE('c)) then
                 sb1 \<^enum>\<^sub>★ c else  sb2 \<^enum>\<^sub>★ c)"
  unfolding sbUnion_def
  apply(subst beta_cfun, intro cont2cont, simp)+
  ..

lemma sbunionm_insert:"(sbUnion·(sb1::'c^Ω)·sb2)★ = Abs_sb (λ c. if
                 (Rep c ∈ chDom TYPE('c)) then
                 sb1 \<^enum>\<^sub>★ c else  sb2 \<^enum>\<^sub>★ c)"
  unfolding sbUnion_def
  apply(subst beta_cfun, intro cont2cont, simp)+
  apply(simp add: sbtypecast_insert)
  apply(rule sb_rep_eqI)
  apply(subst Abs_sb_inverse,simp)
  apply(subst Abs_sb_inverse,simp)
  apply(subst sbgetch_insert)
  apply(subst Abs_sb_inverse,simp)
  apply(case_tac "Rep c∈chDom TYPE('c)";simp)
  by(auto simp add: rep_reduction sbgetch_insert)
```

```
lemma sbunion_rep_eq:"Rep_sb (sbUnion·(sb1::'c^Ω)·sb2) =
  (λ c. if (Rep c ∈ chDom TYPE('c))
         then sb1 \<^enum>\<^sub>* c
         else  sb2 \<^enum>\<^sub>* c)"
  apply(simp add: sbunion_insert sbgetch_insert)
  apply(subst Abs_sb_inverse,simp)
  by auto

lemma sbunionm_rep_eq:"Rep_sb ((sbUnion·(sb1::'c^Ω)·sb2) *) =
  (λ c. if (Rep c ∈ chDom TYPE('c))
         then sb1 \<^enum>\<^sub>* c
         else  sb2 \<^enum>\<^sub>* c)"
  apply(simp add: sbunionm_insert sbgetch_insert)
  apply(subst Abs_sb_inverse,simp)
  by auto


abbreviation sbUnion_abbr :: "'cs1^Ω ⇒ 'cs2^Ω ⇒ ('cs1 ∪ 'cs2)^Ω"
(infixr "⊎" 100) where "sb1 ⊎ sb2 ≡ sbUnion·sb1·sb2"

text⟨The following abbreviations restrict the input and output
domains of @{const sbUnion} to specific cases. These are displayed
by its signature. Abbreviation ⟨⊎\<^sub>*⟩ is the composed function of
@{const sbUnion} and @{const sbTypeCast}, thus, it converts the
output domain.⟩

abbreviation sbUnion_magic_abbr :: "'cs1^Ω ⇒ 'cs2^Ω ⇒ 'cs3^Ω"
(infixr "⊎\<^sub>*" 100) where "sb1 ⊎\<^sub>* sb2 ≡ (sb1 ⊎ sb2)*"

text⟨The third abbreviation only fills in the stream its missing in
its domain ⟨'cs1⟩. It does not use stream on channels that are in
domain ⟨cs2⟩ but not ⟨cs1⟩.⟩

abbreviation sbUnion_minus_abbr :: "('cs1 - 'cs2)^Ω ⇒ 'cs2^Ω ⇒ 'cs1^Ω"
(infixr "⊎\<^sub>-" 500) where "sb1 ⊎\<^sub>- sb2 ≡ sb1 ⊎\<^sub>* sb2"


paragraph ⟨sbUnion Properties \\⟩


lemma sbunion_getch[simp]:fixes c::"'a"
      assumes"Rep c ∈ chDom TYPE('c)"
      shows  "(sbUnion_magic_abbr::'a^Ω⇒ 'b^Ω ⇒ 'c^Ω)cb db \<^enum>\<^sub>* c = cb \<^enum> c"
  apply(simp add: sbunionm_insert)
  apply(simp add: sbgetch_insert)
  apply(subst Abs_sb_inverse,simp add: assms)+
  apply (auto simp add: rep_reduction sbgetch_insert assms)
  using assms cdom_notempty notcdom_empty apply blast
  done

lemma sbunion_eq [simp]: "sb1 ⊎\<^sub>* sb2 = sb1"
  apply(rule sb_eqI)
  apply simp
  using sbunion_getch by fastforce

lemma sbunion_sbtypecast_eq[simp]:"cb ⊎\<^sub>* cb = (cb*)"
  apply(rule sb_eqI,simp)
  by (metis sbtypecast_rep sbunionm_rep_eq)


theorem ubunion_commu:
  fixes   sb1 ::"'cs1^Ω"
  and     sb2 ::"'cs2^Ω"
  assumes "chDom TYPE ('cs1) ∩ chDom TYPE ('cs2) = {}"
  shows   "sb1 ⊎\<^sub>* sb2 = sb2 ⊎\<^sub>* sb1"
  apply(rule sb_rep_eqI)
  apply(simp add: sbunion_rep_eq sbgetch_insert rep_reduction assms)
  using assms cdom_notempty cempty_rule cnotempty_cdom by blast

lemma ubunion_fst[simp]:
  fixes   sb1 ::"'cs1^Ω"
  and     sb2 ::"'cs2^Ω"
  assumes "chDom TYPE ('cs2) ∩ chDom TYPE ('cs3) = {}"
  shows   "sb1 ⊎\<^sub>* sb2 = (sb1* :: 'cs3^Ω)"
  apply(rule sb_rep_eqI)
  apply(simp add: sbunion_rep_eq sbgetch_insert rep_reduction assms)
  using assms cdom_notempty cempty_rule cnotempty_cdom by blast

lemma ubunion_id[simp]: "out*\<^sub>1 ⊎ (out*\<^sub>2) = out"
proof(rule sb_eqI)
  fix c::"'a ∪ 'b"
  assume as:"Rep c ∈ chDom TYPE('a ∪ 'b)"
  have "Rep c ∈ chDom (TYPE ('a))⟹out* ⊎ (out*) \<^enum> c = out \<^enum> c"
    by (metis sbgetch_insert2 sbunion_getch sbunion_rep_eq
        sbunion_sbtypecast_eq)
  moreover have "Rep c ∈ chDom (TYPE ('b))
                ⟹out* ⊎ (out*) \<^enum> c = out \<^enum> c"
```

```
    by (metis sbgetch_insert2 sbunion_getch sbunion_rep_eq
        sbunion_sbtypecast_eq)
  moreover have "Rep c ∈ chDom TYPE ('a) ∨ Rep c ∈ chDom TYPE ('b)"
    using as chdom_in by fastforce
  ultimately show "out⋆ ⊎ (out⋆) \<^enum> c = out \<^enum> c" by fastforce
qed


theorem sbunion_getchl[simp]:
  fixes   sb1 ::"'cs1^Ω"
  and     sb2 ::"'cs2^Ω"
  assumes "Rep c ∈ chDom TYPE('cs1)"
  shows   "(sb1 ⊎ sb2) \<^enum>\<^sub>⋆ c = sb1 \<^enum>\<^sub>⋆ c"
  apply(auto simp add: sbgetch_insert rep_reduction sbunion_rep_eq
        assms)
  done



theorem sbunion_getchr[simp]:
  fixes   sb1 ::"'cs1^Ω"
  and     sb2 ::"'cs2^Ω"
  assumes "Rep c ∉ chDom TYPE('cs1)"
  shows   "(sb1 ⊎ sb2) \<^enum>\<^sub>⋆ c = sb2 \<^enum>\<^sub>⋆ c"
  by(auto simp add: sbgetch_insert rep_reduction sbunion_rep_eq
     assms)

lemma sbunion_conv_fst:
      fixes sb1 :: "'cs1^Ω"
        and sb2 :: "'cs2^Ω"
      assumes "chDom TYPE('cs3) ⊆ chDom TYPE('cs1)"
      shows  "(((sb1 ⊎ sb2)⋆)::'cs3^Ω) = (sb1⋆)"
  apply(rule sb_eqI)
  using assms sbunion_getchl by fastforce

lemma sbunion_conv_snd:
      fixes sb1 :: "'cs1^Ω"
        and sb2 :: "'cs2^Ω"
      assumes "chDom TYPE('cs1) ∩ chDom TYPE('cs3) = {}"
      shows  "(((sb1 ⊎ sb2)⋆)::'cs3^Ω) = (sb2⋆)"
  apply(rule sb_eqI)
  using assms sbunion_getchr by fastforce

lemma sbunion_star_getchr[simp]:
  fixes   sb1 ::"'cs1^Ω"
  and     sb2 ::"'cs2^Ω"
  assumes "Rep c ∉ chDom TYPE('cs1)"
  shows   "(sb1 ⊎\<^sub>⋆ sb2) \<^enum> c = sb2 \<^enum>\<^sub>⋆ c"
  apply(auto simp add: sbgetch_insert rep_reduction sbunion_rep_eq
        assms)
  using cdom_notempty notcdom_empty by blast



lemma sbunion_minus_getchr[simp]:
  fixes sb1 ::"('cs1-'cs2)^Ω"
  and sb2 ::"'cs2^Ω"
  assumes "Rep c ∈ chDom TYPE('cs1)"
  and "Rep c ∉ chDom TYPE('cs1-'cs2)"
  shows   "(sb1 ⊎\<^sub>- sb2) \<^enum>\<^sub>⋆ c = sb2 \<^enum>\<^sub>⋆ c"
  apply(auto simp add: sbgetch_insert rep_reduction sbunion_rep_eq
        assms)
  using assms apply auto[1]
  done

lemma sbunion_minus_getchl[simp]:
  fixes sb1 ::"('cs1-'cs2)^Ω"
  and sb2 ::"'cs2^Ω"
  assumes "Rep c ∈ chDom TYPE('cs1-'cs2)"
  shows   "(sb1 ⊎\<^sub>- sb2) \<^enum>\<^sub>⋆ c = sb1 \<^enum>\<^sub>⋆ c"
  apply(auto simp add: sbgetch_insert rep_reduction sbunion_rep_eq
        assms)
  using assms by auto[1]


lemma sbunion_minus_getchempty[simp]:
  fixes sb1 ::"('cs1-'cs2)^Ω"
  and sb2 ::"'cs2^Ω"
  assumes "Rep c ∉ chDom TYPE('cs1)"
  shows   "(sb1 ⊎\<^sub>- sb2) \<^enum>\<^sub>⋆ c = ϵ"
  by(simp add: assms)

lemma sbunion_minus_getchl2[simp]:
    fixes sb1 ::"('cs1-'cs2)^Ω"
      and sb2 ::"'cs2^Ω"
  assumes "Rep c ∉ chDom TYPE('cs2)"
  shows   "(sb1 ⊎\<^sub>- sb2) \<^enum>\<^sub>⋆ c = sb1 \<^enum>\<^sub>⋆ c"
  by(auto simp add: sbgetch_insert rep_reduction sbunion_rep_eq
```

```
    assms)

lemma sbunion_star_getchl[simp]:
    fixes sb1 ::"'cs1^Ω"
      and sb2 ::"'cs2^Ω"
  assumes "Rep c ∉ chDom TYPE('cs2)"
  shows   "(sb1 ⊎\<^sub>* sb2) \<^enum> c = sb1 \<^enum>\<^sub>* c"
  by (metis assms sbgetch_empty sbgetch_insert2 sbunionm_rep_eq)

lemma sbunion_getch_nomag [simp]:
"sb1 ⊎\<^sub>* sb2  \<^enum>  c = (sb1 ⊎ sb2) \<^enum>\<^sub>* c"
  by(auto simp add: sbgetch_insert2 sbunion_rep_eq)

lemma sbunion_magic:
  fixes sb1 ::"'cs1^Ω"
  and   sb2 ::"'cs2^Ω"
  shows "(sb1 ⊎ sb2)* = sb1 ⊎\<^sub>* sb2"
  apply(rule sb_eqI)
  by auto

lemma sbunion_magic_len_fst:
  fixes sb1 :: "'cs1^Ω"
  and   sb2 :: "'cs2^Ω"
  assumes "chDomEmpty TYPE('cs2)"
    and "¬chDomEmpty TYPE('cs1)"
    and "chDom TYPE('cs3) = chDom TYPE('cs1)"
  shows "#((sb1 ⊎\<^sub>* sb2)::'cs3^Ω) = #sb1"
proof-
  obtain least_ch::'cs1 where
    sb2_len_chdef: "#sb1 = #(sb1 \<^enum>\<^sub>* least_ch)"
    by (metis assms(2) sblen2slen)
  thus ?thesis
    apply (subst sbunion_conv_fst)
    apply (simp add: assms)
    apply (rule sblen_rule [where k="#sb1"])
    using assms apply blast
    apply (metis assms(2) assms(3) cnotempty_cdom sbgetch_insert
           sbgetch_insert2 sblen_min_len sbtypecast_getch)
    apply (rule_tac x="(Abs (Rep least_ch))" in exI)
    by (simp add: assms(2) assms(3) sbgetch_insert)
qed

lemma sbunion_emptyr: fixes sb2::"'cs2^Ω"
  assumes "chDom TYPE('cs2) = {}"
  shows "sb1 ⊎ sb2 = sb1*"
  apply(rule sb_eqI, auto simp add: sbGetCh.rep_eq assms)
  by (simp add: sbgetch_insert sbunion_rep_eq)

lemma sbunion_emptyl: fixes sb1::"'cs1^Ω"
  assumes "chDom TYPE('cs1) = {}"
  shows "sb1 ⊎ sb2 = sb2*"
  apply(rule sb_eqI, auto simp add: sbGetCh.rep_eq assms)
  by (simp add: assms sbgetch_insert sbunion_rep_eq)

lemma sbunion_magic_len_snd:
  fixes sb1 :: "'cs1^Ω"
  and   sb2 :: "'cs2^Ω"
  assumes "chDomEmpty TYPE('cs1)"
  and "¬chDomEmpty TYPE('cs2)"
  and "chDom TYPE('cs3) = chDom TYPE('cs2)"
  shows "#((sb1 ⊎\<^sub>* sb2)::'cs3^Ω) = #sb2"
proof-
  obtain least_ch::'cs2 where
    sb2_len_chdef: "#sb2 = #(sb2 \<^enum>\<^sub>* least_ch)"
    by (metis assms(2) sblen2slen)
  thus ?thesis
    apply (subst sbunion_conv_snd)
    apply (simp add: assms)
    apply (rule sblen_rule [where k="#sb2"])
    using assms apply blast
    apply (metis assms(2) assms(3) cnotempty_cdom sbgetch_insert
           sbgetch_insert2 sblen_min_len sbtypecast_getch)
    apply (rule_tac x="(Abs (Rep least_ch))" in exI)
    by (simp add: assms sbgetch_insert)
qed

lemma sbunion_magic_len:
  fixes sb1 :: "'cs1^Ω"
  and   sb2 :: "'cs2^Ω"
  assumes "chDom TYPE('cs1) ∩ chDom TYPE('cs2) = {}"
  and "chDom TYPE('cs3) = chDom TYPE('cs1) ∪ chDom TYPE('cs2)"
  shows "#((sb1 ⊎\<^sub>* sb2)::'cs3^Ω) = min (#sb1) (#sb2)"
proof (cases "chDomEmpty TYPE('cs1)")
  case cs1_dom_empty: True
  then show ?thesis
  proof (cases "chDomEmpty TYPE('cs2)")
    case cs2_dom_empty: True
    thus?thesis
      by (simp add: assms cs1_dom_empty)
  next
    case False
```

```
    then show ?thesis
      by (simp add: assms cs1_dom_empty sbunion_magic_len_snd)
    qed
next
  case cs1_dom_nempty: False
  then show ?thesis
  proof (cases "chDomEmpty TYPE('cs2)")
    case True
    then show ?thesis
      apply (subst sbunion_magic_len_fst, simp_all)
      using cs1_dom_nempty apply blast
      using assms by blast
  next
    case cs2_dom_nempty: False
    obtain least_cs1::'cs1 where
      sb1_len_chdef: "#sb1 = #(sb1 \<^enum>\<^sub>★ least_cs1)"
      by (metis cs1_dom_nempty sblen2slen)
    obtain least_cs2::'cs2 where
      sb2_len_chdef: "#sb2 = #(sb2 \<^enum>\<^sub>★ least_cs2)"
      by (metis cs2_dom_nempty sblen2slen)
    have least_cs1_dom: "Rep least_cs1 ∈ chDom TYPE('cs1)"
      by (simp add: cs1_dom_nempty)
    have least_cs1_dom: "Rep least_cs2 ∈ chDom TYPE('cs2)"
      by (simp add: cs2_dom_nempty)
    have "⋀c::'cs3. Rep c ∉ chDom TYPE('cs1)⟹
            Rep c ∈ chDom TYPE('cs2)"
      using assms(2) cnotempty_cdom cs1_dom_nempty by auto
    then show ?thesis
      apply (subst sblen_rule [where k="min (#sb1) (#sb2)"])
      apply (simp_all add: assms cs1_dom_nempty)
      apply (case_tac "Rep c ∈ chDom TYPE('cs1)", simp_all)
      apply (metis (full_types) min.coboundedl1 rep_reduction
             sbgetch_insert sblen_min_len2)
      apply (simp add: sbgetch_insert )
      apply (metis cs2_dom_nempty min_le_iff_disj sbgetch_insert2
             sblen_min_len)
      apply (cases "min (#sb1) (#sb2) = #sb1", simp_all)
      apply (rule_tac x="Abs (Rep least_cs1)" in exI)
      apply (simp add: sbgetch_insert)
      apply (simp add: assms cs1_dom_nempty sb1_len_chdef
             sbgetch_insert sbunion_rep_eq)
      apply (rule_tac x="Abs (Rep least_cs2)" in exI)
      apply (subgoal_tac "min (#sb1) (#sb2) = #sb2")
      apply (simp add: sbgetch_insert)
      apply (simp add: assms cs2_dom_nempty sb2_len_chdef
             sbgetch_insert sbunion_rep_eq)
      using assms(1) least_cs1_dom apply blast
      by (metis min_def)
  qed
qed


theorem sbunion_fst: "(sb1 ⊎ sb2)★\<^sub>1 = sb1"
  by simp


theorem sbunion_snd[simp]:
  fixes   sb1 ::"'cs1^Ω"
  and     sb2 ::"'cs2^Ω"
  assumes "chDom TYPE ('cs1) ∩ chDom TYPE ('cs2) = {}"
  shows   "(sb1 ⊎ sb2)★\<^sub>2 = sb2"
  by (metis assms sbconv_eq ubunion_commu ubunion_fst)

lemma sbunion_eql:
  assumes "sb1 = (sb★\<^sub>1)"
    and "sb2 = (sb★\<^sub>2)"
  shows "sb1 ⊎ sb2 = sb"
  by (simp add: assms)

lemma sbunion_beql:
  assumes "sb1 = (sb★\<^sub>1)"
    and "sb2 = (sb★\<^sub>2)"
  shows "sb1 ⊎ sb2 \<triangleq> sb"
  by (simp add: assms sbEQ_def)

lemma sbunion_len[simp]:
  fixes sb1 :: "'cs1^Ω"
  and   sb2 :: "'cs2^Ω"
  assumes "chDom TYPE('cs1) ∩ chDom TYPE('cs2) = {}"
  shows "#(sb1 ⊎ sb2) = min (#sb1) (#sb2)"
proof-
  have "#((sbUnion_magic_abbr::'cs1^Ω ⇒ 'cs2^Ω ⇒ ('cs1 ∪ 'cs2)^Ω)
          sb1 sb2) = min (#sb1) (#sb2)"
    by (rule sbunion_magic_len, simp_all add: assms)
  thus ?thesis
    by simp
qed

lemma union_minus_nomagfst[simp]:
        fixes sb1 ::"(('a ∪ 'b) - 'c ∪ 'd)^Ω"
          and sb2 ::"('c ∪ 'd)^Ω"
```

```
        shows "sb1 ⊎\<^sub>⋆ sb2 = ((sb1 ⊎\<^sub>- sb2)⋆\<^sub>1)"
  apply(rule sb_eql,simp)
  by(case_tac "Rep c∈ chDom TYPE('c ∪ 'd)",auto)

lemma union_minus_nomagsnd[simp]:
        fixes sb1 ::"(('a ∪ 'b) - 'c ∪ 'd)^Ω"
          and sb2 ::"('c ∪ 'd)^Ω"
        shows "sb1 ⊎\<^sub>⋆ sb2 = ((sb1 ⊎\<^sub>- sb2)⋆\<^sub>2)"
  apply(rule sb_eql,simp)
  by(case_tac "Rep c∈ chDom TYPE('c ∪ 'd)",auto)

lemma union_minus:
  fixes sb1 ::"('cs1-'cs2)^Ω"
  and   sb2 ::"'cs2^Ω"
  shows "(sb1 ⊎ sb2)⋆ = sb1 ⊎\<^sub>- sb2"
  by simp

lemma sbunion_belowI:
  fixes sb1::"'cs1^Ω" and sb2::"'cs2^Ω"
    assumes "sb1 ⊑ (out⋆)" and "sb2⊑(out⋆)"
    shows "sb1 ⊎\<^sub>⋆ sb2 ⊑ out"
  apply(rule sb_belowI, rename_tac c)
  apply(case_tac "(Rep c)∈(chDom TYPE('cs1))", simp)
  using assms(1) apply(auto simp add: fun_below_iff)
  using sbgetch_sbelow
  apply (metis (mono_tags, hide_lams) abs_reduction abs_rep_id assms(1) sbGetCh.rep_eq sbconv_eq sbgetch_insert2
        sbtypecast_rep)
  apply(case_tac "(Rep c)∈(chDom TYPE('cs2))", auto)
  using assms(1) apply(auto simp add: fun_below_iff)
  using sbgetch_sbelow
  apply (metis (mono_tags, hide_lams) abs_reduction abs_rep_id assms(2) sbGetCh.rep_eq sbconv_eq sbgetch_insert2
        sbtypecast_rep)
  done

lemma sbunion_belowI2:
  fixes sb1::"'cs1^Ω" and sb2::"'cs2^Ω"
    assumes "sb1 ⊑ (out⋆\<^sub>1)" and "sb2⊑(out⋆\<^sub>2)"
    shows "sb1 ⊎ sb2 ⊑ out"
  by (metis assms(1) assms(2) monofun_cfun monofun_cfun_arg
     ubunion_id)

lemma sbunion_minus_len:
  fixes sb1 :: "('cs1-'cs2)^Ω" and sb2 :: "'cs2^Ω"
  assumes "chDom TYPE('cs2) ⊆ chDom TYPE('cs1)"
  shows "#(sb1 ⊎\<^sub>- sb2) = min (#sb1) (#sb2)"
  apply (rule sbunion_magic_len)
  apply (simp add: inf_commute)
  using assms by auto

lemma sbunion_minus_sbunion_star_eq:
  fixes sb1::"'cs1^Ω"
  and   sb2::"'cs2^Ω"
  shows "(sb1⋆\<^sub>-) ⊎\<^sub>- sb2 = (sb2 ⊎\<^sub>⋆ sb1)"
  apply(rule sb_eql, simp)
  apply(case_tac " Rep c ∈ chDom TYPE ('cs2)")
  apply(simp_all)
  by(simp add: sbgetch_insert)

lemma sbunion_minus_star_minus_fst_id:
  fixes sb1::"'cs1^Ω"
  and   sb2::"'cs2^Ω"
  assumes "⋀c. Rep c∈ chDom TYPE ('cs1) ∩ chDom TYPE ('cs2)⟹sb1 \<^enum> c = sb2 \<^enum>\<^sub>⋆ c"
  shows   "(sb1⋆\<^sub>-) ⊎\<^sub>- sb2 = sb1"
  apply(rule sb_eql)
  apply(case_tac "Rep c ∈chDom TYPE ('cs1) - chDom TYPE ('cs2)", simp_all add: assms)
  by(simp add: sbgetch_insert)

subsubsection ⟨Renaming of Channels⟩


lift_definition sbRenameCh::"('cs1 ⇒ 'cs2) ⇒ 'cs2^Ω → 'cs1^Ω" is
"λf sb. if (∀c. ctype (Rep (f c)) ⊆ ctype (Rep c))
          then Abs_sb (λc. sb \<^enum> (f c))
          else undefined"
  apply(intro cont2cont)
  apply(rule sbwellI) using sbgetch_ctypewell by blast

lemma sbrenamech_rep:
  assumes "⋀c. ctype (Rep (f c)) ⊆ ctype (Rep c)"
  shows "Rep_sb (sbRenameCh f·sb) = (λc. sb \<^enum> (f c)) "
  apply(simp add: assms sbRenameCh.rep_eq)
  apply(rule Abs_sb_inverse, simp)
  apply(rule sbwellI) using sbgetch_ctypewell assms by blast


theorem sbrenamech_getch[simp]:
  assumes "⋀c. ctype (Rep (f c)) ⊆ ctype (Rep c)"
  shows "(sbRenameCh f·sb) \<^enum> c = sb \<^enum> (f c)"
  by (simp add: assms sbgetch_insert2 sbrenamech_rep)
```

```
lemma sbrenamech_len[simp]:
  fixes f::"'cs1 ⇒ 'cs2"
  assumes "⋀c. ctype (Rep (f c)) ⊆ ctype (Rep c)"
    and "¬chDomEmpty TYPE('cs2)"
  shows "#sb ≤ #(sbRenameCh f·sb)"
  by(rule sblengeq, simp add: assms)

lemma sbconvert_eq2:
  fixes sb1::"'a^Ω"
  and   sb2::"'b^Ω"
  assumes "chDom TYPE('a) = chDom TYPE ('b)"
  shows   "sb1⋆ = sb2⟹sb1 = (sb2⋆)"
  apply(rule sb_eql, simp)
  by (metis assms sbunion_getch sbunion_sbtypecast_eq)

text ⟨In some cases only certain channels should be modified, while
  keeping all other channels. For this case we define an alternative
  version of ⟨sbRename⟩. It takes an partial function as argument.
  Only the channels in the domain of the function are renamed.⟩
definition sbRename_part::"('cs1 ⇀ 'cs2) ⇒ 'cs2^Ω → 'cs1^Ω" where
"sbRename_part f = sbRenameCh (λcs1. case (f cs1) of Some cs2 ⇒ cs2
                                               | None ⇒ Abs (Rep cs1))"

lemma sbrenamepart_well[simp]:
    fixes f :: "('cs1 ⇀ 'cs2)"
  assumes "⋀c. c∈dom f⟹ctype (Rep (the (f c))) ⊆ ctype (Rep c)"
      and "⋀c. c∉dom f⟹(Rep c) ∈ chDom TYPE ('cs2)"
    shows "ctype (Rep (case f c of None ⇒ Abs (Rep c) | Some (cs2::'cs2) ⇒ cs2)) ⊆ ctype (Rep c)"
  apply(cases "c∈dom f")
  apply(auto simp add: assms)
  using assms(1) apply fastforce
  by (simp add: assms(2) domIff rep_reduction)

text ⟨The ⟨getch⟩ lemmata is seperated into two cases. The first case is when
  the channel is part of the mapping. This first assumption is directly
  taken from the normal ⟨sbRename⟩ definition. The second assumption
  ensures that unmodified channels also exist in the output bundle.⟩
theorem sbrenamepart_getch_in[simp]:
    fixes f :: "('cs1 ⇀ 'cs2)"
    assumes "⋀c. c∈dom f⟹ctype (Rep (the (f c))) ⊆ ctype (Rep c)"
        and "⋀c. c∉dom f⟹(Rep c) ∈ chDom TYPE ('cs2)"
        and "c∈dom f"
    shows "(sbRename_part f·sb) \<^enum> c = sb \<^enum> the (f c)"
  apply(simp add: sbRename_part_def)
  apply(subst sbrenamech_getch)
  apply(auto simp add: assms)
  using assms(3) by auto


theorem sbrenamepart_getch_out[simp]:
    fixes f :: "('cs1 ⇀ 'cs2)"
    assumes "⋀c. c∈dom f⟹ctype (Rep (the (f c))) ⊆ ctype (Rep c)"
        and "⋀c. c∉dom f⟹(Rep c) ∈ chDom TYPE ('cs2)"
        and "c∉dom f"
    shows "(sbRename_part f·sb) \<^enum> c = sb \<^enum>\<^sub>⋆ c"
  apply(simp add: sbRename_part_def)
  apply(subst sbrenamech_getch)
  apply(auto simp add: assms)
  by (metis assms(2) assms(3) domIff option.simps(4) sbgetch_insert sbgetch_insert2)



end


(*:maxLineLen=68:*)
theory SB_fin
  imports SB
begin


declare %invisible[[show_types]]

default_sort %invisible"{finite,chan}"

subsection ⟨SB Functions with finite Domains⟩




subsubsection ⟨Continuous Version of sbHdElem\_h⟩


text⟨The @{const sbHdElem_h} \ref{subsub:sbhdelem} operator is in
general monotone, but not continuous. The following theorem shows
why bundle chains with infinite domains cause continuity problems.
One can construct a chain of bundle with an infinite domain, where
the first chain element is ⟨⊥⟩ and the next element of all elements
in the chain always have one more stream that is not ⟨ε⟩ anymore.
This results in an infinite chain where all chain bundles have
infinitely many empty stream but the least upper bound has none.⟩
```

```
text ⟨Thus @{const sbHdElem_h} function will output ⟨⊥⟩ for each
chain element, but for the chains loop, it returns a @{type sbElem}.
This is proven in the following theorem, where the chain is
formulated in assumptions.⟩

theorem sbhdelem_h_n_cont:
  fixes   Y::"nat ⇒ ('cs::chan)^Ω"
  assumes "chain Y"
  and     "⋀i. ¬sbHdElemWell (Y i)"
  and     "sbHdElemWell (⨆i. Y i)"
  shows   "(⨆i. sbHdElem_h (Y i)) ≠ sbHdElem_h (⨆i. Y i)"
  apply(auto simp add: sbHdElem_h_def assms)
  using assms(3) sbHdElemWell_def apply auto[1]
proof-
  assume a1:"Iup (Abs_sbElem (Some (λc. shd ((⨆x. Y x)  \<^enum>  c))))=⊥"
  have "¬sbElem_well (Some (λc. shd ((⨆x. Y x)  \<^enum>  c))) ∧
        sbElem_well (Some (λc. shd ((⨆x. Y x)  \<^enum>  c)))"
    by (metis a1 inst_up_pcpo u.distinct(1))
  then show False
    by blast
qed


lemma cont_h2:
  assumes"∃s. s∈UNIV ∧ s∉{c::'c. ∃i::nat. Y i  \<^enum>  c ≠ ϵ}"
  shows"{c::'c. ∃i::nat. Y i  \<^enum>  c ≠ ϵ}≠UNIV"
  using assms by auto

lemma sbisleastadm[simp]:
  "adm (sbIsLeast::('cs::{finite,chan})^Ω⇒bool)"
  apply(simp only: sbHdElemWell_def,simp)
  proof(rule admI)
    fix Y::"nat ⇒ 'a^Ω"
    assume chain:"chain Y"
    assume epsholds:"∀i::nat. ∃c::'a. Y i  \<^enum>  c = ϵ"
    have well:"∀ i.  ¬ sbHdElemWell (Y i) "
      by (simp only: sbHdElemWell_def,simp add: epsholds)
    have h0:"∀c i. ((Y i) \<^enum> c ≠ ϵ)⟶((⨆i::nat. Y i)  \<^enum>  c ≠ ϵ)"
      by (metis (full_types) chain is_ub_thelub minimal
          monofun_cfun_arg po_eq_conv)
    then obtain set_not_eps where set_not_eps_def:
      "set_not_eps = {c::'a. ∃i. Y i \<^enum> c ≠ ϵ}"
    by simp
    have h01:"finite set_not_eps"
      by simp
    have h1:"∀c∈(UNIV - set_not_eps). (⨆i::nat. Y i)  \<^enum>  c = ϵ"
      by (simp add: chain contlub_cfun_arg set_not_eps_def)
    have "set_not_eps ≠ UNIV"
    proof(auto)
      assume a1: "set_not_eps = UNIV"
      have "∃c ∈ UNIV. (⨆i::nat. Y i)  \<^enum>  c ≠ ϵ"
        using a1 h0 set_not_eps_def by blast
      have "∃ i. sbHdElemWell (Y i)"
      proof (rule ccontr, simp)
       assume a10: "∀i::nat. ¬ sbHdElemWell (Y i)"
       have f110: "⋀ i::nat. ¬ sbHdElemWell (Y i)"
         by (simp add: a10)
       obtain i where i_def: "¬ sbHdElemWell (Y i)"
         by (simp add: a10)
       obtain ch_not_eps where ch_not_eps_def:
         "ch_not_eps = {{i. Y i \<^enum> (ch) ≠ ϵ}|ch::'a. True }"
         by blast
       obtain surj_f where surj_f_def:
         "surj_f = (λ ch. {i. Y i \<^enum> (ch::'a) ≠ ϵ})"
         by simp
       have "ch_not_eps ⊆  surj_f ` ({c::'a | c. True})"
         using ch_not_eps_def surj_f_def by auto
       then have ch_not_epsfinite: "finite ch_not_eps"
         by (metis finite_code finite_surj)
       have ch_not_eps_ele_not_emp: "∀ ele ∈ ch_not_eps. ele ≠ {}"
         using ch_not_eps_def a1 set_not_eps_def by blast
       have dom_emty_iff:"⋀c. (ch_not_eps={})⟷
                          (Rep (c::'a) ∈ cEmpty)"
         by (metis (mono_tags, lifting) Collect_empty_eq
            Diff_eq_empty_iff Int1 chDom_def ch_not_eps_def
            ch_not_eps_ele_not_emp chan_botsingle empty_iff
            mem_Collect_eq repinrange sbGetCh.rep_eq)
       have dom_not_emp_false: "ch_not_eps= {}"
       proof -
         have el_ch_not_eps_least: "∀ ele. ele ∈ ch_not_eps
           ⟶(∃ i. i ∈ ele ∧ (∀ j ∈ ele. i ≤ j))"
         proof (rule ccontr, simp)
           assume a1111: "∃ele::nat set. ele ∈ ch_not_eps ∧
                  (∀i::nat. i ∈ ele⟶(∃x::nat∈ele. ¬ i ≤ x))"
           obtain the_ch where the_ch_def:
               "(surj_f the_ch) ∈ ch_not_eps ∧
               (∀i::nat. i ∈ (surj_f the_ch)
               ⟶(∃x::nat ∈ (surj_f the_ch). ¬ i ≤ x))"
                and the_ch_def2:
                "(surj_f the_ch) = {i. Y i \<^enum> the_ch ≠ ϵ}"
             using a1111 ch_not_eps_def surj_f_def by blast
           obtain the_i where the_i_def: "the_i ∈ (surj_f the_ch)"
             using ch_not_eps_ele_not_emp the_ch_def by auto
```

```
            obtain the_subs where the_subst_def:
              "the_subs = {i. i ≤ the_i ∧ Y i \<^enum> the_ch ≠ ε}"
              by simp
            have the_subs_fin: "finite the_subs"
              by (simp add: the_subst_def)
            hence the_min_in_subs: "Min the_subs ∈ the_subs"
              using Min_in the_subs_fin the_i_def the_subst_def
                    the_ch_def2 by blast
            hence the_min_min: "∀ i ∈ (surj_f the_ch).
                                   Min the_subs ≤ i"
              using the_ch_def2 nat_le_linear the_subst_def
              by fastforce
            show False
              using the_ch_def the_min_in_subs the_min_min surj_f_def
                the_ch_def the_subst_def by auto
          qed
           obtain bla::"nat set ⇒ nat set" where bla_def:
      "bla = (λ da_set. {the_i. (∀ i ∈ da_set. the_i ≤ i) ∧
      the_i ∈ da_set})"
             by simp
           obtain min_set_set::"nat set" where min_set_set_def:
      "min_set_set = {THE i. i ∈ bla da_set |
      da_set. da_set ∈ ch_not_eps}"
             by simp
           have i_max_is_max: "∀ ch::'a. ∃ i .
      (i ∈ min_set_set)⟶Y i \<^enum> ch ≠ ε"
             using a1 set_not_eps_def by blast
           have min_set_set_finite: "finite min_set_set"
             by (simp add: ch_not_epsfinite min_set_set_def)
           obtain the_max where the_max_def:
                 "the_max = Max min_set_set"
             by simp
           have "the_max ∈ min_set_set"
             by (metis (mono_tags, lifting) Max_in min_set_set_finite
                 ch_not_eps_def empty_Collect_eq equals0I
                 min_set_set_def the_max_def)
           have "sbHdElemWell (Y the_max)"
           proof (simp only: sbHdElemWell_def, rule)
             fix c::'a
             obtain the_set where the_set_def: "the_set = surj_f c"
               by simp
             then obtain the_min where the_min_def:
               "the_min ∈ the_set ∧ (∀ j ∈ the_set. the_min ≤ j)"
               using el_ch_not_eps_least ch_not_eps_def surj_f_def
                     the_set_def by blast
             have "bla the_set = {the_min}"
               using bla_def the_min_def by force
             hence "(THE i::nat. i ∈ bla the_set) = the_min"
               by auto
             hence the_min_min_set_set_in: "the_min ∈ min_set_set"
               using min_set_set_def ch_not_eps_def surj_f_def
                     the_set_def by blast
             have "Y the_min \<^enum> c ≠ ε"
               using surj_f_def the_min_def the_set_def by blast
             thus "Y the_max  \<^enum>  c ≠ ε"
               by (metis min_set_set_finite the_max_def chain
                   po_class.chain_mono minimal monofun_cfun_arg
                   po_eq_conv Max_ge the_min_min_set_set_in)
           qed
           then show ?thesis
             by (simp add: a10)
         qed
         then show False
           using ch_not_eps_def by auto
       qed
       thus False
         by (metis well)
     qed
     then show "∃c. (⨆i::nat. Y i)  \<^enum>  c = ε"
       using h1 by blast
   qed

lift_definition sbHdElem_h_cont::"'cs::{finite,chan}^Ω→ ('cs^√) u"
is "sbHdElem_h"
  unfolding sbHdElem_h_def
  apply(intro cont2cont)
  apply(rule Cont.contI2)
  apply(rule monofunI)
  apply auto[1]
  apply (metis sbhdelem_mono_eq sbhdelem_some sbhdelemchain)
proof-
  fix Y::"nat ⇒ 'c^Ω"
  assume ch1:"chain Y"
  assume ch2:"chain (λi::nat. if sbIsLeast (Y i) then ⊥ else
                 Iup (Abs_sbElem (Some (λc::'c. shd (Y i  \<^enum>  c)))))"
  have "adm (λsb::'c^Ω. ∃c. sb  \<^enum>  c = ε)"
    apply(insert sbisleastadm)
    by(simp only: sbHdElemWell_def,simp)
  hence "∀i::nat. ∃c::'c. Y i  \<^enum>  c = ε
         ⟹∃c::'c. (⨆i::nat. Y i)  \<^enum>  c = ε"
    using admD ch1 by blast
  hence finiteIn:"∀c::'c. (⨆i::nat. Y i)  \<^enum>  c ≠ ε
         ⟹∃i. ∀c::'c. (Y i) \<^enum> c ≠ ε"
    by blast
```

```
  thus "(if sbIsLeast (⨆i::nat. Y i) then ⊥ else
Iup (Abs_sbElem (Some (λc::'c. shd ((⨆i::nat. Y i)  \<^enum>  c))))) ⊑
       (⨆i::nat. if sbIsLeast (Y i) then ⊥
else Iup (Abs_sbElem (Some (λc::'c. shd (Y i  \<^enum>  c)))))"
  proof(cases "sbIsLeast (⨆i::nat. Y i)")
    case True
    then show ?thesis
      using sbnleast_mex by auto
  next
    case False
    obtain n where n_def:"¬sbIsLeast (Y n)"
      by (metis False finiteIn sbHdElemWell_def)
    have "sbHdElemWell (⨆x::nat. Y x)⟹
    Abs_sbElem (Some (λc::'c. shd ((⨆x::nat. Y x)  \<^enum>  c))) =
    Abs_sbElem (Some (λc::'c. shd (Y n  \<^enum>  c)))"
      by (metis below_shd ch1 is_ub_thelub n_def sbHdElemWell_def
          sbgetch_sbelow)
    hence "(if sbIsLeast (⨆i. Y i) then ⊥ else Iup
(Abs_sbElem (Some (λc::'c. shd ((⨆i::nat. Y i)  \<^enum>  c))))) ⊑
 Iup (Abs_sbElem (Some (λc::'c. shd (Y n  \<^enum>  c))))"
      by auto
    then show ?thesis
      by (metis (mono_tags, lifting) below_lub ch2 n_def)
  qed
qed

subsubsection⟨SB step functions⟩

text⟨Often a \gls{sb} is processed element wise by an automaton.
 If there is no complete element in the \gls{sb} it returns ⟨⊥⟩.
We use the following definition for realising the automaton
semantics, but it could also be used to define a continuous version
of the length definition\ref{subsub:sblen} for \Gls{sb} with a finite
domain or other operators.⟩

definition sb_split::"('cs^√⇒ 'cs^Ω→ 'a::pcpo) → 'cs^Ω→ 'a" where
"sb_split ≡ Λ k sb. fup·(Λ sbe. k sbe·(sbRt·sb))·(sbHdElem_h_cont·sb)"



lemma sb_split_insert:"sb_split·k·sb = (case sbHdElem_h_cont·sb of
up·(sbe::'b^√) ⇒ k sbe·(sbRt·sb))"
  apply(simp add: sb_split_def)
  apply(subst beta_cfun)
  apply(intro cont2cont,simp_all)
  using cont2cont_fst cont_fst cont_snd discr_cont3 by blast



lemma sb_splits_bot[simp]:
"¬(chDomEmpty (TYPE ('cs)))⟹sb_split·f·(⊥::'cs^Ω) = ⊥"
  by(simp add: sb_split_insert sbHdElem_h_cont.rep_eq sbHdElem_h_def
      chDom_def)

theorem sb_splits_leastbot[simp]:
  fixes    sb::"'cs^Ω"
  assumes "¬chDomEmpty (TYPE ('cs))"
      and "sbIsLeast sb"
  shows   "sb_split·f·sb = ⊥"
  using assms
  by(simp add: sb_split_insert sbHdElem_h_cont.rep_eq sbHdElem_h_def
      chDom_def)

lemma sb_splits_len0[simp]:
  fixes    sb::"'cs^Ω"
  assumes "#sb = 0"
  shows   "sb_split·f·sb = ⊥"
  apply(cases "chDomEmpty (TYPE ('cs))")
  using assms apply auto[1]
  using assms sb_splits_leastbot sbnleast_len by blast



theorem sb_splits_sbe[simp]:"sb_split·f·(sbe •^√ sb) = f sbe·sb"
  apply (subst sb_split_insert)
  apply (subst sbrt_sbecons)
  by(simp add: sbHdElem_h_cont.rep_eq sbhdelem_h_sbe)

lemma sbsplit_eql: assumes "⋀sbe sb. spf·(sbe •^√ sb) = f sbe·sb"
      and "⋀sb. #sb = 0⟹spf·sb = ⊥"
    shows "spf = sb_split·f"
  apply(rule cfun_eqI, rename_tac "sb")
  apply(case_tac "sb" rule: sb_cases2)
  by (simp add: assms)+

lemma sb_splits_sbe_empty[simp]:
"chDomEmpty TYPE('cs)⟹sb_split·f·(sb::'cs^Ω) = f sbe·sb"
  by (metis (full_types) sb_splits_sbe sbtypeepmpty_sbbot)

lemma sb_splits_sbe_empty2:
"chDomEmpty TYPE('cs)⟹sb_split·f·(sb::'cs^Ω) =
                                          f (Abs_sbElem None)·⊥"
  by (metis (full_types) sb_splits_sbe sbtypeepmpty_sbbot)
```

```
subsection‹Datatype Constructors for SBs›


text‹Type ‹'a› can be interpreted as a tuple. Because we have
almost no assumptions for ‹'a ›, the user can freely choose ‹'a›.
Hence, he will not use the datatype ‹M›. These locales could for
example create setters from from ‹'a = (nat × bool) stream› and
‹'a = (nat stream × bool stream)› to a bundle with one
‹bool›-channel and one ‹nat›-channel. Thus, we can construct all
bundles with a finite domain.›



subsubsection‹sbElem Locale›

text‹The first locale provides two mappings. The fist one maps some
type ‹'a› to a @{type sbElem}. The second one maps a @{type stream}
of type ‹'a› to a \gls{sb}. For example could a \gls{sb} setter map
a ‹nat×bool› @{type stream} to a \gls{sb} with a ‹nat› and a ‹bool›
stream. The constructor mapping is always injective, but in general
not surjective.›

text‹The locales assumptions depend on the constructors domain.
If the domain is empty, ‹'a› must be a singleton, else, the
constructor has to map to a function that can be interpreted as a
@{type sbElem}. The constructor also has to map to every
possible @{type sbElem} and be injective.›

locale sbeGen =
  fixes lConstructor::"'a::countable ⇒ 'cs::{chan, finite} ⇒ M"
  assumes c_well: "⋀a. ¬chDomEmpty TYPE ('cs)
                    ⟹∀c. lConstructor a c ∈ ctype(Rep c)"
      and c_inj: "¬chDomEmpty TYPE ('cs)⟹inj lConstructor"
      and c_surj: "⋀sbe. ¬chDomEmpty TYPE ('cs)
                    ⟹∀c. sbe c ∈ ctype(Rep c)
                    ⟹sbe∈range lConstructor"
      and c_empty: "chDomEmpty TYPE ('cs)
                    ⟹is_singleton(UNIV::'a set)"

begin



lift_definition setter::"'a ⇒ 'cs^√" is
"if chDomEmpty TYPE ('cs) then (λ_. None) else Some o lConstructor"
  using c_well sbtypeempty_sbewell by auto

text‹In the definition of the setter ‹o› is used to compose the
mappings ‹Some› and ‹lConstructor› into one function where ‹Some›
is applied to the output of ‹lConstructor›.›

text‹The getter work analogous with the inverse constructor. If the
input @{type sbElem} is ‹None›, we know that the domain ‹'cs› is
empty and hence, the type ‹'a› only contains one element. Therefore,
@{const undefined} can be returned.›

definition getter::"'cs^√⇒ 'a" where
"getter sbe ≡ case (Rep_sbElem sbe) of
          None    ⇒ undefined        |
          Some f ⇒ (inv lConstructor) f"



theorem get_set[simp]: "getter (setter a) = a"
  unfolding getter_def
  apply (simp add: setter.rep_eq c_inj c_empty)
  by (metis (full_types)UNIV_I c_empty is_singletonE singleton_iff)

lemma set_inj: "inj setter"
  by (metis get_set injI)

lemma set_surj: "surj setter"
  unfolding setter_def
  apply(cases "¬(chDomEmpty(TYPE('cs)))",auto)
  apply(simp add: chDom_def)
  apply auto
proof-
  fix xb::"'cs^√" and xa::'cs
  assume chnEmpty:"Rep xa ∉ cEmpty"
  obtain f where f_def:"Rep_sbElem xb=(Some f)"
    using chnEmpty sbtypenotempty_fex cempty_rule by blast
  then obtain x where x_def:"f = lConstructor x"
    by (metis (no_types) chnEmpty c_surj empty_iff imageE
        notcdom_empty sbElem_well.simps(2) sbelemwell2fwell)
  then show "xb∈range (λx::'a. Abs_sbElem (Some (lConstructor x)))"
    by (metis (no_types,lifting) Rep_sbElem_inverse f_def range_eqI)
qed

lemma set_bij: "bij setter"
```

```
  by (metis bijI inj_onI sbeGen.get_set sbeGen_axioms set_surj)

lemma get_inv_set: "getter = (inv setter)"
  by (metis get_set set_surj surj_imp_inv_eq)


theorem set_get[simp]: "setter (getter sbe) = sbe"
  apply(simp add: get_inv_set)
  by (meson bij_inv_eq_iff set_bij)

lemma "getter A = getter B⟹A = B"
  by (metis set_get)


fixrec setterSB::"'a stream → 'cs^Ω" where
"setterSB·((up·l)&&ls) = (setter (undiscr l)) •^√ (setterSB·ls)"


lemma settersb_unfold[simp]:
"setterSB·(↑a • s) = (setter a) •^√ setterSB·s"
  unfolding setterSB_def
  apply(subst fix_eq)
  apply simp
  apply(subgoal_tac "∃l. ↑a • s = (up·l)&&s")
  apply auto
  apply (metis (no_types, lifting) lshd_updis stream.sel_rews(4)
         undiscr_Discr up_inject)
  by (metis lscons_conv)

lemma settersb_emptyfix[simp]:
"chDomEmpty (TYPE ('cs))⟹setterSB·s = ⊥"
  by simp

lemma settersb_strict[simp]:"setterSB·ε = ⊥"
  apply(simp add: setterSB_def)
  apply(subst fix_eq)
  by auto

definition getterSB::"'cs^Ω→ 'a stream" where
"getterSB ≡ fix·(Λ h. sb_split·
                (λsbe. Λ sb. updis (getter sbe) && h·sb))"


lemma gettersb_unfold[simp]:
"getterSB·(sbe •^√ sb) = ↑(getter sbe) • getterSB·sb"
  unfolding getterSB_def
  apply(subst fix_eq)
  apply simp
  by (simp add: lscons_conv)

lemma gettersb_emptyfix:
  "chDomEmpty (TYPE ('cs))
 ⟹getterSB·sb = ↑(getter (Abs_sbElem None)) • getterSB·sb"
  by (metis(full_types) gettersb_unfold sbtypeepmpty_sbbot)

lemma gettersb_empty_inf: "chDomEmpty (TYPE ('cs))
 ⟹getterSB·sb = sinftimes (↑(getter (Abs_sbElem None)))"
  using gettersb_emptyfix s2sinftimes by blast

lemma gettersb_realboteps[simp]:
  "¬(chDomEmpty (TYPE ('cs)))⟹getterSB·⊥ = ε"
  unfolding getterSB_def
  apply(subst fix_eq)
  by (simp)

lemma gettersb_boteps[simp]:
  "¬(chDomEmpty (TYPE ('cs)))⟹sbIsLeast sb⟹getterSB·sb = ε"
  unfolding getterSB_def
  apply(subst fix_eq)
  by (simp)

lemma gettersb_inftimes:
  assumes "chDomEmpty (TYPE ('cs))"
  shows "(getterSB·sb) = (sinftimes(↑(a)))"
  apply(insert assms,subst gettersb_emptyfix,simp)
  using gettersb_emptyfix s2sinftimes c_empty
  by (metis (mono_tags) get_set sbtypeepmpty_sbenone)

lemma "¬chDomEmpty TYPE('cs)⟹sbLen (setterSB·s) = #s"
  oops

lemma "a ⊑ getterSB·(setterSB·a)"
  apply(induction a rule: ind)
  apply(auto)
  by (simp add: monofun_cfun_arg)

theorem getset_eq[simp]:
  assumes "¬chDomEmpty (TYPE ('cs))"
  shows   "getterSB·(setterSB·a) = a"
```

```
  apply(induction a rule: ind)
  using assms by auto

lemma "setterSB·(getterSB·sb) ⊑ sb"
  apply(induction sb,simp)
  apply(cases "chDomEmpty(TYPE('cs))",simp,simp)
  apply(subst gettersb_unfold;subst settersb_unfold)
  by (metis cont_pref_eq1I set_get)

lemma "sb1 = sb2⟹sbe •^√ sb1 = sbe •^√ sb2"
  by simp

(*Important TODO*)
lemma setget_eq:"(∀c. #(sb \<^enum> c) = k)⟹setterSB·(getterSB·sb) = sb"
  oops


fun setterList::"'a list ⇒ 'cs^Ω" where
"setterList [] = ⊥" |
"setterList (l#ls) = (setter l) •^√ (setterList ls)"

end


(* Das ist ein Test, ob "sbeGen" auch mit leeren kanälen klappt *)
locale sbeGen_empty =
  fixes t::"'cs::{emptychan, finite} itself"
begin
end

sublocale sbeGen_empty ⊆ sbeGen"(λ(u::unit) cs::'cs. undefined)"
  apply(standard, simp_all)
  by (metis card_UNIV_unit is_singleton_altdef)




end



theory sbLocale

imports SB

begin

subsubsection ⟨Lifting from Stream to Bundle⟩

text⟨This section provides a bijective mapping from ⟨'a ⟩ to
\gls{sb}. Type ⟨'a⟩ could for example be a ⟨nat stream × bool stream⟩.
A locale \cite{ballarin2006interpretation} can be used to lift functions over streams to bundles. The number of
      channels is not
fixed, it can be an arbitrary large number.⟩

text ⟨A ⟨locale⟩ is a special environment within Isabelle. In the beginning of the locale
  are multiple assumptions. Within the locale these can be freely used. To use the locale the
  user has to proof these assumptions later. After that all definitions and theorems in the locale
  are accessible. The locale can be used multiple times. ⟩

text ⟨The definition ⟨lConstructor⟩ maps the ⟨'a⟩ element to a
corresponding \gls{sb}. The constructor has to be injective and maps precisely
to all possible functions, that can be lifted to stream bundles.
Since the setter and getter in this locale are always bijective, all
\glspl{sb} can be constructed.⟩

text⟨The continuity of the setter is given by assuming the
continuity of the constructor. Thus continuity of the getter follows
from assuming that the constructor maintains non-prefix orders and
from the continuity and surjectivity of the setter. Furthermore, assumptions
over the length (⟨#⟩) exist.⟩

(*Todo exchange c_type with sb_well assumption*)

locale sbGen =
 fixes lConstructor::"'a::{pcpo,len} ⇒ 'cs::{chan} ⇒ M stream"
 assumes c_type: "⋀a c. sValues·(lConstructor a c) ⊆ ctype (Rep c)"
   and c_inj: "inj lConstructor"
   and c_surj: "⋀f. sb_well f⟹f∈range lConstructor"
   and c_cont: "cont lConstructor"
   and c_nbelow: "⋀x y. ¬(x ⊑ y)⟹
                        ¬(lConstructor x ⊑ lConstructor y)"
   and c_len:"⋀ a c. ¬chDomEmpty TYPE('cs)⟹#a ≤ # (lConstructor a c)"
   and c_lenex:"⋀ a. ¬chDomEmpty TYPE('cs)⟹∃c.#a = #(lConstructor a c)"

begin



lift_definition setter::"'a → 'cs^Ω"
  is "Abs_sb o lConstructor"
  apply(intro cont2cont)
```

```
  using c_type sbwelll apply blast
  by (simp add: c_cont cont2cont_fun)

lemma setter_rep[simp]: "Rep_sb (setter·a) = lConstructor a"
  apply(simp add: setter.rep_eq)
  by (simp add: Abs_sb_inverse c_type sbwelll)

lemma set_inj: "inj (Rep_cfun setter)"
  apply(rule injI)
  apply(simp add: setter.rep_eq)
  by (metis abs_rep_sb_sb c_inj injD setter_rep)

lemma set_surj: "surj (Rep_cfun setter)"
  unfolding setter.rep_eq setter_def
proof(simp add: surj_def,auto)
  fix y::"'cs^Ω"
 obtain f where f_def:"Rep_sb y=f"
   by simp
 then obtain x where x_def:"f = lConstructor x"
    by (metis c_inj c_surj f_the_inv_into_f sbwell2fwell)
  then have "∃x::'a. y = Abs_sb (lConstructor x)"
    by (metis Rep_sb_inverse f_def)
  thus "∃x::'a. y = Abs_cfun (Abs_sb ∘ lConstructor)·x"
    using setter.rep_eq setter_def by auto
qed

lemma set_bij: "bij (Rep_cfun setter)"
  using bij_betw_def set_inj set_surj by auto

lemma set_inv: "inv (Rep_cfun setter) = (inv lConstructor) ∘  Rep_sb"
  by (simp add: c_inj set_surj surj_imp_inv_eq)

lemma cont_inv_set:"cont (inv (Rep_cfun setter))"
  apply(intro cont2cont)
   apply (simp add: set_surj)
  using c_nbelow cont2monofunE sbrep_cont by (metis setter_rep)


lift_definition getter::"'cs^Ω → 'a"
  is "(inv lConstructor) ∘  Rep_sb"
  apply(intro cont2cont)
  using cont_inv_set set_inv by auto



theorem get_set[simp]: "getter·(setter·a) = a"
  apply (simp add: getter.rep_eq setter.rep_eq )
  using c_inj setter.rep_eq setter_rep by auto



theorem set_get[simp]: "setter·(getter·sb) = sb"
  apply (simp add: getter.rep_eq setter.rep_eq )
  by (simp add: c_surj f_inv_into_f)

lemma get_eq: "getter·A = getter·B⟹A = B"
  by (metis set_get)

(* Should not be used from the user, but still helpful! *)
lemma setter_getch: "setter·a \<^enum> c = lConstructor a c"
  by (simp add: sbgetch_insert2)

lemma setter_getch2[simp]: "(Rep c) ∈ chDom TYPE ('cs)⟹
    setter·a  \<^enum>\<^sub>*  c = lConstructor a (Abs (Rep c))"
  by (auto simp add: sbgetch_insert)

text ‹The length of the resulting bundle is connected to the length of the user-supplied
  datatype ‹'a›:›
theorem setter_len: assumes "chDom TYPE('cs) ≠ {}"
  shows "#(setter·a) = #a"
  apply(rule sblen_rule, simp add: assms)
  apply(simp add: assms setter_getch)
  apply (simp add: assms c_len)
  by (metis assms c_lenex setter_getch)


lemma getter_len: assumes "chDom TYPE('cs) ≠ {}"
  shows "#(getter·sb) = #sb"
  by (metis assms set_get setter_len)


end


(* Das ist ein Test, ob "sbGen" auch mit leeren kanälen klappt *)
locale sbGen_empty =
  fixes t::"'cs::{emptychan} itself"
begin

end

sublocale sbGen_empty ⊆ sbGen "(λ(u::unit) cs::'cs. ϵ)"
  apply(standard, simp_all)
```

```
  apply(rule injI, simp)
  apply(auto simp add: sb_well_def)
  using sValues_notempty by blast

end
```

# Appendix D

# Stream Processing Function Theories

## D.1 SPF Data Type

```
(*:maxLineLen=68:*)
theory SPF

imports bundle.SB

begin

section ⟨Stream Processing Functions in Isabelle⟩


text ⟨A \gls{spf} is written as ⟨('I^Ω → 'O^Ω)⟩. It is an continuous
  function from the input bundle ⟨('I^Ω)⟩ to an output bundle ⟨('O^Ω)⟩.
The signature of the component is directly visible from the
  type-signatur of the \gls{spf}:⟩

definition spfType ::"('I^Ω → 'O^Ω) itself ⇒
  (channel set × channel set)" where
"spfType _ = (chDom TYPE ('I), chDom TYPE ('O))"


definition spfDom ::"('I^Ω → 'O^Ω) itself ⇒ channel set" where
"spfDom = fst o spfType"


definition spfRan ::"('I^Ω → 'O^Ω) itself ⇒ channel set" where
"spfRan = snd o spfType"


definition spfIO::"('I1^Ω → 'O1^Ω) ⇒ ('I1^Ω × 'O1^Ω) set" where
"spfIO spf = {(sb, spf·sb) | sb. True}"

paragraph⟨SPF Equality \\⟩

text⟨Evaluate the equality of bundle functions with same input and
output domains disregarding different types is possible by reusing
the bundle equality ⟨\<triangleq>⟩ operator.⟩

definition spfEq::"('I1^Ω → 'O1^Ω) ⇒ ('I2^Ω → 'O2^Ω) ⇒ bool" where
"spfEq f1 f2 ≡ chDom TYPE('I1) = chDom TYPE('I2) ∧
               chDom TYPE('O1) = chDom TYPE('O2) ∧
                (∀sb1 sb2. sb1 \<triangleq> sb2⟶f1·sb1 \<triangleq> f2·sb2)"

text⟨The operator checks the domain equality of input and output
domains and then the bundle equality of its possible output bundles.
For easier use, a infix abbreviation ⟨\<triangleq>\<^sub>f⟩ is defined.⟩

abbreviation sbeq_abbr :: "('I1^Ω → 'O1^Ω) ⇒ ('I2^Ω → 'O2^Ω) ⇒ bool"
(infixr "\<triangleq>\<^sub>f" 101) where "f1 \<triangleq>\<^sub>f f2 ≡ spfEq f1 f2"
```

```
definition spfConvert::"('I^Ω → 'O^Ω) → ('Ie^Ω → 'Oe^Ω)" where
"spfConvert ≡ Λ f sb. (f·(sb⋆)⋆)"

lemma spfconvert_eq [simp]: "spfConvert·f = f"
  apply(rule cfun_eqI)
  by(simp add: spfConvert_def)

lemma spfconvert_apply:"spfConvert·F·sb = (F·(sb⋆)⋆)"
  by(simp add: spfConvert_def)


subsection ⟨Causal SPFs⟩

text⟨The \gls{spf} types introduced in this framework correspond to
their causality. Beside the continuity of the components its
causality is important for its realizeability. This section
introduces two predicates to distinguish between weak and strong
causal \glspl{spf}. The original definition of weak and strong
causality \cite{BS01} are slightly
different from our predicates. A weak causal component should always
produce the same output for ⟨n⟩ time slots, if its input is also
equal for ⟨n⟩ time slots. A strong causal components output should
even be equal for ⟨n+1⟩ time slots. The causality definitions from
\cite{BS01} can also be formulated in our framework, but are
only defined for components with infinite input and output.⟩

definition weak_causal_broy::"('I^Ω → 'O^Ω) ⇒ bool"where
"weak_causal_broy spf ≡ ∀sb1 sb2 n.
                        # sb1 = ∞ ∧ # sb2 = ∞⟶
                        sbTake n·sb1 = sbTake n·sb2⟶
                        (# (spf·sb1)=∞ ∧ # (spf·sb2)=∞) ∧
                        sbTake n·(spf·sb1) = sbTake n·(spf·sb2)"



text⟨The causal \gls{spf} types of our framework are a direct
consequence from their predicates. Strong causal components are a
subset of the weak causal components. A weak causal component is
realizable, but we also introduce a type for strong causal
components for their compositional properties. In general, both
causal types do not contain a ⟨⊥⟩ element, because this would be
inconsistent with our time model. Hence, the fixed point operator
@{const fix} can not be used for causal \gls{spf} types.⟩

subsubsection ⟨Weak causal SPF⟩

text⟨If the input ⟨sb1⟩ is prefix of another input ⟨sb2⟩, the first
output of a \gls{spf} ⟨spf·sb1⟩ is also prefix of the output
⟨spf·sb2⟩ because the function is monotone. This means, that the
component cannot change output depending on future input, because it
would immediately lead to a contradiction with the monotony
property. Thus, \glspl{spf} can not look into the future because
they are monotone. Their output depends completely on their previous
input. Since we interpret a stream bundle element as a time slice,
sbLen \ref{subsub:sblen} is enough to restrict the
behaviour to a causal one. If the output of a \gls{spf} is longer or
equally long to the input, it consists of as least as many time
slices as the input. Hence, we can define a \gls{spf} as weak, if
the output is in all cases at least as long as the input.⟩



definition weak_well::"('I::len → 'O::len) ⇒ bool" where
"weak_well f ≡ ∀sb. #sb ≤ #(f·sb)"



definition sometimesspfw::"('I^Ω → 'O^Ω)"where
"sometimesspfw = (Λ sb. Abs_sb(λc. sinftimes
                 (↑(SOME a. a ∈ ctype (Rep c)))))"

lemma sometimesspfw_well:
"¬chDomEmpty TYPE('cs)⟹
 sb_well (λc::'cs. sinftimes (↑(SOME a. a ∈ ctype (Rep c))))"
  apply(auto simp add: sb_well_def)
  using cEmpty_def cnotempty_rule some_in_eq by auto

lemma sometimesspfw_len:
"¬chDomEmpty TYPE('cs)⟹# ((sometimesspfw·sb)::'cs^Ω) = ∞"
  apply(rule sblen_rule,simp_all add: sometimesspfw_def
        sbgetch_insert2)
  by(simp add: Abs_sb_inverse sometimesspfw_well)+

lemma weak_well_adm:"adm weak_well"
  unfolding weak_well_def
  apply (rule admI)
  apply auto
  by (meson is_ub_thelub cfun_below_iff len_mono lnle_def
      monofun_def less_lnsuc order_trans)
```

```
lemma strong_spf_exist:" ∃x::('I^Ω → 'O^Ω) .
  (∀sb. lnsuc·(# sb) ⊑ # (x·sb))"
  apply(cases "chDomEmpty TYPE('O)")
  apply simp
  apply(rule_tac x=sometimesspfw in exI)
  by(simp add:  sometimesspfw_len)


cpodef ('I,'O) spfw = "{f::('I^Ω → 'O^Ω) . weak_well f}"
  apply(simp add: weak_well_def)
  apply (metis (full_types) eq_iff strong_spf_exist fold_inf inf_ub
         le2lnle lel le_cases less_irrefl trans_lnle)
  by (simp add: weak_well_adm)

setup_lifting %invisible type_definition_spfw

lemma [simp, cont2cont]:"cont Rep_spfw"
  using cont_Rep_spfw cont_id by blast

lift_definition %invisible Rep_spfw_fun::
"('I,'O)spfw → ('I^Ω →'O^Ω)"is "λ spfs. Rep_spfw( spfs)"
  by(intro cont2cont)


lemma spf_weakI:
  fixes spf :: "'I^Ω → 'O^Ω"
  assumes "⋀sb. (#sb) ⊑ #(spf·sb)"
  shows "weak_well spf"
  by(simp add: weak_well_def assms)

lemma spf_weakI2:
  fixes spf :: "'I^Ω → 'O^Ω"
  assumes "¬chDomEmpty TYPE('I)"
    and "⋀sb. # sb < ∞⟹# sb ⊑ # (spf·sb)"
  shows "weak_well spf"
proof-
  have "⋀sb. # sb = ∞⟹# sb ⊑ # (spf·sb)"
  proof (rule ccontr)
    fix sb::"'I^Ω"
    assume sb_len: "# sb = ∞"
      and not_weak: "¬ # sb ⊑ # (spf·sb)"
    then obtain k where out_len: "# (spf·sb) = Fin k"
      by (metis le_less_linear lnat_well_h2)
    have sbtake_sb_len: "# (sbTake (Suc k)·sb) = Fin (Suc k)"
      by (simp add: assms sbtake_len sb_len)
    have "# (spf·(sbTake (Suc k)·sb)) ⊑ # (spf·sb)"
      using monofun_cfun_arg sblen_monosimp sbtake_below by blast
    thus False
      by (metis Fin_Suc LNat.lnat.distinct(1) assms(2) inf_less_eq le2lnle not_less out_len sbtake_sb_len)
  qed
  thus ?thesis
    by (metis assms(2) inf_ub order.not_eq_order_implies_strict
        weak_well_def)
qed

subsubsection ⟨Strong causal SPF⟩

text⟨Strong causal \Gls{spf} model weak components, whose output
never depends on the current input. Thus, its output only depends on
earlier input. This property is again defined with sbLen ⟨(#)⟩.
Here we have to mind that an input can be infinitely long, hence,
the output will not always be longer than the input. Therefore, we
use a increment instead of the smaller relation.⟩

definition strong_well::"('I::len → 'O::len) ⇒ bool" where
"strong_well spf ≡ ∀sb. lnsuc·(#sb) ⊑ #(spf·sb)"


theorem strong2weak:"strong_well f⟹weak_well f"
  using less_lnsuc strong_well_def trans_lnle weak_well_def by blast

lemma strong_well_adm:
"adm (λx::('I, 'O) spfw. strong_well (Rep_spfw x))"
  unfolding strong_well_def
  apply (rule admI)
  apply auto
  by (meson is_ub_thelub below_spfw_def cfun_below_iff len_mono
      lnle_def monofun_def less_lnsuc order_trans)


cpodef ('I,'O) spfs = "{f::('I,'O)spfw . strong_well (Rep_spfw f)}"
  apply (metis Rep_spfw_cases mem_Collect_eq strong2weak
         strong_spf_exist strong_well_def)
  by(simp add: strong_well_adm)

setup_lifting %invisible type_definition_spfs

lemma [simp, cont2cont]:"cont Rep_spfs"
  using cont_Rep_spfs cont_id by blast

lift_definition %invisible Rep_spfs_fun::
```

```
"('I,'O)spfs → ('I^Ω →'O^Ω)"is "λ spfs. Rep_spfw_fun·(Rep_spfs spfs)"
  by(intro cont2cont)

lemma spf_strongI:
  fixes spf :: "'I^Ω → 'O^Ω"
  assumes "⋀sb. lnsuc·(#sb) ≤ # (spf·sb)"
  shows "strong_well spf"
  by(simp add: strong_well_def assms)


end
```

## D.2 Composition

```
(*:maxLineLen=68:*)
theory SPFcomp

imports bundle.SB SPF
begin

section ‹General Composition Operators›








fixrec spfComp::"('I1^Ω→ 'O1^Ω) → ('I2^Ω→ 'O2^Ω)
→ ((('I1 ∪ 'I2) - ('O1 ∪ 'O2))^Ω→ ('O1 ∪ 'O2)^Ω)" where
"spfComp·spf1·spf2·sbIn = spf1·((sbIn ⊎\<^sub>- spfComp·spf1·spf2·sbIn)⋆\<^sub>1)
                          ⊎ spf2·((sbIn ⊎\<^sub>- spfComp·spf1·spf2·sbIn)⋆\<^sub>2)"

declare %invisible spfComp.simps[simp del]

lemma spfcomp_below1:
"fix·(Λ sbOut. spf1·(sbIn ⊎\<^sub>- sbOut⋆\<^sub>1) ⊎ spf2·(sbIn ⊎\<^sub>- sbOut⋆\<^sub>2))⊑
  spfComp·spf1·spf2·sbIn"
  apply(rule fix_least_below)
  apply simp
  by (metis below_refl spfComp.simps)


definition %invisible spfComp2::"('I1^Ω→ 'O1^Ω) → ('I2^Ω→ 'O2^Ω)
→ ((('I1 ∪ 'I2) - ('O1 ∪ 'O2))^Ω→ ('O1 ∪ 'O2)^Ω)" where
"spfComp2 ≡ Λ spf1 spf2 sbIn .
      fix·(Λ sbOut. spf1·(sbIn ⊎\<^sub>- sbOut⋆\<^sub>1) ⊎ spf2·(sbIn ⊎\<^sub>- sbOut⋆\<^sub>2))"

lemma spfcomp_below2:
"spfComp ⊑
(Λ spf1 spf2 sbIn. fix·(Λ sbOut. spf1·(sbIn ⊎\<^sub>- sbOut⋆\<^sub>1) ⊎ spf2·(sbIn ⊎\<^sub>- sbOut⋆\<^sub>2)))"
    apply(subst spfComp2_def)
  apply(rule fix_least_below)
  apply(rule cfun_belowI)+
  apply(subst spfComp2_def[symmetric])
  apply simp
  apply(subst fix_eq)
  by (simp add: spfComp2_def)
hide_const %invisible spfComp2

lemma spfComp_def2:"spfComp =(Λ spf1 spf2 sbIn.
  fix·(Λ sbOut. spf1·(sbIn ⊎\<^sub>- sbOut⋆\<^sub>1) ⊎ spf2·(sbIn ⊎\<^sub>- sbOut⋆\<^sub>2)))"
  apply(rule below_antisym)
  using spfcomp_below2 apply auto[1]
  apply(rule cfun_belowI)+
  apply simp
  by (simp add: spfcomp_below1)

text‹The standard abbreviation of the composition operator is ‹⊗›.›

abbreviation spfComp_abbr::
"('I1^Ω→ 'O1^Ω) ⇒ ('I2^Ω→ 'O2^Ω)
⇒ ((('I1 ∪ 'I2) - ('O1 ∪ 'O2))^Ω→ ('O1 ∪ 'O2)^Ω)"
(infixr "⊗" 70) where "spf1 ⊗ spf2 ≡ spfComp·spf1·spf2"


abbreviation spfCompm_abbr (infixr "⊗\<^sub>⋆" 70) where
"spf1 ⊗\<^sub>⋆ spf2 ≡ sbTypeCast oo (spfComp·spf1·spf2) oo sbTypeCast"

  (* \cite{Bue17} *)

lemma spfcomp_unfold:
  fixes f::"'fIn^Ω→ 'fOut^Ω"
    and g::"'gIn^Ω→ 'gOut^Ω"
  shows "((f ⊗ g)·sbIn) =
          f·((sbIn ⊎\<^sub>- (f ⊗ g)·sbIn)⋆\<^sub>1) ⊎ g·((sbIn ⊎\<^sub>- (f ⊗ g)·sbIn)⋆\<^sub>2)"
  apply(simp add: spfComp_def)
  by(subst fix_eq, simp)

lemma spfcomp_extract_l:
  fixes f::"'fIn^Ω→ 'fOut^Ω"
    and g::"'gIn^Ω→ 'gOut^Ω"
  shows "((sb ⊎ (f ⊗ g)·sb)⋆) = f·((sb ⊎ (f ⊗ g)·sb)⋆)"
  apply(subst spfcomp_unfold)
  apply(subst sbunion_conv_snd)
   apply(simp, blast)
  apply(subst sbunion_conv_fst)
   apply(simp)
  by (simp)
```

```
lemma spfcomp_extract_r:
  fixes f::"'fIn^Ω → 'fOut^Ω"
    and g::"'gIn^Ω → 'gOut^Ω"
  assumes "chDom TYPE('fOut) ∩ chDom TYPE('gOut) = {}"
  shows "((sb ⊎ (f ⊗ g)·sb)⋆) = g·((sb ⊎ (f ⊗ g)·sb)⋆)"
  apply(subst spfcomp_unfold)
  apply(subst sbunion_conv_snd)
   apply(simp, blast)
  apply(subst sbunion_conv_snd)
   apply(simp add: assms)
  by simp

lemma arg_congsbeq: "x \<triangleq> y⟹(f·x) \<triangleq> (f·y)"
  apply(simp add: sbEQ_def,auto)
  by (metis sb_rep_eql sbgetch_insert2)

lemma spfconvert2sbeq:
  fixes f::"'fIn^Ω → 'fOut^Ω"
  and g::"'gIn^Ω → 'gOut^Ω"
  assumes "chDom TYPE ('fIn) = chDom TYPE('gIn)"
  and "chDom TYPE('fOut) = chDom TYPE('gOut)"
  shows"f = spfConvert·g⟹f \<triangleq>\<^sub>f g"
  apply(auto simp add: spfEq_def assms sbEQ_def spfConvert_def)
  apply(subgoal_tac "⋀sb. f·sb = ((g·(sb⋆))⋆)",auto)
  apply(subgoal_tac "sb1⋆ = sb2", auto)
  apply(rule sb_eql,simp)
  by (metis (full_types) assms(1) sb_eql sbconvert_eq2
      sbtypecast_getch)

lemma spfconvert_strict: "⊥⋆ = ⊥"
  apply(simp add: sbtypecast_insert)
  by (simp add: bot_sb)

lemma sbtypecast_self_inverse:
  fixes  x :: "'a^Ω"
  assumes "chDom (TYPE ('a)) = chDom (TYPE('b))"
  shows "((x⋆)::'b^Ω)⋆ = x"
  by (metis assms sbconvert_eq2)

lemma sbtypecast_fix:
  fixes f :: "'a^Ω ⇒ 'a^Ω"
  assumes "cont f"
  and "chDom TYPE('b) = chDom TYPE('a)"
  shows "(sbTypeCast::'a^Ω → 'b^Ω)·(μ x. f x) = (μ x. sbTypeCast·(f (sbTypeCast·x)))"
  apply(induction rule: cont_parallel_fix_ind)
      apply(auto simp add: assms cont_compose spfconvert_strict)
  apply(subst sbtypecast_self_inverse)
  by (simp_all add: assms)

lemma ubunion_minus_project1:
  fixes x::"(('fIn ∪ 'gIn) - 'fOut ∪ 'gOut)^Ω"
  and xa::"('fOut ∪ 'gOut)^Ω"
shows "x ⊎\<^sub>- xa⋆\<^sub>1 = (x⋆) ⊎\<^sub>- (xa⋆\<^sub>⇌)⋆\<^sub>2"
   apply(rule sb_eql)
    apply(subst union_minus_nomagfst[symmetric])
   apply(subst union_minus_nomagsnd[symmetric])
  apply(rename_tac c)
   apply(case_tac "Rep c ∈ chDom (TYPE(('fIn ∪ 'gIn) - ('fOut ∪ 'gOut)))")
    apply(subst sbunion_star_getchl, simp)+
    apply(simp add: sbgetch_insert)
   apply(subst sbunion_star_getchr, auto)+
  by(auto simp add: sbgetch_insert)[2]

lemma ubunion_minus_project2:
  fixes x::"(('fIn ∪ 'gIn) - 'fOut ∪ 'gOut)^Ω"
  and xa::"('fOut ∪ 'gOut)^Ω"
    shows "x ⊎\<^sub>- xa⋆\<^sub>2 = (x⋆) ⊎\<^sub>- (xa⋆\<^sub>⇌)⋆\<^sub>1"
   apply(rule sb_eql)
    apply(subst union_minus_nomagfst[symmetric])
   apply(subst union_minus_nomagsnd[symmetric])
  apply(rename_tac c)
   apply(case_tac "Rep c ∈ chDom (TYPE(('fIn ∪ 'gIn) - ('fOut ∪ 'gOut)))")
    apply(subst sbunion_star_getchl, simp)+
    apply(simp add: sbgetch_insert)
   apply(subst sbunion_star_getchr, auto)+
  by (auto simp add: sbgetch_insert)[2]



theorem spfcompcommu:
  fixes f::"'fIn^Ω → 'fOut^Ω"
  and g::"'gIn^Ω → 'gOut^Ω"
  assumes "chDom TYPE('fOut) ∩ chDom TYPE('gOut) = {}"
  shows "(f ⊗ g) \<triangleq>\<^sub>f (g ⊗ f)"
  apply(rule spfconvert2sbeq,auto)
  apply(rule cfun_eqI)
  apply(simp add: spfComp_def2 spfConvert_def)
  apply(subst sbtypecast_fix)
  apply(simp add: Un_commute)+
  apply(rule cfun_arg_cong)
  apply(rule cfun_eqI)
```

```
  apply(simp)
  apply(subst (3) ubunion_commu)
  apply(simp add: assms inf_commute)
  apply(rule sbconvert_eq2)
  apply(simp add: Un_commute)+
  apply(rule arg_cong2[where f="(⊎)"])
  apply(rule arg_cong[where f="Rep_cfun f"])
  apply(rule ubunion_minus_project1)
  apply(rule arg_cong[where f="Rep_cfun g"])
  by (rule ubunion_minus_project2)

text⟨The commutativity theorem needs a disjoint output domain
assumption, because the @{const sbUnion} operator is only
commutative for disjoint domains (see \cref{thm:unioncommu}).
Furthermore, the commutativity is proven using the special equality for
\glspl{spf} ⟨(\<triangleq>\<^sub>f)⟩. Otherwise a type error would occur, because the output
type ⟨('fOut ∪ 'gOut)^Ω⟩ is different to ⟨('gOut ∪ 'fOut)^Ω⟩⟩


theorem spfcomp_belowI:
  fixes    f   ::"'fIn^Ω → 'fOut^Ω"
    and    g   ::"'gIn^Ω → 'gOut^Ω"
  assumes "f·(sb ⊎\<^sub>- out⋆\<^sub>1) ⊑ (out⋆\<^sub>1)"
      and "g·(sb ⊎\<^sub>- out⋆\<^sub>2) ⊑ (out⋆\<^sub>2)"
  shows   "(f⊗g)·sb ⊑ out"
  apply(simp add: spfComp_def2)
  apply(rule fix_least_below)
  apply simp
  apply(subst sbunion_belowI2; simp add: assms)
  done


(* More general:  f·(sb ⊎\<^sub>⋆ out) ⊑ (out⋆) *)
theorem spfcomp_eqI:
  fixes    f   ::"'fIn^Ω → 'fOut^Ω"
  and     g   ::"'gIn^Ω → 'gOut^Ω"
  and     out::"('fOut ∪ 'gOut)^Ω"
  assumes "chDom TYPE ('fOut) ∩ chDom TYPE ('gOut) = {}"
  and     "f·(sb ⊎\<^sub>- out⋆\<^sub>1) =(out⋆\<^sub>1)"
  and     "g·(sb ⊎\<^sub>- out⋆\<^sub>2) = (out⋆\<^sub>2)"
  and     "⋀z. f·(sb ⊎\<^sub>⋆ z) = (z⋆\<^sub>1) ∧ g·(sb ⊎\<^sub>⋆ z) = (z⋆\<^sub>2)⟹out ⊑ z"
  shows   "((f⊗g)·sb) = out"
  apply(subst spfComp_def2)
  apply(simp add: spfConvert_def)
  apply(rule fix_eqI)
  apply (insert assms,simp_all)
  by (metis assms(1) sbunion_fst sbunion_snd)


(* better document this:*)
lemma spfcomp_nofeed1 [simp]:
  fixes f::"'fIn^Ω → 'fOut^Ω"
    and g::"'gIn^Ω → 'gOut^Ω"

  assumes (* No self-loops *)
        "chDom TYPE('fOut) ∩ chDom TYPE('fIn) = {}"
    and "chDom TYPE('gOut) ∩ chDom TYPE('gIn) = {}"
      (* No feedback between components, only sequential allowed *)
    and "chDom TYPE('gOut) ∩ chDom TYPE('fIn) = {}"
  shows "(f ⊗ g)·sb = f·(sb⋆) ⊎ g·(sb ⊎\<^sub>⋆ f·(sb⋆))"
  apply(subst spfcomp_unfold)
  apply (rule arg_cong2 [where f="(⊎)"])
  subgoal X
  apply(rule arg_cong [where f="Rep_cfun f"])
    apply(rule sb_eqI, auto)
    apply(subst sbunion_minus_getchI2)
    using assms apply auto
    done
  apply(rule arg_cong [where f="Rep_cfun g"])
  apply(rule sb_eqI, auto)
  apply(case_tac "Rep c ∈ chDom TYPE((('fIn ∪ 'gIn) - 'fOut ∪ 'gOut))")
  apply auto
  apply (metis X sbunion_getchI spfcomp_unfold)
  using assms(2) by blast


lemma spfcomp_nofeed2 [simp]:
  fixes f::"'fIn^Ω → 'fOut^Ω"
    and g::"'gIn^Ω → 'gOut^Ω"
  assumes (* No self-loops *)
        "chDom TYPE('fOut) ∩ chDom TYPE('fIn) = {}"
    and "chDom TYPE('gOut) ∩ chDom TYPE('gIn) = {}"
      (* No feedback between components, only sequential allowed *)
    and "chDom TYPE('fOut) ∩ chDom TYPE('gIn) = {}"
  shows "(f ⊗ g)·sb = f·(sb ⊎\<^sub>⋆ g·(sb⋆)) ⊎ g·(sb⋆)"
  apply(subst spfcomp_unfold)
  apply (rule arg_cong2 [where f="(⊎)"])
   defer
  subgoal X
  apply(rule arg_cong [where f="Rep_cfun g"])
    apply(rule sb_eqI, auto)
```

```
    apply(subst sbunion_minus_getchl2)
    using assms apply auto
    done
  apply(rule arg_cong [where f="Rep_cfun f"])
   apply(rule sb_eql, auto)
  apply(case_tac "Rep c ∈ chDom TYPE((('fIn ∪ 'gIn) - 'fOut ∪ 'gOut))")
    apply auto
  using assms(1) apply blast
  apply(subst spfcomp_unfold)
  apply(subst X)
  apply(subst sbunion_getchr)
  using assms apply blast
  ..

lemma spfcomp_serial1:
  fixes f::"'fIn^Ω→ 'fOut^Ω"
    and g::"'gIn^Ω→ 'gOut^Ω"
  assumes "chDom TYPE ('fOut) ⊆ chDom TYPE ('gIn)"
    and "chDom TYPE ('fOut) ∩ chDom TYPE ('gOut) = {}"
    and "chDom TYPE('gOut) ∩ chDom TYPE('gIn) = {}"
    and "chDom TYPE('fOut) ∩ chDom TYPE('fIn) = {}"
    and "chDom TYPE('gOut) ∩ chDom TYPE('fIn) = {}"
  shows "(f ⊗ g)·sb = f·(sb★) ⊎\<^sub>★ (g·((sb ⊎\<^sub>★ f·(sb★))))"
  apply(subst spfcomp_nofeed1)
  by (simp_all add: assms)

text⟨Sequential and feedback compositions are a special cases of the
general composition ⟨spfComp⟩. They are useful to reduce
the complexity since they work without computing the fixpoint.
If the domains of two functions fulfill the sequential composition
assumptions, following theorem can be used for an easier output evaluation.⟩

theorem spfcomp_serial2:
  fixes f::"'fIn^Ω→ 'fOut^Ω"
    and g::"'gIn^Ω→ 'gOut^Ω"
  assumes "chDom TYPE ('gIn) ⊆ chDom TYPE ('fOut)"
    and "chDom TYPE ('fOut) ∩ chDom TYPE ('gOut) = {}"
    and "chDom TYPE('gOut) ∩ chDom TYPE('gIn) = {}"
    and "chDom TYPE('fOut) ∩ chDom TYPE('fIn) = {}"
    and "chDom TYPE('gOut) ∩ chDom TYPE('fIn) = {}"
  shows "(f ⊗ g)·sb = f·(sb★) ⊎ g·(f·(sb★)★)"
  apply(subst spfcomp_nofeed1)
  apply(simp_all add: assms)
  apply (rule arg_cong2 [where f="(⊎)"])
  apply (rule refl)
  apply(rule arg_cong [where f="Rep_cfun g"])
  apply(rule sb_eql, auto)
  using assms by auto



definition spfCompSeq::"('In^Ω→ 'Intern^Ω) → ('Intern^Ω→ 'Out^Ω)
  → ('In^Ω→ 'Out^Ω)" where
"spfCompSeq ≡ Λ spf1 spf2 sb. spf2·(spf1·sb)"

text ⟨In the sequential case the general composition @{const spfComp}
 is equivalent to @{const spfCompSeq}. The output of the general
composition is restricted to ⟨'Out⟩, because the general
composition also returns the internal channels.⟩
theorem spfcomp_to_sequential:
  fixes f::"'In^Ω→ 'Intern^Ω"
    and g::"'Intern^Ω→ 'Out^Ω"
  assumes "chDom TYPE ('In) ∩ chDom TYPE ('Intern) = {}"
    and "chDom TYPE('In) ∩ chDom TYPE('Out) = {}"
    and "chDom TYPE('Intern) ∩ chDom TYPE('Out) = {}"
  shows "(f ⊗ g)·(sb★) ¦ TYPE('Out) = spfCompSeq·f·g·sb"
  apply(subst spfcomp_serial1)
  using assms apply auto
  unfolding spfCompSeq_def
  apply simp
  apply(rule cfun_arg_cong)
  apply(rule sb_eql, auto)
  by(subst sb_star21, auto)



theorem spfcomp_parallel:
  fixes f::"'fIn^Ω→ 'fOut^Ω"
    and g::"'gIn^Ω→ 'gOut^Ω"
  assumes "chDom TYPE ('fOut) ∩ chDom TYPE ('gOut) = {}"
      and "chDom TYPE ('fOut) ∩ chDom TYPE ('gIn) = {}"
      and "chDom TYPE ('fOut) ∩ chDom TYPE ('fIn) = {}"
      and "chDom TYPE('gOut) ∩ chDom TYPE('gIn) = {}"
      and "chDom TYPE('gOut) ∩ chDom TYPE('fIn) = {}"
    shows "(f ⊗ g)·sb = f·(sb★) ⊎ g·(sb★)"
  apply(subst spfcomp_nofeed1)
  by (simp_all add: assms)



definition spfCompPar:: "('In1^Ω→ 'Out1^Ω) → ('In2^Ω→ 'Out2^Ω) →
```

```
  ('In1 ∪ 'In2)^Ω→ ('Out1 ∪ 'Out2)^Ω" where
"spfCompPar ≡ Λ spf1 spf2 sb. spf1·(sb★\<^sub>1) ⊎  spf2·(sb★\<^sub>2)"

lemma cfun_arg_cong2: "x = y⟹w=z⟹f·x·w = f·y·z"
  by simp


theorem spfcomp_to_parallel:
  fixes f::"'fIn^Ω→ 'fOut^Ω"
    and g::"'gIn^Ω→ 'gOut^Ω"
  assumes "chDom TYPE ('fOut) ∩ chDom TYPE ('gOut) = {}"
      and "chDom TYPE ('fOut) ∩ chDom TYPE ('gIn) = {}"
      and "chDom TYPE ('fOut) ∩ chDom TYPE ('fIn) = {}"
      and "chDom TYPE('gOut) ∩ chDom TYPE('gIn) = {}"
      and "chDom TYPE('gOut) ∩ chDom TYPE('fIn) = {}"
    shows "(f ⊗ g)·(sb★) ¦ TYPE('fOut ∪ 'gOut) = spfCompPar·f·g·sb"
  apply(subst spfcomp_parallel)
  using assms apply auto
  unfolding spfCompPar_def
  apply simp
  apply(rule cfun_arg_cong2; subst sb_star21; auto)
  done


definition spfCompFeed ::"('In^Ω→ 'Out^Ω) → ('In-'Out)^Ω→ 'Out^Ω" where
"spfCompFeed ≡ Λ spf sb. μ sbOut. spf·(sb ⊎\<^sub>- sbOut)"


theorem spfcomp_to_feedback:
  fixes f::"'fIn^Ω→ 'fOut^Ω"
    and g::"'gIn^Ω→ 'gOut^Ω"
  assumes "chDom TYPE ('gOut) = {}"
    shows "(f ⊗ g)·(sb★) ¦ TYPE('fOut) = spfCompFeed·f·sb"
  apply(simp only: spfComp_def2 spfCompFeed_def)
  apply (simp add: sbunion_emptyr assms)
  apply(rule parallel_fix_ind , auto)
  apply (simp add: spfconvert_strict)
  apply(subst sb_star21 , auto)
  apply(rule cfun_arg_cong)
  apply(rule sb_eql , auto simp add: sbGetCh.rep_eq sbunion_rep_eq)
  using assms by blast


end
```

# Appendix E

# Stream Processing Specification Theories

```
(*:maxLineLen=68:*)
theory SPS

imports spf.SPF
begin


 section ‹Stream Processing Specification›
text‹For the definition of \gls{sps} we use the ‹set› datatype
  already included in isabelle. A underspecified component with the input channels
‹'input› and the output channels ‹'output› has the signature:
  ‹('input^Ω→ 'output^Ω) set››


section ‹SPS Completion›

text ‹\gls{sps} ‹S› consists of a set of functions, which each describe
deterministic behaviour of a component. Upon a concrete execution, i.e.
input stream ‹i› the externally visible behaviour is ‹f(i)› for an
‹f∈S›.›
text ‹It may happen that for streams ‹i\<^sub>1,i\<^sub>2› we have ‹f\<^sub>1(i\<^sub>1) = o\<^sub>1› and
‹f\<^sub>2(i\<^sub>2) = o\<^sub>2›, but that no "joint" ‹f∈S› exists, where ‹f(i\<^sub>1) = o\<^sub>1› and
‹f(i\<^sub>2) = o\<^sub>2›. We then speak of an incomplete specification ‹S›. From an
observational point, ‹S› and ‹S∪{f}› cannot be distinguished, but when refinement
is used to specialize ‹S›, this may become a deficit. ›
text ‹We therefore introduce the completion operator ‹spsComplete›
to include all possible functions of a component such that the
black-box behaviour of the component does not change.›

definition spsComplete ::"('I1^Ω→ 'O1^Ω) set ⇒ ('I1^Ω→ 'O1^Ω) set"
  where "spsComplete sps = {spf. ∀sb. ∃spf2∈sps. spf·sb = spf2·sb}"



text‹We give a small example for the completion of two components on
  the datatype containg just ‹a› and ‹b›.
 \<^item> ‹spsConst = {[a ↦ a, b ↦ a], [a ↦ b, b ↦ b]}›
 \<^item> ‹spsID = {[a ↦ a, b ↦ b], [a ↦ b, b ↦ a]}›
›
text‹The first component contains two constant functions which have
the output a or b regardless of the input. The second component
contains the identity function as well as a function that reverses a
 and b. Therefore ‹spsConst› and ‹spsID› are different components.
However they can not be distinguished by their black-box behaviour:
‹spsIO spsConst = {(a,a),(a,b),(b,a),(b,b)} = spsID spsConst›.
If we complete both sets then both components are equal:
‹spsComplete spsConst = spsComplete spsID =
{[a ↦ a, b ↦ a], [a ↦ b, b ↦ b], [a ↦ a, b ↦ b], [a ↦ b, b ↦ a]}›.›

text‹Completion is often used to show that a completed component ‹S2›
is the extension of another component ‹S1›. By definition this holds
if for every function in ‹S1› and possible input there is a function
in ‹S2› with the same output behaviour.›

theorem spscomplete_belowI:
  assumes "⋀spf sb. spf∈S1⟹∃spf2 ∈ S2. spf·sb = spf2·sb"
  shows "S1 ⊆ spsComplete S2"
  unfolding spsComplete_def
  using assms by auto
```

```
theorem spscomplete_below: "sps ⊆ spsComplete sps"
  using spscomplete_belowI by auto


theorem spscomplete_complete [simp]:
 "spsComplete (spsComplete sps) = spsComplete sps"
  unfolding spsComplete_def apply auto
  by metis


definition spsIsComplete :: "('I1^Ω→ 'O1^Ω) set ⇒ bool" where
"spsIsComplete sps ≡ (spsComplete sps) = sps"


theorem spscomplete_empty[simp]: "spsIsComplete {}"
  unfolding spsComplete_def spsIsComplete_def by auto


theorem spscomplete_one[simp]: "spsIsComplete {f}"
  unfolding spsComplete_def spsIsComplete_def apply auto
  by (simp add: cfun_eqI)


theorem spscomplete_univ[simp]: "spsIsComplete UNIV"
  by (simp add: spsIsComplete_def spscomplete_below top.extremum_uniqueI)


section ⟨Refinement⟩



text⟨The @{const spsComplete} function is monotonic.
 Therefore if a component ⟨sps1⟩ refines a second component ⟨sps2⟩
  then this also holds after completion.⟩

theorem spscomplete_mono: assumes "sps1 ⊆ sps2"
  shows "spsComplete sps1 ⊆ spsComplete sps2"
  apply(rule spscomplete_belowI)
  unfolding spsComplete_def
  apply (auto)
  by (meson assms in_mono)





end


(*:maxLineLen=68:*)
theory SPScomp

imports SPS spf.SPFcomp
begin


section⟨General Composition of SPSs⟩


definition spsComp::
"('I1^Ω→ 'O1^Ω) set ⇒ ('I2^Ω→ 'O2^Ω) set ⇒
  ((('I1 ∪ 'I2) - 'O1 ∪ 'O2)^Ω→ ('O1 ∪ 'O2)^Ω) set" (infixr "⊗" 70)
where "spsComp F G = {f ⊗ g | f g. f∈F ∧ g∈G }"

(* TODO: Move to SB.thy if the definitions turns out to be useful *)
definition %invisible sbSameEq:: "'cs1^Ω ⇒ 'cs2^Ω ⇒ bool" where
"sbSameEq sb1 sb2 ≡ ∀c∈chDom TYPE('cs1) ∩ chDom TYPE('cs2).
                          sb1 \<^enum> (Abs c) = sb2 \<^enum> (Abs c) "


lemma "(spsComplete sps1) ⊗ (spsComplete sps2) ⊆
        spsComplete (sps1 ⊗ sps2)"
proof(rule spscomplete_belowI)
  fix spf sb
  assume as: "spf∈(spsComplete sps1) ⊗ (spsComplete sps2)"
(*
  obtain spf1 spf2 where "spf = spf1⊗\<^sub>⋆spf2"
      and spf1_def: "spf1 ∈(spsComplete sps1)"
      and spf2_def: "spf2 ∈(spsComplete sps2)"
    using as by(auto simp add: spsComp_def spscomplete_set)

  obtain spf1' where spf1'_eq: "spf1'·(sb⊎\<^sub>⋆(spf·sb)) = spf1·(sb⊎\<^sub>⋆(spf·sb))" and "spf1'∈sps1"
    using spf1_def apply(auto simp add: spscomplete_set) by metis
  obtain spf2' where spf2'_eq: "spf2'·(sb⊎\<^sub>⋆(spf·sb)) = spf2·(sb⊎\<^sub>⋆(spf·sb))" and "spf2'∈sps2"
    using spf2_def apply(auto simp add: spscomplete_set) by metis
```

```
  have "(spf1' ⊗\<^sub>⋆ spf2') ∈ (sps1 ⊗\<^sub>⋆ sps2)"
      using spsComp_def ⟨spf1'∈sps1⟩ ⟨spf2'∈sps2⟩ by blast
    moreover have "(spf1' ⊗\<^sub>⋆ spf2')·sb = spf·sb"⋆)
      oops
(⋆
      apply(rule spfcomp_eqI)
       apply((subst spf1'_eq | subst spf2'_eq), simp add: ⟨spf = spf1⊗\<^sub>⋆spf2⟩)
      apply(subst spfcomp_l, simp)
  ultimately show "∃spf2∈sps1 ⊗\<^sub>⋆ sps2. spf·sb = spf2·sb"
      by (metis)
qed
⋆)


theorem spscomp_refinement:
fixes F::"('I1^Ω→ 'O1^Ω) set"
  and G::"('I2^Ω→ 'O2^Ω) set"
  and F_ref::"('I1^Ω→ 'O1^Ω) set"
  and G_ref::"('I2^Ω→ 'O2^Ω) set"
assumes "F_ref ⊆ F"
    and "G_ref ⊆ G"
shows "(F_ref ⊗ G_ref) ⊆ (F ⊗ G)"
  apply(simp add: spsComp_def)
  using assms by blast


text⟨This important property enables independent modification of the modules
while preserving properties of the overall system. As long as the
modification is a refinement, it does not influence the other components.
The resulting component ⟨F_ref⟩ can simply replace ⟨F⟩ in the composed
system. Since the result is a refinement, the correctness is still
proven.⟩


text ⟨Properties of the original system ⟨S⟩ directly hold
 for the refined version ⟨S'⟩:⟩
theorem assumes "∀f∈S. P f" and "S' ⊆ S"
  shows "∀f'∈S'. P f'"
  using assms(1) assms(2) by auto


definition spsIsConsistent :: "('I1^Ω→ 'O1^Ω) set ⇒ bool" where
"spsIsConsistent sps ≡ (sps ≠ {})"


theorem spscomp_consistent:
 fixes F::"('I1^Ω→ 'O1^Ω) set"
   and G::"('I2^Ω→ 'O2^Ω) set"
 assumes "spsIsConsistent F"
     and "spsIsConsistent G"
   shows "spsIsConsistent (F ⊗ G)"
proof -
  have f1: "G ≠ {}"
    using assms(2) spsIsConsistent_def by blast
  have "F ≠ {}"
    by (metis assms(1) spsIsConsistent_def)
  then have "{c ⊗ ca |c ca. c ∈ F ∧ ca ∈ G} ≠ {}"
    using f1 by blast
  then show ?thesis
    by (simp add: spsComp_def spsIsConsistent_def)
qed


theorem spscomp_subpred:
 fixes P::"'I1^Ω⇒ 'O1^Ω⇒ bool"
   and H::"'I2^Ω⇒ 'O2^Ω⇒ bool"
 assumes "chDom TYPE ('O1) ∩ chDom TYPE ('O2) = {}"
     and "∀spf∈S1. ∀sb. P sb (spf·sb)"
     and "∀spf∈S2. ∀sb. H sb (spf·sb)"
  shows  "S1 ⊗ S2 ⊆
          {g. ∀sb.
               let all = sb ⊎ g·sb in
                P (all⋆) (all⋆) ∧ H (all⋆) (all⋆)
          }"
  apply (auto simp add: spsComp_def Let_def)
  apply (simp add: spfcomp_extract_l)
  apply (simp add: assms spfcomp_extract_r)+
  done

lemma spscomp_predicate:
      fixes P::"'I1^Ω⇒ 'O1^Ω⇒ bool"
        and H::"'I2^Ω⇒ 'O2^Ω⇒ bool"
      assumes "chDom TYPE ('O1) ∩ chDom TYPE ('O2) = {}"
      shows  "{p . ∀sb. P sb (p·sb)} ⊗ {h . ∀sb. H sb (h·sb)} ⊆
             {g. ∀sb.
                   let all = sb ⊎ g·sb in
                     P (all⋆) (all⋆) ∧ H (all⋆) (all⋆)
```

```
          }"
  apply(rule spscomp_subpred)
  apply (simp_all add: assms)
  done

(* TODO: ähnliches Lemma mit spsIO *)


lemma spscomp_praedicate2:
      fixes P::"'I1^Ω ⇒ 'O1^Ω ⇒ bool"
        and H::"'I2^Ω ⇒ 'O2^Ω ⇒ bool"
      assumes "chDom TYPE ('O1) ∩ chDom TYPE ('O2) = {}"
      shows  "
            {g. ∀sb.
                  let all = sb ⊎ g·sb in
                    P (all*) (all*) ∧ H (all*) (all*)
            } ⊆ spsComp {p . ∀sb. P sb (p·sb)}
                        {h . ∀sb. H sb (h·sb)}" (is "?LHS ⊆ ?RHS")
  oops
(*
proof
  fix g
  assume "g∈?LHS"
  hence "⋀sb. P ((sb⊎g·sb)*) ((sb⊎g·sb)*)"
    by (metis (mono_tags, lifting) mem_Collect_eq)
  have "∃p h. p⊗h = g" oops*)
(*  from this obtain p h where "p⊗h = g" by auto
  have "⋀sb. P (sb) (p·sb)" oops *)
(*  show "g∈?RHS" oops *)

(* Gegenbeispiel ... soweit ich sehe:
    P = H = "ist schwachkausal"
    bleibt nicht unter der feedbackkomposition erhalten *)


section ⟨Special Composition of SPSs⟩


definition spsCompSeq  :: "('In^Ω → 'Intern^Ω) set ⇒ ('Intern^Ω → 'Out^Ω) set
    ⇒ ('In^Ω → 'Out^Ω) set" where
"spsCompSeq sps1 sps2 = {spfCompSeq·spf1·spf2 | spf1 spf2.
                                spf1 ∈ sps1 ∧ spf2 ∈ sps2}"


theorem spscfcomp_set:
  assumes "spf1 ∈ sps1"
      and "spf2 ∈ sps2"
  shows "spfCompSeq·spf1·spf2 ∈ spsCompSeq sps1 sps2"
  apply(simp add: spsCompSeq_def)
  using assms(1) assms(2) by auto


theorem spscfcomp_consistent:
    assumes "spsIsConsistent sps1"
        and "spsIsConsistent sps2"
      shows "spsIsConsistent (spsCompSeq sps1 sps2)"
  apply(simp add: spsIsConsistent_def spsCompSeq_def)
  using assms spsIsConsistent_def by (metis neq_emptyD)


theorem spscfcomp_mono: assumes "sps1_ref ⊆ sps1"
  and "sps2_ref ⊆ sps2"
  shows "(spsCompSeq sps1_ref sps2_ref) ⊆ (spsCompSeq sps1 sps2)"
  apply(simp add: spsCompSeq_def)
  using assms by blast



definition spsCompPar  :: "('In1^Ω → 'Out1^Ω) set ⇒ ('In2^Ω → 'Out2^Ω) set ⇒
  (('In1 ∪ 'In2)^Ω → ('Out1 ∪ 'Out2)^Ω) set" where
"spsCompPar sps1 sps2 = {spfCompPar·spf1·spf2 | spf1 spf2.
                                spf1 ∈ sps1 ∧ spf2 ∈ sps2}"

definition spsCompFeed  :: "('In^Ω → 'Out^Ω) set ⇒
  (('In-'Out)^Ω → 'Out^Ω) set" where
"spsCompFeed sps = {spfCompFeed·spf | spf. spf ∈ sps}"
end
```

# Appendix F

# Case Study

```
(*<*)(*:maxLineLen=68:*)
theory Datatypes

imports inc.Prelude

begin

default_sort type
(*>*)

paragraph‹Channel and Message Datatypes› text‹\label{sub:fdata}›

text‹The case-study consists of three channels. They are named
 ‹cA› ‹cVcurr› and ‹cVprev›.›

datatype channel = cA | cVcurr | cVprev |  cempty


text‹Furthermore, the channel ‹cempty› is added to the datatype,
because there must always be a channel on which nothing can be
transmitted (see \cref{sec:data}).›


text‹The messages are all natural numbers. Hence ‹M› does not
  have to be a new datatype, instead it is set to ‹nat›.›
type_synonym M = nat

text‹Channel ‹cempty› may not contain a message. For every other channel
  every ‹nat›-message can be sent. The definition ‹UNIV› is the set containing all
  ‹nat›-values.›
fun ctype :: "channel ⇒ M set" where
"ctype cempty  = {}" |
"ctype   _     = UNIV"


text‹As always, a theorem that confirms the existence of an empty
channel has to be provided for the framework theories.›
theorem ctypeempty_ex: "∃c. ctype c = {}"
  using ctype.simps(1) by blast
(*<*)
end
(*>*)
```

```
(*<*)
theory Add

imports spf.SPFcomp

begin


(* Could also be in core *)
declare Abs_sb_inverse [simp add]
declare rep_reduction [simp add]
(*>*)

lemma cempty_eq [simp]: "cEmpty = {cempty}"
  unfolding cEmpty_def
  using ctype.elims by force

lemma ctype_univ[simp]: "chDom TYPE('cs) ≠ {}⟹ctype (Rep (c::'cs)) = UNIV"
  apply(subgoal_tac "⋀c::'cs. (Rep c) ≠ cempty")
   apply (meson ctype.elims)
  using cnotempty_rule by auto

(* wellformedness is not a problem here *)
lemma cruiseWell[simp]: fixes f::"'cs ⇒ M stream"
  assumes "chDom TYPE('cs) ≠ {}"
  shows  "sb_well f"
  apply(subgoal_tac "⋀c::'cs. ctype (Rep c) = UNIV")
  unfolding sb_well_def apply simp
  apply(subgoal_tac "⋀c::'cs. (Rep c) ≠ cempty")
  apply (meson ctype.elims)
  using cnotempty_rule using assms by auto

lemma cruiseSbeWell[simp]: fixes f::"('cs ⇒ M)"
  assumes "chDom TYPE('cs) ≠ {}"
  shows  "sbElem_well (Some f)"
  by(auto simp add: assms)

text ‹Now we are going to define the signature of the components. The ‹Add› component
  has the signature ‹{cA,cVprev}^Ω → {cVcurr}^Ω›. The ‹Prefix0› component has the signature
  ‹{cVcurr}^Ω → {cVprev}^Ω›. For each of theses sets we create a new type. Since
   ‹{cVcurr}^Ω› is both the output of ‹Add› and the input of ‹Prefix0› there are only
  three definitions.›

typedef addIn = "{cVprev, cA}"
  by auto

typedef addOut = "{cVcurr}"      \<comment> ‹also ‹prefixIn››
  by auto

typedef prefixOut = "{cVprev}"
  by auto

text ‹To use the datatypes to define bundles, they have to be instantiated in the
  ‹chan› class:›
instantiation addIn::chan
begin
  definition Rep_addIn_def: "Rep = Rep_addIn"

  lemma repadd_range[simp]: "range (Rep::addIn ⇒ channel)
                           = {cVprev, cA}"
  apply(subst Rep_addIn_def)
  using type_definition.Rep_range type_definition_addIn by fastforce

  instance %invisible
    apply (intro_classes)
    apply clarsimp
    unfolding Rep_addIn_def by (meson Rep_addIn_inject injI)
end
text‹As mentioned in \cref{sec:data}, each of the types need a representation function ‹Rep›.›
lemma [simp]: "chDom TYPE(addIn) = {cVprev, cA}"
  unfolding chDom_def
  by auto


instantiation addOut::chan
begin
  definition Rep_addOut_def: "Rep = Rep_addOut"

  lemma repaddout_range[simp]: "range (Rep::addOut ⇒ channel)
                           = {cVcurr}"
  apply(subst Rep_addOut_def)
  using type_definition.Rep_range type_definition_addOut by fastforce

  instance %invisible
    apply (intro_classes)
    apply clarsimp
    unfolding Rep_addOut_def by (meson Rep_addOut_inject injI)
end

lemma [simp]: "chDom TYPE(addOut) = {cVcurr}"
  unfolding chDom_def
  by auto

text‹By
```

```
using typedef to define the domain types over channels, a representation function is provided and
can be used.⟩

instantiation prefixOut::chan
begin
  definition Rep_prefixOut_def: "Rep = Rep_prefixOut"

  lemma repprefixout_range[simp]: "range (Rep::prefixOut ⇒ channel)
                               = {cVprev}"
  apply(subst Rep_prefixOut_def)
  using type_definition.Rep_range type_definition_prefixOut by fastforce

  instance %invisible
    apply (intro_classes)
    apply clarsimp
    unfolding Rep_prefixOut_def by (meson Rep_prefixOut_inject injI)
end

lemma [simp]: "chDom TYPE(prefixOut) = {cVprev}"
  unfolding chDom_def
  by auto



lemma addout_one[simp]: "(c::addOut) = Abs (cVcurr)"
  apply(cases c, auto)
  by (metis Abs_addOut_inverse Rep_addOut_def abs_rep_id singletonI)

lemma prev_one[simp]: "(c::prefixOut) = Abs (cVprev)"
  apply(cases c, auto)
  using Rep_prefixOut_def Rep_prefixOut_inverse range_eq_singletonD repprefixout_range by fastforce


paragraph⟨Prefix component⟩
text⟨The prefix component is essentially a identity component
with an additional initial output. The
identity component with the signature ⟨addOut^Ω → prefixOut^Ω⟩
is definable by renaming the channel
of the input \gls{sb} (⟨cVcurr⟩) to the channel the output \gls{sb} (⟨cVprev⟩).⟩

definition prefixRename :: "addOut^Ω → prefixOut^Ω" where
"prefixRename = sbRename_part [Abs cVcurr ↦ Abs cVprev]"

text⟨Correct behavior is proven in the following theorem, the output stream is equal to the input
stream.⟩

theorem prefrename_getch:
  "prefixRename·sb \<^enum> (Abs cVprev) = sb \<^enum> (Abs cVcurr)"
  unfolding prefixRename_def
  apply(subst sbrenamepart_getch_in)
  apply auto
     apply (metis prev_one)+
  apply (metis Rep_addOut_def Rep_addOut_inject empty_iff insert_iff repaddout_range repinrange)+
  done

text⟨Because one initial output element is needed for the prefix component, the initial output
can be represented by a \gls{sbe}. Thus, a lifting function from natural numbers to an output
\gls{sbe} is defined.⟩

lift_definition initOutput:: "nat ⇒ prefixOut^√" is
"λinit. Some (λ_. init)"
  by simp

text⟨By appending the initial output \gls{sbe} to an output \gls{sb} of the identity component, the
complete output of the prefix component can be defined.⟩

definition prefixPrefix:: "M ⇒ prefixOut^Ω → prefixOut^Ω" where
"prefixPrefix init = sbECons (initOutput init)"

text⟨Therefore, the prefix component is defined by a sequential composition of the identity component
@{const prefixRename} and the appending component @{const prefixPrefix} with an inital output.⟩

definition prefixComp'::"nat ⇒ addOut^Ω → prefixOut^Ω" where
"prefixComp' init = spfCompSeq· prefixRename· (prefixPrefix init)"


text⟨The same prefix component can also be defined in a more direct manner by outputting a stream
that starts with an initial output and then outputs the input stream from the input \gls{sb}.⟩

lift_definition prefixComp::"nat ⇒ addOut^Ω → prefixOut^Ω" is
"λinit sb. Abs_sb (λ_. ↑init • sb \<^enum> (Abs cVcurr))"
  apply(intro cont2cont)
  by(rule cruiseWell, simp)

lemma prefix_getch: "prefixComp init·(sb)  \<^enum>  Abs cVprev = ↑init • sb \<^enum> (Abs cVcurr)"
  by(simp add: prefixComp.rep_eq sbgetch_insert)



lemma [simp]: "sbe2sb (initOutput init)  \<^enum>  c = ↑init"
```

```
  by(simp add: sbgetch_insert2 sbe2sb.rep_eq initOutput.rep_eq)


text⟨Both definitions model the same component. This
is proven in the following theorem:⟩

theorem "prefixComp init = prefixComp' init"
  apply(rule cfun_eqI, rule sb_eqI, subst sbgetch_insert2)
  apply(simp add: prefixComp.rep_eq spfCompSeq_def prefixComp'_def sbECons_def prefixPrefix_def)
  using prefrename_getch
  by (metis prev_one)

text ⟨In the following, @{const prefixComp} is used to define the
complete system.⟩




paragraph⟨Add component⟩
text⟨The add component is defined by using an element-wise add function for streams and applying it
to both input streams.⟩


lift_definition addComp::"addIn^Ω → addOut^Ω" is
"λsb. Abs_sb (λ_. add·(sb \<^enum> Abs cA)·(sb \<^enum> Abs cVprev))"
  by(intro cont2cont, simp)

text ⟨The output on channel ⟨cVcurr⟩ follows directly:⟩
theorem addcomp_getch:
  "addComp·sb \<^enum> (Abs cVcurr) = add·(sb \<^enum> Abs cA)·(sb \<^enum> Abs cVprev)"
  by(simp add: addComp.rep_eq sbgetch_insert2)

text ⟨The length of the output is the minimal length of the two input streams.⟩
theorem add_len:"#(addComp·sb) = min (#(sb \<^enum> Abs cA)) (#(sb \<^enum> Abs cVprev))"
proof -
  have "#(addComp·sb) = #((addComp·sb) \<^enum> Abs cVcurr)"
    apply(auto simp add: len_sb_def sbLen_def)
    apply(rule LeastI2_wellorder_ex, auto)
    by (metis addout_one)
  thus ?thesis
    by (simp add: addcomp_getch add_def)
qed

lemma [simp]: "c∈{cVprev, cA}⟹Abs_addIn c = Abs c"
  by (metis Abs_addIn_inverse Rep_addIn_def abs_rep_id)

text ⟨Since the lenght over bundles is defined as the minimum, the property can be
simplified:⟩
theorem "#(addComp·sb) = #sb"
  apply(simp add: add_len)
  apply(rule sblen_rule[symmetric], auto)
  apply(case_tac "c", auto)
  by (metis min_def)

(*<*)
declare addcomp_getch [simp]
(*>*)

paragraph⟨Acc2vel component⟩
text⟨The composed components behavior is definable by
outputting the addition of the input element
and the previous output element (or 0 for the initial input element).⟩

definition streamSum::"nat stream → nat stream" where
"streamSum ≡ sscanl (+) 0"

lemma sscanl_unfold: "sscanl (+) n·s = add·s·(↑n • sscanl (+) n·s)"
  apply(induction s arbitrary: n rule: ind)
  by (simp add: add.commute add_unfold)+

text ⟨Unfolding the definition once leads to the following recursive equation:⟩
theorem "streamSum·s = add·s·(↑0 • streamSum·s)"
  apply(simp add: streamSum_def)
  using sscanl_unfold by auto

lemma getch_nomag: fixes sb::"'cs1^Ω"
  assumes "c∈chDom TYPE('cs2)"
      and "c∈chDom TYPE('cs1)"
    shows  "sb \<^enum>\<^sub>* ((Abs c)::'cs2) = sb \<^enum> (Abs c)"
  apply(auto simp add: sbGetCh.rep_eq assms)
  done

text ⟨For the composed system unfolding leads to a similar result:⟩
theorem comp_unfold: "((addComp ⊗ (prefixComp init))·sb) \<^enum> Abs cVcurr
    = add·(sb  \<^enum>  Abs cA)·
         (↑init • (addComp ⊗ prefixComp init)·sb  \<^enum>  Abs cVcurr)"
  apply(subst spfcomp_unfold, auto)
  apply(subst getch_nomag, auto)

  apply(subst getch_nomag, auto)
  apply(subst (2) getch_nomag, auto)

  apply(subst spfcomp_unfold, auto)
```

```
  apply(subst (2) getch_nomag, auto)

  apply(subst prefix_getch, auto)
  apply(subst (2)getch_nomag, auto)
  done

text ⟨While the recursive equations are nearly identical, equality
  does not directly follow from it since there might be multiple fixpoints which fulfill
  the recursive equation.⟩
lemma "#(add·s1·s2) = min (#s1) (#s2)"
  by(simp add: add_def)

lemma add_slen [simp]: "#(add·x·y) = min (#x) (#y)"
  apply(simp add: add_def)
  done

lemma smap_srt[simp]: "srt·(smap f·s) = smap f· (srt·s)"
  by (metis sdrop_0 sdrop_forw_rt sdrop_smap)

lemma szip_srt[simp]: "srt·(szip·a·s) = szip·(srt·a)·(srt·s)"
  by (metis (no_types, hide_lams) sdrop_0 sdrop_forw_rt szip_sdrop)

lemma rek2sscanl_h: assumes "Fin n <#s"
  and "⋀input init. z init·input = add·input·(↑init • z init·input)"
shows "snth n (z init·s) = snth n (sscanl (+) init·s)"
using assms(1) proof (induction n arbitrary: s init)
  case 0
  then show ?case
    by (metis add_commutative add_unfold assms(2) empty_is_shortest shd1 snth_shd sscanl_shd surj_scons)
next
  case (Suc n)
  have "#(z init·s) = #s"
    by (metis add_slen assms(2) min_rek slen_scons)
  have "snth (Suc n) (z init·s) = snth n (srt·(z init·s))"
    by (simp add: snth_rt)

  then show ?case apply(subst assms(2), simp add: add_def snth_rt sscanl_srt)
    apply(subst smap_snth_lemma, simp)
     apply (metis Suc.prems ⟨#((z::nat ⇒ nat stream → nat stream) (init::nat)· (s::nat stream)) = #s⟩
          convert_inductive_asm leD lel slen_rt_ile_eq)
    by (metis Suc.IH Suc.prems ⟨#((z::nat ⇒ nat stream → nat stream) (init::nat)· (s::nat stream)) = #s⟩
         add.commute convert_inductive_asm fst_conv not_less slen_rt_ile_eq snd_conv snth_rt sscanl_snth
         sscanl_srt szip_nth)
qed

text⟨Hence we prove that there is only one fixpoint for the equation.
  In the lemma ⟨rek2sscanl⟩ the variable ⟨z⟩ is an arbitrary fixpoint.
  The lemma shows that ⟨z⟩ is the only fixpoint and equivalent to ⟨sscanl⟩. ⟩

theorem rek2sscanl:
  assumes "⋀input init. z init·input = add·input·(↑init • z init·input)"
    shows "z init·s = sscanl (+) init·s"
  apply(rule snths_eq)
  apply (metis add_slen assms fair_sscanl min_rek slen_scons)
  by (metis add_slen assms min_rek rek2sscanl_h slen_scons)


(*<*)
(* Helper: *)
definition "createInput ≡ Λ input. (Abs_sb (λc. input))"

text⟨Composing both components, applying the resulting \gls{spf} to the created input \gls{sb} and
then obtaining the output stream is done by the following function. The input ⟨init⟩ sets the
initial output of the prefix component.⟩

definition comp where
"comp init = (Λ input . ((addComp ⊗ (prefixComp init))·(createInput·input)) \<^enum> (Abs cVcurr) )"

lemma comp2sscanl_h: "comp init·input = sscanl (+) init·input"
  apply(rule rek2sscanl)
  apply(subst comp_def, simp)
  apply(subst comp_unfold)
  apply(subst sbgetch_insert)
  apply(subst createInput_def)
  apply(subst beta_cfun, intro cont2cont; simp)
  by(simp add: comp_def)

lemma sb_eql2:
   fixes   sb1 sb2::"'cs^Ω"
  assumes "⋀c. chDom TYPE('cs) ≠{}⟹c∈Abs`chDom TYPE('cs)⟹sb1 \<^enum> c = sb2 \<^enum> c"
  shows   "sb1 = sb2"
  by (metis abs_rep_id assms image_eql sb_empty_eq sb_eql)

lemma creatinput_eq:
   fixes sb::"'cs^Ω"
  assumes "chDom TYPE ('cs) = {cA}"
  shows "sb = createInput·(sb \<^enum> (Abs cA))"
  apply(rule sb_eql2)
  apply(auto simp add: assms)
  apply(simp add: createInput_def)
  apply(subst beta_cfun, intro cont2cont)
  apply (simp add: assms)
  by (simp add: assms sbgetch_insert2)
```

```
lemma creatinput_eq2:
  fixes sb::"'cs^Ω"
  assumes "chDom TYPE ('cs) = {cA}"
  shows "((createInput·input)::'cs^Ω)\<^enum>(Abs cA) = input"
  apply(subst createInput_def)
  apply(subst beta_cfun, intro cont2cont)
  apply(simp add: comp_def assms)
  by (simp add: assms sbgetch_insert2)
(*>*)

text⟨Following from this statement, the composition of
the ⟨add⟩ and ⟨prefix⟩ component can be
evaluated. ⟩

theorem "(addComp ⊗ (prefixComp init))·sb \<^enum> Abs cVcurr
      = sscanl (+) init·(sb \<^enum> Abs cA)"
  apply(subst creatinput_eq [where sb="sb"], simp)
  apply(subst comp2sscanl_h[symmetric])
  using comp_def  apply auto
  done

lemma add_unfold1[simp]: "add·(↑x)·(↑y• ys) = ↑(x+y)"
  by(simp add: add_def)

lemma add_unfold2[simp]: "add·(↑x•xs)·(↑y) = ↑(x+y)"
  by(simp add: add_def)

text⟨The composition can also be tested over input streams.⟩

theorem
 "(addComp ⊗ prefixComp 0)·(Abs_sb (λc. <[1,1,1,0,0,2]>)) \<^enum> Abs cVcurr
   = <[1,2,3,3,3,5]>"
  apply(subst comp_unfold, subst sbgetch_insert2, simp add: add_unfold)+
  by (simp add: numeral_2_eq_2 numeral_3_eq_3)



paragraph⟨Non-Deterministic Component⟩


text⟨Now we define a non-deterministic component. In
this example the component randomly modifies the output. This is used
to model impreciseness of the actuator. The actuator is unable to exactly
follow the control-command from the ⟨Acc2val⟩ component, instead there
exists a delta. This is modeled in the following definition:⟩
definition realBehaviour::"nat ⇒ nat set" where
"realBehaviour n ≡ if n<50 then {n} else {n-5 .. n+5}"

text ⟨The actuator can perfectly execute the control command for
small values (⟨n<50⟩). There is only one reaction: ⟨{n}⟩. But for
greater input, there may exist an error. Here it is a delta of at most ⟨5⟩,
resulting in the possible outputs ⟨{n-5 .. n+5}⟩.⟩


(*<*) default_sort countable (*>*)

text ⟨Now the ⟨realBehaviour⟩ has to be applied to every element in the
stream. For this we create a general helper-function, similar to the
deterministic @{const smap}. ⟩
definition ndetsmap::"('a ⇒ 'b set)
            ⇒ ('a stream → 'b stream) set" where
"ndetsmap T = gfp (λH. {f | f. (f·ϵ=ϵ)
        ∧ (∀m s. ∃x g. (f·(↑m•s) = ↑x • g·s) ∧ x∈(T m) ∧ g∈H)})"

text ⟨The component is a set of stream processing functions. Each function
  returns ⟨ϵ⟩ on the input ⟨ϵ⟩. When the input starts with a message ⟨m⟩
  the output one of the possible values described in ⟨T⟩.
  The ⟨gfp⟩ operator returns the greatest fixpoint fulfilling the recursive
  equation.⟩

lemma monondetsmap[simp]:"mono (λH. {f | f.  (f·ϵ=ϵ) ∧ (∀m s. ∃x g.
                            (f·(↑m•s) = ↑x • g·s) ∧ x∈(T m) ∧ g∈H)})"
  apply(rule monoI)
  apply(simp add: prod.case_eq_if,auto)
  by (meson subset_iff)

lemma ndetsmap_unfold:"ndetsmap S = {f |f . f·ϵ = ϵ ∧
     (∀m s. ∃x g. f·(↑m • s) = ↑x • g·s ∧
                x ∈ S m ∧ g∈ (ndetsmap S))}"
  unfolding ndetsmap_def
  apply(subst gfp_unfold)
  using monondetsmap by auto

lemma ndetsmap_strict[simp]:"spf ∈ ndetsmap S⟹spf·ϵ=ϵ"
  using ndetsmap_unfold[of S] by auto

lemma ndetsmap_elem: assumes "spf1 ∈ ndetsmap T"
  shows "∃out∈(T m). spf1·(↑m) = ↑out"
  apply (insert assms)
  using ndetsmap_unfold[of T] apply auto
  by (smt assms mem_Collect_eq ndetsmap_strict sconc_snd_empty)
```

```
lemma ndetsmap_step: assumes "spf1 ∈ ndetsmap T"
  shows "∃spf2∈(ndetsmap T). ∃out∈(T m). spf1·(↑m•s) = ↑out • spf2·s"
  apply (insert assms)
  using ndetsmap_unfold[of T] apply auto
  by fastforce

lemma ndetsmap_srt: assumes "spf1 ∈ ndetsmap T"
  shows "∃spf2∈(ndetsmap T). srt·(spf1·(↑m •s)) = spf2·s"
proof-
  obtain spf2 out where spf2_def:"spf1·(↑m •s) = ↑out • spf2·s"
    using ndetsmap_step assms by blast
  then show ?thesis
    by (metis assms inject_scons ndetsmap_step stream.sel_rews(2)
        strictI surj_scons)
qed

(*<*)
lemmas streamind[case_names adm bottom step,
                 induct type: stream] = ind

lemmas streamcases [case_names bottom step,
                   cases type: stream] = scases
(*>*)
lemma ndetsmaplen[simp]:
  assumes "spf ∈ ndetsmap A"
  shows "#(spf·s) = #s"
using assms
proof(induction s arbitrary: spf)
  case adm
  then show ?case
    by(simp add: len_stream_def)
next
  case bottom
  then show ?case
    using assms ndetsmap_unfold by auto
next
  case (step a s)
  then show ?case
    by (smt mem_Collect_eq ndetsmap_unfold slen_scons step.IH
        step.prems)
qed


lemma ndetsmap_snth[simp]:
  assumes "spf ∈ ndetsmap A"
  and "Fin n<#s"
  shows "snth n (spf·s) ∈ (A (snth n s))"
  using assms
proof(induction n arbitrary: spf s A)
  case 0
  then show ?case
    apply(cases s,auto)
    unfolding ndetsmap_def
    by (metis (no_types, lifting) "0.prems"(1) ndetsmap_step shd1)
next
  case (Suc n)
  then show ?case
    apply(cases s,auto)
    apply(simp add: snth_rt)
    using ndetsmap_srt by metis
qed

lemma nnndetsmap_rule[simp]:(*subset rule without dealing with gfp*)
  assumes "S={spf | spf. ∀n s. (Fin n < #s⟶
        snth n (spf·s) ∈ (A (snth n s))) ∧ #(spf·s) = #s ∧ spf·ε=ε}"
  shows "S ⊆ ndetsmap A"
  apply(subst ndetsmap_def)
  apply(rule gfp_ordinal_induct)
  using monondetsmap apply blast
  apply(auto simp add: assms)
proof-
  fix S::"('a stream → 'b stream) set"
  and x::"'a stream → 'b stream" and m::'a and s::"'a stream"
  assume a1:"{uu. ∀n s. (Fin n < #s⟶
        snth n (uu·s) ∈ A (snth n s)) ∧ #(uu·s) = #s ∧ uu·ε = ε} ⊆ S"
  assume a2:"gfp (λH. {uu. uu·ε = ε ∧
       (∀m s. ∃x g. uu·(↑m • s) = ↑x • g·s ∧ x ∈ A m ∧ g ∈ H)}) ⊆ S"
  assume a3:"∀n s. (Fin n < #s⟶
        snth n (x·s) ∈ A (snth n s)) ∧ #(x·s) = #s ∧ x·ε = ε"
  then have h0:"⋀s. #(x·(↑m • s)) = lnsuc·(#s)"
    by simp
  have h2:"∃out. x·(↑m) = ↑out"
    apply(rule_tac x="shd (x·(↑m))" in exI)
    by (simp add: a3 snths_eq)
  have x:"⋀n s m. Fin n < #(↑m • s)⟹
          snth n (x·(↑m • s)) ∈ A (snth n (↑m • s))"
    using a3 by auto
  then have h1:"⋀m s. shd (x·(↑m • s)) ∈ A m"
    using snth_shd
    by (metis Fin_02bot a3 gr_0 lnzero_def shd1 slen_scons)
  have h3':"⋀n s. Fin n < #(srt·s)
          ⟹snth n (srt·(x·s)) ∈ A (snth (Suc n) s)"
    apply (simp add: snth_rt)
  proof-
```

```
    fix n::nat and s::"'a stream"
    assume a1:"Fin n < #(srt·s)"
    obtain out where out_def:"x·s = out"
      by simp
    then have"⋀n. Fin n < #out⟹snth n out ∈ A (snth n s)"
      using a3 by auto
    then have "snth (Suc n) out ∈ A (snth (Suc n) s)"
      by (metis a1 a3 dual_order.strict_implies_order less2eq
           linear neq_iff out_def slen_rt_ile_eq)
    then show "snth n (srt·(x·s)) ∈ A (snth n (srt·s))"
      by(simp add: out_def snth_rt)
  qed
  then have h3:"⋀n s. Fin n < #s⟹
                snth (Suc n) (x·(↑m ● s)) ∈ A (snth n s)"
    by (metis (no_types, lifting) a3 empty_is_shortest h0 lnat.sel_rews(2) snth_rt snth_scons
         srt_decrements_length strictI)
  then have h3:"⋀n s. Fin n < #s⟹
                snth n (srt·(x·(↑m ● s))) ∈ A (snth n s)"
    by (simp add: snth_rt)
  show"∃xa g. x·(↑m ● s) = ↑xa ● g·s ∧ xa ∈ A m ∧ g ∈ S"
    apply(rule_tac x="shd (x·(↑m ● s))" in exI)
    apply(rule_tac x="(Λ s. srt·(x·(↑m●s)))" in exI,auto)
    apply (metis a3 empty_is_shortest slen_scons snths_eq
         stream.sel_rews(2) strictI surj_scons)
    using h1 apply auto[1]
    apply(subgoal_tac "(Λ s. srt·(x·(↑m ● s))) ∈
         {uu. ∀n s. (Fin n < #s⟶
         snth n (uu·s)) ∈ A (snth n s)) ∧ #(uu·s) = #s ∧ uu·ε = ε}")
    using a1 apply blast
    apply auto
    using h3 h0 h2 apply auto
    by (metis a3 empty_is_shortest lnat.sel_rews(2) snths_eq srt_decrements_length stream.sel_rews(2) strictI)
qed

lemma spfinndetsmap[simp]:(*Nice Lemma*)
  assumes "⋀n s. Fin n < #s⟹snth n (spf·s) ∈ (T (snth n s))"
  and "⋀s. #(spf·s) = #s"
  shows "spf∈ ndetsmap T"
  apply(subgoal_tac "{spf} ⊆ {spf |spf. ∀n s.
  (Fin n < #s⟶snth n (spf·s) ∈ T (snth n s)) ∧ #(spf·s) = #s ∧
       spf·ε = ε}")
  using nnndetsmap_rule
  apply (metis (mono_tags, lifting) insert_subset subset_iff)
  by (auto simp add: assms snths_eq)

lemma ndetsmap2smap: (*Reduce ndet automaton to det automaton*)
  assumes"⋀m. is_singleton (T m)"
  and "spf ∈ ndetsmap T"
  shows "spf = smap (λe. SOME x. x∈ T e)"
  apply(rule cfun_eqI)
  apply(rule snths_eq,simp)
  using assms(2) ndetsmaplen apply blast
  apply auto
proof-
  fix x::"'a stream" and n::nat
  assume a1:"Fin n < #(spf·x)"
  then obtain out where out_def: "{out} = T (snth n x)"
    using assms(1)
    by (metis is_singleton_the_elem)
  then have "snth n (spf·x) = out"
    using a1 assms(2) ndetsmaplen by auto
  moreover have"snth n (smap (λe::'a. SOME x::'b. x ∈ T e)·x) = out"
    by (metis a1 all_not_in_conv assms(2) ndetsmaplen out_def
       singleton_iff smap_snth_lemma some_in_eq)
  ultimately show "snth n (spf·x) =
                  snth n (smap (λe::'a. SOME x::'b. x ∈ T e)·x)"
    by simp
qed

lemma ndetsmap_svalue[simp]:
  assumes "spf ∈ ndetsmap A"
  shows "sValues·(spf·s) ⊆⋃ (A ` (sValues·s))"
  using assms
proof(induction s arbitrary: spf)
  case adm
  then show ?case
    apply(rule adm_all, rule adm_imp,auto)
    apply(rule admI,auto)
    apply(simp add: contlub_cfun_arg)
    apply(simp add: lub_eq_Union)
    by fastforce
next
  case bottom
  then show ?case
    by(simp)
next
  case (step a s)
  then show ?case
    apply simp
    using ndetsmap_step[of spf A a s] apply auto
    by blast
qed
```

```
(*<*) default_sort chan (*>*)


(*<*)
definition randomShift::"(addOut^Ω → addOut^Ω) set" where
"randomShift = {Λ sb. Abs_sb (λ_. f·(sb \<^enum> Abs cVcurr)) | f .
    f ∈ ndetsmap realBehaviour}"

(*>*)

  text ⟨The two functions are combined to create the final component:⟩
definition errorActuator::"(nat stream → nat stream) set" where
"errorActuator = ndetsmap realBehaviour"

text ⟨The component is consistent, there exists a function which
  is in the description. For example the identity function (⟨ID⟩).⟩
theorem "ID ∈ errorActuator"
  unfolding errorActuator_def
  apply(rule spfinndetsmap, auto)
  by(auto simp add: realBehaviour_def)

text ⟨The length is not modified by ⟨errorActuator⟩: ⟩
theorem error_len:
  assumes "spf ∈ errorActuator"
  shows "#(spf·s) = #s"
  using assms errorActuator_def ndetsmaplen by blast

text ⟨If the input consists \emph{only} of values less than 50 there is no error.
  The actuator perfectly follows the commands.⟩
theorem assumes "⋀n. n∈sValues·s⟹n<50"
          and "spf ∈ errorActuator"
  shows "spf·s = s"
  apply(rule snths_eq)
  using assms(2) errorActuator_def ndetsmaplen apply blast
  apply auto
proof -
  fix n
  assume "Fin n < #(spf·s)"
  hence "Fin n < #s"
    by (simp add: assms(2) error_len)
  hence "snth n s ∈ sValues·s"
    by (simp add: snth2sValues)
  hence "snth n s < 50"
    by (simp add: assms(1))
  moreover have "snth n (spf·s) ∈ (realBehaviour (snth n s))"
    using ⟨Fin (n::nat) < #(s::nat stream)⟩ assms(2) errorActuator_def ndetsmap_snth by blast
  ultimately show "snth n (spf·s) = snth n s" by(simp add: realBehaviour_def)
qed

text ⟨If the input is larger than 50, errors can occur. Here an example for
  the input with an infinite repetition of ⟨n⟩. The output is non-deterministic.
  But the values must lie between ⟨{n-5 .. n+5}⟩.⟩
theorem assumes "50 ≤ n"
          and "spf ∈ errorActuator"
        shows "sValues·(spf·(sinftimes (↑n))) ⊆ {n-5 .. n+5}"
proof -
  have "sValues·(sinftimes (↑n)) = {n}" by simp
  hence "sValues·(spf·(sinftimes (↑n))) ⊆⋃ (realBehaviour ` {n})"
    using assms(2) errorActuator_def ndetsmap_svalue by fastforce
  thus ?thesis by (auto simp add: realBehaviour_def)
qed


(*<*)
end
(*>*)
```

## Aachener Informatik-Berichte

**This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:**

http://aib.informatik.rwth-aachen.de/

**To obtain copies please consult the above URL or send your request to:**

**Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,**
**Email:** biblio@informatik.rwth-aachen.de

| | |
|---|---|
| 2016-01 * | Fachgruppe Informatik: Annual Report 2016 |
| 2016-02 | Ibtissem Ben Makhlouf: Comparative Evaluation and Improvement of Computational Approaches to Reachability Analysis of Linear Hybrid Systems |
| 2016-03 | Florian Frohn, Matthias Naaf, Jera Hensel, Marc Brockschmidt, and Jürgen Giesl: Lower Runtime Bounds for Integer Programs |
| 2016-04 | Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder: Proving Termination of Programs with Bitvector Arithmetic by Symbolic Execution |
| 2016-05 | Mathias Pelka, Grigori Goronzy, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 2nd KuVS Expert Talk on Localization |
| 2016-06 | Martin Henze, René Hummen, Roman Matzutt, Klaus Wehrle: The SensorCloud Protocol: Securely Outsourcing Sensor Data to the Cloud |
| 2016-07 | Sebastian Biallas : Verification of Programmable Logic Controller Code using Model Checking and Static Analysis |
| 2016-08 | Klaus Leppkes, Johannes Lotz, and Uwe Naumann: Derivative Code by Overloading in C++ (dco/c++): Introduction and Summary of Features |
| 2016-09 | Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, Peter Schneider-Kamp, and Cornelius Aschermann: Automatically Proving Termination and Memory Safety for Programs with Pointer Arithmetic |
| 2016-10 | Stefan Wüller, Ulrike Meyer, and Susanne Wetzel: Towards Privacy-Preserving Multi-Party Bartering |
| 2017-01 * | Fachgruppe Informatik: Annual Report 2017 |
| 2017-02 | Florian Frohn and Jürgen Giesl: Analyzing Runtime Complexity via Innermost Runtime Complexity |
| 2017-04 | Florian Frohn and Jürgen Giesl: Complexity Analysis for Java with AProVE |
| 2017-05 | Matthias Naaf, Florian Frohn, Marc Brockschmidt, Carsten Fuhs, and Jürgen Giesl: Complexity Analysis for Term Rewriting by Integer Transition Systems |

2017-06 Oliver Kautz, Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe: CD2Alloy: A Translation of Class Diagrams to Alloy

2017-07 Klaus Leppkes, Johannes Lotz, Uwe Naumann, and Jacques du Toit: Meta Adjoint Programming in C++

2017-08 Thomas Gerlitz: Incremental Integration and Static Analysis of Model-Based Automotive Software Artifacts

2017-09 Muhammad Hamad Alizai, Jan Beutel, Jó Ágila Bitsch, Olaf Landsiedel, Luca Mottola, Przemyslaw Pawelczak, Klaus Wehrle, and Kasim Sinan Yildirim: Proc. IDEA League Doctoral School on Transiently Powered Computing

2018-01 * Fachgruppe Informatik: Annual Report 2018

2018-02 Jens Deussen, Viktor Mosenkis, and Uwe Naumann: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen

2018-03 Igor Kalkov: A Real-time Capable, Open-Source-based Platform for Off-the-Shelf Embedded Devices

2018-04 Andreas Ganser: Operation-Based Model Recommenders

2018-05 Matthias Terber: Real-World Deployment and Evaluation of Synchronous Programming in Reactive Embedded Systems

2018-06 Christian Hensel: The Probabilistic Model Checker Storm - Symbolic Methods for Probabilistic Model Checking

2019-01 * Fachgruppe Informatik: Annual Report 2019

2019-02 Tim Felix Lange: IC3 Software Model Checking

2019-03 Sebastian Patrick Grobosch: Formale Methoden für die Entwicklung von eingebetteter Software in kleinen und mittleren Unternehmen

2019-05 Florian Göbe: Runtime Supervision of PLC Programs Using Discrete-Event Systems

* These reports are only available as a printed version.
Please contact biblio@informatik.rwth-aachen.de

to obtain copies.

# Related Interesting Work from the SE Group, RWTH Aachen

The following section gives an overview on related work done at the SE Group, RWTH Aachen. More details can be found on the website `www.se-rwth.de/topics/` or in [HMR$^+$19]. The work presented here mainly has been guided by our mission statement:

Our mission is to define, improve, and industrially apply *techniques*, *concepts*, and *methods* for *innovative and efficient development* of software and software-intensive systems, such that *high-quality* products can be developed in a *shorter period of time* and with *flexible integration* of *changing requirements*. Furthermore, we *demonstrate the applicability* of our results in various domains and potentially refine these results in a domain specific form.

## Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process.", [JWCR18] addresses the question how digital and organizational techniques help to cope with physical distance of developers and [RRSW17] addresses how to teach agile modeling. Modeling will increasingly be used in development projects, if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum16], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR$^+$06, GKR$^+$08, HR17] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR$^+$09] and refactoring in various modeling and programming languages [PR03]. In [FHR08] we describe a set of general requirements for model quality. Finally, [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG$^+$14] we discuss how to improve the reliability of adaptivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation.

## Artifacts in Complex Development Projects

Developing modern software solutions has become an increasingly complex and time consuming process. Managing the complexity, size, and number of the artifacts developed and used during a project together with their complex relationships is not trivial [BGRW17]. To keep track of relevant structures, artifacts, and their relations in order to be able e.g. to evolve or adapt models and their implementing code, the *artifact model* [GHR17] was introduced. [BGRW18] explains its applicability in systems engineering based on MDSE projects.

An artifact model basically is a meta-data structure that explains which kinds of artifacts, namely code files, models, requirements files, etc. exist and how these artifacts are related to each other. The artifact model therefore covers the wide range of human activities during the development down to fully automated, repeatable build scripts. The artifact model can be used to optimize parallelization during the development and building, but also to identify deviations of the real architecture and dependencies from the desired, idealistic architecture, for cost estimations, for requirements and bug tracing, etc. Results can be measured using metrics or visualized as graphs.

## Artificial Intelligence in Software Engineering

MontiAnna is a family of explicit domain specific languages for the concise description of the architecture of (1) a neural network, (2) its training, and (3) the training data [KNP+19]. We have developed a compositional technique to integrate neural networks into larger software architectures [KRRvW17] as standardized machine learning components [KPRS19]. This enables the compiler to support the systems engineer by automating the lifecycle of such components including multiple learning approaches such as supervised learning, reinforcement learning, or generative adversarial networks. According to [MRR11g] the semantic difference between two models are the elements contained in the semantics of the one model that are not elements in the semantics of the other model. A smart semantic differencing operator is an automatic procedure for computing diff witnesses for two given models. Smart semantic differencing operators have been defined for Activity Diagrams [MRR11a], Class Diagrams [MRR11d], Feature Models [DKMR19], Statecharts [DEKR19], and Message-Driven Component and Connector Architectures [BKRW17, BKRW19]. We also developed a modeling language-independent method for determining syntactic changes that are responsible for the existence of semantic differences [KR18].

We apply logic, knowledge representation and intelligent reasoning to software engineering to perform correctness proofs, execute symbolic tests or find counterexamples using a theorem prover. And we have applied it to challenges in intelligent flight control systems and assistance systems for air or road traffic management [KRRS19, HRR12] and based it on the core ideas of Broy's Focus theory [RR11, BR07]. Intelligent testing strategies have been applied to automotive software engineering [EJK+19, DGH+19, KMS+18], or more generally in systems engineering [DGH+18]. These methods are realized for a variant of SysML Activity Diagrams and Statecharts.

Machine Learning has been applied to the massive amount of observable data in energy management for buildings [FLP+11a, KLPR12] and city quarters [GLPR15] to optimize the operation efficiency and prevent unneeded CO2 emissions or reduce costs. This creates a structural and behavioral system theoretical view on cyber-physical systems understandable as essential parts of digital twins [RW18, BDH+20].

## Generative Software Engineering

The UML/P language family [Rum12, Rum11, Rum16] is a simplified and semantically sound derivate of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P based on the MontiCore language workbench [KRV10, GKR+06, GKR+08, HR17]. In [KRV06], we discuss additional roles necessary in a model-based software development project. [GKRS06, GHK+15b] discuss mechanisms to keep generated and handwritten code separated. In [Wei12], we demonstrate how to systematically derive a transformation language in concrete syntax. [HMSNRW16] presents how to generate extensible and statically type-safe visitors. In [MSNRR16], we propose the use of symbols for ensuring the validity of generated source code. [GMR+16] discusses product lines of template-based code generators. We also developed an approach for engineering reusable language components [HLMSN+15b, HLMSN+15a]. To understand the implications of executability for UML, we discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03], and the advantages and perils of using modeling languages for programming in [Rum02].

## Unified Modeling Language (UML)

Starting with an early identification of challenges for the standardization of the UML in [KER99] many of our contributions build on the UML/P variant, which is described in the books [Rum16, Rum17] respectively [Rum12, Rum13] and is implemented in [Sch12]. Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "System Model" [BCGR09a], [BCGR09b], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied, when checking variants of class diagrams [MRR11c] and objects diagrams [MRR11e] or the consistency of both kinds of diagrams [MRR11f]. We also apply these concepts to activity diagrams [MRR11b] which allows us to check for semantic differences of activity diagrams [MRR11a]. The basic semantics for ADs and their semantic variation points is given in [GRR10]. We also discuss how to ensure and identify model quality [FHR08], how models, views and the system under development correlate to each other [BGH+98], and how to use modeling in agile development projects [Rum04], [Rum02]. The question how to adapt and extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99], [FELR98] and [SRVK10].

## Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR+06, KRV10, Kra10, GKR+08, HR17] allows the specification of an integrated abstract and concrete syntax format [KRV07b, HR17] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR+07, Völ11, HLMSN+15b, HLMSN+15a, HRW18, BEK+18a, BEK+18b, BEK+19] and can, thus, easily be reused. We discuss the roles in software development using domain specific languages in [KRV14]. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11, GMR+16]. [BDL+18] presents a method to derive internal DSLs from grammars. In [BJRW18], we discuss the translation from grammars to accurate metamodels. Successful applications have been carried out in the Air Traffic Management [ZPK+11] and television [DHH+20] domains. Based on the concepts described above, meta modeling, model analyses and model evolution have been discussed in [LRSS10] and [SRVK10]. DSL quality [FHR08], instructions for defining views [GHK+07], guidelines to define DSLs [KKP+09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

## Software Language Engineering

For a systematic definition of languages using composition of reusable and adaptable language components, we adopt an engineering viewpoint on these techniques. General ideas on how to engineer a language can be found in the GeMoC initiative [CBCR15, CCF+15] and the concern-oriented language development approach [CKM+18]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10, HR17, HRW18, BEK+19]. In [SRVK10] we discuss the possibilities and the challenges using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Völ11, KRV08, HLMSN+15b, HLMSN+15a, HMSNRW16, HR17, BEK+18a, BEK+18b, BEK+19] and the backend [RRRW15, MSNRR16, GMR+16, HR17, BEK+18b]. In [GHK+15a, GHK+15b], we discuss the integration of handwritten and generated object-oriented code. [KRV14] describes the roles in software development using domain specific languages. Language derivation is to our believe a promising technique to develop new languages for a specific purpose that rely on existing basic languages [HRW18]. How to automatically derive such a transformation language using concrete syntax of the base language is described in [HRW15, Wei12] and successfully applied to various DSLs. We also applied the language derivation technique to tagging languages that decorate a base language [GLRR15] and delta languages [HHK+15a, HHK+13], where a delta language is derived from a base language to be able to constructively describe differences between model variants usable to build feature sets. The derivation of internal DSLs from grammars is discussd in [BDL+18] and a translation of grammars to accurate metamodels in [BJRW18].

## Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services. We use streams, statemachines and components [BR07] as well as expressive forms of composition and refinement [PR99, RW18] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13b]. In [RRW13a], we introduce a code generation framework for MontiArc. MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11, HKR+11] and evolution on deltas [HRRS12]. Other extensions are concerned with modeling cloud architectures [NPR13] and with the robotics domain [AHRW17a, AHRW17b]. [GHK+07] and [GHK+08a] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. [MRR14b] provides a precise technique to verify consistency of architectural views [Rin14, MRR13] against a complete architecture in order to increase reusability. We discuss the synthesis problem for these views in [MRR14a]. Co-evolution of architecture is discussed in [MMR10] and modeling techniques to describe dynamic architectures are shown in [HRR98, BHK+17, KKR19].

## Compositionality & Modularity of Models

[HKR+09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07, RW18] and algebraically grounded in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10, HR17] that can even be used to develop modeling tools in a compositional form [HR17, HLMSN+15b, HLMSN+15a, HMSNRW16, MSNRR16, HRW18, BEK+18a, BEK+18b, BEK+19]. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09]. [Völ11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15] applies compositionality to Robotics control. [CBCR15] (published in [CCF+15]) summarizes our approach to composition and remaining challenges in form of a conceptual model of the "globalized" use of DSLs. As a new form of decomposition of model information we have developed the concept of tagging languages in [GLRR15]. It allows to describe additional information for model elements in separated documents, facilitates reuse, and allows to type tags.

## Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision and detailedness is discussed in [HR04]. We defined a semantic domain called "System Model" by using mathematical theory in [RKB95, BHP+98] and [GKR96, KRB96]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08]. To better understand the effect of an evolved design, detection of semantic differencing as opposed to pure syntactical differences is needed [MRR10]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11f, MRR11f] compare class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH+98] discusses the relationships between a system, a view and a complete model in the context of the UML. [GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these on class and object diagrams in [MRR11f] as well as activity diagrams in [GRR10]. [Rum12] defines the semantics in a variety of code and test case generation, refactoring and evolution techniques. [LRSS10] discusses evolution and related issues in greater detail. [RW18] discusses an elaborated theory for the modeling of underspecification, hierarchical composition, and refinement that can be practically applied for the development of CPS.

## Evolution and Transformation of Models

Models are the central artifacts in model driven development, but as code they are not initially correct and need to be changed, evolved and maintained over time. Model transformation is therefore essential to effectively deal with models. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], decomposition [PR99, KRW20], synthesis [MRR14a], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12], and systematic model transformation language development [Wei12]. [Rum04] describes how comprehensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

## Variability and Software Product Lines (SPL)

Products often exist in various variants, for example cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK+08a] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12]. Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRR+11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13] and [HRW15] describe an approach to systematically derive delta languages. We also apply variability modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02] and generators [GMR+16]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09], leverage features for compositional reuse [BEK+18b], and applied it as a semantic language refinement on Statecharts in [GR11].

## Modeling for Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. In [RW18], we discuss how an elaborated theory can be practically applied for the development of CPS. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12], autonomous driving [BR12a, KKR19], and digital twin development [BDH+20] to processes and tools to improve the development as well as the product itself [BBR07]. In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest for the European airspace [ZPK+11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13b, RRW14]. In [RRW13a], we describe a code generation framework for this language. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

## Model-Driven Systems Engineering (MDSysE)

Applying models during Systems Engineering activities is based on the long tradition on contributing to systems engineering in automotive [GHK+08b], which culminated in a new comprehensive model-driven development process for automotive software [KMS+18, DGH+19]. We leveraged SysML to enable the integrated flow from requirements to implementation to integration. To facilitate modeling of products, resources, and processes in the context of Industry 4.0, we also conceived a multi-level framework for machining based on these concepts [BKL+18]. Research within the excellence cluster Internet of Production considers fast decision making at production time with low latencies using contextual data traces of production systems, also known as Digital Shadows (DS) [SHH+20]. We have investigated how to derive Digital Twins (DTs) for injection molding [BDH+20], how to generate interfaces between a cyber-physical system and its DT [KMR+20] and have proposed model-driven architectures for DT cockpit engineering [DMR+20].

## State Based Modeling (Automata)

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts [GKR96, BCR07b, BCGR09b, BCGR09a], (2) understanding the refinement [PR94, RK96, Rum96, RW18] and composition [GR95, GKR96, RW18] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96, RW18] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [GKR96, BR07]. We apply these techniques, e.g., in MontiArcAutomaton [RRW13a, RRW14, RRW13a, RW18] as well as in building management systems [FLP+11b].

## Model-Based Assistance and Information Services (MBAIS)

Assistive systems are a special type of information system: they (1) provide situational support for human behaviour (2) based on information from previously stored and real-time monitored structural context and behaviour data (3) at the time the person needs or asks for it [HMR+19]. To create them, we follow a model centered architecture approach [MMR+17] which defines systems as a compound of various connected models. Used languages for their definition include DSLs for behavior and structure such as the human cognitive modeling language [MM13], goal modeling languages [MRV20] or UML/P based languages [MNRV19]. [MM15] describes a process how languages for assistive systems can be created.

We have designed a system included in a sensor floor able to monitor elderlies and analyze impact patterns for emergency events [LMK+11]. We have investigated the modeling of human contexts for the active assisted living and smart home domain [MS17] and user-centered privacy-driven systems in the IoT domain in combination with process mining systems [MKM+19], differential privacy on event logs of handling and treatment of patients at a hospital [MKB+19], the mark-up of online manuals for devices [SM18] and websites [SM20], and solutions for privacy-aware environments for cloud services [ELR+17] and in IoT manufacturing [MNRV19]. The user-centered view on the system design allows to track who does what, when, why, where and how with personal data, makes information about it available via information services and provides support using assistive services.

## Modelling Robotics Architectures and Tasks

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW13a] extends the ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b, RRW14, RRRW15, HR17] that perfectly fit robotic architectural modeling. The LightRocks [THR+13] framework allows robotics experts and laymen to model robotic assembly tasks. In [AHRW17a, AHRW17b], we define a modular architecture modeling method for translating architecture models into modules compatible to different robotics middleware platforms.

## Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed and tested. A consistent requirements management that connects requirements with features in all phases of the development for the automotive domain is described in [GRJA12]. The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08a]. [HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. [RSW+15] describes an approach to use model checking techniques to identify behavioral differences of Simulink models. In [KKR19], we introduce a framework for modeling the dynamic reconfiguration of component and connector architectures and apply it to the domain of cooperating vehicles. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions. As tooling infrastructure, the SSElab storage, versioning and management services [HKR12] are essential for many projects.

## Smart Energy Management

In the past years, it became more and more evident that saving energy and reducing CO2 emissions is an important challenge. Thus, energy management in buildings as well as in neighbourhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales. During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for technical specification of building services already. We adapted the well-known concept of statemachines to be able to describe different states of a facility and to validate it against the monitored values [FLP+11b]. We show how our data model, the constraint rules, and the evaluation approach to compare sensor data can be applied [KLPR12].

## Cloud Computing & Enterprise Information Systems

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality, and new application domains. It promises to enable new business models, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development [KRR14]. Application classes like Cyber-Physical Systems and their privacy [HHK+14, HHK+15b], Big Data, App, and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools [KRS12]. We tackle these challenges by perusing a model-based, generative approach [NPR13]. The core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale. We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our own development platforms. New services, e.g., collecting data from temperature, cars etc. can now easily be developed.

## Model-Driven Engineering of Information Systems

Information Systems provide information to different user groups as main system goal. Using our experiences in the model-based generation of code with MontiCore [KRV10, HR17], we developed several generators for such data-centric information systems. *MontiGem* [AMN+20] is a specific generator framework for data-centric business applications that uses standard models from UML/P optionally extended by GUI description models as sources [GMN+20]. While the standard semantics of these modeling languages remains untouched, the generator produces a lot of additional functionality around these models. The generator is designed flexible, modular and incremental, handwritten and generated code pieces are well integrated [GHK+15b], tagging of existing models is possible [GLRR15], e.g., for the definition of roles and rights or for testing [DGH+18].

# References

[AHRW17a] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Engineering Robotics Software Architectures with Exchangeable Model Transformations. In *International Conference on Robotic Computing (IRC'17)*, pages 172–179. IEEE, April 2017.

[AHRW17b] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Modeling Robotics Software Architectures with Modular Model Transformations. *Journal of Software Engineering for Robotics (JOSER)*, 8(1):3–16, 2017.

[AMN+20] Kai Adam, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In *40 Years EMISA: Digital Ecosystems of the Future: Methodology, Techniques and Applications (EMISA'19)*, LNI P-304, pages 59–66. Gesellschaft für Informatik e.V., May 2020.

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.

[BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.

[BCGR09b] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009.

[BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.

[BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.

[BDH+20] Pascal Bibow, Manuela Dalibor, Christian Hopmann, Ben Mainz, Bernhard Rumpe, David Schmalzing, Mauritius Schmitz, and Andreas Wortmann. Model-Driven Development of a Digital Twin for Injection Molding. In Schahram Dustdar, Eric Yu, Camille Salinesi, Dominique Rieu, and Vik Pant, editors, *International Conference on Advanced Information Systems Engineering (CAiSE'20)*, Lecture Notes in Computer Science 12127, pages 85–100. Springer International Publishing, June 2020.

[BDL+18] Arvid Butting, Manuela Dalibor, Gerrit Leonhardt, Bernhard Rumpe, and Andreas Wortmann. Deriving Fluent Internal Domain-specific Languages from Grammars. In *International Conference on Software Language Engineering (SLE'18)*, pages 187–199. ACM, 2018.

[BEK+18a] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Controlled and Extensible Variability of Concrete and Abstract Syntax with Independent Language Features. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS'18)*, pages 75–82. ACM, January 2018.

[BEK+18b] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Modeling Language Variability with Reusable Language Components. In *International Conference on Systems and Software Product Line (SPLC'18)*. ACM, September 2018.

[BEK+19] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Composition of Independent Language Features. *Journal of Systems and Software*, 152:50–69, June 2019.

[BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737, Germany, 1997. TU Munich.

[BGH+98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998.

[BGRW17] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. Taming the Complexity of Model-Driven Systems Engineering Projects. Part of the Grand Challenges in Modeling (GRAND'17) Workshop, July 2017.

[BGRW18] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. On the Need for Artifact Models in Model-Driven Systems Engineering Projects. In Martina Seidl and Steffen Zschaler, editors, *Software Technologies: Applications and Foundations*, LNCS 10748, pages 146–153. Springer, January 2018.

[BHK+17] Arvid Butting, Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Classification of Dynamic Reconfiguration in Component and Connector Architecture Description Languages. In *Proceedings of MODELS 2017. Workshop ModComp*, CEUR 2019, September 2017.

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.

[BJRW18] Arvid Butting, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Translating Grammars to Accurate Metamodels. In *International Conference on Software Language Engineering (SLE'18)*, pages 174–186. ACM, 2018.

[BKL+18] Christian Brecher, Evgeny Kusmenko, Achim Lindt, Bernhard Rumpe, Simon Storms, Stephan Wein, Michael von Wenckstern, and Andreas Wortmann. Multi-Level Modeling Framework for Machine as a Service Applications Based on Product Process Resource

Models. In *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control (ISCSIC'18)*. ACM, September 2018.

[BKRW17] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Semantic Differencing for Message-Driven Component & Connector Architectures. In *International Conference on Software Architecture (ICSA'17)*, pages 145–154. IEEE, April 2017.

[BKRW19] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution. *Journal of Systems and Software*, 149:437–461, March 2019.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[BR12a] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.

[BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.

[CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.

[CCF+15] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015.

[CEG+14] Betty Cheng, Kerstin Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi Müller, Patrizio Pelliccione, Anna Perini, Nauman Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 2014.

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009.

[CKM+18] Benoit Combemale, Jörg Kienzle, Gunter Mussbacher, Olivier Barais, Erwan Bousse, Walter Cazzola, Philippe Collet, Thomas Degueule, Robert Heinrich, Jean-Marc Jézéquel, Manuel Leduc, Tanja Mayerhofer, Sébastien Mosser, Matthias Schöttle, Misha Strittmatter, and Andreas Wortmann. Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering. *Computer Languages, Systems & Structures*, 54:139 – 155, 2018.

[DEKR19] Imke Drave, Robert Eikermann, Oliver Kautz, and Bernhard Rumpe. Semantic Differencing of Statecharts for Object-oriented Systems. In Slimane Hammoudi, Luis Ferreira Pires, and Bran Selić, editors, *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'19)*, pages 274–282. SciTePress, February 2019.

[DGH+18] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Matthias Markthaler, Bernhard Rumpe, and Andreas Wortmann. Model-Based Testing of Software-Based System Functions. In *Conference on Software Engineering and Advanced Applications (SEAA'18)*, pages 146–153, August 2018.

[DGH+19] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Matthias Markthaler, Philipp Orth, Karin Samira Salman, Johannes Richenhagen, Bernhard Rumpe, Christoph Schulze, Michael Wenckstern, and Andreas Wortmann. SMArDT modeling for automotive software testing. *Software: Practice and Experience*, 49(2):301–328, February 2019.

[DHH+20] Imke Drave, Timo Henrich, Katrin Hölldobler, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Modellierung, Verifikation und Synthese von validen Planungszuständen für Fernsehausstrahlungen. In Dominik Bork, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2020*, pages 173–188. Gesellschaft für Informatik e.V., February 2020.

[DKMR19] Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Semantic Evolution Analysis of Feature Models. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 245–255. ACM, September 2019.

[DMR+20] Manuela Dalibor, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Towards a Model-Driven Architecture for Interactive Digital Twin Cockpits. In Gillian Dobbie, Ulrich Frank, Gerti Kappel, Stephen W. Liddle, and Heinrich C. Mayr, editors, *Conceptual Modeling*, pages 377–387. Springer International Publishing, October 2020.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluver Academic Publisher, 1999.

[EJK+19] Rolf Ebert, Jahir Jolianis, Stefan Kriebel, Matthias Markthaler, Benjamin Pruenster, Bernhard Rumpe, and Karin Samira Salman. Applying Product Line Testing for the Electric Drive System. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 14–24. ACM, September 2019.

[ELR+17] Robert Eikermann, Markus Look, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Architecting Cloud Services for the Digital me in a Privacy-Aware Environment. In Ivan Mistrik, Rami Bahsoon, Nour Ali, Maritta Heisel, and Bruce Maxim, editors, *Software Architecture for Big Data and the Cloud*, chapter 12, pages 207–226. Elsevier Science & Technology, June 2017.

[FELR98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, November 1998.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.

[FLP+11a] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. Der Energie-Navigator - Performance-Controlling für Gebäude und Anlagen. *Technik am Bau (TAB) - Fachzeitschrift für Technische Gebäudeausrüstung*, Seiten 36-41, März 2011.

[FLP+11b] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.

[FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference(IEECB'12)*, 2012.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007.

[GHK+08a] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.

[GHK+08b] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. View-Centric Modeling of Automotive Logical Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV*, Informatik Bericht 2008-02. TU Braunschweig, 2008.

[GHK+15a] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 74–85. SciTePress, 2015.

[GHK+15b] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. Integration of Handwritten and Generated Object-Oriented Code. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 112–132. Springer, 2015.

[GHR17] Timo Greifenberg, Steffen Hillemacher, and Bernhard Rumpe. *Towards a Sustainable Artifact Model: Artifacts in Generator-Based Model-Driven Projects*. Aachener Informatik-Berichte, Software Engineering, Band 30. Shaker Verlag, December 2017.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference*, LNI 82, Seiten 67-81, 2006.

[GLPR15] Timo Greifenberg, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Energieeffiziente Städte - Herausforderungen und Lösungen aus Sicht des Software Engineerings. In Linnhoff-Popien, Claudia and Zaddach, Michael and Grahl, Andreas, Editor, *Marktplätze im Umbruch: Digitale Strategien für Services im Mobilen Internet*, Xpert.press, Kapitel 56, Seiten 511-520. Springer Berlin Heidelberg, April 2015.

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.

[GMN+20] Arkadii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In Bonnie Anderson, Jason Thatcher, and Rayman Meservy, editors, *25th Americas Conference on Information Systems (AMCIS 2020)*, AIS Electronic Library (AISeL), pages 1–10. Association for Information Systems (AIS), August 2020.

[GMR+16] Timo Greifenberg, Klaus Müller, Alexander Roth, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Modeling Variability in Template-based Code Generators for Product Line Engineering. In *Modellierung 2016 Conference*, LNI 254, pages 141–156. Bonner Köllen Verlag, March 2016.

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012.

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.

[HHK+14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014.

[HHK+15a] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.

[HHK+15b] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Future Generation Computer Systems*, 56:701–718, 2015.

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007.

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009.

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.

[HLMSN+15a] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Composition of Heterogeneous Modeling Languages. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 45–66. Springer, 2015.

[HLMSN+15b] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 19–31. SciTePress, 2015.

[HMR+19] Katrin Hölldobler, Judith Michael, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Innovations in Model-based Software and Systems Engineering. *The Journal of Object Technology*, 18(1):1–60, July 2019.

[HMSNRW16] Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. Compositional Language Engineering using Generated, Extensible, Static Type Safe Visitors. In *Conference on Modelling Foundations and Applications (ECMFA)*, LNCS 9764, pages 67–82. Springer, July 2016.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.

[HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017*. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, pages 1 – 10. fortiss GmbH, 2011.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181-192, 2012.

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.

[HRW18] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Computer Languages, Systems & Structures*, 54:386–405, 2018.

[JWCR18] Rodi Jolak, Andreas Wortmann, Michel Chaudron, and Bernhard Rumpe. Does Distance Still Matter? Revisiting Collaborative Distributed Software Design. *IEEE Software*, 35(6):40–47, 2018.

[KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.

[KKR19] Nils Kaminski, Evgeny Kusmenko, and Bernhard Rumpe. Modeling Dynamic Architectures of Self-Adaptive Cooperative Systems. *The Journal of Object Technology*, 18(2):1–20, July 2019. The 15th European Conference on Modelling Foundations and Applications.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012.

[KMR+20] Jörg Christian Kirchhof, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 90–101. ACM, October 2020.

[KMS+18] Stefan Kriebel, Matthias Markthaler, Karin Samira Salman, Timo Greifenberg, Steffen Hillemacher, Bernhard Rumpe, Christoph Schulze, Andreas Wortmann, Philipp Orth, and Johannes Richenhagen. Improving Model-based Testing in Automotive Software Engineering. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE'18)*, pages 172–180. ACM, June 2018.

[KNP+19] Evgeny Kusmenko, Sebastian Nickels, Svetlana Pavlitskaya, Bernhard Rumpe, and Thomas Timmermanns. Modeling and Training of Neural Processing Systems. In Marouane Kessentini, Tao Yue, Alexander Pretschner, Sebastian Voss, and Loli Burgueño, editors, *Conference on Model Driven Engineering Languages and Systems (MODELS'19)*, pages 283–293. IEEE, September 2019.

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in*

*Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012.

[KPRS19] Evgeny Kusmenko, Svetlana Pavlitskaya, Bernhard Rumpe, and Sebastian Stüber. On the Engineering of AI-Powered Systems. In Lisa O'Conner, editor, *ASE19. Software Engineering Intelligence Workshop (SEI19)*, pages 126–133. IEEE, November 2019.

[KR18] Oliver Kautz and Bernhard Rumpe. On Computing Instructions to Repair Failed Model Refinements. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 289–299. ACM, October 2018.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapmann & Hall, 1996.

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.

[KRRS19] Stefan Kriebel, Deni Raco, Bernhard Rumpe, and Sebastian Stüber. Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g. Based on Broy's Streams Become Feasible? In Stephan Krusche, Kurt Schneider, Marco Kuhrmann, Robert Heinrich, Reiner Jung, Marco Konersmann, Eric Schmieders, Steffen Helke, Ina Schaefer, Andreas Vogelsang, Björn Annighöfer, Andreas Schweiger, Marina Reich, and André van Hoorn, editors, *Proceedings of the Workshops of the Software Engineering Conference. Workshop on Avionics Systems and Software Engineering (AvioSE'19)*, CEUR Workshop Proceedings 2308, pages 87–94. CEUR Workshop Proceedings, February 2019.

[KRRvW17] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 34–50. Springer, July 2017.

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113-116. VDI Verlag, 2012.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, LNCS 4735, pages 286–300. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[KRV14] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modeling Languages. In *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM‘ 06)*. CoRR arXiv, 2014.

[KRW20] Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Automated semantics-preserving parallel decomposition of finite component and connector architectures. *Automated Software Engineering*, 27:119–151, April 2020.

[LMK+11] Philipp Leusmann, Christian Möllering, Lars Klack, Kai Kasugai, Bernhard Rumpe, and Martina Ziefle. Your Floor Knows Where You Are: Sensing and Acquisition of Movement Data. In Arkady Zaslavsky, Panos K. Chrysanthis, Dik Lun Lee, Dipanjan Chakraborty, Vana Kalogeraki, Mohamed F. Mokbel, and Chi-Yin Chow, editors, *12th IEEE International Conference on Mobile Data Management (Volume 2)*, pages 61–66. IEEE, June 2011.

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010.

[MKB+19] Felix Mannhardt, Agnes Koschmider, Nathalie Baracaldo, Matthias Weidlich, and Judith Michael. Privacy-Preserving Process Mining: Differential Privacy for Event Logs. *Business & Information Systems Engineering*, 61(5):1–20, October 2019.

[MKM+19] Judith Michael, Agnes Koschmider, Felix Mannhardt, Nathalie Baracaldo, and Bernhard Rumpe. User-Centered and Privacy-Driven Process Mining System Design for IoT. In Cinzia Cappiello and Marcela Ruiz, editors, *Proceedings of CAiSE Forum 2019: Information Systems Engineering in Responsible Information Systems*, pages 194–206. Springer, June 2019.

[MM13] Judith Michael and Heinrich C. Mayr. Conceptual modeling for ambient assistance. In *Conceptual Modeling - ER 2013*, LNCS 8217, pages 403–413. Springer, 2013.

[MM15] Judith Michael and Heinrich C. Mayr. Creating a domain specific modelling method for ambient assistance. In *International Conference on Advances in ICT for Emerging Regions (ICTer2015)*, pages 119–124. IEEE, 2015.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.

[MMR+17] Heinrich C. Mayr, Judith Michael, Suneth Ranasinghe, Vladimir A. Shekhovtsov, and Claudia Steinberger. *Model Centered Architecture*, pages 85–104. Springer International Publishing, 2017.

[MNRV19] Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Towards Privacy-Preserving IoT Systems Using Model Driven Engineering. In Nicolas Ferry, Antonio Cicchetti, Federico Ciccozzi, Arnor Solberg, Manuel Wimmer, and Andreas Wortmann, editors, *Proceedings of MODELS 2019. Workshop MDE4IoT*, pages 595–614. CEUR Workshop Proceedings, September 2019.

[MRR10] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10)*, LNCS 6627, pages 194–203. Springer, 2010.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011.

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CDDiff: Semantic Differencing for Class Diagrams. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming*, pages 230–254. Springer Berlin Heidelberg, 2011.

[MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.

[MRR11f] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.

[MRR11g] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Summarizing Semantic Model Differences. In Bernhard Schätz, Dirk Deridder, Alfonso Pierantonio, Jonathan Sprinkle, and Dalila Tamzalit, editors, *ME 2011 - Models and Evolution*, October 2011.

[MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 444–454. ACM New York, 2013.

[MRR14a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views (extended abstract). In Wilhelm Hasselbring and Nils Christian Ehmke, editors, *Software Engineering 2014*, LNI 227, pages 63–64. Gesellschaft für Informatik, Köllen Druck+Verlag GmbH, 2014.

[MRR14b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014.

[MRV20] Judith Michael, Bernhard Rumpe, and Simon Varga. Human behavior, goals and model-driven software engineering for assistive systems. In Agnes Koschmider, Judith Michael, and Bernhard Thalheim, editors, *Enterprise Modeling and Information Systems Architectures (EMSIA 2020)*, pages 11–18. CEUR Workshop Proceedings, June 2020.

[MS17] Judith Michael and Claudia Steinberger. Context modeling for active assistance. In Cristina Cabanillas, Sergio España, and Siamak Farshidi, editors, *Proc. of the ER Forum 2017 and the ER 2017 Demo Track co-located with the 36th Int. Conference on Conceptual Modelling (ER 2017)*, pages 221–234, 2017.

[MSNRR16] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. An Extended Symbol Table Infrastructure to Manage the Composition of Output-Specific Generator Information. In *Modellierung 2016 Conference*, LNI 254, pages 133–140. Bonner Köllen Verlag, March 2016.

[NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013.

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.

[PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15*. Northeastern University, 2001.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014.

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.

[RR11] Jan Oliver Ringert and Bernhard Rumpe. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics*, 2011.

[RRRW15] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015.

[RRSW17] Jan Oliver Ringert, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Teaching Agile Model-Driven Engineering for Cyber-Physical Systems. In *International Conference on Software Engineering: Software Engineering and Education Track (ICSE'17)*, pages 127–136. IEEE, May 2017.

[RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013.

[RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.

[RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.

[RSW+15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, London, 2002.

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003.

[Rum04] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004.

[Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012.

[Rum13] Bernhard Rumpe. Towards Model and Language Composition. In Benoit Combemale, Walter Cazzola, and Robert Bertrand France, editors, *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*, pages 4–7. ACM, 2013.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017.

[RW18] Bernhard Rumpe and Andreas Wortmann. Abstraction and Refinement in Hierarchically Decomposable and Underspecified CPS-Architectures. In Lohstroh, Marten and Derler, Patricia Sirjani, Marjan, editor, *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, LNCS 10760, pages 383–406. Springer, 2018.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[SHH+20] Günther Schuh, Constantin Häfner, Christian Hopmann, Bernhard Rumpe, Matthias Brockmann, Andreas Wortmann, Judith Maibaum, Manuela Dalibor, Pascal Bibow, Patrick Sapel, and Moritz Kröger. Effizientere Produktion mit Digitalen Schatten. *ZWF Zeitschrift für wirtschaftlichen Fabrikbetrieb*, 115(special):105–107, April 2020.

[SM18] Claudia Steinberger and Judith Michael. Towards Cognitive Assisted Living 3.0. In *International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops 2018)*, pages 687–692. IEEE, march 2018.

[SM20] Claudia Steinberger and Judith Michael. Using Semantic Markup to Boost Context Awareness for Assistive Systems. In *Smart Assisted Living: Toward An Open Smart-Home Infrastructure*, Computer Communications and Networks, pages 227–246. Springer International Publishing, 2020.

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010.

[THR+13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.

[ZPK$^+$11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.