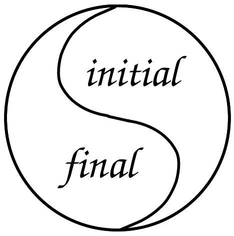
Swinging Data Types

– The dielectic between actions and constructors –

August 13, 2004

Peter Padawitz
Fachbereich Informatik
Universität Dortmund, Germany
peter@ls5.informatik.uni-dortmund.de



Abstract

Initial structures are good for modelling constructor-based data types because they fit the intuition about these types and admit resolution- and rewrite-oriented inductive theorem proving. The corresponding specification and verification methods do not comply so well with *non-free* or *permutative* types such as sets, bags and maps and are still less appropriate when infinite structures like streams or processes come into play. Non-free and infinite structure are better modelled as dynamic objects, which are identified through reactions upon actions (methods, messages, state transitions) rather than through constructors they might be built of. *Extensional, contextual, behavioural, observational* or *bisimilarity* relations model object equality and the suitable domains are *final structures* that are *conservative* with respect to *visible* subtypes. Consequently, a collection of data types and programs should be designed hierarchically as a “swinging” chain of specifications each of which extends its predecessor by either *constructor types* or *action types*. Constructor types introduce the visible domains and come with *inductively* defined *total* functions, structural equality and *safety predicates* with Horn clause axioms, while action types provide the *hidden* domains together with *coinductively* defined *partial* functions, behavioural equality and *liveness predicates* with liveness axioms that are dual to Horn clauses. A swinging specification is interpreted as a sequence of initial and final models. General proof rules capture this semantics and exploit the duality of induction and coinduction to its utmost extent. The deductive tractability is further enhanced by making both constructor and action types amenable to rewrite-oriented proof methods so that we can reason about swinging specifications in the same way we are used to reason about exclusively constructor-based types.

To my friends

in Miami and San Juan

who taught me

quite different lessons

about safety and liveness

Contents

1 Introduction **5**

1.1 Final semantics 5

1.2 Algebras and coalgebras 7

1.3 Infinite types 8

1.4 Safety and liveness predicates and their axioms 10

1.5 Further input to this work 13

1.6 From models to theories 15

1.7 Overview 16

2 Syntax **17**

3 Semantics **22**

4 Swinging Semantics **26**

5 Swinging Theory **32**

6 Functional Specifications **42**

7 Proof Rules **47**

8 Strong Functionality **58**

9 Consistency and Inductive Validity **64**

10 Applications **78**

10.1 Parametric types 78

10.1.1 Proofs by fixpoint induction 81

10.2 Permutative types 86

10.3 Streams 91

10.3.1 Distributed systems 94

10.3.2 Process networks 95

10.3.3 Concurrent languages 97

10.4 Process algebra 99

10.4.1 Basic process algebra 99

10.4.2 Concurrency and communication 102

10.4.3 Abstraction and refinement 104

10.5 Temporal logic of actions . . . 105
10.5.1 Simple TLA . . . 106
10.5.2 Abstraction and refinement . . . 107
10.6 The SECD machine . . . 108
10.7 Object-oriented modules . . . 110
10.8 Dynamic logic . . . 111
10.8.1 A complete assertion rule for recursive procedures . . . 114
10.9 Iteration . . . 123

11 Mind - Math - Movement **126**

References **128**

1 Introduction

Algebraic specifications are widely used for designing data types as a collection of carrier sets and functions and predicates operating on the carriers (cf., e.g., [GTW 78], [EM 85], [Wir 90]). In particular, any specification SP whose functions and predicates are axiomatized by Horn clauses has a canonical model, the *initial* one, which usually fits the intuition about the type to be specified as well as it admits inductive theorem proving (cf., e.g., [GM 86], [Pad 88], [Pad 92]).In practice, inductive theorem proving works well only if SP is *inhabited*, i.e. the functions of SP allow us to construct ground (i.e. variable-free) terms for each of the carriers specified by SP. Moreover, inductive theorem proving becomes really effective only if the set of functions can be decomposed into *constructors* on the one hand and *defined functions* on the other hand. Each ground term must be semantically equivalent to a term built up of constructors and ground terms built up of constructors are semantically equivalent only if they are syntactically identical. The axioms of SP are merely recursive equations for defined functions and predicates. Several equations yield several cases due to different constructors forming the arguments of a defined function or predicate as in the following example:

```
vars      m,n:nat  L,L':list
axioms    0+n = n
          succ(m)+n = succ(m+n)
          0 < succ(m)
          succ(m) < succ(n) <== m < n
          prefix(nil,L)
          prefix(n&L,n&L') <== prefix(L,L')
```

The constructor discipline entails proof-theoretical properties of SP such as *ground confluence*, a property that is often indispensable for performing inductive proofs effectively. It admits augmenting actual proof systems by powerful lemmas and rules, which considerably enhance the "automation degree" of a derivation (cf. [Pad 92], [Pad 94], [Pad 95]). It turns out that the more SP resembles a program written in a functional or logic language with constructor types, such as ML or Prolog, the greater is the chance that proofs about the program become tractable because then SP will ground confluent.

1.1 Final semantics

Data constructors are called free if each object of the data type has a unique ground term representation built up of constructors. There have been several attempts to generalize the notion of ground confluence (see above) so that *permutative structures* with non-free constructors can be treated deductively in a similar way as specifications with only free constructors. Permutative structures such as *sets, bags* (or multisets) and *maps* (or arrays) are usually presented by *constructor equations* from which all semantical equivalences between constructor terms can be derived:

```
vars      x,y:entry  b:bag  s:set  f:map  i,j:index
axioms    insert(x,insert(y,b)) = insert(y,insert(x,b))
          insert(x,insert(x,s)) = insert(x,s)
          insert(x,insert(y,s)) = insert(y,insert(x,s))
          put(i,x,put(i,y,f)) = put(i,x,f)
          put(i,x,put(j,y,f)) = put(j,y,put(i,x,f)) <== i <> j
```

(cf. Section 10.2). Each attempt to deal with non-free constructors is a variation of one of the following two approaches:

- *theory resolution* and *theory rewriting* (cf., e.g., [JLM 84], [Sti 85], [JKi 86], [Höl 89], [AB 92]),

- **final** or *terminal semantics* based on *behavioural* or *observational equivalences* (cf., e.g., [GGM 76], [Wan 79], [Kam 83], [WPPDB 83], [ST 87], [Hen 91], [GD 94]).

Both approaches assume that specifications are built up hierarchically. Theory resolution and rewriting regard the constructor equations of a specification SP as a separate primitive or constraint theory PTh and extend it to a theory Th given by further axioms of SP. Final semantics reverses this hierarchy: The basic theory is given by SP *without* constructor equations, which are later *induced* by behavioural equivalence relations.

Theory resolution and rewriting aim at effective proof rules, quite similar to *constraint logic programming* (cf. [JL 87]) where the separation of a constraint theory PTh admits employing efficient PTh-solution procedures during program execution whose actual goal is to compute solutions with respect to the entire theory Th. However, the embedding of PTh in Th is difficult both to formalize and to implement. In fact, only few data types with non-free constructors can be handled in this way.

Final semantics generalizes the construction of a final, i.e. minimal, automaton to the construction of a model of a specification SP with non-free constructors. In automata theory, behavioural equivalence comes as *Nerode equivalence* $\sim_{ner}$, which identifies each two states s, s' with the same input-output behaviour (cf. [MG 85], Sect. 5):

$$s \sim_{ner} s' \quad \text{iff} \quad \text{for all } w \in Input^*,\ \beta\delta^*(s,w) = \beta\delta^*(s',w)$$

where β and δ are the output resp. transition functions of the automaton. Final semantics generalizes $\sim_{ner}$ to the **behavioural** or **observational equivalence** $\sim$ between functional terms:

$$t \sim t' \quad \text{iff} \quad \text{for all } \textbf{contexts } c,\ c(t) \equiv_{SP} c(t')$$

where a context is a term of *primitive* or *visible* sort and $\equiv_{SP}$ is semantic equivalence. This is the general reference to automata theory in all the variations of final semantics. Differences occur with respect to the admissable sorts of *context parameters*, i.e., the schema for defining $\sim$ usually reads a little more general:

$$t \sim t' \quad \text{iff} \quad \text{for all contexts } c \text{ and context parameters } u,\ c(t,u) \equiv_{SP} c(t',u).$$

[ST 87] and [Pad 88] replace the above condition $\forall c : c(t,u) \equiv_{SP} c(t',u)$ by more general *context formulas* or *observations.* [MG 85] and [NO 88] restrict context parameters to visible sorts, while [BHW 95] distinguishes *input sorts* for context parameters from the visible – or output – sorts as automata theory separates input sets from output sets. Input sorts may be motivated by non-term-generated parameter models (cf. [BHW 95]). But since we focus on term-generated models, all context parameters are supposed to be visible. Hence the definition of behavioural equivalence is simpler and we avoid complicated proof rules, which were not justified by the applications we have in mind. The notion *input sorts* is reserved for the parameter sorts of a parametric specification, which, in terms of [BHW 95], are the *unconstrained sorts* of a *reachability constraint.*

Theory resolution and theory rewriting might be too restrictive, but final semantics so far is too general from the viewpoint of verification methodology. This is mainly due to the above notion of a context, which makes proofs of behavioural equivalence almost untractable. [Hen 91] and [GD 94] propose induction on contexts, but this only adds a further induction-hypothesis formation problem to the one we already face when reasoning within initial models. For reducing the number of contexts to be taken into account [BH 95] singles out *crucial contexts* as those ones to which we may restrict the definition of behavioural equivalence without loosing its *congruence property*, which is indispensible for constructing a model from $\sim$. But how can we recognize crucial contexts besides proving the congruence property directly? The problem is that final semantics approaches have pointed out proof obligations without providing us with simple criteria to fulfil them.

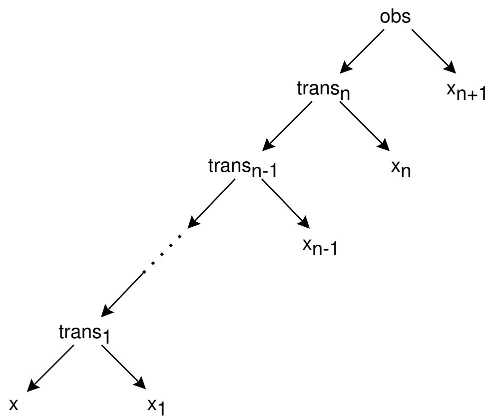


Figure 1. A context[1]

A closer look at the automaton metaphor reveals how these proof-theoretical problems might be solved. The contexts of an automaton are all terms of the form $\beta(\delta_w(s))$ where for inputs w, δ_w is defined by $\delta_w(s) = \delta^*(w, s)$ for all states s. This suggests that each component function, which we will generally call an **action**, is either a **transition** like δ_w, which maps into a non-primitive or *hidden* type, or an **observer** like β, which maps into a primitive or visible type. Conceptually, it is not important that β and δ_w are unary, but it is essential that each action has a unique argument of non-primitive sort. This complies with the object-oriented design paradigm of associating attributes and methods with a single class (= hidden type) and also with the object-oriented execution principle of sending methods as messages to an object of that class for the purpose of changing the receivers' attributes. A context reflects a sequence of transitions, which are induced by received messages, and a final observation or output of primitive values. Hence, formally, a context c is a term shaped as in Fig. 1 where x is the unique hidden-sorted variable of c and $x_1, \ldots, x_{n+1}$ are (lists of) variables of visible sorts. x stands for a *dynamic object* that is observed through *obs* after it has passed the sequence of state transitions, or, in other words, received the sequence of messages $trans_1, \ldots, trans_n$. Based on this notion of a context, behavioural equivalence is defined as in 4.2.

1.2 Algebras and coalgebras

Given a signature SIG, the class of SIG-algebras defines an endofunctor F on the category Set of sets such that a SIG-algebra A can be presented as a morphism $\alpha : F(A) \to A$ where F maps A to the **sum** of the domains of all operations of A (cf. [LS 81]). In general, morphisms $\alpha : F(A) \to A$ are called F-*algebras*.

Conversely, given an endofunctor $G : Set \to Set$, G-*coalgebras* are morphisms $\beta : A \to G(A)$. An action structure with hidden part A and visible part V, transitions $trans_1, \ldots, trans_k : A \times V \to A$ and observers $obs_1, \ldots, obs_l : A \times V \to V$ can be presented as the G-coalgebra $\beta : A \to G(A)$ where G maps A to the **product** $(\prod_{v \in V} A^k) \times (\prod_{v \in V} V^l)$ such that for all $a \in A$, $v \in V$, $1 \le i \le k$ and $1 \le j \le l$, $proj_{v,i}(\beta(a)) = trans_i(a, v)$ and $proj_{v,k+j}(\beta(a)) = obs_j(a, v)$ (cf. [Hag 89], [Rei 95]).

If the category of F-algebras has an initial object $\iota : F(I) \to I$, then ι is an isomorphism, which, due to the *sums* returned by F, consists of injections into I. These embeddings define the *constructors* of the type given by F. Conversely, if the category of G-coalgebras has a final object $\tau : T \to G(T)$, then τ is an isomorphism, which, due to the *products* returned by G, consists of projections on T. Dually to the constructors derived from F, these projections define the *actions* of the type given by G. Operations defined inductively on I come as initial morphisms from ι to other F-algebras (cf. [LS 81]). Final morphisms from G-coalgebras to τ represent

[1] The arrows point to function arguments.

operations on T and are called to be *defined by coinduction* (cf. [Rei 95]).

The coalgebra view confirms the necessity to construct specifications hierarchically. The above example of G shows that we usually start out from carriers for the visible sorts. This is in accordance with categories of algebras with fixed carriers for primitive sorts, which are used in final semantics approaches where the final objects are term algebras factored through behavioural equivalences (cf. e.g. [Wan 79], [MG 85]). Intuitively, the fixed visible carriers ensure that final objects do not collapse into one-point structures. This is obvious in final semantics and also applies to corresponding categories of coalgebras. Moreover, the distinction between actions, which form the contexts (cf. Fig. 1), and other functions, which are defined by coinduction *in terms of* actions, stems from the coalgebra view and significantly improves the deductive tractability of final semantics.

1.3 Infinite types

Data types with non-free constructors are not the only application of the coalgebra approach. It also allows us to model infinite data structures such as streams and processes. For instance, an action type of streams has a transition $tail : stream \rightarrow stream$, which deletes the first element of a stream, and an observer $head : stream \rightarrow entry$, which returns the first element of a stream (cf. Section 10.1). Process algebra (cf. [BW 90]) can be modelled as an action type with an observer $prefix : proc \rightarrow (tree \rightarrow bool)$ such that $prefix(p)(t)$ holds true iff the finite tree t is a prefix of the process p (cf. Section 10.5).

The **cpo approach** (cf., e.g., [GTWW 77], [LM 82], [Möl 84], [TW 86]) models infinite structures as equivalence classes of term chains. This model is initial in a category of algebras equipped with a complete partial order (*cpo*) $\leq$ and satisfying given inequations, i.e. axioms of the form $t \leq t'$ (cf. [Blo 76]; [Pad 83]). In the case of processes, the category of partially ordered algebras is usually replaced by a class of metric spaces (cf. [BT 85]; [EPBRDG 90]). The cpo approach maintains the idea of data constructibility, which characterizes initial semantics approaches. The problem is the same as in the finite case: if non-free constructors are involved, we need constructor equations whose different nature compared to function or predicate definitions enforces theory resolution and theory rewriting (see above). Moreover, proofs by induction on the term structure are no longer possible. Structural induction is replaced by **computational** or (least) **fixpoint induction** (cf. [Sco 70], [Mor 71]) and *recursive program schemes* (cf. [Niv 75]) take the place of constructor-based function definitions. *Infinite term rewriting* (cf. [CO 90]) is also subjected to the proof-theoretical restrictions of the cpo approach. The crucial point is that only *admissable* predicates can be proved by fixpoint induction. A predicate P is admissable (cf. [Manna 74]) or *chain complete* (cf. [BW 88]) if P holds true for the supremum S of a chain of objects already if all these objects (*finite approximations* of S) satisfy P. For instance, the "cpo version" of the predicate "f is a partial function" is not admissable: finite approximations of some total functions are partial (cf. [Manna 74], Ex. 5-21; [LS 87], p.86). However, many interesting properties of infinite objects are *liveness conditions* (fairness, termination, etc.), which are *non*-admissable (cf. 1.4). The fixpoint induction principle mentioned here is based on Kleene's fixpoint theorem (cf. [LNS 82]). Hence the properties to be proved using this principle must be presented in terms of a *continuous* function, which is mostly given by the second-order function whose least fixpoint interprets a recursive program scheme. In Sections 1.5 and 7, we are concerned with the other fixpoint induction principle that employs the more general Knaster-Tarski fixpoint theorem (cf. [LNS 82]) and whose applications always start out from a *monotonic* function on a *powerset* cpo.

Although infinite objects cannot be presented as finite terms built up of constructors, we can keep to final semantics and behavioural equivalence. Again, let us refer to the automaton metaphor. In final semantics, two objects are different iff they are behaviourally inequivalent. In a final automaton, two states are different iff they have different behaviours (= input-output relations). As not *every* relation between given input and output sets need to have a state representation, so not *every* object of an *action type* need to have a term representation. For instance, we construct certain "initial" streams by calling stream-generating functions (cf. Ex. 2.5). Further streams are obtained by applying stream-manipulating functions to initial streams. But stream-generating and -manipulating functions are not free constructors because the equality of objects is determined by behavioural equivalence and not by the term structure. On the one hand, an action type model can only consist of those objects that are represented as ground terms built up of generating functions for and transitions on action sorts. On the other hand, each object may take a number of different term representations. Both characteristics can also be found in automata if we identify *term* with *state*: many behaviours may have no state representation at all; different states may be behaviourally equivalent.

To sum up, we distinguish between two kinds of data types: **constructor types** and **action types**. The former present finite structures with free constructors, the latter comprise the rest. Sets, bags and maps (cf. 1.1) are action types where the actions are the membership of an element, the frequency of an element and the access to an element, respectively. Actions of infinite types were mentioned above.

If actions are the basic functions of an action type such as constructors are the basic functions of a constructor type, then what are the *defined* functions of an action type? And *how* are these defined in a corresponding specification? The coalgebra approach answers the first question: defined functions are presentable as morphisms from well-chosen (!) coalgebras to the final coalgebra. This is called a definition by coinduction (cf. 1.2). For instance, streams with elements from a set A build up the final colgebra $\tau : T \to G(T)$ where $G : Set \to Set$ is defined by $G(B) =_{def} A \times B$ for sets B and $G(f) =_{def} id_A \times f$ for maps $f : B \to C$. Hence any G-coalgebra $f : B \to A \times B$ is a pair of maps $(head_f : B \to A, tail_f : B \to B)$. In case of τ we have:

- $T =_{def} [I\!N \to A]$,
- $head_\tau(s) =_{def} s(0)$ for all $s : I\!N \to A$,
- $(tail_\tau(s))(n) =_{def} s(n+1)$ for all $s : I\!N \to A$ and $n \geq 0$.

The equations

- $head_\tau(merge(s, s')) = head_\tau(s)$
- $tail_\tau(merge(s, s')) = merge(s', tail_\tau(s))$

define the *zipping* or *merging* of two streams as the unique coalgebra morphism $merge : f \to \tau$ where the G-coalgebra $f : B \to A \times B$ is defined as follows:

- $B =_{def} [I\!N \to A] \times [I\!N \to A]$,
- $head_f(s, s') =_{def} s(0)$ for all $s, s' : I\!N \to A$,
- $tail_f(s, s') =_{def} (s', tail_\tau(s))$ for all $s, s' : I\!N \to A$.

We note that the set of finite lists *and* infinite streams over A can also be modelled as the object I of the initial algebra $\iota : F(I) \to I$ where F is the endofunctor on the category $CPO_\perp$ of cpos with least element and strict continuous functions, defined by $F(B) = A \otimes B_\perp$ where $\otimes$ is the *smash product* and $\_{\perp}$ adds a new least element (cf. [Hag 89], Ex. 5.5). But here we are again in the cpo approach with all its "structural overhead"!

For capturing the idea of coinductive definitions in the coalgebra approach (cf. 1.2) we introduce the **coinduction principle**, which all operations of an action type have to satisfy (cf. 4.6). It ensures that the behavioural equivalence induced by our restricted notion of a context is compatible with those operations and thus a *congruence* relation. Previous approaches get this property automatically at the expense of the deductive tractability of their more general definition of a context (cf. 1.1).

Operations whose range sort is primitive are called **attributes**, those whose range sort is non-primitive are called **methods**. For instance, the concatenation $@ : list \times stream \to stream$ of a list and a stream and the function $merge : stream \times stream \to stream$ (see above) are methods of an action stream type. A coinductive specification of these operations reads as follows:

```
vars      x:entry  L:list  s,s':stream
axioms    nil@s = s
          (x&L)@s = x&(L@s)
          merge(s,s') = head(s)&merge(s',tail(s))
```

For constructing the final model, an action type must be *consistent* with respect to its primitive subtype. Consistency criteria are given in Section 9.

1.4 Safety and liveness predicates and their axioms

Like functions, predicates of a constructor type SP are usually defined by induction on term structures. If $P_1, \ldots, P_n$ are the predicates of SP, then the definition (the *logic program*) of P_i is the set $AX(P_i)$ of those Horn clause axioms of SP whose leading predicate is P_i. Initial semantics interprets $P_1, \ldots, P_n$ as the **least** relations satisfying $AX(P_1), \ldots, AX(P_n)$. $P_1, \ldots, P_n$ include the equality predicates for which SP implicitly contains congruence axioms (cf. Section 2). Along these lines a first approach to axiomatize behavioural equivalence might be the following. Given that $act_1, \ldots, act_n$ are all actions of an action sort s, add to SP the axiom

$$x \sim y \;\Leftarrow\; act_1(x) \sim act_1(y) \wedge \ldots \wedge act_n(x) \sim act_n(y)$$

and the congruence axioms for $\sim$. But – if there are no other axioms relating hidden-sorted terms – the least relation satisfying these axioms is *syntactic* identity! Hence the initial semantics interpretation of predicates is definitely not suitable for action types. Instead, behavioural equivalence is captured by the **greatest** relation satisfying the inverse of the above axiom:

$$x \sim y \;\Rightarrow\; act_1(x) \sim act_1(y) \wedge \ldots \wedge act_n(x) \sim act_n(y).$$

For instance, in the case of streams this axiom reads as follows (cf. 1.3):

$$s \sim s' \;\Rightarrow\; head(s) = head(s') \wedge tail(s) \sim tail(s').$$

These clauses suggest that $P(x) \Rightarrow gs$ is the general form of an axiom defining a predicate P on action sorts. gs may include "recursive calls" of P that are often given by atoms of the form $P(act_1(\ldots(act_n(x))\ldots))$. Then $P(x) \Rightarrow gs$ means that P holds true for t only if P is valid for the result t' of applying the action sequence $act_1, \ldots, act_n$ to t. In turn, this requires that P is valid for the result t" of applying $act_1, \ldots, act_n$ to t'. Intuitively, P expresses a liveness property of infinite objects used for modelling data flows or state sequences of concurrent systems. Hence we call P a **liveness predicate** and $P(x) \Rightarrow gs$ a **liveness axiom**. For instance, the fairness of a stream s in the sense that s contains infinitely many zeros is captured by the liveness axiom

$$fair(s) \;\Rightarrow\; \exists L, s' : (s \sim L@(0\&s') \wedge fair(s')).$$

Or suppose that a relation $\rightsquigarrow$ defines the evaluation steps of an interpreter for functional expressions (cf. [Gor 95]), $iterate(f, x)$ generates the stream of iterations $x, f(x), f(f(x), \ldots$ and $repeat(x)$ produces the constant stream $x, x, x, \ldots$. Then the partiality of f and the termination of a loop over f can both be stated as liveness axioms:

$$\begin{aligned}
&partial(f) \;\Rightarrow\; \exists x : divergent(f(x)) \\
&divergent(x) \;\Rightarrow\; \exists y : (x \rightsquigarrow y \wedge divergent(y)) \\
&terminates(f, x) \;\Rightarrow\; \exists L, y : iterate(f, x) \sim L@repeat(y).
\end{aligned}$$

A liveness predicate is interpreted as the *greatest* relation satisfying a liveness axiom, while a predicate of a constructor type is interpreted as the *least* relation satisfying a set of Horn clauses. The latter usually expresses a safety property of a finite structure and thus we call it a **safety predicate**. A safety property ensures the overall absence of "something bad", while a liveness property guarantees the eventual occurrence of "something good" (cf. [Lam 94], [Best 95]). Least and greatest interpretations of predicates are well-known in both logic and functional programming (cf., e.g., [Apt 90], [Pau 94], [Gor 95]).

The cpo approach provides the proof principle of fixpoint induction (cf. 1.2) that permits proving only *admissable* properties. A predicate P on a cpo C is admissable iff P holds true for $x \in C$ whenever P is valid for all *finite approximations* of x. Admissability reminds of the usual characterizations of safety and liveness in concurrent systems (cf. [AS 87], [MP 92], [AL 93a]). Here P is called a safety property iff P is *finitely refutable*, i.e. P does *not* hold for a stream s *only if* (any infinite extension of) some finite prefix of s does not satisfy P. P is called a liveness property iff P is *infinitely verifiable*, i.e. any finite sequence can be extended to a stream satisfying P.

All this confirms the intuition that liveness properties can only be *expressed in the infinity*, while safety properties are reducible to properties of finite structures. Maybe, this is a more striking distinction than "sometimes good" versus "never bad". What is sometimes good about divergence (see above)? Termination is certainly something good, but it only becomes a liveness property in the above sense when it is expressed in terms of (state) streams. The same applies to *partial*, which refers to the *infinite* transitive closure of $\leadsto$. The association of liveness with greatest and safety with least interpretations also seems to be more justified if we take into account the difference between the underlying domains. In the domain of streams, liveness in the sense of infinite verifiability is not the same as expressability in the infinity. For instance, the property that all elements of a stream satisfy a given condition p is a *safety* property, which, in modal logic, comes up as as the *greatest* solution $\Box P$ of the formula $X \Leftrightarrow P \wedge \circ X$. An instance of a safety property may be a liveness property: fairness (see above) is given $\Box\Diamond is0$ (cf. Section 1.5). In our approach, $\Box P$ were interpreted as the greatest relation Q satisfying the *liveness* axiom $Q(s) \Rightarrow P(head(s)) \wedge Q(tail(s))$. Hence we call $\Box P$ *and* $fair$ liveness predicates. Both properties refer to *whole* streams, i.e. can only be expressed in the infinity.

Since safety and liveness properties may exclude each other, [DW 90] calls P a liveness property *for a safety property* Q if each finite list L satisfying Q can be extended to an infinite stream satisfying P *and* Q.

As the above examples suggest, the duality between the least and the greatest interpretation of a safety P and a liveness predicate Q, respectively, has a syntactic counterpart. While all axioms defining P, i.e. all Horn clauses with P in their conclusion, can be combined into a single equivalent formula of the form

$$P(x) \;\Leftarrow\; \exists y_1 : (e_{1,1} \wedge \ldots \wedge e_{1,n_1}) \vee \ldots \vee \exists y_k : (e_{k,1} \wedge \ldots \wedge e_{k,n_k}),$$

a liveness axiom for Q has the same form, but with a reversed arrow:

$$Q(x) \;\Rightarrow\; \exists y_1 : (e_{1,1} \wedge \ldots \wedge e_{1,n_1}) \vee \ldots \vee \exists y_k : (e_{k,1} \wedge \ldots \wedge e_{k,n_k}).$$

Some $e_{i,j}$ may be recursive calls of P or Q, respectively.

As safety properties are associated with finite structures and liveness properties are concerned with infinite structures (cf. [Kin 94]), **inductive** Horn clause definitions are characteristic for constructor types, while liveness axioms define predicates of action types. This also becomes apparent if liveness axioms are simulated by Horn clauses and vice versa. On the deductive level, the duality emerges again, namely as the proof principles of (least) fixpoint induction and (greatest) fixpoint coinduction (cf. Section 1.5).

A simulation of Horn clauses by a liveness axiom is used by logic programmers for coping with negation. The resulting liveness axiom $P(x) \Rightarrow gs$ is called the **only-if-completion** *ONLY(P)* of the logic program (= set of Horn clauses) defining a predicate P (cf. [Apt 90], Sect. 5.5). $\neg P(t)$ holds true iff t does not satisfy the conclusion of *ONLY(P)*. In other words, *ONLY(P)* is valid with respect to *initial* semantics (cf. Lemma 5.14) and hence justifies the *closed world assumption* of logic programming (cf. [Rei 78]). In Section 5, we go the other way around and introduce the **if-completion** of a liveness predicate and show its validity with respect to *final* semantics. If-completions are sets of Horn clauses. Hence a predicate P can be specified in two ways: either by a set C of Horn clauses and then interpreted as the least relation satisfying C, or by a liveness axiom c and then interpreted as the greatest relation satisfying c. P will be the same relation in the final model of c and the initial model of C. For instance, Horn clauses defining the predicate $fact(x,y)$ (y is the factorial of x) read as follows:

$$fact(0,1)$$

$$fact(succ(x), y * succ(x)) \;\Leftarrow\; fact(x,y),$$

while a liveness axiom defining $fact$ is given by:

$$fact(x,z) \;\Rightarrow\; (x = 0 \wedge z = 1) \vee \exists y : (fact(x-1,y) \wedge z = y * x).$$

The interchangeability of Horn clauses and liveness axioms ends when it comes to the question of proof methods. The Horn clauses for P must comprise an inductive definition on the structure of the arguments of P. Otherwise we almost have no chance to prove inductive theorems about P. But only predicates on finite structures, i.e. predicates of constructor types, can be defined inductively. The if-completion of a predicate P on streams or processes is not powerful enough for proving conjectures about P. For instance, the if-completion of $fair$ (see above) is just the inverse of the liveness axiom defining $fair$:

$$fair(s) \;\Leftarrow\; s \sim L@(0\&s') \wedge fair(s').$$

The if-completion of a liveness predicate P does not define P as the original liveness axiom for P does. In particular, it does not provide an induction principle for proving $P(t)$ for a given object t. The general definition and proof principle for action types that complements structural induction for constructor types is **computational induction**. It is well-know from the cpo approach (cf. 1.2) and actually means induction on *proof structure*. Computational or proof induction is reduced to structural or data induction by defining all predicates that cannot be defined by structural induction via auxiliary **approximation predicates**. This leads to the **Horn clause completion** and the **coinductive theory** of an action type T (cf. Section 5). The coinductive theory of T is the inductive theory of the Horn clause completion of T. For instance, $fair$ would there be specified as follows:

$$fair(s) \;\Leftarrow\; \forall i : fair_i(s)$$

$$fair_0(s)$$

$$fair_{i+1}(s) \;\Leftarrow\; s \sim L@(0\&s') \wedge fair_i(s').$$

Behavioural equivalence is just a particular liveness predicate. Its liveness axiom is translated into Horn clauses in the same way as other liveness axioms are compiled:

$$x \sim y \;\Leftarrow\; \forall i : x \sim_i y$$

$$x \sim_0 y$$

$$x \sim_{i+1} y \;\Leftarrow\; act_1(x) \sim_i act_1(y) \wedge \ldots \wedge act_n(x) \sim_i act_n(y).$$

Approximation predicates resemble the approximation functions used in the cpo approach for building up solutions of recursive program schemes. Both approximation functions and predicates fix the number of derivation steps needed to execute a specific recursive function resp. predicate call. Mathematics provides numerous examples of the duality between structural and computational induction. For instance, many equivalences are shown in one direction by structural induction, while computational induction does the job in the other direction. Just think of how one usually proves the correctness on the one hand and the completeness on the other hand of a proof calculus with respect to a model class.

1.5 Further input to this work

Programming language interpreters are natural candidates for action types (cf. Sections 10.3.3 and 10.4). While the programs come as action-sorted ground terms, their interpreter *eval* is given by a collection of observers. *eval* defines a **fully abstract** semantics of the language if it identifies each two programs p and p' such that $eval(c(p)) = eval(c(p'))$ holds true for all program contexts c (cf. [Mil 77]). For functional languages adopting a lazy evaluation strategy, Abramsky's *lazy λ-calculus* provides such an action type in pure form (cf. [Abr 90], [AO 93], [Tur 91]). A λ-term t is said to *converge*, written as $t \Downarrow$, if there is a leftmost β-reduction sequence starting from t into a *weak head normal form*, i.e. into an abstraction $\lambda.t'$, written as $t \Downarrow \lambda.t'$. Conversely, t *diverges*, written as $t \Uparrow$ if t does not converge. The associated equality on closed λ-terms is called *applicative bisimilarity* and can be defined in accordance with the above schema for behavioural equivalences:

$$
\begin{aligned}
& t \sim t' \;\Leftarrow\; \forall i : t \sim_i t' \\
& t \sim_0 t' \\
& t \sim_{i+1} t' \;\Leftarrow\; \forall \text{ closed } \lambda\text{-terms } u : eval(t,u) \approx_i eval(t',u)
\end{aligned}
$$

where

$$
\begin{aligned}
t \approx_i t' \quad &\Longleftrightarrow_{def} \quad t \sim_i t' \vee (t \Uparrow \wedge\, t' \Uparrow), \\
eval(t,u) = t' \quad &\Longleftrightarrow_{def} \quad \exists x, v : t \Downarrow \lambda x.v \wedge v[u/x] = t'.
\end{aligned}
$$

Hence the actions for the lazy λ-calculus are the partial (!) functions $eval(-,u)$. This example deviates from the above schema for behavioural equivalences only in the premise of the clause for $\sim_{i+1}$ where $\approx_i$ occurs instead of $\sim_i$. $\approx_i$ is a weakening of behavioural equivalence, analogously to weak equality relations, which identify all expressions that are *undefined*, i.e. have no interpretations in the respective semantic domain.

As action types are hierarchically based on primitive types (cf. 1.1), all objects of primitive sort that cannot be represented as primitive terms are reasonably called "undefined". Undefined terms arise whenever certain observers are partial. Then behavioural equivalence must be defined as in the case of applicative bisimilarity:

$$
\begin{aligned}
& x \sim y \;\Leftarrow\; \forall i : x \sim_i y \\
& x \sim_0 y \\
& x \sim_{i+1} y \;\Leftarrow\; act_1(x) \approx_i act_1(y) \wedge \ldots \wedge act_n(x) \approx_i act_n(y).
\end{aligned}
$$

$\approx_i$ relates to $\sim_i$ as above where, however, divergence is now defined as follows:

$$t \Uparrow \;\Longleftrightarrow_{def}\; t \text{ is of primitive sort and for all primitive terms } u,\ t \not\equiv u.$$

Hence for all transitions *act* among $act_1, \ldots, act_n$ we can return from $act(x) \approx_i act(y)$ to $act(x) \sim_i act(y)$, while for all observers *act*, $act(x) \approx_i act(y)$ means

$$act(x) \equiv act(y) \vee (act(x) \Uparrow \wedge\, act(y) \Uparrow)$$

where $\equiv$ is the equality associated with the primitive type.

[Par 69] and, recently, [Gor 95] exposed the duality of induction and coinduction in a set-theoretical setting based on the Knaster-Tarski fixpoint theorem (cf. [LNS 82]). Starting out from a set D and a monotone function $\Phi : \wp(D) \to \wp(D)$, the least fixpoint $lfp(\Phi)$ of Φ and the greatest fixpoint $gfp(\Phi)$ of Φ are given by $\cap\{A \subseteq D \mid \Phi(A) \subseteq A\}$ and $\cup\{A \subseteq D \mid A \subseteq \Phi(A)\}$, respectively. Given a subset A of D, these representations of $lfp(\Phi)$ and $gfp(\Phi)$[2] are the basis for (least fixpoint) induction, which means concluding $lfp(\Phi) \subseteq A$ from

[2] If Φ is upward resp. downward continuous, then Kleene's fixpoint theorem (cf. [LNS 82]) yields further characterizations, namely $lfp(\Phi) = \cup_i \Phi^i(\emptyset)$ resp. $gfp(\Phi) = \cap_i \Phi^i(D)$.

$\Phi(A) \subseteq A$, and for coinduction (or greatest fixpoint induction), which means concluding $A \subseteq gfp(\Phi)$ from $A \subseteq \Phi(A)$. Note that the implications are not always reversible, i.e. one may have to **generalize** resp. **specialize** the predicate P presenting A for proving $lfp(\Phi) \subseteq A$ resp. $A \subseteq gfp(\Phi)$. If $\Phi(A) \not\subseteq A$, we must look for a proper subset B of A (presented as a stronger, more general predicate than P) such that $\Phi(B) \subseteq B$ and we can conclude $lfp(\Phi) \subseteq B \subseteq A$. If $A \not\subseteq \Phi(A)$, there must be a proper superset B of A (presented as a weaker, less general predicate than P) such that $B \subseteq \Phi(B)$ and thus $A \subseteq B \subseteq gfp(\Phi)$.

For instance, the set of natural numbers can be defined as $lfp(\Phi)$ where $\Phi : \wp(I\!N) \to \wp(I\!N)$ is given by

$$\Phi(X) \ = \{0\} \cup \{x+1 \mid x \in X\}.$$

Proving $I\!N \subseteq A$ by induction means showing (1) $0 \in A$ and (2) $succ(x) \in A$ whenever $x \in A$. But (1)$\wedge$(2) is equivalent to $\Phi(A) \subseteq A$. Analogously, the Horn clauses

$$\begin{aligned} &ordered(nil) \\ &ordered([x]) \\ &ordered(x :: y :: L) \ \Leftarrow \ x \leq y \wedge ordered(y :: L) \end{aligned}$$

define the set of ordered lists of natural numbers as the set $lfp(\Phi)$ where $\Phi : \wp(List) \to \wp(List)$ is given by:

$$\Phi(X) \ = \{nil\} \cup \{[x] \mid x \in I\!N\} \cup \{x :: y :: L \mid x \leq y \wedge y :: L \in X\}.$$

Hence for proving that a predicate P holds true for all ordered lists it is sufficient to show $P(nil)$, $P([x])$ and $(x \leq y \wedge P(y :: L)) \Rightarrow P(x :: y :: L)$. Remember that the least fixpoint interpretation of logic programs is the one Prolog systems realize.

Conversely, the liveness axiom for $fair$ (cf. 1.4) defines the set of fair streams as $gfp(\Phi)$ where $\Phi : \wp(Stream) \to \wp(Stream)$ is given by

$$\Phi(X) \ = \{s \mid \exists L, s' : (s \sim L@(0\&s') \wedge s' \in X)\}.$$

Suppose that the two streams whose elements alternate between zeros and ones, 01 and 10, shall be shown to be fair. To prove this by coinduction means showing that $A =_{def} \{01, 10\}$ is a subset of $\Phi(A)$. Indeed, we have $01 \sim nil@(0\&01) \in \Phi(A)$ and $01 \sim [1]@(0\&01) \in \Phi(A)$.

In Section 7, the equivalences

$$lfp(\Phi) \subseteq A \ \Leftrightarrow \ \exists B : \Phi(B) \subseteq B \subseteq A \quad \text{and} \quad A \subseteq gfp(\Phi) \ \Leftrightarrow \ \exists B : A \subseteq B \subseteq \Phi(B)$$

will be expressed as (backward) deduction rules for Gentzen clauses, called **fixpoint induction** and **fixpoint coinduction**, respectively. The monotonicity of Φ is guaranteed because both the premises of Horn clauses and the conclusions of liveness axioms are $\neg$-free.

Greatest fixpoint interpretations are well-known in the *μ-calculus* that allows us to define sets of states or processes recursively (cf. [Sti 92]). The μ-calculus is based on modal logic, which extends propositional logic to a language for presenting state sets. Given an alphabet *Act* of *actions* and a *transition system* $T = \{\stackrel{act}{\longrightarrow} \subseteq State \times State \mid act \in Act\}$ and an interpretation I of atomic formulas as state sets, modal formulas are interpreted as follows. Let $s \in State$ and $\longrightarrow = \cup T$.

$$\begin{array}{lcl}
s \models \phi & \Leftrightarrow & s \in I(\phi) \text{ for all atomic formulas } \phi \\
s \models \phi \wedge \psi & \Leftrightarrow & s \models \phi \text{ and } s \models \psi \\
s \models \neg\phi & \Leftrightarrow & s \not\models \phi \\
s \models [act]\phi & \Leftrightarrow & s' \models \phi \text{ for all } s \xrightarrow{act} s' \\
s \models \circ\phi & \Leftrightarrow & s' \models \phi \text{ for all } s \longrightarrow s' \\
s \models \Box\phi & \Leftrightarrow & s' \models \phi \text{ for all } s \xrightarrow{*} s' \\
s \models \Diamond\phi & \Leftrightarrow & s' \models \phi \text{ for some } s \xrightarrow{*} s' \\
s \models \mu Z.\phi & \Leftrightarrow & s \in lfp(\Phi) \text{ if } \phi \text{ is } \neg\text{-free} \\
s \models \nu Z.\phi & \Leftrightarrow & s \in gfp(\Phi) \text{ if } \phi \text{ is } \neg\text{-free}
\end{array}$$

where $\Phi : \wp(State) \to \wp(State)$ is defined by $\Phi(X) = \{s \in State \mid s \models \phi_{Z \leftarrow X}\}$. In the last two equivalences ϕ is supposed to $\neg$-free because this guarantees the monotonicity of Φ and thus the existence of $lfp(\Phi)$ and $gfp(\Phi)$ (see above). Without working out all details we claim that each (T, I) can be presented as an action type T' with action set Act[3] and the following function *comp* translates pairs of a state and a modal formula into first order formulas over T' such that $s \models \phi$ iff the final model of T' satisfies $comp(s, \phi)$.

$$\begin{array}{rcl}
comp(s,\phi) & = & P_\phi(s) \text{ for all atomic formulas } \phi \\
comp(s,\phi \wedge \psi) & = & comp(s,\phi) \wedge comp(s,\psi) \\
comp(s,\neg\phi) & = & \neg comp(s,\phi) \\
comp(s,[act]\phi) & = & comp(act(s),\phi) \\
comp(s,\circ\phi) & = & \bigwedge_{act \in Act} comp(act(s),\phi) \\
comp(s,\Box\phi) & = & comp(s,\nu Z.(\phi \wedge \circ Z)) \\
comp(s,\Diamond\phi) & = & comp(s,\mu Z.(\phi \vee \circ Z)) \\
comp(s,\mu Z.\phi) & = & P_Z(s) \wedge c \text{ where } c = (P_Z(x) \Leftarrow comp(x,\phi_{Z \leftarrow P_Z(x)})) \\
comp(s,\nu Z.\phi) & = & P_Z(s) \wedge c' \text{ where } c' = (P_Z(x) \Rightarrow comp(x,\phi_{Z \leftarrow P_Z(x)})).
\end{array}$$

Indeed, if c is a Horn clause and c' is a liveness axiom, the final model would interpret P_Z as the least resp. greatest relation satisfying c resp. c'.

The translation of modal logic into final semantics reveals that the latter is a formal approach to dynamic structures where states *and* transitions reside in a single model. This differs conceptionally from modal logic as well as from *dynamic logic* (cf. [Pra 76]), *temporal logic* (cf. [Kro 87]), *dynamic data types* (cf. [CR 91]), *evolving algebras* (cf. [Gur 91]) and *rewriting logic* (cf. [Mes 93]) where each state has its own model and transitions relate several models to each other. Such multi-level models can often be reduced to action types with sorts for states, state streams or process graphs (cf. Section 10.6). It would be worthwhile to study applications of modal logic such as model checking in concurrent systems in the final framework (cf. [SW 89]). The interpretation of modal logic[4] in final semantics is also suggested by the result that two states or processes are equivalent iff the sets of modal formulas they represent are equal (cf. [HM 85]).

1.6 From models to theories

To sum up, the coalgebra approach tells us how to define functions by coinduction, namely as morphisms into final coalgebras (cf. 1.2). The fixpoint approach enables us to define relations by coinduction, i.e. as greatest

[3] T' must have a predicate P_Z on *State* for each atomic modal formula Z such that $T' \models P_Z(s)$ iff $s \in I(Z)$.
[4] and the quite similar process algebra; cf. Section 10.4

fixpoints of monotone powerset functions (cf. 1.5).

However, both approaches give only model-theoretic characterizations of coinduction. For instance, no axiomatic presentation of streams is given that could justify the above equivalences $01 \sim nil@(0\&10)$ and $10 \sim [1]@(0\&10)$. Similarly, the finality of a coalgebra does not say anything about the structure of its elements. What remains to be done and what is one of the main issues of this paper is a more deduction-oriented presentation of action types and liveness predicates.[5] To this end we need the general schema for specifying contexts and behavioural equivalences that was outlined in Section 1.1. Previous work focused on behavioural equivalences for particular types such as permutative structures (cf. 1.1), functional programming languages or concurrent processes. Only recently, some of the specialized notions have been compared, and their common nature is going to be revealed. For instance, both [AO 93] (Prop. 2.2.4) and [Gor 95] (Thm. 4) state *operational extensionality* results, which show that contextual equivalence and bisimilarity coincide in the lazy λ-calculus and, respectively, in a functional language with products and streams. Contextual equivalence is employed mainly in programming language and data type theory, while bisimilarity was first introduced in process algebra and theories of communicating systems (cf. Section 10.5).

When starting from a transition system T (see above) one usually defines bisimilarity as the greatest relation $\sim$ satisfying the following implication (cf. [Mil 83], [Gor 95]):

$$\begin{array}{rl} s \sim s' \;\Rightarrow\; \forall act \in Act\ \forall t \in State: & (s \xrightarrow{act} t \Rightarrow \exists t' \in State : s' \xrightarrow{act} t' \wedge t \sim t') \\ \wedge & (s' \xrightarrow{act} t \Rightarrow \exists t' \in State : s \xrightarrow{act} t' \wedge t \sim t'). \end{array}$$

If T can be presented as an action type T' with action set Act, then this implication becomes the liveness axiom for behavioural equivalence (cf. 1.4):

$$x \sim y \;\Rightarrow\; \bigwedge_{act \in Act} act(x) \sim act(y).$$

Constructor and action types are more than complementary to each other. The former affect the latter insofar as action types are built upon *visible* (constructor or action) types. More than with a duality we are faced with a dielectic, an instance of Bateson's zigzag ladder of dielectic between form and process (cf. [Bat 79]): constructors constitute forms, actions build up processes.[6] So we actually deal with **swinging specifications**, i.e. chains of specifications, often alternating between constructor types and action types.

1.7 Overview

The paper is organized as follows. Section 2 provides syntactical notions. Section 3 recapitulates many-sorted structures and models. Section 4 defines initial and final semantics and introduces the coinduction principle (cf. 1.3). Def. 4.3 and Thm. 4.9 yield the construction of the final model of an action type, taking into account the two modifications of the classical approach: restricted contexts and liveness predicates. Def. 4.10 combines initial and final models for defining the semantics of a swinging specification. Section 5 deals with the theory of swinging specifications. Thm. 5.8 says that the theory of an action type SP coincides with the inductive theory of the Horn clause completion of SP. In Sections 6 to 9, results of [Pad 92] on *ground confluent* specifications are simplified resp. extended and applied to swinging specifications for obtaining the proof rules listed in Section 7. Swinging specifications are confined stepwise first to *constructor-based*, then to *functional* and finally to *strongly functional* specifications (Sections 6 and 8). All these notions are based on a partition of the set of operations into a set of *constructors* and a set of *defined functions*. Constructors constitute the *normal forms* of a signature. The *consistency* of a specification w.r.t. a subspecification is - besides the coinduction principle -

[5] A complete derivation of the fairness of 01 and 10 is given in Example 7.8.
[6] Readers with a more philosophical touch than the "normal scientist" are invited to meditate about our *swinging conclusion* (Section 11).

needed for defining the final semantics of an action type. Ground confluence, the main property of a functional specification, suggests using the method of *proof by consistency* (cf. [KM 87]) for proving theorems of swinging specifications. Section 9 adapts this method to constructor as well as action types. The criteria for *critical clause convergence* presented in Section 8 play an important rôle when the method is applied to non-trivial proof obligations such as the correctness of an implementation (cf. Example 9.12) or the operation compatibility of behavioural equivalences (cf. Sections 10.2 and 10.5.1). Most of the material presented in Sections 6 to 9 is applicable both to constructor and action types. The proof rules for reasoning about an action type are based on its Horn clause completion. This makes many proof techniques that were developed for constructor types as well amenable to action types. Section 10 discusses various application areas for swinging specifications.

2 Syntax

We assume some familiarity with the basic notions of algebraic specification and first-order logic (cf. e.g. [GTW 78], [EM 85], [Pad 88], [Wir 90]). For the purpose of presenting constructor types along with action types we recapitulate and modify the syntax of many-sorted logic with equality as follows.

Definition 2.1 (signature) A **signature** $SIG = (CS, AS, AC, OP, PR)$ consists of CS of **constructor sorts**, a set AS of **action sorts** and three S^+-sorted sets AC, OP and PR where $S = CS \cup AS$. The elements of AC, OP, $FU = AC \cup OP$ and PR are called **actions**, **operations**, **functions** and **predicates**, respectively.

Given $w \in S^*$ and $s \in S$, $F \in FU_{ws}$ is written as $F : w \to s \in FU$. The length of w is called the **arity of** F. If $w = \epsilon$, then F is called a **constant** of sort s. For all actions $F : w \to s$, $w = s'v$ for some $s' \in AS$ and $v \in CS^*$. An operation $F : w \to s$ with $s \in AS$ is called a **method** or a **procedure**. If $s \in CS$, then F is an **attribute**.

Given $w \in S^+$, $P \in PR_w$ is written as $P : w \in PR$. For each $w \in S^+$, the **equality predicate for** w, $\equiv_w: ww$,[7] is an implicit element of PR. The explicit elements of PR are called **logical predicates**. ❑

Given a signature SIG with sort set S and an S-sorted set X of variables, SIG**-terms over** X or **terms over** SIG **and** X are defined as usual: the elements of X and the functions of SIG provide the function symbols terms are built of. $T_{SIG}(X)$ denotes the set of SIG-terms over X. If $t \in T_{SIG}(X)_s$, then **sort**$(t) = s$. The function **label**: $T_{SIG}(X) \times \mathbb{N}^* \to FU \cup X$ is defined inductively as follows:

- For all $F(t) \in T_{SIG}(X)$, $label(F(t), \epsilon) = F$.
- For all $F(t_1, \ldots, t_n) \in T_{SIG}(X),\ i \in \mathbb{N}$ and $w \in \mathbb{N}^*$, $label(F(t_1, \ldots, t_n), iw) = label(t_i, w)$.

A binary relation R on $T_{SIG}(X)$ is SIG**-compatible** if for all $F : w \to s$ and $t = (t_1, \ldots, t_n), u = (u_1, \ldots, u_n) \in T_{SIG}(X)_w$, $(t_i, u_i) \in R$ for all $1 \le i \le n$ implies $(Ft, Fu) \in R$. A SIG**-congruence** is a SIG-compatible equivalence relation.

For $w = s_1 \ldots s_n \in S^+$ and an S-sorted set A, A_w stands for $A_{s_1} \times \ldots \times A_{s_n}$. For any Cartesian product $\times_{i \in I} A_i$, the projection on A_i is denoted by pr_i.

Given $P \in PR_w$ and $t \in T_{SIG}(X)_w$, Pt is an **atom** over SIG. Hence equations $t \equiv u$ between terms or term tuples t and u are particular atoms. All other atoms are called **logical atoms**.

Definition 2.2 (context) Let SIG be a signature and s be a sort of SIG. An action $F : sw \to s'$ is called an s**-action**. The set of s**-contexts** is inductively defined as follows:

- For all $x \in X_s,\ x : s \to s$ is an s-context.

[7] The index w is omitted if it is clear from the context

- For all s-contexts $c : sv \to s'$, s'-actions $F : s'w \to s'' \in SIG$ and $(x_1, \ldots, x_n) \in X_w$, $F(c, x_1, \ldots, x_n) : svw \to s''$ is an s-context. ❑

An S-sorted function σ from X to $T_{SIG}(X)$ is called a **substitution** over SIG. We write $x\sigma$ for $\sigma(x)$. The **domain of** σ, $dom(\sigma)$, is the set of all variables x such that $x \neq x\sigma$. The **identity substitution** id has an empty domain. Let $V, Z \subseteq X$. σ is a **renaming of** V **away from** Z if $V\sigma \subseteq X$, $|V\sigma| = |V|$ and $V\sigma \cap Z = \emptyset$. The **restricted substitution** $\sigma_V : X \to T_{SIG}(X)$ is defined by $x\sigma_V = x\sigma$ for all $x \in V$ and $x\sigma_V = x$ otherwise. Given a further substitution τ, we write $\sigma =_V \tau$ if $\sigma_V = \tau_V$. For all $t \in T_{SIG}(X)$, the **instance** $t\sigma$ **of** t is obtained from t by replacing each variable x by $x\sigma$. A term or atom t **matches** or **subsumes** or is a **prefix** of a term or atom u, written $t \leq u$, if $t\sigma = u$ for some σ. t **unifies** with u if $t\sigma = u\sigma$ for some σ. σ **subsumes** a substitution τ, written $\sigma \leq \tau$, if for all $x \in X$, $x\sigma \leq x\tau$. σ is a **most general** or **minimal** substitution satisfying a predicate P on substitutions if σ subsumes all substitutions satisfying P. Given two substitutions σ and τ, the **composed substitutions** $\sigma\tau$ and $\sigma + \tau$ are defined by $x(\sigma\tau) = (x\sigma)\tau$ for all $x \in X$, $x(\sigma + \tau) = x\sigma$ for all $x \in dom(\sigma)$ and $x(\sigma + \tau) = x\tau$ for all $x \in dom(\tau)$.

Let $V, Z \subseteq X$. A **goal** is either the constant $FALSE$ or an expression h of the form $\exists V\forall Zg$ where g is a conjunction (often regarded as a set) of atoms and $\forall Zg$ is a short notation for the conjunction $\wedge\{g\sigma \mid \sigma \in T^X_{SIG},\ dom(\sigma) = Z\}$. The elements of V are the **existential** or **output variables** of h, while $\mathbf{in(h)} =_{def} var(h) \setminus V$ is the set of **universal** or **input variables** of h. A substitution σ replaces only input variables, i.e. $h\sigma =_{def} \exists V(\forall Zg)\sigma_{in(h)}$. $\mathbf{EQ(\sigma)}$ denotes the goal $\wedge\{x \equiv x\sigma \mid x \in dom(\sigma)\}$.

A **goal set** is a disjunction (often regarded as a set) of goals. A goal set of the form $gs \vee FALSE$ is identified with gs. Given a goal set gs and a goal h, the expression $gs \Leftarrow h$ is called a **Gentzen clause** over SIG. If $h = \emptyset$, we write gs instead of $gs \Leftarrow h$. $\neg h$ is a short notation for $FALSE \Leftarrow h$. $gs \Leftarrow h_1 \vee \ldots \vee h_n$ denotes the conjunction of all Gentzen clauses $gs \Leftarrow h_i$, $1 \leq i \leq n$. Given a non-empty goal $g \neq FALSE$ without output variables, the Gentzen clause $g \Leftarrow h$ is called a **Horn clause**. $g \Leftarrow h$ is a **conditional equation** if g consist of an equation. Otherwise $g \Leftarrow h$ is called a **non-equational Horn clause**.

Free variables of a Gentzen clause $c = gs \Leftarrow h$ that occur in h, but not in gs, are automatically existential variables of h. Free variables that occur in gs, but not in h, are automatically universal variables of gs. Free variables of both gs and h are automatically universal variables of the entire clause c. Quantifiers that bind an empty set of bounded variables are usually omitted.

For all above-defined expressions t, $\mathbf{var(t)}$ denotes the set of all variables occurring in t. If $x \in var(t)$, we also write $t(x)$ for t and $t(x\sigma)$ for $t\sigma$ if $dom(\sigma) \subseteq \{x\}$. t is **ground** if $var(t) = \emptyset$. The set of ground SIG-terms is denoted by T_{SIG}. A substitution σ is **ground** if $dom(\sigma)\sigma \subseteq T_{SIG}$. We write T^X_{SIG} for the set of ground substitutions over SIG. $t\sigma$ is defined as above. $\mathbf{t}[t_1/s_1, \ldots, t_n/s_n]$ denotes the expression obtained from t by replacing all occurrences of the symbol s_i by the expression t_i, $1 \leq i \leq n$.

Gentzen clauses combine the "naturality" of arbitrary first-order formulas with the proof orientation of the clausal form used in automated theorem proving (ATP). Since each first-order formula can be transformed into an equivalent set of Gentzen clauses, we do not restrict the expressive power of first-order logic by keeping to Gentzen clauses. Fortunately, most proof obligations occurring in data type verification, such as pre/postconditions and invariants, need not be transformed at all. They originally come as Gentzen clauses. On the one hand, Gentzen clauses are implications and thus can be proved or disproved in a natural-deduction style. On the other hand, the clausal form of premises and conclusions allows us to employ proof rules that are similar and at least as powerful as those used in ATP.

Given an atom $P(x)$ with $x \in X^+$, a Gentzen clause $gs \Leftarrow P(x)$ with $in(gs) \subseteq var(x)$ is called a **liveness axiom for** P and mostly written as $P(x) \Rightarrow gs$. Important examples of liveness axioms are the only-if-completions of logic programs (cf. 1.4). Conversely, a liveness axiom for P can be turned into Horn clauses by constructing the *if-completion* of P (cf. 5.10). For instance, the if-completion of the Gentzen clause

$$s \subseteq s' \;\Rightarrow\; \forall\{x\}\ not(contains(s,x))or\ contains(s',x) \equiv true$$

defining set inclusion is given by:

$$s \subseteq s' \;\Leftarrow\; \forall\{x\}\ not(contains(s,x))or\ contains(s',x) \equiv true.$$

Definition 2.3 (specification and extension) A **(Gentzen clause) specification** SP is a pair (SIG, AX) consisting of a signature $SIG = (CS, AS, AC, OP, PR)$ and a set AX of Gentzen clauses over SIG. AX is the set of **axioms of** SP. Let $S = CS \cup AS$ and $FU = AC \cup OP$. SP is called **inhabited** if for all $s \in S$, $T_{SIG,s}$ is non-empty. SP is called a **Horn clause specification** if all axioms of SP are Horn clauses. Any expression over SIG and X is also called an expression **over** SP.

A specification SP' is an **extension of** SP if SP is componentwise included in SP' and if the following conditions hold true: Let S be the set of sorts of SP.

- For all $w \in S^+$, SP (implicitly) includes a predicate $Def_w : w$, called the **definedness predicate for** w.
- For all $s \in S$, $SP' \setminus SP$ (implicitly) contains a constant $\bot_s : \epsilon \to s$, called the **bottom constant for** s.[8]
- The only axioms of SP' with definedness predicates in their conclusions are given by:

$$Def_s(F(x)) \;\Leftarrow\; Def_w(x) \quad \text{for all functions } F : w \to s \text{ of } SP,$$
$$Def_w(x_1,\dots,x_n) \;\Leftarrow\; Def_{s_1}(x_1) \wedge \dots \wedge Def_{s_n}(x_n) \quad \text{for all } w = s_1 \dots s_n \in S^+.$$

Moreover, SP' is a **constructor extension** and the logical predicates of $SP' \setminus SP$ are called **safety predicates** if the sorts and axioms of $SP' \setminus SP$ are constructor sorts and Horn clauses, respectively. SP' is an **action extension** and the logical predicates of $SP' \setminus SP$ are called **liveness predicates** if

- all sorts of $SP' \setminus SP$ are action sorts,
- all actions of $SP' \setminus SP$ are s-actions for some $s \in SP' \setminus SP$ (cf. 2.2),
- each axiom of $SP' \setminus SP$ is either a liveness axiom for some liveness predicate of $SP' \setminus SP$ or a Horn clause whose premise does not contain (liveness or equality) predicates of $SP' \setminus SP$. ❏

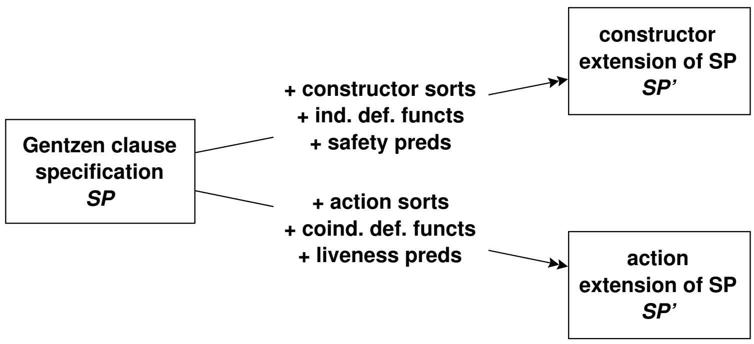


Figure 2. Constructor and action extensions[9]

[8]Sort indices are omitted if they are clear from the context.
[9]A double-headed arrow denotes an inclusion.

With the help of definedness predicates and bottom constants SP may be extended by *partial functions*. Semantical conditions on SP' to ensure that this works correctly are given in subsequent sections.

Note that liveness predicates may be defined on action sorts as well as constructor sorts. In particular, an action extension may add no action sorts, but only liveness predicates to a base specification.

The condition on the axioms of an action extension implies that all equations occurring in premises of non-liveness axioms are of primitive sort. This assumption is crucial in [Wir 90], Thm. 5.4.5, which establishes final models of hierarchical types.[10] Similarly, we need this assumption for concluding that the final model Fin of SP' satisfies $AX' \setminus AX$ (cf. Thm. 4.9). Fin can be presented as the quotient of Ini, the *initial* SP'-model, by behavioural equivalence. If SP' contains only unconditional axioms, then $Mod(SP')$ is closed under quotient constructions. This need not hold if conditional axioms are involved unless all premise predicates are primitive. [BH 95a] call a class C of SIG'-algebras *behaviourally consistent* w.r.t. a family of (partial) congruences $\sim = \{\sim_A \mid A \text{ is a } SIG'\text{-algebra}\}$ if for all $A \in C$, $A/\sim_A \in C$. If $\sim_A$ is defined as behavioural equivalence on A, the syntactic restriction of $AX' \setminus AX$ ensures that $Mod(SP')$ is behaviourally consistent w.r.t. $\sim$, in particular $Fin \cong Ini/\sim \in Mod(SP')$.

Syntactically, constructor and action extensions, respectively, extend an arbitrary specification as in Fig. 2.

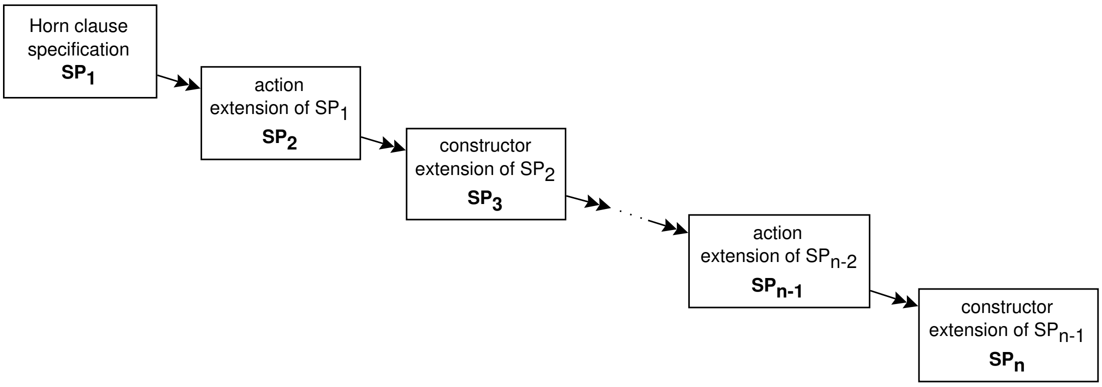


Figure 3. A swinging specification[11]

Definition 2.4 (swinging specification) A **swinging specification** is a sequence $(SP_1, \ldots, SP_n)$ of inhabited specifications such that SP_1 is a Horn clause specification and for all $2 \leq i \leq n$, SP_i is either a constructor extension or an action extension of SP_{i-1}. ❏

Example 2.5 (lists and streams) An inhabited specification

$$SP_1 = ((CS_1, AS_1, AC_1, OP_1, PR_1), AX_1)$$

is given by:

- $CS_1 = \{nat, list\}$
- $AS_1 = AC_1 = \emptyset$
- $OP_1 = \{0 : \epsilon \rightarrow nat,\ succ : nat \rightarrow nat,\ + : nat\ nat \rightarrow nat,\ nil : \epsilon \rightarrow nat,\ \& : nat\ list \rightarrow list\}$
- $PR_1 = \{< : nat\ nat.\ prefix : list\ list\}$

[10] The proof of this result can be found in [BW 82] (Cor. 4).

[11] Constructor extensions need not always alternate with action extensions. Def. 2.4 admits successive constructor extensions as well as successive action extensions.

- $AX_1 = \{0 + n \equiv n,\ succ(m) + n \equiv succ(m + n),\ 0 < succ(n),\ succ(m) < succ(n) \Leftarrow m < n,$
  $prefix(nil, L),\ prefix(n\&L, n\&L') \Leftarrow prefix(L, L')\}$

An action extension $SP_2 = ((CS_2, AS_2, AC_2, OP_2, PR_2), AX_2)$ of SP_1 is defined as follows:

- $CS_2 = CS_1$
- $AS_2 = \{stream\}$
- $AC_2 = \{head : stream \rightarrow nat,\ tail : stream \rightarrow stream\}$
- $OP_2 = OP_1 \cup \{\& : nat\ stream \rightarrow stream,\ 01 : \epsilon \rightarrow stream,\ 10 : \epsilon \rightarrow stream,\ nats : nat \rightarrow stream,$
  $@ : list\ stream \rightarrow stream,\ merge : stream\ stream \rightarrow stream\}$
- $PR_2 = PR_1 \cup \{fair : stream\}$
- $AX_2 = AX_1 \cup \{head(n\&s) \equiv n,\ tail(n\&s) \equiv s,\ 01 \equiv 0\&10,\ 10 \equiv 1\&01,\ nats(n) \equiv n\&nats(succ(n)),$
  $nil@s \equiv s,\ (n\&L)@s \equiv n\&(L@s),\ merge(n\&s, s') \equiv n\&merge(s', s),$
  $fair(s) \Rightarrow \exists\{L, s'\}(s \equiv L@(0\&s') \wedge fair(s'))\}$

Let us reformulate SP_1 and SP_2 using an Expander-like syntax (cf. [Pad 94]):

```
SP_1
     sorts     nat  list
     opns      0:nat
               succ:nat->nat
               +:nat*nat->nat
               nil:list
               &:nat*list->list
     preds     <:nat*nat
               prefix:list*list
     infixes   +  &  <
     vars      m,n:nat  L,L':list
     axioms    0+n = n
               succ(m)+n = succ(m+n)
               0 < succ(m)
               succ(m) < succ(n) <== m < n
               prefix(nil,L)
               prefix(n&L,n&L') <== prefix(L,L')
```

With respect to the initial SP_1-model (cf. 3.2), $prefix(L, L')$ holds true iff L is a prefix of L'.

```
SP_2
     base      SP_1
     sorts     stream
     acts      head:stream->nat
               tail:stream->stream
     opns      &:nat*stream->stream
               01,10:stream
               nats:nat->stream
               @:list*stream->stream
               merge:stream*stream->stream
     preds     fair:stream
     infixes   & @ merge
     vars      s,s':stream
     axioms    head(n&s) = n
               tail(n&s) = s
               01 = 0&10
               10 = 1&01
```

```
nats(n) = n&nats(succ(n))
nil@s = s
(n&L)@s = n&(L@s)
merge(n&s,s') = n&merge(s',s)
fair(s)  ==>  \/{L,s'}(s = L@(0&s') /\ fair(s'))
```

& appends a number to the left of a stream. 01 and 10 are the two streams whose elements alternate between zeros and ones. $nats(n)$ generates the stream of all numbers starting from n. @ concatenates a list and a stream into a stream. $merge$ merges two streams into a single stream by alternatively appending an element of one stream to an element of the other stream. $fair(s)$ holds true if s contains infinitely many zeros. $\wedge$ denotes a universal quantifier or a conjunction combining atoms to goals. $\vee$ denotes an existential quantifier or a disjunction combining goals to goal sets. Finally, we "swing" to a constructor extension of SP_2:

```
SP_3
     base      SP_2
     sorts     streamlist
     opns      nil:streamlist
               &:stream*streamlist->streamlist
```

(SP_1, SP_2, SP_3) is a swinging specification. ❏

3 Semantics

As usual, the semantics starts out from SIG-structures and a satisfaction relation between formulas and SIG-structures.

Given a signature SIG with sort set S, a SIG**-structure** A (with identity) consists of an S-sorted set, the **carrier set** of A, also denoted by A, a function $F^A : A_w \to A_s$ for each function $F : w \to s \in SIG$ and a relation $P^A \subseteq A_w$ for each predicate $P : w \in SIG$ such that for all $s \in S$, $\equiv_s^A$ is the equality relation on A_s. Given two SIG-structures A and B, a SIG**-homomorphism** is an S-sorted function $h : A \to B$ such that for each function $F : w \to s \in SIG$, $F^B \circ h_w = h_s \circ F^A$ and for each predicate $P : w \in SIG$, $h(P^A) \subseteq P^B$. A and B are SIG**-isomorphic** if $g \circ h = id_A$ and $h \circ g = id_B$ for some SIG-homomorphism $g : B \to A$.

Let SIG' be a subsignature of SIG. The SIG'**-restriction of** A, $A_{SIG'}$, is the SIG'-structure defined by $A_{SIG',s} = A_s$ for all sorts $s \in SIG'$ and $F^{A_{SIG'}} = F^A$ for all functions and predicates $F \in SIG'$.

The interpretation of SIG-terms in A depends on a **valuation of variables in** A, i.e. an S-sorted function $b : X \to A$. The unique SIG-homomorphism extending b to a function from $T_{SIG}(V)$, $V \subseteq X$, to A is denoted by b^*. If $V = \emptyset$, then b^* agrees with the **evaluation** of ground terms in A, which is denoted by $eval^A : T_{SIG} \to A$. For all $t \in T_{SIG}$, we write t^A instead of $eval^A(t)$. If $eval^A$ is surjective, then A is called **reachable** or **finitely generated**. Each SIG-structure A has a unique reachable substructure, denoted by **gen(A)**.

A valuation $b : X \to A$ **solves an atom** $p = P(t_1, \ldots, t_n)$ over SIG, written: $A \models_b p$, if $(b^*t_1, \ldots, b^*t_n) \in P^A$. b **solves a goal** $\exists V g$, written: $A \models_b \exists V g$, if there is a valuation c such that for all $p \in g$, $A \models_c p$ and for all $x \in X \setminus V$, $c(x) = b(x)$. b **solves a goal set** gs, written: $A \models_b gs$, if b solves some $g \in gs$.

A **satisfies an atom, goal, etc.,** g, or g **is valid in** A, written $A \models g$, if all valuations of variables in A solve g. By definition, the contradictory goal $FALSE$ is not satisfiable. A **satisfies a Gentzen clause** $gs \Leftarrow h$ if for all valuations $b : X \to A$, $A \models_b h$ implies $A \models_b gs$. The set of all Gentzen clauses satisfied by A is called the **theory Th(A) of** A.

Let SIG' be a subsignature of SIG. The set of all Gentzen clauses satisfied by $gen(A_{SIG'})$ is called the SIG'**-theory** $\mathbf{Th}_{SIG'}\mathbf{(A)}$ **of** A.

Let $SP = (SIG, AX)$ be a specification. A SIG-structure A is an SP**-model** if A satisfies all elements of AX. The class of SP-models is denoted by **Mod(SP)**.

Definition 3.1 (initial and final structures) Given a class C of SIG-structures, $I \in C$ is **initial w.r.t.** C if for all $A \in C$ there is a unique SIG-homomorphism $ini^A : I \to A$. $T \in C$ is **final** or **terminal w.r.t.** C if for all $A \in C$ there is a unique SIG-homomorphism $fin^A : A \to T$. ❑

The initial semantics of Horn clause specifications is based on the following theorem.

Theorem 3.2 (existence of initial models) ([GM 86], Thm. 1; [Pad 88], Thm. 4.4.3) Let SP be a Horn clause specification. Then there is (up to SIG-isomorphism) a unique initial SIG-structure w.r.t. $Mod(SP)$, called the **initial** SP**-model Ini(SP)**. ❑

The choice of $Ini(SP)$ as *the* semantics of a Horn clause specification has been thoroughly motivated in the literature (cf. e.g. [MG 85], [EM 85], [Pad 88]). The initial SP-model both follows the intuition of a "canonical" structure representing a data type and makes reasoning about the data type equivalent to inductive theorem proving. For the second consequence, however we must restrict our attention to inhabited specifications (cf. Def. 2.3). Otherwise certain invalid goals would become provable (cf. [MG 85]).

Initial semantics does not rule out neither functional sorts nor polymorphic types, not even parametric specifications. For handling higher-order functions such as map, $fold$ and $filter$ (cf. [BW 88]) we add functional sorts $w \to s$ to S and the inductive definition of $T_{SIG}(X)$ by function application: if $t \in T_{SIG}(X)_{w\to s}$ and $t' \in T_{SIG}(X)_w$, then $(t\ t') \in T_{SIG}(X)_s$. A SIG-structure A interprets a functional sort $w \to s$ as the function space $[A_w \to A_s]$. Thm. 3.2 remains valid provided that $Mod(SP)$ is replaced by the class of all finitely generated SP-models (cf. [MTW 88]).

Polymorphic types come up if S is given its own term structure with sort constructors such as $\times$ (product), $\to$ (function space) and sort variables, which either stand for arbitrary sorts (the polymorphic case) or for sorts obeying certain *constraints* on functions or predicates that operate only on sort variables (the parametric case). If SP specifies a polymorphic or parametric type, the semantics of SP is given by a class of initial models of inhabited extensions SP' of SP. Consequently, proofs of conjectures about SP must not refer to any special property of SP'. Only the axioms and constraints of SP may be taken into account (see also Sect. 10.1).[12]

$Ini(SP)$ is usually presented as the quotient of T_{SIG} by the SIG-congruence consisting of all equations that are derivable from SP via the *cut calculus for SP*:

Definition 3.3 (cut calculus) Given a specification $SP = (SIG, AX)$ with sort set S, the **cut calculus for** SP consists of two rules for deriving Horn clauses from Horn clause axioms of SP and SIG-congruence axioms for equality predicates of SP. The latter are Horn clauses as well: Let $w = s_1 \dots s_n \in S^+$ and $x = (x_1, \dots, x_n), y = (x_1, \dots, x_n) \in X_w$.

- $x \equiv_w x$,
- $x \equiv_w y \Leftarrow y \equiv_w x$,
- $F(x) \equiv_s F(y) \ \Leftarrow\ x \equiv_w y$ for all functions $F : w \to s$ of SIG,
- $P(x) \ \Leftarrow\ P(x) \wedge x \equiv_w y$ for all predicates $P : w$ of SIG,
- $x \equiv_w y \ \Leftarrow\ x_1 \equiv_{s_1} y_1 \wedge \dots \wedge x_n \equiv_{s_n} y_n$.

The rules transform one or two Horn clauses into a single Horn clause:

substitution rule $\dfrac{g \Leftarrow h}{g\sigma \Leftarrow h\sigma}$ if σ is a substitution over SIG

[12] For type-theoretical semantic resp. axiomatic foundations of polymorphism, cf., e.g., [MPS 86] and [Tur 91].

cut rule $$\frac{g \Leftarrow h \wedge g', \quad g' \Leftarrow h'}{g \Leftarrow h \wedge h'}$$

We write $SP \vdash_{cut} g \Leftarrow h$ if $g \Leftarrow h$ is derivable from the Horn clause axioms of SP via the cut calculus for SP. The set of all derivable equations $t \equiv t'$ induces a congruence relation $\equiv_{SP}$ on $T_{SIG}(X)$, called SP**-equivalence** and defined by:

$$t \equiv_{SP} t' \iff_{def} SP \vdash_{cut} t \equiv t'. \; ❑$$

Definition 3.4 (inductive theory) Given a specification SP, a Gentzen clause $gs \Leftarrow h$ over SP is called an **inductive theorem of** SP if for all ground substitutions σ over SP, $SP \vdash_{cut} h\sigma$ implies $SP \vdash_{cut} g\tau$ for some $h' = \exists V g \in gs$ and $\tau =_{in(h')} \sigma$. The **inductive theory of** SP, **ITh(SP)**, is the set of all inductive theorems of SP. ❑

In particular, $\neg h$ is an inductive theorem of $SP = (SIG, AX)$ if for all $\sigma \in T^X_{SIG}$, $SP \not\vdash_{cut} h\sigma$, and a Horn clause $g \Leftarrow h$ is in $ITh(SP)$ if for all $\sigma \in T^X_{SIG}$, $SP \vdash_{cut} h\sigma$ implies $SP \vdash_{cut} g\sigma$.

Theorem 3.5 (construction of the initial model) ([Pad 88], Thm. 4.4.3) Let $SP = (SIG, AX)$ be a Horn clause specification. Then the initial SP-model $Ini(SP)$ (cf. 3.2) is (up to SIG-isomorphism) given by the following SIG-structure Ini:

- For all sorts $s \in SIG$, $Ini_s = T_{SIG,s}/\equiv_{SP}$.
- For all functions $F : w \to s \in SIG$ and $([t_1], \ldots, [t_n]) \in Ini_w$,

$$F^{Ini}([t_1], \ldots, [t_n]) \quad =_{def} \quad [F(t_1, \ldots, t_n)].$$

- For all predicates $P : w \in SIG$ and $([t_1], \ldots, [t_n]) \in Ini_w$,

$$([t_1], \ldots, [t_n]) \in P^{Ini} \iff_{def} SP \vdash_{cut} P(t_1, \ldots, t_n). \; ❑$$

This construction reveals the crucial assumption involved when specifications are associated with initial semantics: data must be representable as ground terms. This means we have to focus on inhabited specifications and reachable models. When it comes to effective theorem proving, this will be obvious. The soundness of many efficient proof rules depends on the condition that SP is *ground confluent*, which in turn requires identifying constructors among the functions of SP (cf. Sections 6 and 7).

Thm. 3.5 immediately implies

Fact 3.6 Let SP be an inhabited Horn clause specification. Then for all Gentzen clauses c over SP,

$$Ini(SP) \models c \quad \text{iff} \quad c \in ITh(SP). \; ❑$$

Definition 3.7 (conservative extension and model) Let $SP = (SIG, AX)$ be a specification with sort set S, A be a reachable SP-model, $SP' = (SIG', AX')$ be an extension of SP and B be an SP'-model. B is **consistent w.r.t.** A or **conservative over** A if A is isomorphic to $gen(B_{SIG})$ or, equivalently, for all ground atoms p over SIG, $B \models p$ implies $A \models p$, in other words: all ground atoms in the SIG-theory of B are theorems of A.

$t \in T_{SIG'}$ is **defined**, written as $t \Downarrow$, if $Ini(SP') \models Def(t)$ or, equivalently, if there is a $u \in T_{SIG}$ such that $t \equiv_{SP'} u$. Otherwise t is **undefined**, written as $t \Uparrow$. SP' is **totalized** if for all sorts $s \in SP$ and $t \in T_{SIG',s}$, t is undefined iff $t \equiv_{SP'} \perp$. SP' is SP**-complete** if all $t \in T_{SIG'}$ are defined. A reachable SIG'-structure B is **definedness-preserving** if for all logical predicates $P : w \in SIG$ and $t \in T_{SIG',w}$,

$$B \models P(t) \quad \text{implies} \quad t \Downarrow \quad \text{or} \quad Ini(SP') \models P(t).$$

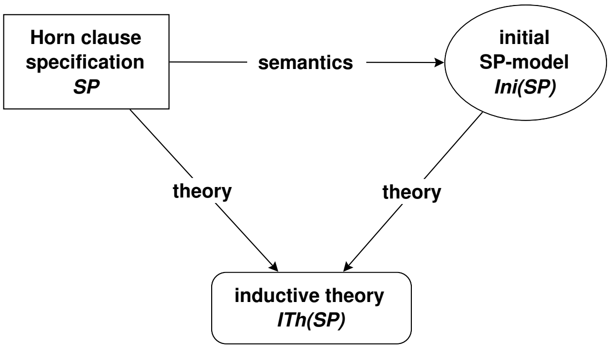


Figure 4. Semantics and theory of a Horn clause specification

CMod(A,SP') denotes the class of definedness-preserving conservative SP'-models over A. ❑

In terms of [WPPDB 83], $CMod(A,SP')$ is a class of *hierarchy-faithful* models over A. Since definedness predicates are logical predicates of SIG, they are interpreted uniformly in all definedness-preserving SP'-models B:

$$Def_s^B \;=\; \{t^B \mid t \in T_{SIG',s},\ t^B = u^B \text{ for some } u \in T_{SIG}\}.$$

Note that $Ini(SP')$ is always definedness-preserving.

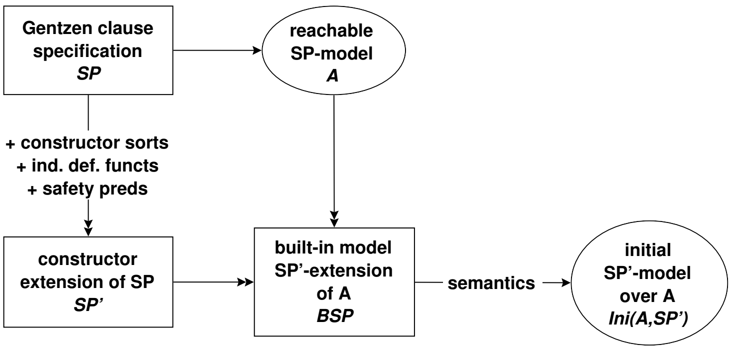


Figure 5. Semantics of a constructor extension

Definition 3.8 (built-in model extension) Let SP be a specification, A be a reachable SP-model, $SP' = (SIG', AX')$ be an extension of SP, HC be the set of Horn clauses of AX' and $HC(A)$ be a set of Horn clauses over SP such that $Th(A) = ITh(SIG, HC(A))$. Then

$$\mathbf{BSP(A,SP')} \;=_{def}\; (SIG', HC(A) \cup HC)$$

is the **built-in model SP'-extension of A**. **Ini(A,SP')**$=_{def}$ $Ini(BSP(A,SP'))$ and **ITh(A,SP')**$=_{def}$ $ITh(BSP(A,SP'))$ are called the **initial SP'-model over A** and the **inductive theory of SP' over A**, respectively. ❑

We could replace the minimal set $HC(A)$ of Horn clauses capturing the theory of A by the set ground atoms over SP that are valid in A. However, the confluence condition that will later be imposed on $BSP(A,SP')$ might enforce a different choice of $HC(A)$. But in any case, even if SP' is an action extension of SP, all axioms of $BSP(A,SP')$ are Horn clauses. Hence the initial SP'-model over A is conservative over A iff $Ini(A,SP')$ is consistent w.r.t. A. The latter property is crucial for the construction of the final structure w.r.t. $CMod(A,SP')$ (cf. Thm. 4.9).

4 Swinging Semantics

Let us now introduce the particular notions that allow us to define the semantics of a swinging specification (cf. 4.10 below).

Assumption 4.1 The rest of this section deals with action extensions. We assume a specification $SP = (SIG, AX)$ with sort set S, a reachable SP-model A and an action extension $SP' = (SIG', AX')$ of SP with sort set S' (cf. 2.3). **BOP** ("base operations") denotes the set of actions of SIG' and operations of SIG. **NOP** ("new operations") is the set of operations of $SIG' \setminus SIG$. **BSP** denotes the built-in model SP'-extension of A. ❑

Definition 4.2 (weak and behavioural equivalence) The **weak equivalence** $\approx_{BSP}$ and the **behavioural equivalence** $\sim_{BSP}$ are S'-sorted relations on $T_{SIG'}$ defined as follows: Let $s' \in S'$ and $t, t' \in T_{SIG',s'}$.

$$t \approx_{BSP} t' \iff_{def} t \equiv_{BSP} t' \vee (t \Uparrow \wedge\, t' \Uparrow)$$

$$t \sim_{BSP} t' \iff_{def} \begin{cases} \text{for all } s \in S,\ s'\text{-contexts } c : s'w \to s \text{ and } (t_1, \dots, t_n) \in T_{SIG',w}, \\ c(t, t_1, \dots, t_n) \approx_{BSP} c(t', t_1, \dots, t_n). \end{cases}$$

$\approx_{BSP}$ and $\sim_{BSP}$ are extended componentwise to S'^+-sorted relations. ❑

Note that for all $s \in S$, $\sim_{BSP,s}$ agrees with $\approx_{BSP,s}$ because the only s-contexts are variables.

If SP' (or BSP) is totalized or SP-complete, then $\approx_{BSP}$ coincides with $\equiv_{BSP}$. In general, $\approx_{BSP}$ includes $\equiv_{BSP}$ and identifies all undefined terms. Building behavioural equivalence on top of weak equivalence instead of $\equiv_{BSP}$ is necessary for making the following construction of a canonical SP'-model a *final* object of $CMod(A, SP')$ (cf. 3.1 and 3.7): while the equivalence of defined terms is determined by the equality relation of A, $CMod(A, SP')$ makes no assumptions about undefined terms. Hence the final object of $CMod(A, SP')$ must identify them.

For ensuring that SP-predicates compatible with $\sim_{BSP}$ (cf. Lemma 4.5(2)) we will require that BSP is totalized. This does not exclude the specification of partial-recursive functions with undecidable domains. The cpo approach (cf. 1.3) interprets recursive functions F as suprema of their approximations F_i, $1 \le i \le n$, and identifies all "non-terminating" calls of F with a bottom element. Without employing the "cpo machinery" a specification of F by axioms of the form $F(t) \equiv u \Leftarrow h$ can be totalized simply by turning these axioms into axioms for the approximations of F:

$$F'(0, t) \equiv \bot,$$
$$F'(succ(i), t) \equiv u[F'(i, t')/F(t')] \;\Leftarrow\; h[F'(i, t')/F(t')],$$

i.e. any subterm $F(t')$ of u or h with leading symbol F is replaced by $F'(i, t)$. F itself is then defined by the Horn clause (!) axioms

$$F(t) \equiv F'(i, t) \;\Leftarrow\; Def(F'(i, t)),$$
$$F(t) \equiv \bot \;\Leftarrow\; \forall\{i\} F'(i, t) \equiv \bot.$$

As we will see in Section 6, specifications with functions should be *functional.* To this end the following theorem must hold true:

$$F'(i, x) \equiv F'(j, x) \;\Leftarrow\; Def(F'(i, x)) \wedge Def(F'(j, x)). \tag{1}$$

A sufficiently powerful criterion for this condition is the *regularity* of all functions (except F' itself) occurring in axioms for F' (cf. 4.4).

Suppose that all these functions are regular and suppose that the axioms for F' cover all possible (ground term) arguments. If $i = j$ or there are no recursive calls of F' in the axioms for F', then (1) holds true trivially.

Otherwise w.l.o.g. let $i < j$. For all ground terms t, $F'(i,t)$ and $F'(j,t)$ are equivalent to terms of the form $v(\perp)$ and $v(v')$, respectively, such that v consists of constants, variables and regular functions. Hence $\lambda x.v(x)$ is regular, too. Therefore, $SP \vdash_{cut} Def(F'(i,t))$ and thus $SP \vdash_{cut} v(\perp)$ imply $F'(i,t) \equiv_{SP} F'(j,t)$.

Definition 4.3 (construction of the final model) The **final** SP'**-model over** A, **Fin(A,SP')**, is defined as the following SIG'-structure Fin:

(4.3.1) For all $s \in S'$, $Fin_s = T_{SIG',s}/\sim_{BSP}$.

(4.3.2) For all functions $F : w \to s \in SP'$ and $([t_1],\dots,[t_n]) \in Fin_w$,
$$F^{Fin}([t_1],\dots,[t_n]) \quad =_{def} \quad [F(t_1,\dots,t_n)].$$

(4.3.3) For all logical predicates $P : w \in SP$ and $([t_1],\dots,[t_n]) \in Fin_w$,
$$([t_1],\dots,[t_n]) \in P^{Fin} \quad \Longleftrightarrow_{def} \quad BSP \vdash_{cut} P(t_1,\dots,t_n).$$

(4.3.4) For all $s \in S'$ and $[t],[t'] \in Fin_s$, $\quad [t] \equiv_s^{Fin} [t'] \iff_{def} t \sim_{BSP} t'$.

(4.3.5) For all liveness predicates of $SP' \setminus SP$, P^{Fin} is defined as the greatest relation such that Fin satisfies all liveness axioms of $SP' \setminus SP$ (cf. Def. 2.3). ❑

The construction of Fin is well-defined only if BSP is a conservative extension of A and $\sim_{BSP}$ is SIG'-compatible (cf. Section 2). The latter condition is guaranteed if NOP is defined by *coinduction* on *head normal forms* (cf. 4.6) and if (A, SP') is *regular*:

Definition 4.4 (regularity) A function $F : s_1 \dots s_n \to s$ is **regular** if for all $1 \le i \le n$,

$$Def(F(x_1,\dots,x_n)) \quad \Rightarrow \quad Def(x_i) \ \vee \ F(x_1,\dots,x_n) \equiv F(x_1,\dots,x_{i-1},x,x_{i+1},\dots,x_n)$$

is an inductive theorem of BSP. (A, SP') is **regular** if BSP is totalized (cf. 3.7) and all $F \in BOP$ are regular. ❑

Regularity captures the continuity of functions on *flat* cpos (cf. 1.3). Let us recall the syntactic restriction we imposed on Horn clause axioms $p \Leftarrow g$ of $SP' \setminus SP$ (cf. Def. 2.3): all predicates of the premise g are in SP. The following lemma shows that regularity of BOP implies BOP-compatibility of behavioural equivalence. For guaranteeing NOP-compatibility we introduce the *coinduction principle* for specifying operations (cf. Def. 4.6 and the proof of Thm. 4.9).

Lemma 4.5

(1) Let $F : w \to s \in BOP$ be regular. Then for all $t,t' \in T_{SIG',w}$, $t \sim_{BSP} t'$ implies $F(t) \sim_{BSP} F(t')$.

(2) Let (A, SP') be regular. Then for all goals $g(x)$ over BOP and predicates of SP, $t \sim_{BSP} t'$ implies $BSP \vdash_{cut} g(t)$ iff $BSP \vdash_{cut} g(t')$.

Proof. (1) Let $w = (s_1,\dots,s_n)$, $t = (t_1,\dots,t_n)$ and $t' = (t'_1,\dots,t'_n)$. If $F \in SP$, then for all $1 \le i \le n$, $sort(t_i) \in SP$ and thus $t_i \sim_{BSP} t'_i$ implies $t_i \approx_{BSP} t'_i$. Hence $F(t) \approx_{BSP} F(t')$ because F is regular, and we conclude $F(t) \sim_{BSP} F(t')$. Otherwise F is an action, i.e. $sort(t_1) \in S' \setminus S$ and for all $2 \le i \le n$, $sort(t_i) \in SP$ and thus $t_i \approx_{BSP} t'_i$. If $s \in SP$, then $t_1 \sim_{BSP} t'_1$ and the regularity of F imply

$$F(t_1,\dots,t_n) \approx_{BSP} F(t'_1,t_2,\dots,t_n) \approx_{BSP} F(t'_1,\dots,t'_n)$$

because $F(x_1,\dots,x_n)$ is an s_1-context. If $s \in S' \setminus S$, then $t_1 \sim_{BSP} u_1$ and the regularity of F imply for all s-contexts $c : sv \to s'$ and $(v_1,\dots,v_k) \in T_{SIG',v}$,

$$c(F(t_1,\dots,t_n),v_1,\dots,v_k) \approx_{BSP} c(F(t'_1,t_2,\dots,t_n),v_1,\dots,v_k) \approx_{BSP} c(F(t'_1,\dots,t'_n),v_1,\dots,v_k)$$

because $c(F(x_1,\dots,x_n),x_1,\dots,x_k)$ is an s_1-context. Hence in both cases $F(t)\sim_{BSP} F(t')$.

(2) follows from (1) if we take into account that for all predicates $P:w$ of SP and $t,t'\in T_{SIG',w}$, $BSP\vdash_{cut} P(t)$ and $t\sim_{BSP} t'$ imply $t\approx_{BSP} t'$ and thus, since BSP is totalized, $t\equiv_{BSP} t'$ so that $BSP\vdash_{cut} P(t')$ holds true as well. ❑

Definition 4.6 (A,SP') **satisfies the coinduction principle** if for all $F:s_1\dots s_n\to s\in NOP$ and $1\le i\le n$ there is $HNF_i\subseteq T_{SIG'}(X)_{s_i}$, called a set of s_i**-head normal forms for** (A,SP'), such that the following conditions hold true:

(1) For all $t\in T_{SIG',s_i}$ there are $t'\in HNF_i$ and $\sigma\in T^X_{SIG'}$ such that $t\equiv_{BSP} t'\sigma$.

(2) For all $t,u\in HNF_i$ and $\sigma,\tau\in T^X_{SIG'}$, $t\sigma\sim_{BSP} u\tau$ implies $t=u$ and $x\sigma\sim_{BSP} x\tau$ for all $x\in var(t)$.

(3) Let $t\in HNF_{s_1}\times\dots\times HNF_{s_n}$ and $V=X\setminus var(t)$. For all s-contexts $c:sv\to s'$ and term tuples $u\in T_{SIG',v}$ there are $k\ge 0$, terms $v_1,\dots,v_k$ over BOP and goals $g_1,\dots,g_k$ over BOP and predicates of SP such that

$$c(F(t),u)\equiv v_1 \ \Leftarrow\ g_1,\ \ \dots,\ \ c(F(t),u)\equiv v_k\ \Leftarrow\ g_k$$

and $\exists V g_1\vee\dots\exists V g_k$ are inductive theorems of BSP. ❑

4.6(1) and 4.6(3) imply that all ground terms of the form $c(F(t,u))$ with context c, $F\in NOP$ and head normal form (tuple) t can be reduced into a term over BOP. In other words, NOP-symbols are removable. NOP may be called *contextually total* functions.

The default set of s-head normal forms is a singleton consisting of a variable of sort s. On the one hand, 4.6(3) were violated for this choice of HNF if some axioms for F depend on the structure of the argument t. On the other hand, if s has non-free constructors (cf. 1.1), then 4.6(2) enforces that all s-head normal forms are variables (cf. 10.2 and 10.4.1).

Example 4.7 (streams) Let (SP_1,SP_2,SP_3) be the swinging specification defined in Ex. 2.5. SP_2 is an action extension of SP_1 with the action sort *stream*. Let s be a variable of sort *stream*. Each *stream*-context $c:w\to s'$ with $s'\in SP_1$ has the form $head(tail^i(s))$ for some $i\in\mathbb{N}$.[13] A set of *stream*-head normal forms for $(Ini(SP_1),SP_2)$ is given by $HNF=\{n\&s\}$. Condition 4.6(2) for HNF reduces to: $n\sigma\&s\sigma\sim_{BSP} n\tau\&s\tau$ implies $SP_2\vdash_{cut} n\sigma\approx_{BSP} n\tau$ and $s\sigma\sim_{BSP} s\tau$.

Let NOP be the set of operations of $SP_2\setminus SP_1$, i.e. $NOP=\{\&,01,10,nats,@,merge\}$. Then 4.6(3) holds true if for all $i\in\mathbb{N}$, $F:w\to stream\in NOP$ and head normal form tuples t, $head(tail^i(F(t)))\equiv_{SP_2} t'$ for some t' over $SP_2\setminus NOP$. This can be shown easily by induction on i with the help of 4.6(1) and the axioms for NOP (cf. Ex. 2.5):

```
head(n&s) = n
tail(n&s) = s
01 = 0&10
10 = 1&01
nats(n) = n&nats(succ(n))
nil@s = s
(n&L)@s = n&(L@s)
merge(n&s,s') = n&merge(s',s)
```

These axioms ensure the coinduction principle because we have chosen HNF accordingly. For instance, the equation for *merge* would violate the coinduction principle if $HNF=\{s\}$. Then the alternative *merge* axiom

```
merge(s,s') = head(s)&merge(s',tail(s))
```

[13] For unary functions f, $f^0(x)=_{def} x$ and $f^{i+1}(x)=_{def} f(f^i(x))$.

would comply with the coinduction principle as the following axiomatizations of the stream-generating function *evnats* and the stream-processing function *evmerge* would also do:

```
evnats(n) = n&evnats(succ(n))  <==  n = 2*k
evnats(n) = evnats(succ(n))  <==  n = succ(2*k)
evmerge(s,s') = head(s)&evmerge(tail(s),s')  <==  head(s) = 2*k
evmerge(s,s') = head(s')&evmerge(tail(s),tail(s'))  <==  head(s) = succ(2*k)
```

Conversely, $HNF = \{n\&s\}$ would force us to specify *evmerge* as follows:

```
evmerge(m&s,n&s') = m&evmerge(s,n&s')  <==  m = 2*k
evmerge(m&s,n&s') = n&evmerge(s,s')  <==  m = succ(2*k)
```

Note that $\exists\{k\}(m = 2*k) \vee \exists\{k\}(m = succ(2*k))$ is an inductive theorem, which is required by Condition 4.6(3) of the coinduction principle. ❏

In process algebra, the reducibility of action-sorted terms to head normal form instances is known as the definition of processes by **guarded recursion** (cf. [BW 90]). The guards are given by constructor-sorted subterms. Here these are the head elements of the streams on the right-hand sides of equational axioms.[14]

Definition 4.8 (action consistency) SP' is **action consistent w.r.t.** A if $Ini(A, SP')$ is consistent w.r.t. A and (A, SP') is regular and satisfies the coinduction principle. ❏

Theorem 4.9 (The final model over A is final in the class of conservative models over A) Suppose that SP' is action consistent w.r.t. A. $Fin(A, SP')$ is a well-defined SIG'-structure. Moreover, $Fin(A, SP')$ is final in the class $CMod(A, SP')$ of conservative SP'-models over A (cf. Def. 3.7).

Proof. Let $Fin = Fin(A, SP')$ (cf. 4.3) is well-defined if

(4.9.1)	$\sim_{BSP}$ is SIG'-compatible.
(4.9.2)	The relations of 4.3.3 are well-defined.
(4.9.3)	The greatest relations required by 4.3.5 do exist.

First we prove 4.9.1. Let $F : w \to s \in SP'$ and $t, u \in T_{SIG',w}$ such that $t \sim_{BSP} u$. We have to show

(4.9.4) $\quad F(t) \sim_{BSP} F(u)$.

If $F \in BOP$, then 4.9.4 follows from Lemma 4.5(1). So let $F \in NOP$ and $w = s_1 \ldots s_n$. By 4.6(1), there are $t', u' \in HNF_{s_1} \times \ldots \times HNF_{s_n}$ and $\sigma, \tau : X \to T_{SIG'}$ such that $t \equiv_{BSP} t'\sigma$ and $u \equiv_{BSP} u'\tau$. Hence $t \sim_{BSP} u$ implies $t'\sigma \sim_{BSP} u'\tau$. By 4.6(2),

(4.9.5) $\quad t' = u'$ and $x\sigma \sim_{BSP} x\tau$ for all $x \in var(t')$.

Let $V = X \setminus var(t')$. By 4.6(3), for all sorts $s' \in SP$, s-contexts $c : sv \to s'$ and term tuples $u \in T_{SIG',v}$ there are $r \geq 0$, terms $v_1, \ldots, v_r$ and goals $g_1, \ldots, g_r$ over BOP and predicates of SP such that

$$c(F(t), u) \equiv v_1 \ \Leftarrow\ g_1, \quad \ldots, \quad c(F(t), u) \equiv v_k \ \Leftarrow\ g_k$$

and $\exists V g_1 \vee \ldots \exists V g_k$ are inductive theorems of BSP. From the last of these theorems we conclude that there are $1 \leq i \leq r$ and $\theta \in T^X_{SIG'}$ such that $BSP \vdash_{cut} g_i\theta$ and $\theta =_{var(t')} \sigma$. Hence by 4.9.5 and Lemma 4.5(2), $BSP \vdash_{cut} g_i\eta$ where $\eta = \tau_{var(t')} + \theta_V$. Therefore,

(4.9.6) $\quad c(F(t'\sigma), u) \equiv_{BSP} v_i\theta \quad$ and $\quad c(F(t'\tau), u) \equiv_{BSP} v_i\eta$

[14] See also Section 10.4.

because $(c(F(t'),u) \equiv v_i \Leftarrow g_i) \in ITh(A,SP')$. By 4.9.5 and Lemma 4.5(1), $v_i\theta \sim_{BSP} v_i\eta$ and thus $v_i\theta \approx_{BSP} v_i\eta$ since v_i is S-sorted. Hence 4.9.6 implies $c(F(t'),u)\sigma \approx_{BSP} c(F(t'),u)\tau$ and thus 4.9.1.

Let $P : w$ be a logical predicate of SP and $t, t' \in T_{SIG',w}$ such that $BSP \vdash_{cut} P(t)$ and $t \sim_{BSP} t'$. Since P is regular, Lemma 4.5(2) implies $BSP \vdash_{cut} P(t')$ and thus 4.9.2.

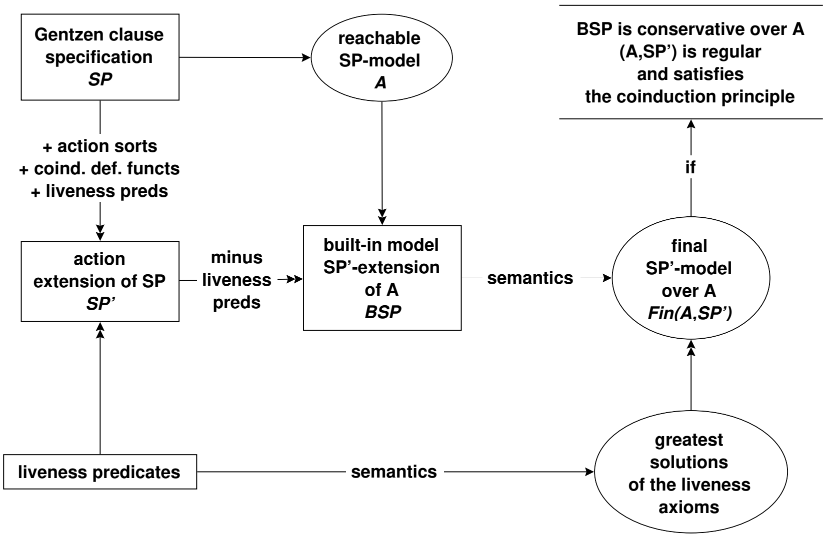


Figure 6. Semantics of an action extension

Next we show 4.9.3. Let **PreFin** be the class of SIG'-structures B satisfying 4.3.1 to 4.3.4 with B instead of Fin. Hence it is only the interpretations of liveness predicates of $SP' \setminus SP$ over which the structures of $PreFin$ vary. These interpretations define a partial order $\leq$ on $PreFin$:

$$B \leq C \iff_{def} \text{ for all predicates } P,\ P^B \subseteq P^C.$$

$PreFin$ has a greatest element All w.r.t. $\leq$: for all liveness predicates $P : w$ of $SP' \setminus SP$, let $P^{All} = Fin_w$. Each descending chain $\{B_i\}_{i \in \mathbb{N}} \subseteq PreFin$ has an infimum $inf(B_i)$: $P^{inf(B_i)} =_{def} \cap_{i \in \mathbb{N}} P^{B_i}$.

A function $\Phi : PreFin \to PreFin$ is defined as follows. For all $B \in PreFin$ and each liveness predicate P of $SP' \setminus SP$,

$$P^{\Phi(B)} =_{def} \{b(x) \mid b : X \to B,\ B \models_b gs \text{ for some liveness axiom } P(x) \Rightarrow gs \text{ of } SP'\}.$$[15]

Φ is monotone and downward continuous w.r.t. $\leq$, i.e. all descending chains $\{B_i\}_{i \in \mathbb{N}} \subseteq PreFin$ satisfy $inf(\Phi(B_i)) \leq \Phi(inf(B_i))$. We conclude from Kleene's fixpoint theorem that $gfp(\Phi) =_{def} inf(\Phi^i(All))$ is the greatest fixpoint of Φ (cf. [LNS 82]). Hence 4.3.5 follows from defining P^{Fin} as $P^{gfp(\Phi)}$ for all liveness predicates P of $SP' \setminus SP$. This finishes the proof of 4.9.3.

By 4.3.1 to 4.3.4, Fin is a quotient of $Ini(A, SP')$ and thus reachable. By 4.3.5, Fin satisfies the liveness axioms of $SP' \setminus SP$. By 2.3, the remaining axioms of SP' are Horn clauses $p \Leftarrow g$ such that g does not contain (liveness or equality) predicates of $SP' \setminus SP$. We must show that $p \Leftarrow g$ is valid in Fin.

Let σ be a ground substitution such that Fin satisfies $g\sigma$. Let $g = \{P_1(t_1), \ldots, P_n(t_n)\}$ and $1 \leq i \leq n$. If P_i is a logical predicate, then $P_i \in SP$ (cf. 2.3). Hence by 4.3.3, $BSP \vdash_{cut} P_i(t_i\sigma)$. If P_i is an equality predicate, then $P_i(t_i) = (u \equiv_s u')$ for some $s \in S$. Hence by 4.3.4, $u\sigma \approx_{BSP} u'\sigma$ and thus $u\sigma \equiv_{BSP} u'\sigma$ because BSP is totalized (cf. 3.7). We conclude that for all $1 \leq i \leq n$, $BSP \vdash_{cut} P_i(t_i\sigma)$, i.e. $BSP \vdash_{cut} g\sigma$. Hence $BSP \vdash_{cut} p\sigma$ and thus $Fin \models p\sigma$ because Fin is a quotient of $Ini(A, SP')$.

[15] W.l.o.g. we assume that P is unary.

Therefore Fin satisfies $p \Leftarrow g$. Consequently, Fin is an SP'-model.

By 4.3.3, Fin is definedness-preserving. We show that Fin is conservative over A. Let $P(t)$ be a ground atom over SP such that $Fin \models P(t)$. If P is logical, then by 4.3.3, $BSP \vdash_{cut} P(t)$. Otherwise $P = \equiv_s$ and $t = (u, u')$ for some $s \in S$ and $u, u' \in T_{SIG}$. Then by 4.3.4, $u \sim_{BSP} u'$ and thus $u \approx_{BSP} u'$ because $s \in S$. Hence $u \equiv_{BSP} u'$ and thus $BSP \vdash_{cut} P(t)$ because $u, u' \in T_{SIG}$ are defined. In both cases we have derived $BSP \vdash_{cut} P(t)$ from $Fin \models P(t)$. Hence $A \models P(t)$ because $Ini(A, SP')$ is consistent w.r.t. A.

Hence Fin is conservative over A.

It remains to show that Fin is final in $CMod(A, SP')$. So assume a conservative SP'-model B over A. Define a function $h : B \to Fin$ by $h(t^B) = t^{Fin}$ for all $t \in T_{SIG'}$. Let SIG'' be the subsignature of SIG' consisting of the sorts and functions of SIG'. Since $eval^B$ is a surjective SIG''-homomorphism and $eval^{Fin}$ is SIG''-homomorphic, h is the unique SIG''-homomorphism with $h \circ eval^B = eval^{Fin}$. For h to be SIG'-homomorphic it remains to show that h is compatible with all predicates of SP'. Hence we are left with two proof obligations:

(4.9.7) h is well-defined.

(4.9.8) For all predicates $P \in SP'$, $h(P^B) \subseteq P^{Fin}$.

First we prove 4.9.7. Let s be a sort of SP' and $t, t' \in T_{SIG',s}$ such that $t^B = t'^B$.

Let $s \in S' \setminus S$. Then for all s-contexts $c : sw \to s'$ and term tuples $v \in T_{SIG',w}$, $c(t, v)^B = c(t', v)^B$.

Case 1. There are ground terms u, u' over SP such that $c(t, v) \equiv_{BSP} u$ and $c(t', v) \equiv_{BSP} u'$. Since B is a BSP-model,

$$u^B = c(t, v)^B = c(t', v)^B = u'^B$$

and thus $u^A = u'^A$ because B is conservative over A. Hence $A \models u \equiv u'$ and thus $BSP \models u \equiv u'$, i.e. $u \equiv_{BSP} u'$. Therefore $c(t, v) \equiv_{BSP} c(t', v)$, and we conclude $t \sim_{BSP} t'$, i.e. $t^{Fin} = t'^{Fin}$.

Case 2. $c(t, v)$ is not defined. If $c(t', v)$ were defined, then $c(t', v) \equiv_{BSP} u$ for some $u' \in T_{SIG}$. Since B is an SP'-model B,

$$c(t, v)^B = c(t', v)^B = u'^B$$

and thus $c(t, v) \equiv_{BSP} u'$ because B is conservative over A. Hence $c(t, v)$ were defined as well. Therefore $c(t, v) \Uparrow$ and $c(t', v) \Uparrow$ and thus $c(t, v) \approx_{BSP} c(t', v)$. We conclude $t \sim_{BSP} t'$, i.e. $t^{Fin} = t'^{Fin}$.

If $s \in S$, then replace $c(t, v)$ by t and $c(t', v)$ by t' in the previous argument, and $t^{Fin} = t'^{Fin}$ can be derived analogously.

Finally we prove 4.9.8. Let $P \in SP$ and $t \in T_{SIG'}$ such that $t^B \in P^B$, i.e. $B \models P(t)$. If $t^B = u^B$ for some $u \in T_{SIG}$, then $t \equiv_{BSP} u'$ for some $u' \in T_{SIG}$ because B is definedness-preserving. Hence $t^B = u'^B$ because B is an SP'-model. Hence $B \models P(t)$ implies $B \models P(u')$, $A \models P(u')$ and thus $BSP \vdash_{cut} P(u')$. Hence $BSP \vdash_{cut} P(t)$ and thus $t^{Fin} \in P^{Fin}$ by 4.3.3. If $t^B \neq u^B$ for all $u \in T_{SIG}$, then t is not defined. Since B is definedness-preserving, $B \models P(t)$ implies $BSP \vdash_{cut} P(t)$ and thus $t^{Fin} \in P^{Fin}$.

Let $s \in S' \setminus S$ and $t, t' \in T_{SIG',s}$ such that $t^B = t'^B$. Hence $t^{Fin} = h(t^B) = h(t'^B) = t'^{Fin}$. We conclude $h(\equiv_s^B) \subseteq \equiv_s^{Fin}$.

Define a $BSIG$-structure $C \in PreFin$ by $P^C = h(P^B)$ for all liveness predicates P of $SP' \setminus SP$. Since h is SIG''-homomorphic (see above) and $h(P^B) \subseteq P^{Fin} = P^C$ holds true for all other predicates of SP', $B \models_b gs$ implies $C \models_{h \circ b} gs$ for all valuations $b : X \to B$ and goal sets gs. Let c be a valuation of variables in C. Then $h \circ b = c$ for some valuation $b : X \to B$. Let $P(x) \Rightarrow gs$ be a liveness axiom of $SP' \setminus SP$ and $c(x) \in P^C$. Then $(h \circ b)(x) \in P^C$ and thus $b(x) \in P^B$ because $P^C = h(P^B)$. Since B satisfies $P(x) \Rightarrow gs$, $B \models_b gs$ and thus $C \models_c gs$. Hence C satisfies $P(x) \Rightarrow gs$. Since Fin is the greatest structure of $PreFin$ (w.r.t. $\leq$) that satifies all liveness axioms of $SP' \setminus SP$, we conclude $h(P^B) = P^C \subseteq P^{Fin}$ for all liveness predicates P of $SP' \setminus SP$.

This finishes the proof that Fin is final in $CMod(A, SP')$. ❑

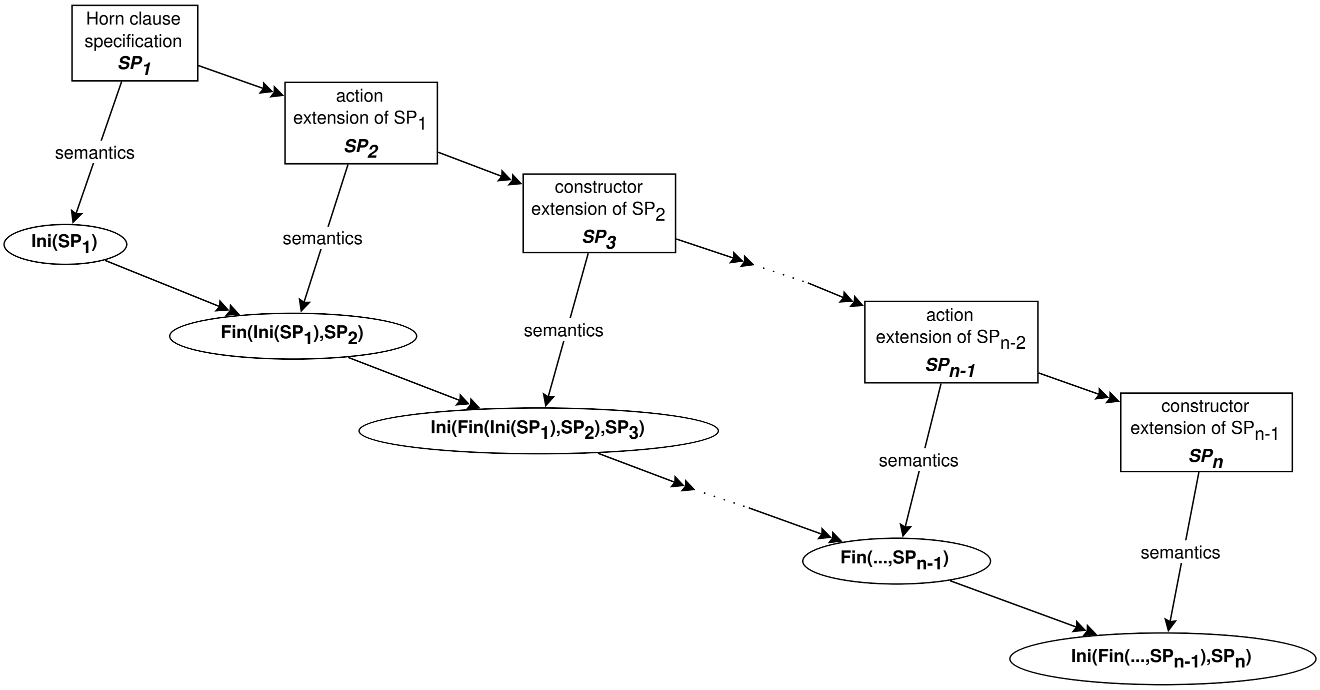


Figure 7. Semantics of a swinging specification

Definition 4.10 (semantics of a swinging specification) Let $SP = (SP_1, \ldots, SP_n)$ be a swinging specification (cf. Def. 2.4). Suppose that the following sequence $Sem = (Sem_1, \ldots, Sem_n)$ of structures is well-defined:

- $Sem_1 = Ini(SP_1)$,
- for all $2 \leq i \leq n$,
  - $Sem_i = Ini(Sem_{i-1}, SP_i)$ if SP_i is a constructor extension of SP_{i-1},
  - $Sem_i = Fin(Sem_{i-1}, SP_i)$ if SP_i is an action extension of SP_{i-1} that is action consistent w.r.t. Sem_{i-1}.

Then Sem is called the **semantics of** SP. ❑

Example 4.11 Let (SP_1, SP_2, SP_3) be the swinging specification of Ex. 2.5. The reader may convince himself that $Sem_1 = Ini(SP_1)$, $Sem_2 = Fin(Sem_1, SP_2)$ and $Sem_3 = Ini(Sem_2, SP_3)$ agree with the "intended" data types specified by SP_1, SP_2 and SP_3, respectively. ❑

5 Swinging Theory

Based on an SP-model A, the semantics of an extension SP' of SP is either the initial SP'-model over A (cf. 3.8) or the final conservative SP'-model over A (cf. 4.3). It follows directly from the definition of the former that for all ground atoms $P(t)$,

$$Ini(A, SP') \models P(t) \quad \text{iff} \quad BSP(A, SP') \vdash_{cut} P(t).$$

Hence, in case of a constructor extension, the cut calculus for $BSP(A, SP')$ (cf. 3.3) allows us to derive *all* ground atoms satisfied by $Ini(A, SP')$. In case of an action extension, we know from Thm. 4.9 that $Fin(A, SP')$ satisfies all axioms of $BSP(A, SP')$. Hence the cut calculus for $BSP(A, SP')$ is correct w.r.t. $Fin(A, SP')$ insofar as only Horn clauses satisfied by this structure can be derived. In analogy to the constructor case we build an extension $HSP(A, SP')$ of $BSP(A, SP')$ such that for all ground atoms $P(t)$,

$$Fin(A, SP') \models P(t) \quad \text{iff} \quad HSP(A, SP') \vdash_{cut} P(t).$$

The definition of $Fin(A, SP')$ (cf. 4.3) suggests that, in fact, additional axioms are needed to derive those ground atoms whose leading predicate P belongs to $SP' \setminus SP$. There are two such cases: P is an equality predicate for a sort of $SP' \setminus SP$ or a liveness predicate of $SP' \setminus SP$.

Definition 5.1 (Horn clause completion) For all sorts $s \in SP$ and $i \in I\!N$, define $\sim_s = \sim_{s,i} = \equiv_s$. For all sorts $s \in SP' \setminus SP$, let $\sim_s : ss$ and $\sim_{s,i} : ss$, $i \in I\!N$, be predicates not occurring in SP', called the **approximation predicates for** $\sim_s$, and $act_1 : sw_1 \to s_1, \ldots, act_n : sw_n \to s_n$ be all s-actions of SP' (cf. 2.2). The **approximation axioms for** $\sim_s$ are the following Horn clauses. Let $i \in I\!N$ and $x, y \in X_s$.

- $x \sim_{s,0} y$,
- $x \sim_{s,i+1} y \;\Leftarrow\; \forall\{z_1, \ldots, z_n\}(act_1(x, z_1) \sim_{s_1,i} act_1(y, z_1) \wedge \ldots \wedge act_n(x, z_n) \sim_{s_n,i} act_n(y, z_n))$.

Let LPR be the set of liveness predicates of $SP' \setminus SP$, $P : w \in LPR$ and $P_i : w$, $i \in I\!N$, be a predicate not occurring in SP', called the **approximation predicates for** P. The **approximation axioms for** P are the following Horn clauses: Let $i \in I\!N$ and $x \in X_w$.

- $P_0(x)$,
- $P_{i+1}(x) \;\Leftarrow\; gs[Q_i/Q \mid Q \in LPR][\sim_s / \equiv_s \;\mid s \in SP' \setminus SP]$[16] for all liveness axioms $P(x) \Rightarrow gs$.

The **Horn clause completion of** (A, SP')**, HSP(A,SP')**, is the least extension of $BSP(A, SP')$ (cf. 3.8) that contains for all predicates $P \in (SP' \setminus SP) \cup \{\sim_s | s \in SP' \setminus SP\}$, the approximation predicates and axioms for P and the Horn clause

$$P(x) \;\Leftarrow\; \bigwedge_{i \in I\!N} P_i(x),$$

called the **gfp-axiom for** P.[17] ❑

In contrast to SP', all axioms of $HSP(A, SP')$ are Horn clauses. By introducing approximation predicates we obtain Horn clause axiomatizations of all liveness predicates of SP' including behavioural equivalence. The Horn clause completion yields a general *construction* of such axiomatizations and not only a *characterization* as [BH 95], Thm. 42 does.

When implementing a Horn clause completion one combines all approximation predicates for a predicate $P : w$ into $P' : nat \times w$ such that $P'(i, x)$ is equivalent to $P_i(x)$.

Example 5.2 (streams) The Horn clause completion of $(Ini(SP_1), SP_2)$ reads as follows (cf. Ex. 2.5):

```
HSP
     base     SP_1
     sorts    stream
     acts     head:stream->nat
              tail:stream->stream
     opns     &:nat*stream->stream
```

[16] cf. Section 2

[17] gfp = greatest fixpoint; cf. the proof of Thm. 4.9) for the interpretation of liveness predicates in $Fin(A, SP')$.

```
          01,10:stream
          nats:nat->stream
          @,merge:stream*stream->stream
preds     ~i:stream*nat*stream
          ~:stream*stream
          fair_i:nat*stream
          fair:stream
infixes   &  @  merge  ~i  ~
vars      s,s':stream
axioms    head(n&s) = n
          tail(n&s) = s
          01 = 0&10
          10 = 1&01
          nats(n) = n&nats(succ(n))
          nil@s = s
          (n&L)@s = n&(L@s)
          merge(n&s,s') = n&merge(s',s)
          s ~0 s'
          s ~succ(i) s'  <==  head(s) = head(s') /\ tail(s) ~i tail(s')
          s ~ s'  <==  /\{i}(s ~i s')
          fair_0(s)
          fair_succ(i)(s)  <==  s ~ L@(0&s') /\ fair_i(s')
          fair(s)  <==  /\{i}fair_i(s)
```

Intuitively, $s \sim_i s'$ holds true iff the prefixes of the streams s and s' consisting of their first i elements coincide. $fair_i(s)$ holds true iff s contains at least i zeros. ❏

Assumption 4.1 (continued) HSP denotes the Horn clause completion of (A, SP'). ❏

Since HSP is a Horn clause specification, we may use the cut calculus (cf. 3.3) for deriving all ground atoms satisfied by $Ini(HSP)$. The main theorem of this section (Thm. 5.8) tells us that these atoms are exactly the ground atoms satisfied by $Fin(A, SP')$ if $Fin(A, SP')$ interprets $\sim_s$ as equality. 5.8 refers to the following two lemmas, which relate the cut-derivable ground atoms $P_i(t)$ to the "i-th approximation" of P^{Fin}. We take over the assumptions of Thm. 4.9.

Lemma 5.3 For each context c, let $\#c$ be the number of actions of c. The **i-th approximation $\sim_i$ of** $\sim_{BSP}$, $i \in I\!N$, is an S'-sorted relation on $T_{SIG'}$ defined as follows: Let s' be a sort of SP' and $t, t' \in T_{SIG',s'}$.

$$t \sim_i t' \quad \Longleftrightarrow_{def} \quad \begin{cases} \text{for all } s \in S,\ s'\text{-contexts } c : s'w \to s \text{ with } \#c = i \text{ and } (t_1, \ldots, t_n) \in T_{SIG',w}, \\ c(t, t_1, \ldots, t_n) \approx_{BSP} c(t', t_1, \ldots, t_n). \end{cases}$$

Suppose that BSP is totalized. Then for all $s \in S'$, $t, t' \in T_{SIG',s}$ and $i \in I\!N$,

$$t \sim_i t' \quad \text{iff} \quad HSP \vdash_{cut} t \sim_{s,i} t'$$

and thus

$$t \sim_{BSP} t' \quad \text{iff} \quad HSP \vdash_{cut} t \sim_s t'.$$

Proof. By the approximation axiom $x \sim_{s,0} y$, $HSP \vdash_{cut} t \sim_{s,0} t'$. $t \sim_0 t'$ holds true because there is no s-context $c : sw \to s'$ with $s' \in SP$ and $\#c = 0$.

Let $i \in I\!N$ and $t \sim_{i+1} t'$. If $s \in S$, then $t \sim_{i+1} t'$ means $t \approx_{BSP} t'$. Since BSP is totalized, $t \approx_{BSP} t'$ is equivalent to $t \equiv_{BSP} t'$ and thus to $HSP \vdash_{cut} t \sim_{s,i+1} t'$.

Let $s \in S' \setminus S$. Then $t \sim_{i+1} t'$ implies

$$c(act(t, u_1, \ldots, u_k), t_1, \ldots, t_n) \ \approx_{BSP} \ c(act(t', u_1, \ldots, u_k), t_1, \ldots, t_n)$$

for all s-actions $act : sv \to s'$, $(u_1,\dots,u_k) \in T_{SIG',v}$, s'-contexts $c : s'w \to s''$ with $s'' \in SP$ and $\#c = i$ and $(t_1,\dots,t_n) \in T_{SIG',w}$. Hence $act(t,u_1,\dots,u_k) \sim_i act(t',u_1,\dots,u_k)$. By induction hypothesis,

$$HSP \vdash_{cut} \forall\{y_1,\dots,y_k\}\ act(t,y_1,\dots,y_k) \sim_{s',i} act(t',y_1,\dots,y_k)$$

and thus $HSP \vdash_{cut} t \sim_{s,i+1} t'$. Conversely, $HSP \vdash_{cut} t \sim_{s,i+1} t'$ implies

$$HSP \vdash_{cut} \forall\{y_1,\dots,y_k\}\ act(t,y_1,\dots,y_k) \sim_{s',i} act(t',y_1,\dots,y_k)$$

for all s-actions $act : sw \to s'$. By induction hypothesis, $act(t,u_1,\dots,u_k) \sim_i act(t',u_1,\dots,u_k)$ for all $(u_1,\dots,u_k) \in T_{SIG',w}$. Hence $t \sim_{i+1} t'$. ❑

Lemma 5.4 Let $PreFin$ be defined as in the proof of Thm. 4.9). For all $i \in I\!N$, $s \in S'$ and all liveness predicates $P \in SP' \setminus SP$, let $B_i \in PreFin$ interpret $\sim_s$ as the equality on Fin_s and P as follows:

$$[t] \in P^{B_i} \quad \Longleftrightarrow_{def} \quad HSP \vdash_{cut} P_i(t).$$

Let $All \in PreFin$ and $\Phi : PreFin \to PreFin$ be defined as in the proof of Thm. 4.9. Then for all $i \in I\!N$, $B_i = \Phi^i(All)$.

Proof. Let LPR be the set of liveness predicates of $SP' \setminus SP$ and $P \in LPR$. First we show by induction on i that $\sim_{BSP}$ is P_i-compatible, i.e. $c_i = (P_i(x) \Leftarrow (P_i(y) \wedge x \sim_s y))$ is an inductive theorem of HSP. c_0 follows from the axiom $P_0(x)$. Let $t,t' \in T_{SIG'}$ such that $HSP \vdash_{cut} \{P_{i+1}(t), t \sim_s t'\}$. By Lemma 5.3, $t \sim_{BSP,s} t'$. $HSP \vdash_{cut} P_{i+1}(t)$ implies $HSP \vdash_{cut} g_i(t)$ where $P_{i+1}(x) \Leftarrow g_i(x)$ is an approximation axiom for P. By 4.9.1, $\sim_{BSP}$ is SIG'-compatible. By induction hypothesis, $\sim_{BSP}$ is P_i-compatible. Hence $HSP \vdash_{cut} g_i(t)$ implies $HSP \vdash_{cut} g_i(t')$ and thus $HSP \vdash_{cut} P_{i+1}(t')$ by applying the approximation axiom $P_{i+1}(x) \Leftarrow g_i(x)$. Therefore $\sim_{BSP}$ is P_{i+1}-compatible.

From this we conclude that P^{B_i} is well-defined: if $[t] = [t']$ and $HSP \vdash_{cut} P_i(t)$, then $t \sim_{BSP} t'$ and thus $HSP \vdash_{cut} P_i(t')$.

Now we prove $B_i = \Phi^i(All)$ by induction on i. Let $[t] \in P^{B_0}$, i.e. $HSP \vdash_{cut} P_0(t)$. Equivalently, $[t] \in P^{All} = P^{\Phi^0(All)}$. Hence $B_0 = \Phi^0(All)$. By induction hypothesis, assume that $B_i = \Phi^i(All)$. Moreover, suppose that for all $i \in I\!N$,

(5.4.1) $B_{i+1} = \Phi(B_i)$.

Then $B_{i+1} = \Phi(B_i) = \Phi(\Phi^i(All)) = \Phi^{i+1}(All)$. It remains to show 5.4.1. By Lemma 5.3, for all ground terms t,t' over SP',

$$B_i \models t \equiv t' \ \text{ iff } \ Fin \models t \equiv t' \ \text{ iff } \ t \sim_{BSP} t' \ \text{ iff } \ HSP \vdash_{cut} t \sim t'.$$

Hence goal set validity in B_i is characterized as follows where $gs_i =_{def} gs[Q_i/Q \mid Q \in LPR][\sim_s / \equiv_s \mid s \in S' \setminus S]$:

(5.4.2) For all goal sets gs over SP' and $b : X \to B_i$, $B_i \models_b gs$ iff
$B_i \models_c gs$ for some $c : X \to B_i$ with $c(y) = b(y)$ for all $y \in in(gs)$
iff $HSP \vdash_{cut} gs_i\sigma$ for some ground substitution σ such that $[y\sigma] = b(y)$ for all $y \in in(gs)$.

Let $i \in I\!N$ and $[t] \in P^{B_{i+1}}$, i.e. $HSP \vdash_{cut} P_{i+1}(t)$. Equivalently, $HSP \vdash_{cut} gs_i\sigma$ for some liveness axiom $P(x) \Rightarrow gs$ and a ground substitution σ such that $x\sigma = t$. By 5.4.2, this is equivalent to:

(5.4.3) $B_i \models_b gs_i$ for some $b : X \to B_i$ with $b(y) = [y\sigma]$ for all $y \in in(gs)$.

Since $in(gs) \subseteq var(x)$, 5.4.3 is equivalent to:

(5.4.4) $B_i \models_b gs_i$ for some $b : X \to B_i$ with $b(x) = [x\sigma]$.

Since $x\sigma = t$, the definition of Φ implies that 5.4.4 is equivalent to $[t] \in P^{\Phi(B_i)}$. This finishes the proof of 5.4.1. ❏

Lemma 5.5 Suppose that SP' is action consistent w.r.t. A (cf. 4.8). Let $Fin = Fin(A, SP')$. Then for all ground atoms $P(t)$ over SP',

$$Fin \models P(t) \quad \text{iff} \quad HSP \vdash_{cut} P'(t)$$

where $P' =_{def} \sim_s$ if $P = \equiv_s$ for some $s \in SP'$ and $P' =_{def} P$ otherwise.

Proof. Suppose that $Fin \models P(t)$.

Let P be a logical predicate of SP. By 4.3.3, $BSP \vdash_{cut} P(t)$ and thus $HSP \vdash_{cut} P(t)$. Since $P' = P$, we have $HSP \vdash_{cut} P'(t)$. For equations $P(t)$, $HSP \vdash_{cut} P'(t)$ follows from Lemma 5.3.

Let P be a liveness predicate of $SP' \setminus SP$. Define $PreFin$, All, inf and Φ as in the proof of Thm. 4.9. Since $P^{Fin} = P^{inf(\Phi^i(All))}$, $Fin \models P(t)$ implies $[t] \in P^{\Phi^i(All)}$ for all $i \in I\!N$. Hence by Lemma 5.4, $[t] \in P^{B_i}$, i.e. $HSP \vdash_{cut} P_i(t)$. Therefore, $HSP \vdash_{cut} P(t)$ by the gfp-axiom for P. Since $P' = P$, we conclude $HSP \vdash_{cut} P'(t)$.

Conversely, suppose that $HSP \vdash_{cut} P'(t)$. The SP'-model Fin becomes an HSP-model by $Q_i^{Fin} =_{def} Q^{B_i}$ for all predicates $Q \in SP' \setminus SP$ and $i \in I\!N$ (cf. Lemma 5.4),

$$[u] \sim_{s,i}^{Fin} [u'] \quad \Longleftrightarrow_{def} \quad u \sim_i u'$$

for all sorts s of SP', $u, u' \in T_{SIG',s}$ and $i \in I\!N$ and if $\sim_s$ is interpreted as the equality on Fin_s. Since $\sim_{BSP,s} = \cap\{\sim_{i,s} | i \in I\!N\}$, $\sim_{s,i}^{Fin}$ is well-defined. Moreover, Fin satisfies all axioms of HSP and thus all Horn clauses derivable from HSP via the cut calculus for HSP because the cut calculus is sound w.r.t. all HSP-models. Hence $HSP \vdash_{cut} P'(t)$ implies $Fin \models P'(t)$. Therefore, $Fin \models P(t)$ because for all sorts $s \in SP'$, Fin interprets $\sim_s$ and $\equiv_s$ as the same relation, namely the equality on Fin_s. ❏

Example 5.6 Let (SP_1, SP_2) be the swinging specification defined in Ex. 2.5 and $Fin = Fin(Ini(SP_1), SP_2)$. We show that

(5.6.1) $\quad Fin \models fair(01) \wedge fair(10).$

Suppose that for all $i \in I\!N$,

(5.6.2) $\quad Fin \models fair_i(01) \wedge fair_i(10).$

By applying the gfp-axiom for $fair$ (cf. 5.1), 5.6.1 follows from 5.6.2. 5.6.2 is proved by induction on i: $Fin \models \{fair_0(01),\ fair_0(10)\}$ follows from the approximation axioms $fair_0(s)$. We have

$$Fin \models 01 \equiv 0\&10 \equiv nil@(0\&10)$$

and

$$Fin \models 10 \equiv 1\&01 \equiv 1\&(0\&10) \equiv 1\&(nil@(0\&10)) \equiv (1\&nil)@(0\&10)$$

by applying axioms of $HSP(Ini(SP_1), SP_2)$ (cf. Ex. 5.2). By induction hypothesis, $Fin \models fair_i(10)$. Hence

$$Fin \models fair_{i+1}(01) \wedge fair_{i+1}(10)$$

by applying an approximation axiom for $fair$. ❏

Definition 5.7 (coinductive theory) The **behaviour variant** c_{beh} of a (set of) Gentzen clause(s) c over HSP is obtained from c by replacing each equality predicate $\equiv_s$, $s \in S' \setminus S$, by $\sim_s$. Conversely, the **coinductive**

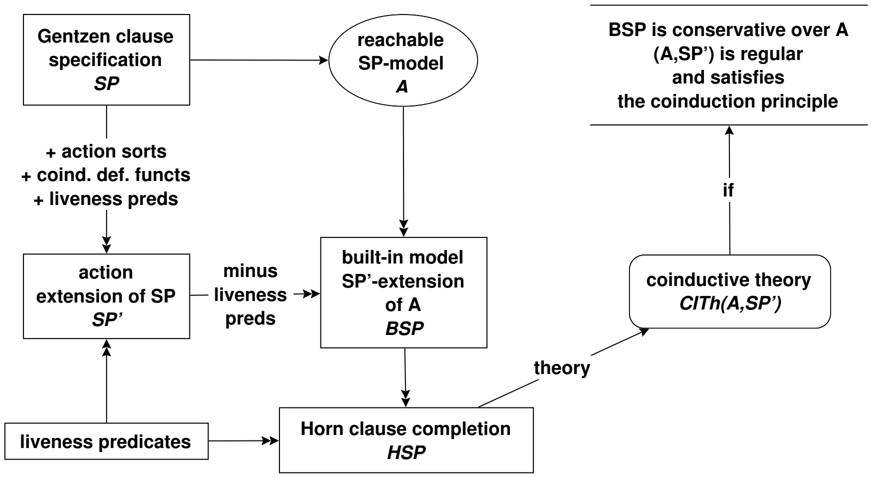


Figure 8. Theory of an action extension (cf. Fig. 6)

theory of (A, SP')**, CITh(A,SP')**, is the set of Gentzen clauses obtained from inductive theorems of HSP by replacing $\sim$ by $\equiv$. ❑

Lemma 5.5 immediately implies:

Theorem 5.8 (adequacy of the Horn clause completion) Suppose that SP' is action consistent w.r.t. A. Then for all Gentzen clauses c over SP',

$$Fin(A, SP') \models c \quad \text{iff} \quad c_{beh} \in ITh(HSP) \quad \text{iff} \quad c \in CITh(A, SP'). \; ❑$$

Since $\sim_{BSP}$ is a congruence relation, all congruence axioms of the cut calculus for SP' (cf. 3.3) are coinductive theorems of (A, SP'). Hence for each goal g and substitutions σ, τ, the **goal congruence clause**

$$g\sigma \;\Leftarrow\; g\tau \cup \{x\sigma \equiv x\tau \mid x \in var(g)\}$$

is a coinductive theorem of (A, SP'). This means that replacing action-sorted subterms of a goal by behaviorally equivalent terms is a sound step in the proof of a coinductive theorem (see the term replacement rule of 7.2). A further consequence of Thm. 5.8 is the following fixpoint characterization of behavioural equivalence:

Definition and Lemma 5.9 (behavioural equivalence is a liveness predicate) Suppose that SP' is action consistent w.r.t. A. Let $Fin = Fin(A, SP')$. For all $s \in S' \setminus S$, $\sim_s^{Fin}$ is the greatest relation such that for all $s \in S' \setminus S$, Fin satisfies the congruence axiom

$$x \equiv_s y \;\Rightarrow\; \forall\{z_1, \ldots, z_n\}(act_1(x, z_1) \equiv_{s_1} act_1(y, z_1) \wedge \ldots \wedge act_n(x, z_n) \equiv_{s_n} act_n(y, z_n)), \qquad (cong_s)$$

called the **liveness axiom for** $\equiv_s$, where $act_1 : sw_1 \to s_1, \ldots, act_n : sw_n \to s_n$ are the s-actions of SP'. The liveness axiom for $\equiv_s$ is an implicitly included in SP'.

Proof. For all $s \in S' \setminus S$, let $R_s \subseteq Fin_s \times Fin_s$ and Fin' be the structure that agrees with Fin except that $\equiv_s$ is interpreted by R_s. Suppose that Fin' satisfies $cong_s$. For proving $R_s \subseteq \equiv_s^{Fin}$ it is sufficient to show $R_s \subseteq \sim_{s,i}^{Fin}$ for all $i \in \mathbb{N}$ (see the proof of Lemma 5.5). Since $\sim_{s,0}^{Fin} = Fin_s \times Fin_s$, the proof is complete for $i = 0$. Let $(a, b) \in R_s$. Since Fin' satisfies $cong_s$, for all $(z_1, \ldots, z_n) \in Fin_{w_1 \ldots w_n}$ and $1 \leq j \leq n$, $(act_j^{Fin}(a, z_j), act_j^{Fin}(b, z_j)) \in R_{s_j}$. By induction hypothesis, $act_j^{Fin}(a, z_j) \sim_{s_j,i}^{Fin} act_j^{Fin}(b, z_j)$. Since Fin satisfies the approximation axioms for $\sim_s$ (cf. 5.1), we conclude that $a \sim_{s,i+1}^{Fin} b$. ❑

Lemma 5.9 reveals that behavioural equivalence is indeed a liveness predicate and $cong_s$ is a liveness axiom for $\equiv_s$ in the sense of 2.3. In other words, $\equiv_s$ is *bisimilarity* or a greatest *bisimulation* in the sense of [Mil 89][18] (cf. Sections 1.6 and 10.5). The inverse of $cong_s$ is called the if-completion of $\equiv_s$:

Definition 5.10 (if-completion) Let P be a liveness predicate of $SP' \setminus SP$ or an equality predicate for some $s \in S' \setminus S$. The set of Horn clauses $P(x) \Leftarrow g$ such that $g \in gs$ for some liveness axiom $P(x) \Rightarrow gs$ of SP' is called the **if-completion $IF(P)$ of P.** ❑

Lemma 5.11 Suppose that SP' is action consistent w.r.t. A (cf. 4.8). Let PR be the set of liveness predicates of $SP' \setminus SP$ and equality predicates for sorts of $S' \setminus S$. Then for all $P \in PR$, the if-completion of P is a coinductive theorem of (A, SP').

Proof. Let $P \in PR$, $P(x) \Leftarrow g \in IF(P)$ and σ be a ground substitution over HSP such that $var(g) \subseteq dom(\sigma)$ and $HSP \vdash_{cut} g_{beh}\sigma$ (cf. Def. 3.4).

Let $P = \equiv_s$ for some $s \in S' \setminus S$ and $act_1 : sw_1 \to s_1, \ldots, act_n : sw_n \to s_n$ be the s-actions of SP'. Then $g_{beh}\sigma = \forall\{z_1, \ldots, z_n\}(act_1(x\sigma, z_1) \sim_{s_1} act_1(y\sigma, z_1) \wedge \ldots \wedge act_n(x\sigma, z_n) \sim_{s_n} act_n(y\sigma, z_n))$. By Lemma 5.5, $Fin(A, SP') \models g_{beh}\sigma$, i.e. for all $1 \le i \le n$ and suitable ground terms t_i over SP', $act_i(x\sigma, t_i) \sim_{BSP} act_i(y\sigma, t_i)$. Hence for all $j \ge 0$, $act_i(x\sigma, t_i) \sim_j act_i(y\sigma, t_i)$ and thus $HSP \vdash_{cut} act_i(x\sigma, t_i) \sim_{s_i,j} act_i(y\sigma, t_i)$ by Lemma 5.3. Therefore, by induction on j and the approximation axioms for $\sim_s$, $HSP \vdash_{cut} x\sigma \sim_{s,j} y\sigma$ for all $j \ge 0$, and thus the gfp-axiom for $\sim_s$ implies $HSP \vdash_{cut} x\sigma \sim_s y\sigma$.

Let LPR be the set of liveness predicates of $SP' \setminus SP$ and $P \in LPR$. Since for all ground atoms $Q(t)$ with $Q \in PR$, $HSP \vdash_{cut} Q(t)$ only if $HSP \vdash_{cut} Q_i(t)$ for all $i \in I\!N$, $HSP \vdash_{cut} g_{beh}\sigma$ implies

$$HSP \ \vdash_{cut} \ g_{beh}[Q_i/Q \mid Q \in LPR]\sigma \ = \ g[Q_i/Q \mid Q \in LPR][\sim_s / \equiv_s \ \mid s \in SP' \setminus SP]\sigma.$$

Hence, by applying approximation axioms for P, we obtain $HSP \vdash_{cut} P_i(x\sigma)$ for all $i \in I\!N$, and thus, by applying the gfp-axiom for P, $HSP \vdash_{cut} P(x\sigma)$. ❑

Example 5.12 Let (SP_1, SP_2) be the swinging specification defined in Ex. 2.5 and HSP be the Horn clause completion of $(Ini(SP_1), SP_2)$.

$$s \equiv s' \ \Leftarrow \ head(s) \equiv head(s') \wedge tail(s) \equiv tail(s')$$

is the if-completion of $\equiv_{stream}$,

$$fair(s) \ \Leftarrow \ s \equiv L@(0\&s') \wedge fair(s')$$

is the if-completion of $fair$. By Lemma 5.11, both clauses are coinductive theorems of $(Ini(SP_1), SP_2)$. ❑

Lemma 5.11 is dual to the fact that, with respect to a *Horn clause* specification SP, only-if-completions of logical predicates of SP are *inductive* theorems of SP:

Definition 5.13 (only-if-completion of a predicate) Let SP be a Horn clause specification and $P : w$ be a logical predicate of SP. The **(logic) SP-program for P, $prog(P)$,** is the set of all axioms of SP with leading predicate P, i.e. for all $c \in prog(P)$ there are t, g such that $c = P(t) \Leftarrow g$. Let $x \in X_w$ such that for all $c \in prog(P)$, $var(c)$ does not contain components of x. The liveness axiom

$$P(x) \ \Rightarrow \bigvee_{c=(P(t)\Leftarrow g)\in prog(P)} \exists var(c) \ (x \equiv t \wedge g)$$

is called the **only-if-completion $ONLY(P)$ of P.** ❑

For instance,

$$ordered(L) \ \Rightarrow \ L = nil \ \vee \ \exists\{x\}(L = [x]) \ \vee \ \exists\{x, y, L'\}(L = x :: y :: L' \wedge x \le y \wedge ordered(y :: L'))$$

[18] [Rut 93] and [Gor 95] define bisimulations category- resp. set-theoretically as relations satisfying the above axiom. See also [Pau 94].

is the only-if-completion of *ordered* (cf. 1.4).

Lemma 5.14 Let SP be a Horn clause specification. For all logical predicates P of SP, the only-if-completion of P is an inductive theorem of SP.

Proof. A ground atom $P(t)$ can be derived by the cut calculus for SP only if for some $P(u) \Leftarrow g \in prog(P)$ and some ground substitution σ with $u\sigma = t$, $g\sigma$ is derivable as well. The statement can also be concluded from Fact 3.6 or [AvE 82], Lemma 8.3, where a function Ψ[19] is constructed in order to characterize $Ini(SP)$ as a least fixpoint. Ψ is "dual" to the function Φ defined in the proof of Thm. 4.9 for characterizing final models as greatest fixpoints. We recall the construction of Ψ.

Let $SP = (SIG, AX)$ and **PreIni** be the class of SIG-structures that agree with $Ini(SP)$ except for the logical predicates of SP. The predicate interpretations define a partial order $\leq$ on $PreIni$:

$$B \leq C \iff_{def} \text{ for all logical predicates } P,\ P^B \subseteq P^C.$$

$PreIni$ has a least element $Empty$ w.r.t. $\leq$: for all $P : w \in SP$, let $P^{Empty} = \emptyset$. Each ascending chain $\{B_i\}_{i \in I\!N} \subseteq PreIni$ has a supremum $sup(B_i)$: $P^{sup(B_i)} =_{def} \cup_{i \in I\!N} P^{B_i}$. $\Psi : PreIni \to PreIni$ is defined as follows. For all $B \in PreFin$ and predicates $P \in SP$,

$$P^{\Psi(B)} =_{def} \{b^*(t) \mid b : X \to B,\ B \models_b g \text{ for some } P(t) \Leftarrow g \in prog(P)\}.$$

Ψ is monotone and upward continuous w.r.t. $\leq$, i.e. all ascending chains $\{B_i\}_{i \in I\!N} \subseteq PreIni$ satisfy $\Psi(sup(B_i)) \leq sup(\Psi(B_i))$. One concludes from Kleene's fixpoint theorem that $lfp(\Psi) =_{def} sup(\Psi^i(Empty))$ is the least fixpoint of Ψ (cf. [LNS 82]).

From the definition of Ψ we immediately conclude the following: $B \in PreIni$ is a fixpoint of Ψ iff for all logical predicates $P \in SP$, P^B is a solution of $prog(P)$, i.e. $B \models prog(P)$. Hence $Ini(SP)$ is a fixpoint of Ψ. It is also easy to see that Ψ can be defined in terms of only-if-completions: for all $B \in PreIni$,

$$P^{\Psi(B)} = \{b(x) \mid b : X \to B,\ B \models_b gs \text{ where } P(x) \Rightarrow gs = ONLY(P)\}.$$

Hence B is a fixpoint of Ψ iff

$$P^B = \{b(x) \mid b : X \to B,\ B \models_b gs \text{ where } P(x) \Rightarrow gs = ONLY(P)\}.$$

i.e. $B \models ONLY(P)$. Since $Ini(SP)$ is a fixpoint of Ψ, we conclude that $ONLY(P)$ is an inductive theorem of SP. ❏

The following lemma completes the semantic duality between liveness and safety predicates:

Lemma 5.15 (logic programs define least relations) Let $SP = (SIG, AX)$ be a Horn clause specification and $Ini = Ini(SP)$. For all logical predicates P of SP, P^{Ini} is the least relation such that Ini satisfies $prog(P)$.

Proof. Let $\leq$ and Ψ be defined as in the proof of Lemma 5.14. Since $Ini(SP)$ is a fixpoint of Ψ and B is a fixpoint of Ψ iff $B \models prog(P)$, we have to show $Ini \leq lfp(\Psi)$, i.e. for all logical predicates P of SP, $P^{Ini} \subseteq P^{lfp(\Psi)}$. For all $i \in I\!N$, let A_i be the set of all $a \in P^{Ini}$ such that there is a ground term t with $a = t^{Ini}$, the derivation of $P(t)$ by the cut calculus for SP contains i occurrences of non-equational axioms and there are no derivations of some $P(u)$ with $u^{Ini} = a$ and fewer occurrences of non-equational axioms.

We show $A_i \subseteq P^{\Psi^i(Empty)}$ by induction on i. Since $A_0 = \emptyset$, the proof is complete for $i = 0$. Let $i > 0$ and $a \in A_i$. Then there are $P(t) \Leftarrow g \in prog(P)$ and a ground substitution σ such that $(t\sigma)^{Ini} = a$ and $g\sigma$ has a derivation with $i-1$ non-equational axioms. By induction hypothesis, $\Psi^{i-1}(Empty) \models g\sigma$. Hence by definition of Ψ, $a = (t\sigma)^{Ini} \in P^{\Psi^i(Empty)}$.

[19] called T_P in [AvE 82], Lemma 8.3

Since $P^{Ini} \subseteq \cup_{I \in I\!N} A_i$, we conclude that $P^{Ini} \subseteq \cup_{I \in I\!N} P^{\Psi^i(Empty)} = P^{sup(\Psi^i(Empty))} = P^{lfp(\Psi)}$. ❑

Only-if-completions of inductively defined *functions* can be derived from Horn clause specifications if HSP is normal form complete (cf. 6.12). As a further consequence of Lemma 5.5 we may construct inequality predicates for action sorts from inequality predicates for primitive sorts.

Definition 5.16 Let $SP = (SIG, AX)$ be a specification and A be an SP-model. A **has inequality predicates** if for all sorts $s \in SP$, SP can be extended by an inequality predicate $\not\equiv_s$ such that for all $t, t' \in T_{SIG,s}$, $A \models t \not\equiv_s t'$ iff $t^A \neq t'^A$. ❑

Definition 5.17 Let Assumption 4.1 hold true. Suppose that A has inequality predicates. The **inequality extension** $HSPN$ **of** (A, SP') is the least extension of HSP that contains for all $s \in S$ the axioms

$$x \not\equiv_s \bot \;\Leftarrow\; Def(x), \quad \bot \not\equiv_s x \;\Leftarrow\; x \not\equiv_s \bot$$

and for all $s \in S' \setminus S$ and $i \in I\!N$ the predicates $\not\equiv_s$: ss and $\not\equiv_{s,i}$: ss and the axioms

$$x \not\equiv_s y \;\Leftarrow\; x \not\equiv_{s,i} y,$$
$$x \not\equiv_{s,i+1} y \;\Leftarrow\; act_1(x, z_1) \not\equiv_{s_1,i} act_1(y, z_1) \vee \ldots \vee act_n(y, z_n) \not\equiv_{s_n,i} act_1(y, z_n)$$

where $act_1 : sw_1 \to s_1, \ldots, act_n : sw_n \to s_n$ are the s-actions of SP'. For all $s \in S$ and $i \in I\!N$, define $\not\equiv_{s,i} = \not\equiv_s$. ❑

Under the assumptions of the previous definition we identify HSP and $HSPN$.

Definition and Theorem 5.18 (action extensions with inequality predicates) Let Assumption 4.1 hold true. Suppose that A has inequality predicates and SP' is action consistent w.r.t. A. Then $Fin(A, SP')$ has inequality predicates. Moreover, for all $s \in S' \setminus S$, the following clauses are coinductive theorems of (A, SP'):

(1) $x \not\equiv_s y \;\Rightarrow\; \exists\{z_1\}(act_1(x, z_1) \not\equiv_{s_1} act_1(y, z_1)) \vee \ldots \vee \exists\{z_n\}(act_n(x, z_n) \not\equiv_{s_n} act_n(y, z_n))$

(2) $x \not\equiv_s y \;\Leftarrow\; act_1(x, z_1) \not\equiv_{s_1} \vee \ldots \vee act_n(x, z_n) \not\equiv_{s_n} act_n(y, z_n)$

where $act_1 : sw_1 \to s_1, \ldots, act_n : sw_n \to s_n$ are the s-actions of SP'. (1) is called the **liveness axiom for** $\not\equiv_s$**.** and implicitly included in SP'. (2) is called the **if-completion of** $\not\equiv_s$ (cf. 5.10).

Proof. Let $s \in SP$. First we show that $x \not\equiv_s \bot \Leftrightarrow Def(x)$ is an inductive theorem of $HSPN$. "$\Leftarrow$" is an axiom of $HSPN$. For "$\Rightarrow$" assume that both $HSPN \vdash_{cut} t \not\equiv_s \bot$ and $HSPN \not\vdash_{cut} Def_s(t)$ hold true for some $t \in T_{SIG'}$. Since BSP and thus $HSPN$ are totalized, we would have $HSPN \vdash_{cut} t \equiv_s \bot$ and thus $HSPN \vdash_{cut} \bot \not\equiv_s \bot$. But there is no derivation of $\bot \not\equiv_s \bot$ from the axioms of $HSPN$. Hence $x \not\equiv_s \bot \Rightarrow Def(x) \in ITh(HSPN)$.

Now let $t, t' \in T_{SIG',s}$. If $t \Downarrow$ and $t' \Downarrow$, then there are $u, u' \in T_{SIG,s}$ such that $t \equiv_{BSP} u$ and $t' \equiv_{BSP} u'$. Since $Fin(A, SP')$ is conservative over A,

$$Fin(A, SP') \models t \not\equiv_s t' \;\text{ iff }\; Fin(A, SP') \models u \not\equiv_s u' \;\text{ iff }\; A \models u \not\equiv_s u'$$
$$\text{iff }\; A \not\models u \equiv_s u' \;\text{ iff }\; Fin(A, SP') \not\models u \equiv_s u' \;\text{ iff }\; Fin(A, SP') \not\models t \equiv_s t'$$

because SP has inequality predicates w.r.t. A. If $t \Uparrow$, then by Lemma 5.5, totalizedness of $HSPN$ and since $\bot \not\equiv_s x \Leftrightarrow Def(x) \in ITh(HSPN)$,

$$Fin(A, SP') \models t \not\equiv_s t' \;\text{ iff }\; HSPN \vdash_{cut} t \not\equiv_s t' \;\text{ iff }\; HSPN \vdash_{cut} \bot \not\equiv_s t'$$
$$\text{iff }\; HSPN \vdash_{cut} Def(t') \;\text{ iff }\; HSPN \not\vdash_{cut} \bot \equiv_s t' \;\text{ iff }\; HSPN \not\vdash_{cut} t \equiv_s t' \;\text{ iff }\; Fin(A, SP') \not\models t \equiv_s t'.$$

Analogously, one obtains

$$Fin(A, SP') \models t \not\equiv_s t' \;\text{ iff }\; Fin(A, SP') \not\models t \equiv_s t'$$

if $t' \Uparrow$. Finally, let $s \in S' \setminus S$ and $t, t \in T_{SIG',s}$. By Lemma 5.5 and the axioms for $\equiv_s$ and $\not\equiv_s$,

$$Fin(A, SP') \models t \not\equiv_s t' \text{ iff } HSPN \vdash_{cut} t \not\equiv_s t' \text{ iff } HSPN \not\vdash_{cut} t \equiv_s t' \text{ iff } Fin(A, SP') \not\models t \equiv_s t'.$$

Therefore $HSPN$ has inequality predicates w.r.t. $Fin(A, SP')$. Hence (1) is a coinductive theorem because by 5.11, the if-completion of $\equiv_s$ is a coinductive theorem. (2) follows from congruence axioms for equality predicates. ❏

Lemmata 5.11, 5.14 and Thm. 5.18 are fundamental for efficient deduction rules for proving inductive or coinductive theorems (cf. Section 7).

In accordance with the sequence of initial and final models defining the semantics of a swinging specification SP (cf.4.10), the theory of SP is a sequence of inductive and coinductive theories:

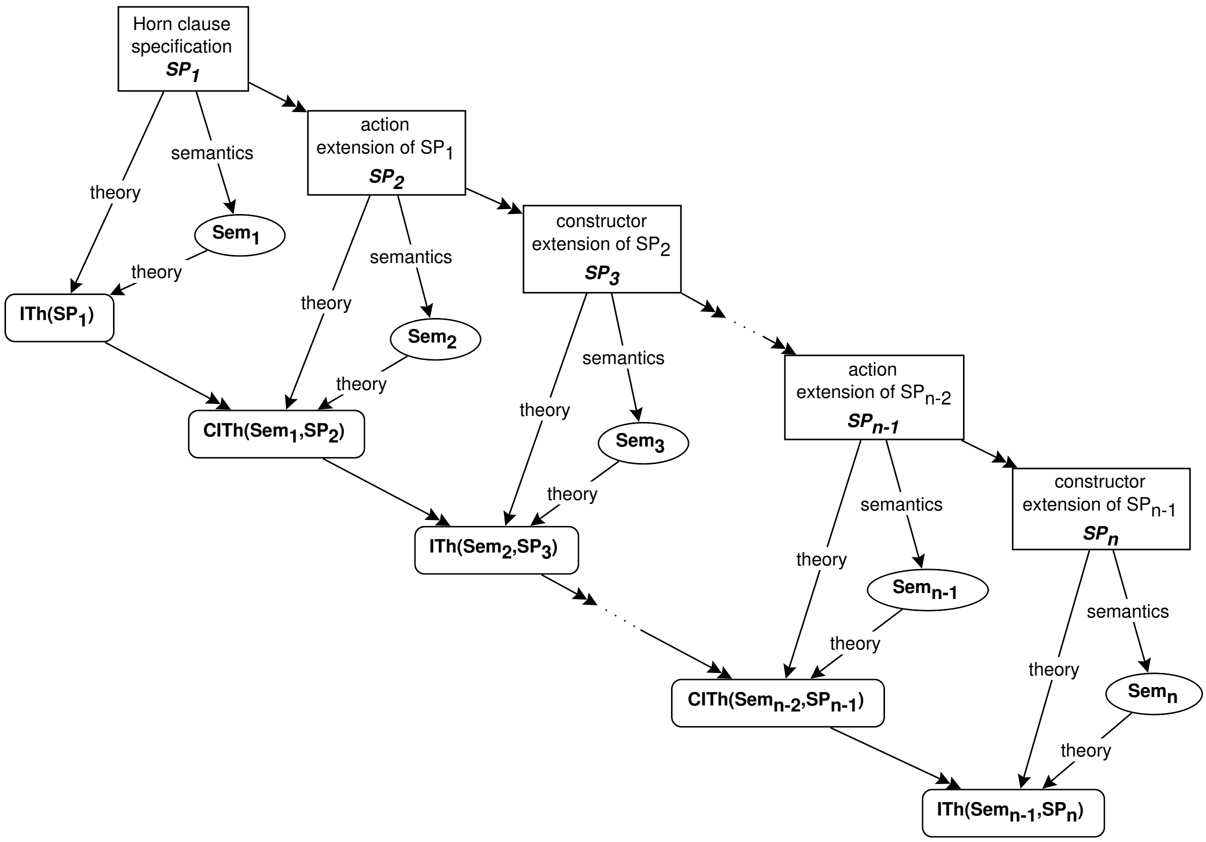


Figure 9. Theory of a swinging specification (cf. Fig. 7)

Definition and Theorem 5.19 (theory of a swinging specification) Let $SP = (SP_1, \dots, SP_n)$ be a swinging specification with semantics $(Sem_1, \dots, Sem_n)$. Then a sequence $Th = (Th_1, \dots, Th_n)$ of theories and a sequence $HSP = (HSP_1, \dots, HSP_n)$ of specifications are well-defined as follows:

- $Th_1 = ITh(SP_1)$ and $HSP_1 = SP_1$.
- For all $2 \leq i \leq n$,
  - $Th_i = ITh(Sem_{i-1}, SP_i)$ and $HSP_i = BSP(Sem_{i-1}, SP_i)$ if SP_i is a constructor extension of SP_{i-1} (cf. 3.8),
  - $Th_i = CITh(Sem_{i-1}, SP_i)$ and $HSP_i = HSP(Sem_{i-1}, SP_i)$ if SP_i is an action extension of SP_{i-1} (cf. 5.7).

Then Th and HSP are called the **theory of** SP and the **Horn clause completion of** SP, respectively. By Fact 3.6 and Thm. 5.8, for all $1 \leq i \leq n$, $Th_i = Th(Sem_i)$. ❏

Fact 5.20 Let SIG be a signature and A, B be two reachable SIG-structures with $Th(A) = Th(B)$. Then A and B are SIG-isomorphic. ❏

Corollary 5.21 Let $SP = (SP_1, \ldots, SP_n)$ be a swinging specification with semantics $(Sem_1, \ldots, Sem_n)$ theory $(Th_1, \ldots, Th_n)$ and Horn clause completion $(HSP_1, \ldots, HSP_n)$. Then for all $1 \leq i \leq n$, $Th_i = ITh(HSP_i)$ if SP_i is a constructor extension of SP_{i-1}, $Th_{i,beh} = ITh(HSP_i)$ if SP_i is an action extension of SP_{i-1}, and thus by Fact 5.20, $Sem_i \cong Ini(HSP_i)$.

Proof. We have $Th_1 = Th(Sem_1) = Th(Ini(SP_1)) = ITh(SP_1) = ITh(HSP_1)$. Let $i > 1$. If SP_i is a constructor extension of SP_{i-1}, then

$$Th_i = Th(Sem_i) = Th(Ini(Sem_{i-1}, SP_i)) = ITh(Sem_{i-1}, SP_i) = ITh(BSP(Sem_{i-1}, SP_i)) = ITh(HSP_i).$$

If SP_i is an action extension of SP_{i-1}, then

$$\begin{aligned} Th_{i,beh} &= Th(Sem_i)_{beh} = Th(Fin(Sem_{i-1}, SP_i))_{beh} = CITh(Sem_{i-1}, SP_i)_{beh} \\ &= ITh(HSP(Sem_{i-1}, SP_i)) = ITh(HSP_i). \ ❏ \end{aligned}$$

6 Functional Specifications

Thm. 5.8 provides the basic result for proving that an *action* extension fulfils requirements presented as Gentzen clauses. Fact 3.6 is the "constructor counterpart" of 5.8 insofar as it allows us to prove Gentzen clauses with respect to a *constructor* extension. Section 5 also provided a couple of inductive and coinductive "meta"-theorems, which will be turned into proof rules in Section 7. Further theorems are obtained if the given specification is *functional*. Functionality comprises two conditions: *normal form completeness* and *ground confluence*. In Section 8, we will specialize functionality and come up with axiom schemata that cover – at least – all current functional-logic programs. The basic notions presented in the present section originate from the theory of conditional term rewriting systems (CTRS), but had to be generalized for becoming applicable to the whole area of functional-logic specification and programming. The generalization led to a reduction calculus for goals (6.4), a corresponding reduction relation (6.5) and the notions of *reductive* and *subreductive validity* (6.9, 8.2), which have been introduced and motivated thoroughly in [Pad 92] (to some extent already in [Pad 88]) and which are reviewed here for the purpose of establishing efficient proof rules for swinging specifications in the subsequent section.

Depending upon the underlying specification SP, our reduction calculus is parameterized by a set NF of normal forms. In contrast to other approaches, NF is not *defined* as the set of SP-reduced terms. This would entail a circular definition because our rewriting rule *refers* to NF as the set of terms, which may be substituted for *fresh* or *extra variables* (cf. 6.4).

Assumption 6.1 For notational convenience let us regard each logical predicate P as a Boolean function and each logical atom $P(t)$ as an equation $P(t) \equiv true$. Conversely, a Boolean function F that stems from a predicate occurs only at the outermost position of an equation $t \equiv true$. Terms $F(t)$ where F stems from a predicate are called **atomic terms**. This property is taken into account by normal form completeness: atomic ground terms need not have normal forms (cf. 2.3 and 6.6). ❏

Definition 6.2 (constructor-based specification) An inhabited Horn clause specification SP with sort set S is **constructor-based** if S consists of **base sorts** and **non-base sorts** such that each operation of SP is either a bottom constant for a base sort (cf. 2.3), a **constructor** or a **defined function** and the following conditions are satisfied:

(1) For all constructors $F : s_1 \dots s_n \to s$ with base sort s and all $1 \le i \le n$, s_i is a base sort or there is a set S_i of non-base sorts such that $s_i \in S_i$ and for all constructors $G : w \to s'$, $s' \in S_i$ implies $w \in S_i^*$.

(2) For all axioms $l \equiv r \Leftarrow h$ of SP, l is neither a variable nor a bottom constant.

(3) All actions of SP are defined functions.

Let $SP = (SIG, AX)$. The S-sorted set of SIG**-normal forms**, $\mathbf{NF_{SIG}(X)}$, is the set of all SIG-terms over X consisting of constructors and variables. The S-sorted set of $SIG\bot$**-normal forms**, $\mathbf{NF_{SIG\bot}(X)}$, is the set of all SIG-terms t over X consisting of constructors, bottom constants and variables such that all subterms of t of the form $F(\dots, \bot, \dots)$ are base-sorted. $\mathbf{NF_{SIG}}$ and $\mathbf{NF_{SIG\bot}}$ denote the sets of ground SIG-normal forms and ground $SIG\bot$-normal forms, respectively.

SP is **free-constructor-based** if for all axioms $l \equiv r \Leftarrow h$ of SP, $l \notin NF_{SIG\bot}$ and l contains a bottom constant only if $r = \bot$. ❑

If, according to Assumption 6.1, logical predicates are regarded as functions, they are always regarded as defined functions and not as constructors. SIG-normal forms are $SIG\bot$-normal forms. Conversely, (1) ensures that each base-sorted $SIG\bot$-normal form is either a SIG-normal form or a bottom constant. A bottom constant for a sort s is usually introduced when the specification with the constructors for s is extended (cf. 2.3). This makes bottom constants conceptually different from constructors. However, certain terms with bottom constants are regarded as normal forms for permitting the specification of (totalized) partial functions even if SP is supposed to be *normal form complete*, i.e. all ground terms are reducible into normal forms (cf. 6.5 and 6.6).

Example 6.3 (streams) The stream specification $HSP = HSP(Ini(SP_1), SP_2)$ (cf. 5.2) suggests to declare the operations $\&, 01, 10$ and $nats$ as constructors and $@, merge$ as defined functions. But then HSP were not free-constructor-based: the left-hand sides of the axioms

```
01 = 0&10
10 = 1&01
nats(n) = n&nats(succ(n))
```

consist of constructors and and variables. So, for the moment, let us declare $01, 10$ and $nats$ as defined functions. ❑

A variable x is a **fresh** or **extra variable** of a conditional equation $c = t \equiv u \Leftarrow h$ if x occurs in u or h, but not in t. **fresh**(c) denotes the set of fresh variables of c.

Definition 6.4 (reduction calculus) Let $SP = (SIG, AX)$ be a constructor-based specification. The **reduction calculus for** SP consists of the following two rules for transforming goals over SIG.

rewriting $\quad \dfrac{g(l\sigma)}{g(r\sigma) \wedge h\sigma} \quad$ if $c = (l \equiv r \Leftarrow h) \in AX$, $\sigma : X \to T_{SIG}(X)$ and $fresh(c)\sigma \subseteq NF_{SIG}(X)$

reflection $\quad \dfrac{g \wedge t \equiv t}{g}$

A sequence $g_1, \dots, g_n$ of goals is called a **goal reduction of** g_1 **into** g_n **upon** SP if for all $1 \le i \le n$, g_{i+1} is obtained from g_i by a single rewriting or reflection step. The corresponding inference relation is denoted by $\vdash_{SP}$. g is SP**-convergent** if $g \vdash_{SP} \emptyset$. ❑

Definition 6.5 (reduction relation) Let $SP = (SIG, AX)$ be a constructor-based specification. The SP**-reduction relation** $\longrightarrow_{SP}$ is a binary relation both on $T_{SIG}(X)$ and on the set of goals over SIG defined as follows:

- $g \longrightarrow_{SP} g' \quad \Longleftrightarrow_{def} \quad g = t(l\sigma)$ and $g' = t(r\sigma)$ for some term (goal) t, $c = (l \equiv r \Leftarrow h) \in AX$ and a substitution σ such that $h\sigma$ is SP-convergent and $fresh(c)\sigma \subseteq NF_{SIG}(X)$.

g' is an SP**-reduct** of g if $g \longrightarrow^*_{SP} g'$. A substitution τ is an SP**-reduct** of a substitution σ if $x\sigma \longrightarrow^*_{SP} x\tau$ for all $x \in X$. In this case we write $\sigma \longrightarrow^*_{SP} \tau$. g and g' are SP**-joinable**, written: $g \downarrow_{SP} g'$, if g and g' have a common SP-reduct. g is SP**-reducible into** each SP-reduct of g. If g is the only SP-reduct of g, then g is SP**-reduced**. A goal g is **strongly** SP**-convergent** if all SP-reducts of g are SP-convergent. ❑

Note that two terms t and t' are SP-joinable if and only if the equation $t \equiv t'$ is SP-convergent:

$$t \downarrow_{SP} t' \quad \text{iff} \quad t \equiv t' \vdash_{SP} \emptyset.$$

If $SP = (SIG, AX)$ is free-constructor-based, then all $SIG\bot$-normal forms are SP-reduced.

Definition 6.6 (functional specification) Let $SP = (SIG, AX)$ be a constructor-based specification. SP is **normal form complete** if each non-atomic (!) ground SIG-term is SP-reducible into a $SIG\bot$-normal form. SP is **ground confluent** if SP-convergence is closed under $\longrightarrow_{SP}$, i.e. if all SP-convergent ground goals are strongly SP-convergent.

SP is **functional** if SP is normal form complete and ground confluent. ❑

Example 6.7 (streams) The choice of constructors and defined functions in Ex. 6.3 violates normal form completeness: neither the constants $01, 10$ nor ground terms of the form $nats(t)$ or $merge(u, v)$ are reducible into normal forms. Hence we declare $01, 10$, $nats$ and $merge$ as constructors and replace the axioms

```
01 = 0&10
10 = 1&01
nats(n) = n&nats(succ(n))
merge(n&s,s') = n&merge(s',s)
```

by the following semantically equivalent (!) ones:

```
head(01) = 0
tail(01) = 10
head(10) = 1
tail(10) = 01
head(nats(n)) = n
tail(nats(n)) = nats(succ(n))
head(merge(n&s,s')) = n
tail(merge(n&s,s')) = merge(s',s)
```

❑

Since our reduction calculus is based on normal forms, certain "stupid" axiom sets such as

$$\{a \equiv b, \quad a \equiv c, \quad b \equiv c \Leftarrow (b \equiv x \wedge c \equiv x)\}$$

are no longer confluent. This specification were ground confluent only if b and c were joinable, which could only be shown by reducing b into c with the help of the axiom $b \equiv c \Leftarrow (b \equiv x \wedge c \equiv x)$. But this axiom were applicable to b only if $b \equiv x \wedge c \equiv x$ had a *normal form* solution in the fresh variable x. If we regard the above axioms as a constructor-based specification, then the (only) solution a of $b \equiv x \wedge c \equiv x$ is not a normal form. If we do not require constructor-basedness, then reductions can no longer be lifted to narrowing expansions (cf. 7.7) and we loose the only effective method for solving goals such as $b \equiv x \wedge c \equiv x$.

[BG 89] and [DO 90], Section 5, also consider reduction calculi with restricted substitution into fresh variables. However, these approaches combine the substitution restriction with the requirement that the specification is *decreasing*, which is a much stronger termination condition than strong termination as defined below (Def. 8.1).

$\vdash_{SP}$ is **sound** insofar as SP-convergent goals are derivable via the cut calculus for SP (cf. 3.3): for all goals g over SIG,

$$g \vdash_{SP} \emptyset \quad \text{implies} \quad SP \vdash_{cut} g.$$

$\vdash_{SP}$ is **complete** w.r.t. $\vdash_{cut}$ if and only if SP is ground confluent:

Theorem 6.8 (Church-Rosser Theorem) ([Pad 92], Thm. 6.5(2)) A normal form complete specification SP is ground confluent iff for all ground goals g, $SP \vdash_{cut} g$ implies $g \vdash_{SP} \emptyset$. ❑

The Church-Rosser Theorem allows us to reduce inductive validity to *reductive* validity:

Definition 6.9 (reductive validity) Given a constructor-based specification $SP = (SIG, AX)$, a Gentzen clause $gs \Leftarrow h$ over SIG is **reductively valid w.r.t.** SP if for all $\sigma : X \to NF_{SIG\perp}$ such that $h\sigma$ is *strongly* SP-convergent there are $h' = \exists V g \in gs$ and a ground substitution $\tau =_{in(h')} \sigma$ such that $g\tau$ is *strongly* SP-convergent. ❑

In particular, $\neg h$ is reductively valid if there is no $\sigma : X \to NF_{SIG\perp}$ such that $h\sigma$ is strongly SP-convergent.

Corollary 6.10 Let SP be a normal form complete specification. SP is ground confluent iff each Gentzen clause c over SP is reductively valid w.r.t. SP iff $c \in ITh(SP)$ (cf. 3.4).

Proof. Let SP be ground confluent. "$\Rightarrow$": Let $c = gs \Leftarrow h$ be reductively valid and σ be a ground substitution such that $SP \vdash_{cut} h\sigma$. By Thm. 6.8, $h\sigma \vdash_{SP} \emptyset$. Since SP is ground confluent, $h\sigma$ is strongly SP-convergent. Since SP is a normal form complete, there is a $\rho : X \to NF_{SIG\perp}$ with $\sigma \longrightarrow^*_{SP} \rho$. Since $h\sigma$ is strongly SP-convergent, $h\rho$ is also strongly SP-convergent. Since $gs \Leftarrow h$ is reductively valid, there are $h' = \exists V g \in gs$ and a ground substitution $\tau =_{in(h')} \sigma$ such that $g\tau$ is (strongly) SP-convergent. Hence $SP \vdash_{cut} g\tau$.

"$\Leftarrow$": Let $gs \Leftarrow h \in ITh(SP)$ and $\sigma : X \to NF_{SIG\perp}$ be a ground substitution such that $h\sigma$ is (strongly) SP-convergent. Then $SP \vdash_{cut} h\sigma$. Since $gs \Leftarrow h \in ITh(SP)$, there are $g \in gs$ and a ground substitution $\tau =_{in(g)} \sigma$ such that $SP \vdash_{cut} g\tau$. Since SP is ground confluent, Thm. 6.8 implies that $g\tau$ is SP-convergent and thus strongly SP-convergent.

Suppose that each Gentzen clause $gs \Leftarrow h$ over SP is reductively valid w.r.t. SP iff $gs \Leftarrow h \in ITh(SP)$. Let g be an SP-convergent ground goal. Then $SP \vdash_{cut} g$ and thus $g \in ITh(SP)$. Hence by assumption, g is reductively valid, i.e. strongly convergent. We conclude that SP is ground confluent. ❑

Note that, if SP is ground confluent, then we can replace "strong SP-convergence" in the definition of reductive validity by "SP-convergence".

Let us now invert functional programs, i.e. sets of conditional equations defining functions, analogously to the way we inverted logic programs for predicates (cf. 5.13).

Definition 6.11 (only-if-completion of a defined function) Let SP be a constructor-based specification and $F : w \to s$ be a defined function of SP. The **(functional) SP-program for F, $prog(F)$,** is the set of all axioms of SP with leading function F, i.e. for all $c \in prog(F)$ there are t, u, g such that $c = F(t) \equiv u \Leftarrow g$. Let $x \in X_w$ and $y \in X_s$ such that for all $c \in prog(P)$, $var(c)$ does not contain y nor components of x. The liveness axiom

$$F(x) \equiv y \;\Rightarrow \bigvee_{c=(F(t)\equiv u \Leftarrow g)\in prog(F)} \exists var(c)\; (x \equiv t \wedge y \equiv u \wedge g)$$

is called the **only-if-completion $ONLY(F)$ of F.** ❑

Lemma 6.12 Let SP be a normal form complete specification. Then for all defined functions F of SP, the only-if-completion of F is an inductive theorem of SP.

Proof. Let $SP = (SIG, AX)$ and σ be a ground substitution such that $SP \vdash_{cut} F(x\sigma) \equiv y\sigma$. Since SP is normal form complete, there are $t, u \in NF_{SIG\perp}$ with $x\sigma \longrightarrow^*_{SP} t$ and $F(t) \longrightarrow_{SP} u$. Hence $t = t'\tau$, $u = u'\tau$,

$g\tau \vdash_{SP} \emptyset$ and $fresh(c)\tau \subseteq NF_{SIG}$ for some $c = (F(t') \equiv u' \Leftarrow g) \in prog(F)$ and some τ. W.l.o.g. $\tau =_{\{x,y\}} \sigma$ because $var(c)$ does not contain neither y nor components of x. Hence $SP \vdash_{cut} x\tau = x\sigma \equiv t = t'\tau$ and $SP \vdash_{cut} y\tau = y\sigma \equiv F(x\sigma) \equiv F(t) \equiv u = u'\tau$. Moreover, $g\tau \vdash_{SP} \emptyset$ implies $SP \vdash_{cut} g\tau$. Hence σ solves the conclusion of $ONLY(F)$. ❑

Under the assumption that SP is normal form complete and *strongly terminating* (cf. Section 8) we obtain a functional counterpart of Lemma 5.15:

Definition and Lemma 6.13 (functional programs define least relations) Let SP be a normal form complete and strongly terminating specification (cf. 8.1), $Ini = Ini(SP)$ and DF be a set of defined functions of SP such that for all $F \in DF$, $prog(F)$ is **flat w.r.t.** DF, i.e. function symbols of DF occur in $prog(F)$ only as root symbols of left-hand sides of equations.[20]

For all $F : w \to s \in DF$ add the predicate $graph_F : ws$ and the axiom $graph_F(x,y) \Leftarrow F(x) \equiv y$ to SP. $graph_F$ is called the **graph of** F. The **(logic) program for** $graph_F$ **w.r.t.** DF, $prog_{DF}(F)$, is the set of Horn clauses obtained from the program for F by replacing each equation $F(t) \equiv u$, $F \in DF$, with $graph_F(t,u)$. Obviously, for all $F \in DF$, $prog_{DF}(F)$ does not contain symbols from DF. Moreover, $graph_F^{Ini}$ is the least relation such that Ini satisfies $prog_{DF}(F)$.

Proof. For all $F : w \to s \in DF$, let $R_F \subseteq Ini_w \times Ini_s$ and Ini' be the structure that agrees with Ini except that $graph_F$ is interpreted by R_F. Suppose that Ini' satisfies $prog_{DF}(F)$. We show $graph_F^{Ini} \subseteq R_F$ by Noetherian induction along the reduction ordering $>$ for SP, which exists by assumption. Let $SP = (SIG, AX)$ and $(a,b) \in graph_F^{Ini}$. Since SP is normal form complete, there are $t,u \in NF_{SIG\perp}$ and $v \in T_{SIG}$ with $t^{Ini} = a$, $u^{Ini} = b$ and $F(t) \longrightarrow_{SP} v$. Hence $t = t'\sigma$, $v = v'\sigma$, $g\sigma \vdash_{SP} \emptyset$ and $fresh(c)\sigma \subseteq NF_{SIG}$ for some $c = (F(t') \equiv v' \Leftarrow g) \in prog(F)$ and some σ. $g\sigma \vdash_{SP} \emptyset$ implies $(F(t) \equiv u) > g$, in particular $(F(t) \equiv u) > (G(u'\sigma) \equiv u''\sigma)$ for all equations $G(u') \equiv u'' \in g$ with $G \in DF$. We have $SP \vdash_{cut} G(u'\sigma) \equiv u''\sigma$ and thus $SP \vdash_{cut} graph_G(u',u'')\sigma$, i.e. $((u',u'')\sigma)^{Ini} \in graph_G^{Ini}$. Hence by induction hypothesis, $((u',u'')\sigma)^{Ini} \in R_G$. Since $c \in prog(F)$ and Ini' satisfies $prog_{DF}(F)$, we conclude that $(a,b) = ((t,u)\sigma)^{Ini} \in R_F$. ❑

[20] Cf. [BGM 88] for a corresponding procedure for flattening non-flat programs.

7 Proof Rules

Let us first recall those results of Sections 5 and 6, which yield basic inference rules for proofs of inductive or coinductive theorems. Suppose that SP is a Horn clause specification, A is a reachable SP-model and SP' is an action extension of SP. Let $Ini = Ini(SP)$ and $Fin = Fin(A, SP')$.

(1) If SP' is action consistent w.r.t. A, then the liveness predicates of SP' are the greatest relations on Fin satisfying their liveness axioms (Thm. 4.9).

(2) If SP' is action consistent w.r.t. A, then the action-sorted equality or inequality predicates of SP' are the greatest relations on Fin satisfying their liveness axioms (Lemma 5.9).

(3) If SP' is action consistent w.r.t. A, then Fin satisfies the if-completions of the liveness and equality predicates of SP' (Lemma 5.11).

(4) Ini satisfies the only-if-completions of logical predicates of SP (Lemma 5.14).

(5) The logical predicates of SP are the least relations on Ini satisfying their programs (Lemma 5.15).

(6) If A has inequality predicates and SP' is action consistent w.r.t. A, then Fin has inequality predicates and satisfies their liveness axioms (Thm. 5.18).

(7) If SP is normal form complete, then Ini satisfies the only-if-completions of defined functions of SP (Lemma 6.12).

(8) If SP is normal form complete and strongly terminating, then the graphs of defined functions are the least relations on Ini satisfying their programs (Lemma 6.13).

Invariants and other proof obligations that come up with formal progarm verification usually take the form of Gentzen clauses. Hence theorem provers and proof checkers like Expander (cf. [Pad 94]) employ **Gentzen rules**, i.e. inference rules for transforming Gentzen clauses. A Gentzen clause c is proved by stepwise reducing c into a tautology, denoted by $TRUE$. It turns out that most steps of a derivation $c = c_1 \vdash \ldots \vdash c_n = TRUE$ affect *either* the premise *or* the conclusion of a Gentzen clause c_i. Therefore, Gentzen rules are based upon lower-level **goal set rules**, which transform goal sets. A goal set rule R is called an **expansion**, **generalization** or **backward rule** if the succedent of R implies the antecedent. R is a **contraction**, **specialization** or **forward rule** if the antedent of R implies the succedent.[21] Due to the type of the underlying specification the implication is valid in an inductive or a coinductive theory. Given a conjecture $gs \Leftarrow hs$, expansions are used to transform the conclusion gs, while contractions are applied to the premise hs. The proof of $gs \Leftarrow hs$ is complete if the expansions and contractions lead to goal sets gs' and hs', respectively, such that $gs' \Leftarrow hs'$ is a tautology. A purely syntactical tautology criterion is subsumption:

Definition 7.1 A goal set $hs = (h_1 \vee \ldots \vee h_m)$ **subsumes** a goal set $gs = (g_1 \vee \ldots \vee g_n)$ if for all $1 \leq i \leq m$ there is $1 \leq j \leq n$ such that h_i subsumes g_j. A goal h **subsumes** a goal g if $h = FALSE$ or $g = \emptyset$ or $g = \exists V g'$ and there is a substitution σ such that $g'\sigma_V \subseteq h$.[22] ❑

The following basic Gentzen rules reduce the proof of a Gentzen clause to expansions and contractions of its conclusion and premise, respectively. We attach arrows to each Gentzen or goal set rule to indicate in which direction the rule establishes an implication.

expansion $\dfrac{gs \Leftarrow hs}{gs' \Leftarrow hs} \Uparrow$ if gs' is obtained from gs by an expansion step

[21] In terms of *inductive logic programming*, expansions are inductive and contractions are deductive rules (cf. [MR 94]).

[22] Goal set subsumption generalizes to goal sets Plotkin's θ-subsumption of disjunctions of literals (cf. [Plo 70]).

contraction $\dfrac{gs \Leftarrow hs}{gs \Leftarrow hs'} \Uparrow$ if hs' is obtained from hs by a contraction step

subsumption $\dfrac{gs \Leftarrow hs}{TRUE} \Uparrow$ if hs subsumes gs

A Gentzen or goal set rule is a **simplification** or **equivalence transformation** if its antecedent is equivalent to its succedent. Simplifications are preferable to proper expansions and contractions because they are sound transformations of both the conclusion and the premise of the conjecture, say $gs \Leftarrow hs$. However, simplification steps are *sufficient* only if gs and hs are equivalent. In contrast to simplifications, which are often performed automatically, proper expansions and contractions must be selected more carefully for achieving a successful derivation of $gs \Leftarrow hs$. The expansions of gs must produce goal sets that are stronger than gs, but still weak enough for being implied by hs. The contractions of hs must return goal sets that are weaker than hs, but still strong enough for implying gs. Simplification steps are reversible. Expansion and contraction steps, such as applications of assumptions, lemmas or induction hypotheses, may result in a failing derivation so that their use should be controlled interactively.

The names of the following simplification rules stem from the underlying clauses whose validity under certain conditions has been shown in the previous section (cf. Results (3), (4), (6) and (7)). Existential quantifiers, which are propagated from the antecedent to the succedent of a rule are omitted. The variables bound by such quantifiers are supposed to be different from the existential variables that are introduced by the rule. A goal set rule is applied to a goal set gs by selecting a **redex**, i.e. a subset gs' of gs that matches the rule antecedent, and replacing gs' by the corresponding succedent. If the succedent is the empty goal set or $FALSE$, then gs is replaced by $gs \setminus gs'$ if $gs \neq gs'$, otherwise by $FALSE$.

Throughout this section suppose that $SP = (SP_1, \ldots, SP_n)$ is a swinging specification with semantics $Sem = (Sem_1, \ldots, Sem_n)$, theory $(Th_1, \ldots, Th_n)$ and Horn clause completion $(HSP_1, \ldots, HSP_n)$ (cf. 4.10 and 5.19).

Definition 7.2 (basic simplifications) Let g, h be goals, t, u be terms and σ be a substitution over SP_n.

if-completion of P $\dfrac{(P(x) \wedge g)\sigma}{(\exists X_1(h_1 \wedge g) \vee \ldots \vee \exists X_k(h_k \wedge g))\sigma} \Updownarrow$

if $\{P(x) \Leftarrow h_1, \ldots, P(x) \Leftarrow h_k\}$ is the if-completion of a liveness or action-sorted equality predicate P of SP_n and for all $1 \leq i \leq k$, $X_i = var(h_i) \setminus var(x)$ (cf. 5.9, 5.10, 5.18)

only-if-completion of P $\dfrac{(P(x) \wedge g)\sigma}{(\exists X_1(x \equiv t_1 \wedge h_1 \wedge g) \vee \ldots \vee \exists X_k(x \equiv t_k \wedge h_k \wedge g))\sigma} \Updownarrow$

if P is a safety predicate of SP_n,

$$prog(P) \;=\; \{P(t_1) \Leftarrow h_1, \ldots, P(t_k) \Leftarrow h_k\}$$

and for all $1 \leq i \leq k$, $X_i = var(P(t_i) \Leftarrow h_i)$ (cf. 5.13)

only-if-completion of F $\dfrac{g(F(x))\sigma}{(\exists X_1(x \equiv t_1 \wedge g(u_j) \wedge h_1) \vee \ldots \vee \exists X_k(x \equiv t_k \wedge g(u_k) \wedge h_k))\sigma} \Updownarrow$

if F is a defined function of SP_n,

$$prog(F) \;=\; \{F(t_1) \equiv u_1 \Leftarrow h_1, \ldots, F(t_k) \equiv u_k \Leftarrow h_k\}$$

and for all $1 \leq i \leq k$, $X_i = var(F(t_i) \equiv u_i \Leftarrow h_i)$ (cf. 6.11)

term replacement $\dfrac{g(t)}{g(u)} \Updownarrow$ if $t \equiv u \in Th_n$

stop rule $\dfrac{TRUE^{23} \vee hs}{TRUE} \Updownarrow$

¬-rule $\dfrac{gs \vee hs}{hs} \Updownarrow$ if $\neg gs \in Th_n$

gfp-rule $\dfrac{p\sigma \wedge g}{h\sigma \wedge g} \Updownarrow$ if $p_{beh} \Leftarrow h$ is a gfp-axiom of HSP_n (cf. 5.1)

approximation $\dfrac{(P(x) \wedge g)\sigma}{(\exists X_1(h_1 \wedge g) \vee \ldots \vee \exists X_k(h_k \wedge g))\sigma} \Updownarrow$

if P is an approximation predicate of HSP_n, $P(x) \Leftarrow h_1, \ldots, P(x) \Leftarrow h_k$ are all approximation axioms for P and for all $1 \leq i \leq k$, $X_i = var(h_i) \setminus var(x)$ (cf. 5.1) ❑

Only the last two rules refer to approximation predicates and thus to the particular symbols of the Horn clause completion of SP. They allow us to carry out proofs by *computational induction*, i.e. by induction on the approximation index i (cf. Ex. 5.6). A more elegant alternative that avoids the use of approximations is provided by the fixpoint coinduction rule introduced below.

If- and only-if-completion may increase the number of goals of a goal set enormously. If these rules are employed in a Gentzen clause *expansion*, all goals that are not used in the remaining proof can be removed:

goal set elimination $\dfrac{gs \vee hs}{hs} \Uparrow$

Note that goal set elimination is not a simplification. Hence a goal set removed in a Gentzen clause *contraction* must have a valid complement (cf. the ¬-rule).

Only-if-completion resembles the rules of *lazy resolution* and *lazy narrowing*, which are used for solving goals (cf. [Red 85], [Pad 88], [Höl 89]), but produce only one of the k goals generated by only-if-completion. Hence lazy resolution and lazy narrowing are proper expansion rules. They often come together with the rules of *constructor decomposition, constructor clash* and *variable elimination*, which are used for testing the solvability of equations and originate from the unification rules introduced by [MM 82]. In the context of a free-constructor-based functional specification they are simplification rules that are inevitable for proving or solving equations. While only-if-completion simplifies a goal set by removing defined functions, constructor decomposition and clash are needed for deleting constructors. The soundness of constructor decomposition and clash is guaranteed only if HSP_n is free-constructor-based and functional.[24]

constructor decomposition $\dfrac{F(t) \equiv_s F(u) \wedge g}{(t \equiv u \wedge g) \vee (F(t) \equiv_s F(u) \equiv_s \bot \wedge g)} \Updownarrow$

if s is a base sort and F is a constructor of SP_n

$\dfrac{F(t) \equiv_s F(u) \wedge g}{t \equiv u \wedge g} \Updownarrow$ if s is a non-base sort and F is a constructor of SP_n

constructor clash $\dfrac{F(t) \equiv_s G(u) \wedge g}{F(t) \equiv_s G(u) \equiv_s \bot \wedge g} \Updownarrow$

if s is a base sort and F and G are different constructors of SP_n

$\dfrac{F(t) \equiv_s G(u) \wedge g}{FALSE} \Updownarrow$

[23] We write $TRUE$ not only for the always true Gentzen clause, but also for the empty goal $\emptyset$, which stands for an empty conjunction and thus for the always true goal.

[24] In most cases one need not construct the Horn clause completion for proving functionality: if the built-in model SP_n-extension of Sem_{n-1} is *strongly* functional (cf. Section 8), then this property holds true for HSP_n as well.

if s is a non-base sort and F and G are different constructors of SP_n

variable elimination $\quad \dfrac{\exists\{x\}(g(x) \wedge x \equiv t)}{g(t)} \Updownarrow \quad$ if $x \in X \setminus var(t)$

Note that variable elimination may lead to multiple occurrences of t in g and thus to multiple redices for only-if-completion. Of course, the latter cannot happen if t is a normal form.

Lemma 7.3 (constructor decomposition and clash are sound) Let $SP = (SIG, AX)$ be a free-constructor-based functional specification, s be a sort of SP and $F : w \to s$, $G : w \to s$ be two different constructors of SP. Then the clauses

$$F(x) \equiv_s F(y) \quad \Leftrightarrow \quad x \equiv_w y \vee F(x) \equiv_s F(y) \equiv_s \bot, \tag{1}$$

$$F(x) \equiv_s G(y) \quad \Leftrightarrow \quad F(x) \equiv_s G(y) \equiv_s \bot \tag{2}$$

are inductive theorems of SP. For non-base sorts s of SP (cf. 6.2), (1) and (2) reduce to

$$F(x) \equiv_s F(y) \quad \Leftrightarrow \quad x \equiv_w y, \tag{1}$$

$$F(x) \equiv_s G(y) \quad \Leftrightarrow \quad FALSE. \tag{2}$$

Proof. By Cor. 6.10, it is sufficient to show that (1) and (2) are reductively valid w.r.t. SP.

(1) Let $t = F(x)$, $t' = F(y)$ and $\sigma : X \to NF_{SIG\bot}$ such that $t\sigma \equiv t'\sigma$ is SP-convergent. Then $t\sigma$ and $t'\sigma$ are SP-joinable. Since $x\sigma$ and $y\sigma$ are $SIG\bot$-normal forms, $t\sigma$ and $t'\sigma$ are also $SIG\bot$-normal (if F maps into a non-base sort) or F maps into a non-base sort. In the first case $t\sigma$ and $t'\sigma$ are SP-reduced and thus equal, in particular $x\sigma = y\sigma$ because SP is free-constructor-based. In the second case both terms are SP-reducible into a bottom constant because SP is normal form complete and for all $l \equiv r \Leftarrow h$, l contains $\bot$ only if $r = \bot$. Hence $x\sigma \equiv y\sigma$ or $t\sigma \equiv \bot \equiv t'\sigma$ is SP-convergent and thus strongly SP-convergent because SP is functional. The $\Leftarrow$-part of (1) follows from the F-compatibility of $\equiv$.

(2) Let $t = F(x)$, $t' = G(y)$ and $\sigma : X \to NF_{SIG\bot}$ such that $t\sigma \equiv t'\sigma$ is SP-convergent. We conclude as in the proof of (1) that $t\sigma$ and $t'\sigma$ are either equal or SP-reducible into a bottom constant. The first case contradicts the assumption that F and G are different. ❑

Thm. 5.18 presented conditions for the existence of inequality predicates in action extensions. From Lemma 7.3 we obtain an analogous result for functional specifications:

Definition 7.4 Let $SP = (SIG, AX)$ be a constructor-based specification. The **inequality extension** SPN **of** SP, is the least extension of SP that contains for all sorts s, sort tuples $w = s_1 \dots s_n$ and each two different constructors $F : w \to s$ and $G : v \to s$ of SP the predicates $Def_w : w$ and $\not\equiv_w$: ww and the axioms

$$\begin{aligned}
&Def_s(F(x)) \;\Leftarrow\; Def_w(x),\\
&Def_w(x_1, \dots, x_n) \;\Leftarrow\; Def_{s_1}(x_1) \wedge \dots \wedge Def_{s_n}(x_n),\\
&x \not\equiv_s \bot \;\Leftarrow\; Def_s(x),\\
&F(x) \not\equiv_s F(y) \;\Leftarrow\; (x \not\equiv_w y \wedge F(x) \not\equiv_s \bot) \vee (x \not\equiv_w y \wedge F(y) \not\equiv_s \bot),\\
&F(x) \not\equiv_s G(y) \;\Leftarrow\; F(x) \not\equiv_s \bot \vee G(y) \not\equiv_s \bot,\\
&(x_1, \dots, x_n) \not\equiv_w (y_1, \dots, y_n) \Leftarrow\; x_1 \not\equiv_{s_1} y_1 \vee \dots \vee x_n \not\equiv_{s_n} y_n.
\end{aligned}$$

For non-base sorts s of SP, these axioms reduce to:

$$\begin{aligned}
&F(x) \not\equiv_s F(y) \;\Leftarrow\; x \not\equiv_w y,\\
&F(x) \not\equiv_s G(y),\\
&(x_1, \dots, x_n) \not\equiv_w (y_1, \dots, y_n) \Leftarrow\; x_1 \not\equiv_{s_1} y_1 \vee \dots \vee x_n \not\equiv_{s_n} y_n. \quad ❑
\end{aligned}$$

Theorem 7.5 (functional specifications with inequality predicates) Let SP be a free-constructor-based functional specification. Then $Ini(SP)$ has inequality predicates (cf. 5.16).

Proof. Let $SPN = (SIG, AX)$, s be a sort of SIG and $t, t' \in NF_{SIG\bot,s}$. First we show:

(1) For all $t, t' \in NF_{SIG\bot,s}$, $SPN \vdash_{cut} t \not\equiv t'$ iff $SP \not\vdash_{cut} t \equiv t'$.

Suppose that $SPN \vdash_{cut} t \not\equiv t'$. By induction on the size of t and t' we prove that t and t' are not SP-equivalent. Since $\bot \not\equiv \bot$ cannot be derived from the axioms of SPN, t or t' is not a bottom constant. If t and t' were SP-equivalent, then $t \downarrow_{SP} t'$ because SP is functional. Hence $t = t'$ because t and t' are $SIG\bot$-normal and SP is free-constructor-based. Therefore neither t nor t' is a bottom constant and thus $t = F(u)$ and $t' = G(u')$ for some constructors F, G and $SIG\bot$-normal forms u, u'.

Let $F \neq G$. If t and t' were SP-equivalent, then by Lemma 7.3, $t \equiv_{SP} t' \equiv_{SP} \bot$. Above we have shown that $\bot \neq t \equiv_{SP} \bot$ leads to a contradiction. Hence t and t' are not SP-equivalent.

Let $F = G$. If t and t' were SP-equivalent, then by Lemma 7.3, $u \equiv_{SP} u'$ or $t \equiv_{SP} t' \equiv_{SP} \bot$. As above, the second case leads to a contradiction. Hence $u \equiv_{SP} u'$. By induction hypothesis, $SPN \not\vdash_{cut} u \not\equiv u'$. This contradicts $SPN \vdash_{cut} t \not\equiv t'$ because all derivations of $t \not\equiv t'$ lead via $u \not\equiv u'$. Hence t and t' are not SP-equivalent.

Conversely, suppose that t and t' are not SP-equivalent. Then t or t' is not a bottom constant. W.l.o.g. let $t \neq \bot$. By induction on the size of t and t' we show $SPN \vdash_{cut} t \not\equiv t'$. Hence $t = F(u)$ for some constructor F and $SIG\bot$-normal form u.

If s is a base sort, then t consists of constructors and thus $SPN \vdash_{cut} t \not\equiv \bot$. If $t' = \bot$, then the proof is complete. Otherwise $t' = G(u')$ for some constructor G and $SIG\bot$-normal form u'. If $F = G$, then $SP \not\vdash_{cut} t \equiv t'$ implies $SP \not\vdash_{cut} u \equiv u'$ and thus by induction hypothesis, $SPN \vdash_{cut} u \not\equiv u'$, and the axiom

$$F(x) \not\equiv_s F(y) \;\Leftarrow\; x \not\equiv y \wedge F(x) \not\equiv_s \bot$$

implies $SPN \vdash_{cut} t \not\equiv t'$. If $F \neq G$, then the axiom

$$F(x) \not\equiv_s G(y) \;\Leftarrow\; F(x) \not\equiv_s \bot$$

implies $SPN \vdash_{cut} t \not\equiv t'$.

If s is a non-base sort, then $t' = G(u')$ for some constructor G and $SIG\bot$-normal form u'. If $F = G$, then $SP \not\vdash_{cut} t \equiv t'$ implies $SP \not\vdash_{cut} u \equiv u'$ and thus by induction hypothesis, $SPN \vdash_{cut} u \not\equiv u'$. Hence the axiom

$$F(x) \not\equiv_s F(y) \;\Leftarrow\; x \not\equiv y$$

implies $SPN \vdash_{cut} t \not\equiv t'$. If $F \neq G$, then the axiom $F(x) \not\equiv_s G(y)$ implies $SPN \vdash_{cut} t \not\equiv t'$. This finishes the proof of (1).

Now let t, t' be arbitrary ground terms over SP. Since SP is normal form complete, t and t' are SP-reducible into some $SIG\bot$-normal forms u and u', respectively. Since $t \equiv_{SP} u$ and $t' \equiv_{SP} u'$, we have

$$SP \vdash_{cut} t \equiv t' \;\text{ iff }\; SP \vdash_{cut} u \equiv u'$$

as well as

$$SPN \vdash_{cut} t \not\equiv t' \;\text{ iff }\; SPN \vdash_{cut} u \not\equiv u'.$$

Hence by (1),

$$SPN \vdash_{cut} t \not\equiv t' \;\text{ iff }\; SP \not\vdash_{cut} t \equiv t'. \;❏$$

Thm. 7.5 allows us to regard inequality predicates for *constructor sorts* of SP_n as implicit safety predicates whose programs are given by the above axioms. Hence we may use the *only-if-completions* of these predicates as simplification rules in proofs of Gentzen clauses analogously to the *if-completions* of inequality predicates for *action sorts* (cf. 7.2). An action-sorted equality predicate $\equiv_s$ is also handled by if-completion. By Lemma 5.9, the if-completion rule for $\equiv_s$ reads as follows:

$$\frac{t \equiv_s u \wedge g}{act_1(t,z_1) \equiv act_1(u,z_1) \wedge \ldots \wedge act_k(t,z_k) \equiv act_k(u,z_k) \wedge g} \Updownarrow$$

where $act_1, \ldots, act_k$ are the s-actions of SP_n. The constructor-sort counterpart of this rule is provided by constructor decomposition and clash.

A constructor type can be turned into an action type by specifying projections for constructors and declaring them as actions:

$$proj_{F,i}(F(x_1,\ldots,x_k)) \equiv x_i \quad \text{for all } 1 \leq i \leq k$$

$$proj_{F,i}(G(x)) \equiv \bot \quad \text{for all } 1 \leq i \leq k \text{ and all constructors } G \neq F$$

Then constructor decomposition for F can be derived from if-completions of equality predicates and only-if-completions of projections.

Head normal forms (cf. 4.6) are usually built up of constructors (cf. Ex. 6.7). Conditions 4.6(1) and (3) of the coinduction principle imply normal form completeness w.r.t. these constructors. Moreover, 4.6(2) says that head normal forms are uniquely decomposable modulo behavioural equivalence. Consequently, the following decomposition rule for head normal forms is a simplification.

head normal form decomposition

$$\frac{t\sigma \equiv_s t\tau \wedge g}{x_1\sigma \equiv x_1\tau \wedge \ldots \wedge x_k\sigma \equiv x_k\tau \wedge g} \Updownarrow$$

if t belongs to a set of s-head normal forms for some (Sem_{i-1}, SP_i), $1 < i \leq n$, and $var(t) = \{x_1, \ldots, x_k\}$

Proofs of negated goals such as required by the $\neg$-rule are often carried out by *narrowing*. This procedure for solving equations was introduced by [Sla 74] and meanwhile became the basic execution mechanism of functional-logic programming languages (cf. [Han 94]). The underlying calculus is a generalization of the reduction calculus (cf. 6.4):

Definition 7.6 (narrowing calculus) Let $SP = (SIG, AX)$ be a constructor-based specification. A pair $\langle g, \tau\rangle$ consisting of a goal g over SIG and a substitution $\tau : X \to NF_{SIG}(X)$ such that $var(g) \cap dom(\tau) = \emptyset$ is called a SIG**-narrowing pair**. The **narrowing calculus for** SP consists of the following inference rules each of which transforms a SIG-narrowing pair into a SIG-narrowing pair. Let σ be a substitution over SIG.

narrowing rule

$$\frac{\langle g(t), \tau\rangle}{\langle g(v)\sigma \wedge h\sigma, \tau\sigma_{var(g(t))}\rangle}$$

if $t \notin X$, $t\sigma = u\sigma$, $c = (u \equiv v \Leftarrow h) \in AX$, $fresh(c)\sigma \subseteq NF_{SIG}(X)$ and $\langle g(v)\sigma \wedge h\sigma, \tau\sigma_{var(g(t))}\rangle$ is a SIG-narrowing pair

unification rule

$$\frac{\langle g \wedge t \equiv u, \tau\rangle}{\langle g\sigma, \tau\sigma\rangle}$$

if $t\sigma = u\sigma$ and $\langle g\sigma, \tau\sigma\rangle$ is a SIG-narrowing pair

A sequence $np_1, \ldots, np_n$ of SIG-narrowing pairs is called a **narrowing expansion of** np_1 **into** np_n **upon** SP if for all $1 \leq i \leq n$, np_{i+1} is obtained from np_i by a single narrowing or unification step. The corresponding inference relation is denoted by $\vdash\!\surd_{SP}$. Let $np = \langle g, \tau\rangle$ be a SIG-narrowing pair. g is SP**-narrowable** if $np \vdash\!\surd_{SP} np'$ for some np'. Otherwise g is SP**-narrowed**. ❑

The narrowing calculus is a generalization of the reduction calculus. Conversely, goal reductions can be lifted to narrowing expansions:

Fact 7.7 Let $SP = (SIG, AX)$ be a free-constructor-based specification and $\tau : X \to NF_{SIG}(X)$. Then

$$g\tau \vdash_{SP} \emptyset \quad \text{iff} \quad \langle g, id\rangle \vdash\!\!\surd_{SP} \langle \emptyset, \tau\rangle. \quad ❏$$

Moreover, if SP is functional, then narrowing computes all ground normal form solutions of a goal g. Indeed, Fact 7.7 and Thm. 6.8 imply

$$SP \vdash_{cut} g\tau \quad \text{iff} \quad \langle g, id\rangle \vdash\!\!\surd_{SP} \langle \emptyset, \tau\rangle$$

for all $\tau : X \to NF_{SIG}$. Hence the following refutation rule is a simplification provided that $HSP_n = (SIG, AX)$ is functional:

refutation by narrowing $\quad \dfrac{g}{FALSE} \Updownarrow \quad$ if there is no $\tau : X \to NF_{SIG}$ such that $\langle g, id\rangle \vdash\!\!\surd_{HSP_n} \langle \emptyset, \tau\rangle$

The usual criterion for the applicability condition of this rule is non-narrowability of g (cf. 7.6). The general case, however, is a tree of narrowing expansions of $\langle g, id\rangle$ such that each path either leads into a cycle or ends with a non-narrowable goal.

If SP is free-constructor-based, functional and strongly terminating, then narrowing computes all ground normal form solutions even if the narrowing rule is applied only to the leftmost-innermost redex t in the current goal g (cf. [Pad 95], Lemmata 9.3, 9.6, 9.7).

This gives rise to the following alternative to only-if-completion, which, by [Pad 95], Lemma 8.6, is also a simplification rule.

innermost narrowing of F $\quad \dfrac{g(F(t))}{\exists X_1(g(u_1)\sigma \wedge h_1\sigma \wedge EQ(\sigma_1')) \vee \ldots \vee \exists X_k(g(u_k)\sigma \wedge h_k\sigma \wedge EQ(\sigma_k'))^{25}} \Updownarrow$

if $F(t)$ is the leftmost-innermost HSP_n-narrowable subterm of $g(F(t))$,

$$prog(F) \;=\; \{F(t_1) \equiv u_1 \Leftarrow h_1, \ldots, F(t_k) \equiv u_k \Leftarrow h_k\}$$

and for all $1 \leq i \leq k$, σ_i is a minimal unifier of t and t_i, $X_i = var(dom(\sigma_i)\sigma_i)$ and $\sigma_i' = (\sigma_i)_{in(g(F(t)))}$

All goal set rules defined so far are simplifications and thus can be used for transforming both the conclusion and the premise of a Gentzen clause. Let us recapitulate the individual purposes. Only-if-completion and innermost narrowing expand a function or predicate according to its defining axioms. Constructor decomposition, if- and only-if-completion of equality and inequality predicates simplify equations and inequations. For eliminating an action equality or liveness predicate P we may also apply the gfp-rule for P and derive the succedents of this rule by induction using approximation rules. However, below we will see how these induction steps can be replaced by a single application of the *fixpoint coinduction rule*. In simple cases where (co)induction steps are not necessary, if- and only-if-completion and term replacement will do most of the job. Constructor clash and refutation by narrowing remove entire goals. Variable elimination is applied to equations $x \equiv t$, especially if t cannot be simplified any further unless within a new goal context.

Only-if-completion applies axioms at outermost positions. Innermost narrowing does the same at innermost positions. Hence the latter rule automatically discards all axioms whose lefthand sides do not unify with the chosen redex. On the one hand, this is more efficient than only-if-completion because less subgoals are produced.

[25] cf. Section 2

On the other hand, the combination of only-if-completion and constructor clash (cf. Ex. 7.8) yields an outermost strategy that often detects and removes unsolvable equations faster than innermost narrowing.

Innermost narrowing performs in parallel several *lemma* or *deductive paramodulation* steps upon axioms (cf. [Pad 91], [Pad 92], [Pad 95]). Deductive paramodulation and deductive resolution are rules for applying an arbitrary Gentzen clause theorem c. Due to the generation of the unifier σ (see below) these rules are proper expansion rules rather than simplifications. The following version called *deductive expansion* combines the paramodulation of subterms with the resolution of subgoals. All rules defined above are compositions of deductive expansion steps.

Suppose that $c \in Th_n$ (cf. 7.2) has the form $(\exists X_1 u \equiv u_1 \vee \ldots \vee \exists X_k u \equiv u_k \vee \exists X_{k+1} u_{k+1} \vee \ldots \vee \exists X_l u_l) \Leftarrow h$, i.e. $u_1, \ldots, u_k$ are terms, while $u_{k+1}, \ldots, u_l$ are goals. Deductive expansion upon c comprises all ways of applying c backward: Let $gs = g_1(t_1) \vee \ldots \vee g_k(t_k) \vee (g_{k+1} \wedge t_{k+1}) \vee \ldots \vee (g_l \wedge t_l)$.

deductive expansion upon c

$$\frac{gs}{g_1(u)\sigma \wedge \ldots \wedge g_k(u)\sigma \wedge g_{k+1}\sigma \wedge \ldots \wedge g_l\sigma \wedge h\sigma \wedge EQ(\sigma_{in(gs)})} \Uparrow$$

if for all $1 \leq i \leq l$, σ is a renaming of X_i away from $var(c)$ (cf. Sect. 2) and a minimal unifier of t_i and u_i and $X_i\sigma \cap (in(gs) \cup var(in(gs)\sigma \cup g_i(u)\sigma)) = \emptyset$

For instance, deductive expansion upon the reflexivity axiom $x \equiv x$ amounts to the unification of the two sides of equation:

unification

$$\frac{g \wedge t \equiv u}{g\sigma \wedge EQ(\sigma_{in(g \wedge t \equiv u)})} \Uparrow \quad \text{if } \sigma \text{ is a minimal unifier of } t \text{ and } u$$

A correspondingly general *contraction* rule comprising all forward applications of c as well as expansion and contraction rules for applying induction hypotheses were developed and employed in a number of proofs in [Pad 92], [Pad 94], [Pad 95] and [Pad 96]. Based on given induction orderings, inductive expansion and contraction produce *descent conditions* to ensure that the induction hypotheses applied in the proof of a conjecture c are indeed "smaller" than c itself. This allows us to perform induction steps in without referring to particular induction schemata. However, these proofs use induction orderings explicitly, which is avoided by fixpoint induction and coinduction as described in Section 1.5. Given a monotone function $\Phi : \wp(D) \to \wp(D)$, fixpoint induction and coinduction means employing the equivalence transformations

$$\frac{lfp(\Phi) \subseteq A}{\exists B : \Phi(B) = B \subseteq A} \Updownarrow \quad \text{and} \quad \frac{A \subseteq gfp(\Phi)}{\exists B : A \subseteq B = \Phi(B)} \Updownarrow$$

respectively. They correspond to the following Gentzen rules whose correctness follows from Lemmata 5.15 and 6.13 (for induction) resp. Thm. 4.9 and Lemma 5.9 (for coinduction), i.e. four of the results mentioned at the beginning of this section. Note that the predicate Q occurring in the rule succedents is existentially quantified. Q corresponds to the set B above. Hence fixpoint induction and coinduction are second-order rules. This is not surprising because all rules capturing induction schemas quantify over predicates. Moreover, the following rules can be generalized to $n > 1$ predicates $Q_1, \ldots, Q_n$ for handling (e.g. mutual-recursively defined) predicates $P_1, \ldots, P_n$ (cf. Section 10.9).

fixpoint induction on P

$$\frac{gs \;\Leftarrow\; P(x)}{\exists Q : \; prog(P)[Q/P], \;\; gs \;\Leftarrow\; Q(x)} \Updownarrow$$

if P is a safety predicate or the graph of a defined function and $prog(P)$ is the SP_n-program for P (cf. 5.13 and 6.13)

fixpoint coinduction on P

$$\frac{P(x) \;\Leftarrow\; gs}{\exists Q : \; live(P)[Q/P], \;\; Q(x) \;\Leftarrow\; gs} \Updownarrow$$

if P is a liveness or action-sorted equality predicate of SP_n and $live(P)$ is the set of liveness axioms for P (cf. 5.9)

The first rule succedent $prog(P)[Q/P]$ resp. $live(P)[Q/P]$ actually implies $Q(x) \Leftarrow P(x)$ resp. $P(x) \Leftarrow Q(x)$. Due to the second rule succedent $gs \Leftarrow Q(x)$ resp. $Q(x) \Leftarrow gs$, $Q(x)$ is in fact a generalization resp. specialization of the original conclusion resp. premise gs. When applying fixpoint induction/coinduction one may always start with $Q(x) =_{def} gs$. Only if the proof of $prog(P)[Q/P]$ resp. $live(P)[Q/P]$ fails with this choice of Q, then Q must be given as a proper generalization/specialization of gs. Hence the expansion property of fixpoint induction and coinduction follows from an intermediate *cut step*:

$$\cfrac{gs \;\Leftarrow\; P(x)}{\cfrac{\exists Q : gs \;\Leftarrow\; Q(x) \;\Leftarrow\; P(x)}{prog(P)[Q/P],\;\; gs \;\Leftarrow\; Q(x)}\Uparrow}\Updownarrow \qquad \cfrac{P(x) \;\Leftarrow\; gs}{\cfrac{\exists Q : P(x) \;\Leftarrow\; Q(x) \;\Leftarrow\; gs}{live(P)[Q/P],\;\; Q(x) \;\Leftarrow\; gs}\Uparrow}\Updownarrow$$

Both rules are contractions because we may choose Q as P.

The following simplifications are useful for establishing clauses that match the antecedents of the fixpoint rules.

premise splitting $$\frac{gs \;\Leftarrow\; h_1 \vee \ldots \vee h_n}{gs \;\Leftarrow\; h_1, \ldots,\; gs \;\Leftarrow\; h_n}\Updownarrow$$

variable introduction $$\frac{gs(t) \;\Leftarrow\; h}{gs(x) \;\Leftarrow\; x \equiv t \wedge h}\Updownarrow \qquad \text{if } x \notin var(gs(t) \Leftarrow h)$$

For proofs employing fixpoint induction see Section 10.1.1. The following example illustrates fixpoint coinduction along with typical applications of other rules presented in this section. Further proofs with fixpoint coinduction steps can be found in Examples 9.12 and 9.15.

Example 7.8 We start out from an action extension of SP_1 (cf. 2.5):

```
SP_4
     base      SP_1
     sorts     stream
     acts      head:stream->nat
               tail:stream->stream
     opns      &:nat*stream->stream
               01,10,oi:stream
               @:list*stream->stream
     preds     fair:stream
     infixes   &  @
     vars      n:nat  L:list  x,y:stream
     axioms    head(n&x) = n
               tail(n&x) = x
               head(01) = 0
               tail(01) = 10
               head(10) = 1
               tail(10) = 01
               head(oi) = 0
               head(tail(oi)) = 1
               tail(tail(oi)) = oi
               nil@x = x
               (n&L)@x = n&(L@x)
               fair(x)  ==>  \/{L,y}(s = L@(0&y) /\ fair(y))
```

It can be shown that the Horn clause completion of $(Ini(SP_1), SP_4)$ is free-constructor-based, functional and strongly terminating. Hence any of the above rules can be used to show that (1) $01 \equiv 0\&10$, (2) $01 \equiv oi$, (3) $01 \not\equiv 10$ and (4) $fair(01)$ are coinductive theorems of $(Ini(SP_1), SP_4)$.

Proof of (1):

$01 \equiv 0\&10$

if-completion of $\equiv_{stream}$

$\vdash \quad head(01) \equiv head(0\&10) \wedge tail(01) \equiv tail(0\&10)$

term replacement upon axioms

$\vdash \quad 0 \equiv 0 \wedge 10 \equiv 10$

constructor decomposition and $\neg$-rule

$\vdash \quad TRUE$

Proof of (2):

$01 \equiv oi$

variable introduction

$\vdash \quad x \equiv y \ \Leftarrow\ x \equiv 01 \wedge y \equiv oi$

fixpoint coinduction on $\equiv_{stream}$

$\vdash \quad head(x) \equiv head(y) \wedge Q(tail(x), tail(y)) \ \Leftarrow\ Q(x,y),$

$Q(x,y) \ \Leftarrow\ x \equiv 01 \wedge y \equiv oi$ *specialization*

only-if-completion of Q upon $prog(Q) = \{Q(01, oi),\ Q(10, tail(oi))\}$

$\vdash \quad (head(x) \equiv head(y) \wedge tail(x) \equiv 01 \wedge tail(y) \equiv oi)$

$\vee\ (head(x) \equiv head(y) \wedge tail(x) \equiv 10 \wedge tail(y) \equiv tail(oi))$

$\Leftarrow\ (x \equiv 01 \wedge y \equiv oi) \vee (x \equiv 10 \wedge y \equiv tail(oi)),$

$(x \equiv 01 \wedge y \equiv oi) \vee (x \equiv 10 \wedge y \equiv tail(oi)) \ \Leftarrow\ x \equiv 01 \wedge y \equiv oi$

subsumption

$\vdash \quad (head(x) \equiv head(y) \wedge tail(x) \equiv 01 \wedge tail(y) \equiv oi)$

$\vee\ (head(x) \equiv head(y) \wedge tail(x) \equiv 10 \wedge tail(y) \equiv tail(oi))$

$\Leftarrow\ (x \equiv 01 \wedge y \equiv oi) \vee (x \equiv 10 \wedge y \equiv tail(oi))$

variable introduction

$\vdash \quad (head(01) \equiv head(oi) \wedge tail(01) \equiv 01 \wedge tail(oi) \equiv oi)$

$\vee\ (head(01) \equiv head(oi) \wedge tail(01) \equiv 10 \wedge tail(oi) \equiv tail(oi)),$

$(head(10) \equiv head(tail(oi)) \wedge tail(10) \equiv 01 \wedge tail(tail(oi)) \equiv oi)$

$\vee\ (head(10) \equiv head(tail(oi)) \wedge tail(10) \equiv 10 \wedge tail(tail(oi)) \equiv tail(oi))$

term replacement upon axioms

$\vdash \quad (head(01) \equiv head(oi) \wedge tail(01) \equiv 01 \wedge tail(oi) \equiv oi)$

$\vee\ (0 \equiv 0 \wedge 10 \equiv 10 \wedge tail(oi) \equiv tail(oi)),$

$(1 \equiv 1 \wedge 01 \equiv 01 \wedge oi \equiv oi)$

$\vee\ (head(10) \equiv head(tail(oi)) \wedge tail(10) \equiv 10 \wedge tail(tail(oi)) \equiv tail(oi))$

unification

$\vdash \quad (head(01) \equiv head(oi) \wedge tail(01) \equiv 01 \wedge tail(oi) \equiv oi) \vee\ TRUE,$

$TRUE \vee\ (head(10) \equiv head(tail(oi)) \wedge tail(10) \equiv 10 \wedge tail(tail(oi)) \equiv tail(oi))$

stop rule

$\vdash \quad TRUE$

Proof of (3):

$01 \not\equiv 10$

if-completion of $\not\equiv_{stream}$

$\vdash \quad head(01) \not\equiv head(10) \vee tail(01) \not\equiv tail(10)$

term replacement upon axioms

$\vdash \quad 0 \not\equiv 1 \vee tail(01) \not\equiv tail(10)$

only-if-completion of $\not\equiv_{nat}$ and constructor clash

$\vdash \quad TRUE \vee tail(01) \not\equiv tail(10)$

stop rule

$\vdash \quad TRUE$

Proof of (4):

$fair(01)$

variable introduction

$\vdash \quad fair(x) \Leftarrow x \equiv 01$

fixpoint coinduction on $fair$

$\vdash \quad \exists\{L,y\}(x \equiv L@(0\&y) \wedge Q(y)) \Leftarrow Q(x),$

$Q(x) \Leftarrow x \equiv 01$ *specialization*

only-if-completion of Q upon $prog(Q) = \{Q(01),\ Q(10)\}$

$\vdash \quad \exists\{L,y\}(x \equiv L@(0\&y) \wedge y \equiv 01) \vee \exists\{L,y\}(x \equiv L@(0\&y) \wedge y \equiv 10) \Leftarrow x \equiv 01 \vee x \equiv 10,$

$x \equiv 01 \vee x \equiv 10 \Leftarrow x \equiv 01$

subsumption

$\vdash \quad \exists\{L,y\}(x \equiv L@(0\&y) \wedge y \equiv 01) \vee \exists\{L,y\}(x \equiv L@(0\&y) \wedge y \equiv 10) \Leftarrow x \equiv 01 \vee x \equiv 10$

goal set elimination

$\vdash \quad \exists\{L,y\}(x \equiv L@(0\&y) \wedge y \equiv 10) \Leftarrow x \equiv 01 \vee x \equiv 10$

only-if-completion of @

$\vdash \quad \exists\{L,y\}(L \equiv nil \wedge x \equiv 0\&y \wedge y \equiv 10)$

$\vee\exists\{L,y,n,L'\}(L \equiv n\&L' \wedge x \equiv n\&(L'@(0\&y)) \wedge y \equiv 10)$

$\Leftarrow x \equiv 01 \vee x \equiv 10$

variable elimination

$\vdash \quad \exists\{y\}(x \equiv 0\&y \wedge y \equiv 10) \vee \exists\{y,n,L'\}(x \equiv n\&(L'@(0\&y)) \wedge y \equiv 10)$

$\Leftarrow x \equiv 01 \vee x \equiv 10$

term replacement upon (1)

$\vdash \quad (x \equiv 01 \wedge 10 \equiv 10) \vee \exists\{n,L'\}(x \equiv n\&(L'@01) \wedge 10 \equiv 10) \Leftarrow x \equiv 01 \vee x \equiv 10$

unification

$\vdash \quad x \equiv 01 \vee \exists\{n,L'\}x \equiv n\&(L'@01) \Leftarrow x \equiv 01 \vee x \equiv 10$

only-if-completion of @

$\vdash \quad x \equiv 01 \vee \exists\{n,L'\}(L' \equiv nil \wedge x \equiv n\&01)$

$\vee\exists\{n,L',m,L''\}(L' \equiv m\&L'' \wedge x \equiv n\&m\&(L''@01))$

$\Leftarrow x \equiv 01 \vee x \equiv 10$

goal set elimination

$\vdash \quad x \equiv 01 \vee \exists\{n,L'\}(L' \equiv nil \wedge x \equiv n\&01) \Leftarrow x \equiv 01 \vee x \equiv 10$

variable elimination

$\vdash \quad x \equiv 01 \vee \exists\{n\}x \equiv n\&01 \Leftarrow x \equiv 01 \vee x \equiv 10$

term replacement upon $1\&01 \equiv 10$

$\vdash \quad x \equiv 01 \vee x \equiv 10 \Leftarrow x \equiv 01 \vee x \equiv 10$

subsumption

$\vdash \quad TRUE$

Alternative *inductive* proofs of (2) and (4) would employ gfp- and approximation rules (cf. 7.2) and thus explicitly use axioms of the Horn clause completion of $(Ini(SP_1), SP_4)$ (cf. Ex. 5.6). ❏

8 Strong Functionality

Let us now turn to the question how to ensure that a constructor-based specification is ground confluent. The main additional requirement to SP that allows to "localize" confluence is *strong termination*:

Definition 8.1 (strong termination) A constructor-based specification $SP = (SIG, AX)$ is **strongly terminating** if there is a **reduction ordering for** SP, i.e. a transitive and well-founded relation $>$ on T_{SIG} such that the constant *true* is minimal w.r.t. $>$ (cf. 6.1) and the following two conditions hold true:

(*rewrite compatibility upon* AX) For all $c = (l \equiv r \Leftarrow h) \in AX$, $t(x) \in T_{SIG}(X)$ and $\sigma \in T_{SIG}^X$, $fresh(c)\sigma \subseteq NF_{SIG}$ and $h\sigma \vdash_{SP} \emptyset$ imply $t(l\sigma) > t(r\sigma)$ and $t\sigma > h\sigma$.[26]

(*subterm compatibility*) For all $t \in T_{SIG}$ and all proper subterms u of t, $t > u$. ❑

A uniform construction of $>$ that takes into account both the syntax and the semantics of SP is given in [Pad 92], Sect. 6.2, and [Pad 95], Sect. 7. The first requirement to $>$ (rewrite compatibility) is weaker then similar conditions introduced in the CTRS community[27]: neither *reductive* (cf. [JW 86]) nor *simplifying* (cf. [Kap 87]) nor *decreasing*[28] (cf. [DOS 88]) nor *quasi-reductive* CTRS (cf. [BG 89], [GW 93]) restrict their "descent conditions" to joinable axiom premise instances $h\sigma$.

Similarly, we have weakened the second condition on SP, which – together with strong termination – ensures ground confluence, that is the notion of a joinable critical pair. While [DOS 87] demands that overlapping axioms with joinable premises must have a joinable conclusion, [Pad 92] shows that only **strongly** convergent premises need to have a joinable conclusion (cf. Def. 6.6). The corresponding result uses the notion of subreductive validity (cf. [Pad 92], p. 108):

Definition 8.2 (subreductive validity w.r.t. SP) Given a specification $SP = (SIG, AX)$ with reduction ordering $>$ and a SIG-term t, a Horn clause $p \Leftarrow h$ over SIG is **sub-t-reductively valid w.r.t. SP** if for all $\sigma \in T_{SIG}^X$ such that all SP-convergent ground goals $g < t\sigma$ are strongly SP-convergent,

$$h\sigma \vdash_{SP} \emptyset \quad \text{implies} \quad p\sigma \vdash_{SP} \emptyset. \ ❑$$

[Pad 92] requires *strong* convergence, but a short glance at the proof of [Pad 92], Thm. 6.10, reveals that convergence is sufficient. The stronger definition of subreductive validity in [Pad 92] was motivated by the proof method presented in [Pad 92], Section 6.4. We replace this method here by Criteria 8.9 and 8.11, which are tailored to proving that *critical pairs* are subreductively valid. If all critical pairs induced by axioms of SP are subreductively valid, then a strongly terminating specification is ground confluent (Thm. 8.4). Critical pairs arising from conditional equations are extended to *conditional critical pairs (CCPs)* or *critical clauses*:

Definition 8.3 (critical clause) Given a Horn clause specification $SP = (SIG, AX)$, let $t(x) \in T_{SIG}(X)$,

$$c = (l \equiv r \Leftarrow g), \quad c' = (l' \equiv r' \Leftarrow h) \in \ AX$$

and σ, τ be minimal substitutions such that $l\sigma = t(l'\tau)$ and $label(t, w) = x$ implies $label(l, w) \notin X$ (cf. Sect. 2). Then the Horn clause

$$cc \quad = \quad r\sigma \equiv t(r'\tau) \ \Leftarrow \ g\sigma \cup h\tau$$

is the SP**-critical clause of** c **on** c' **at (position** w **of)** $t\sigma$. We also say that cc is **induced by** c **and** c' **at** $t\sigma$. **SP** is **critical clause convergent** if for each $t \in T_{SIG}(X)$, all SP-critical clauses induced at t are sub-t-reductively valid w.r.t. SP. ❑

Theorem 8.4 (Superposition Theorem I) ([Pad 92], Thm. 6.10 and Lemma 6.13) A strongly terminating and normal form complete specification SP is ground confluent iff SP is critical clause convergent. ❑

[27] CTRS = conditional term rewriting systems

[28] "Decreasing" is "strongly terminating" without the premise $h\sigma \vdash_{SP} \emptyset$ in Def. 8.1.

Proofs of ground confluence can be carried out in a hierarchical way if a subspecification of SP is already known to be ground confluent. For this purpose we weaken the notion of subreductive validity and adapt it to hierarchical specifications (cf. 3.7). The extensions considered here are restricted to additional axioms, i.e. the signature is not changed. For adding new operations or predicates in a consistent way one may use Thm. 9.2 and Cor. 9.3.

Definition 8.5 (subreductive validity w.r.t. (SP,C)) Given a specification $SP = (SIG, AX)$ with reduction ordering $>$, a set C of Horn clauses over SIG, let $SP' = (SIG, AX \cup C)$ and e be a SIG-equation. A Horn clause $p \Leftarrow h$ over SIG is **sub-e-reductively valid w.r.t. (SP,C)** if for all $\sigma \in T_{SIG}^X$ such that all SP'-convergent ground goals $g < e\sigma$ are strongly SP'-convergent *and SP-convergent*,

$$h\sigma \vdash_{SP'} \emptyset \quad \text{implies} \quad p\sigma \vdash_{SP'} \emptyset.$$

Let $SP' = (SIG, AX \cup C)$. **(SP,C)** is **critical clause convergent** if for all $t \in T_{SIG}(X)$ and SP'-critical clauses $cc = (u \equiv v \Leftarrow h)$ induced at t, c is sub-$(t \equiv u)$-reductively valid w.r.t. (SP, C). ❑

Critical clause convergence of (SP, C) implies ground confluence of SP' and the stronger "rewrite version" of consistency w.r.t. $Ini(SP)$ (cf. 3.7):

Definition 8.6 Given a constructor-based specification $SP = (SIG, AX)$ and a set C of Horn clauses over SIG, $SP' = (SIG, AX \cup C)$ is **reductively consistent w.r.t.** SP if for all ground atoms p over SIG, $p \vdash_{SP'} \emptyset$ implies $p \vdash_{SP} \emptyset$. ❑

Proposition 8.7 Given a ground confluent specification $SP = (SIG, AX)$ and a set C of Horn clauses over SIG, $SP' = (SIG, AX \cup C)$ is ground confluent if SP' is reductively consistent w.r.t. SP.

Proof. Let g be an SP'-convergent ground goal such that $g \longrightarrow_{SP'} g'$ for some g'. Since SP' is reductively consistent w.r.t. SP, there is a ground goal g'' with $g \longrightarrow_{SP}^* g'' \longleftarrow_{SP}^* g'$. Since SP is ground confluent, g'' and thus g' are SP-convergent. ❑

Theorem 8.8 (Superposition Theorem II) Let $SP = (SIG, AX)$ be a constructor-based specification and C be a set of Horn clauses over SIG such that $SP' = (SIG, AX \cup C)$ is strongly terminating and normal form complete. SP' is ground confluent and reductively consistent w.r.t. SP iff (SP, C) is critical clause convergent.

Proof. Almost the same as the proof of Thm. 8.4. For that theorem one has to show that each SP-convergent ground atom p is strongly SP-convergent, and the proof is done by induction on p along the given reduction ordering for SP (cf. [Pad 92], Thm. 6.10). For the present theorem, one has to show that each SP'-convergent ground atom p is strongly SP'-convergent and SP-convergent. Analogously to Thm. 8.4, the proof is done by induction on p along the given reduction ordering for SP. ❑

This theorem will be used for proofs of consistency (cf. 9.4). Here we apply it in order to infer the ground confluence of SP' from critical clause convergence of (SP, C). A critical clause cc is subreductively valid if cc is an inductive theorem of SP (cf. Lemma 8.9) or if cc can be derived from *e-bounded* inductive theorems of SP and clauses of C (cf. Thm. 8.11).

Lemma 8.9 (Criterion I for subreductive validity w.r.t. (SP, C)) Given a strongly terminating functional specification $SP = (SIG, AX)$ and a set C of Horn clauses over SIG, let $SP' = (SIG, AX \cup C)$, $cc = (u \equiv v \Leftarrow h)$ be an SP'-critical clause induced at a term t and $e = (t \equiv u)$.

cc is sub-e-reductively valid w.r.t. (SP, C) if cc is an inductive theorem of SP.

Proof. Let $>$ be the reduction ordering for SP, $cc \in ITh(SP)$ and σ be a ground substitution such that $h\sigma$ is SP'-convergent and all SP'-convergent ground goals $g < e\sigma$ are SP-convergent. $e\sigma > t\sigma > h\sigma$ implies that $h\sigma$ is SP-convergent. Hence $SP \vdash_{cut} h\sigma$. Since $cc \in ITh(SP)$, $SP \vdash_{cut} u\sigma \equiv v\sigma$ and thus, by Thm. 6.8, $u\sigma \equiv v\sigma \vdash_{SP} \emptyset$. ❑

Definition 8.10 (e-bounded clause) Given a specification $SP = (SIG, AX)$ with reduction ordering $>$ and a SIG-equation e, a clause $l \equiv r \Leftarrow h$ is ***e*-bounded w.r.t.** $>$ if for all terms t and ground substitutions σ, τ, $SP \vdash_{cut} h\tau$ and $e\sigma > t(l\tau)$ imply $e\sigma > t(r\tau)$.

Let C, D be sets of Horn clauses. C is e-bounded if all $c \in C$ are e-bounded. C is D-bounded if for all $e \Leftarrow h \in D$, C is e-bounded. ❏

Since $>$ is a reduction ordering for SP, the axioms of SP are e-bounded for any SIG-equation e. However, the condition that C is e-bounded for a given e is usually weaker than the strong termination of $SP \cup C$.

For a set C of Horn clauses, let **prem(C)** denote the union of all premises of clauses of C and **concl(C)** stand for the goal consisting of all conclusions of clauses of C.

Theorem 8.11 (Criterion II for subreductive validity w.r.t. (SP, C)**)** Given the assumptions of Lemma 8.9, let $C' \subseteq C$, $C_1, \ldots, C_n \subseteq ITh(SP)$, $SP_i = (SIG, AX \cup C' \cup concl(C_i)\rho)$ and ρ substitute unique constants, which do not occur in SIG, for the variables of $concl(C_i)$.[29]

cc is sub-e-reductively valid w.r.t. (SP, C) if $C' \cup C_1 \cup \ldots \cup C_n$ is e-bounded w.r.t. $>$,

$$c \;=\; prem(C_1) \vee \ldots \vee prem(C_n) \Leftarrow h$$

is an inductive theorem of SP and for all $1 \leq i \leq n$, $u \equiv v$ is SP_i-convergent.

Proof. Let σ be a ground substitution such that $h\sigma$ is SP'-convergent and all SP'-convergent ground goals $g < e\sigma$ are SP-convergent. Hence $e\sigma > t\sigma > h\sigma$ and $c \in ITh(SP)$ imply $SP \vdash_{cut} prem(C_i)\sigma$ for some $1 \leq i \leq n$. Since $u \equiv v$ is SP_i-convergent, there is a term t' such that $u \longrightarrow^*_{SP_i} t' \longleftarrow^*_{SP_i} v$. Suppose that

(1) $u\sigma \downarrow_{SP} t'\sigma$,

(2) $v\sigma \downarrow_{SP} t'\sigma$,

(3) $e\sigma > t'\sigma$.

Since SP is ground confluent, (1) and (2) imply $u\sigma \downarrow_{SP} v\sigma$. Hence it is sufficient to show (1) to (3). We prove (1) by induction on the number of SP_i-reduction steps leading from u to t'. (3) can be shown analogously by induction on the number of SP_i-reduction steps leading from v to t'.

If $u = t'$, then $e\sigma > t\sigma > u\sigma = t'\sigma$ because *cc* is induced at t. Otherwise there are terms u', v' such that

$$u \longrightarrow^*_{SP_i} u' \longrightarrow_{SP_i} v' \longrightarrow^*_{SP} t',$$

and the step from from u' to v' is performed

(3) either by applying a clause $l \equiv r \Leftarrow h' \in C'$ such that $u' = c(l\tau)$, $v' = d(r\tau)$ and $h'\tau \vdash_{SP'} \emptyset$ for some d, τ

(4) or by applying a clause $l \equiv r \Leftarrow h' \in C_i$ such that $u' = c(l\rho)$ and $v' = d(r\rho)$ for some d.

By induction hypothesis, $u \longrightarrow^*_{SP_i} u'$ implies $u\sigma \downarrow_{SP} u'\sigma$ and $e\sigma > u'\sigma$.

In Case (3), $u'\sigma \longrightarrow_{SP'} v'\sigma$ and $e\sigma > u'\sigma \geq l\tau\sigma > h'\tau\sigma \vdash_{SP'} \emptyset$. Hence $h'\tau\sigma \vdash_{SP} \emptyset$ and thus $e\sigma > v'\sigma$ because $l \equiv r \Leftarrow h'$ is e-bounded w.r.t. $>$. Moreover, $u'\sigma \downarrow_{SP} v'\sigma$ because $u'\sigma \longrightarrow_{SP'} v'\sigma$ and $e\sigma > \{u'\sigma, v'\sigma\}$.

In Case (4), $SP \vdash_{cut} prem(C_i)\sigma$ implies $SP \vdash_{cut} h'\sigma$. Hence $SP \vdash_{cut} u'\sigma \equiv v'\sigma$ and thus $u'\sigma \downarrow_{SP} v'\sigma$ because SP is ground confluent. Since $l \equiv r \Leftarrow h'$ is e-bounded w.r.t. $>$, we have $e\sigma > v'\sigma$.

[29] This technical assumption prevents the variables of $concl(D_i)$ from being replaced within goal reductions.

Hence in both cases, $e\sigma > v'\sigma \longrightarrow^*_{SP} t'\sigma$, and thus (3) holds true. Moreover,

$$u\sigma \downarrow_{SP} u'\sigma \downarrow_{SP} v'\sigma \longrightarrow^*_{SP} t'\sigma$$

implies (1) because SP is ground confluent. ❑

Lemma 8.9 and Thm. 8.11 provide hierarchical criteria for critical clause convergence and thus for ground confluence because they assume that a part of the specification is already ground confluent. Both criteria are useful for *proofs by inductive convergence* (cf. Thm. 9.10). The crucial advantage of Thm. 8.11 over Lemma 8.9 is that the conjectures given by C can be applied in a proof of C and need not be checked for being induction hypotheses.

For SP-critical clauses, subreductive validity w.r.t. SP (cf. 8.2) coincides with reductive validity (cf. 6.9):

Lemma 8.12 (reductive and subreductive validity) A normal form complete and strongly terminating specification SP with reduction ordering $>$ is critical clause convergent iff all SP-critical clauses are reductively valid w.r.t. SP.

Proof. "$\Rightarrow$": Suppose that SP is critical clause convergent. By Thm. 8.4, SP is ground confluent. Let $cc = u \equiv v \Leftarrow h$ be an SP-critical clause induced at a term t and $\sigma : X \to NF_{SIG\perp}$ such that $h\sigma$ is strongly SP-convergent. Then $h\sigma$ is SP-convergent. Moreover, *all* SP-convergent ground goals are strongly SP-convergent because SP is ground confluent. Since cc is sub-t-reductively valid, we conclude that $u\sigma \equiv v\sigma$ is SP-convergent and thus strongly SP-convergent because SP is ground confluent. Hence cc is reductively valid.

"$\Leftarrow$": Let $>$ be the reduction ordering for SP and $cc = p \Leftarrow h$ be a reductively valid SP-critical clause induced at a term t. Let σ be a ground substitution such that $h\sigma$ is SP-convergent and all SP-convergent ground goals $g < t\sigma$ are strongly SP-convergent. Since SP is normal form complete, σ has an SP-reduct $\tau : X \to NF_{SIG\perp}$. Since cc is induced at t, $h\sigma < t\sigma$. Hence $h\sigma$ and thus $h\tau$ are strongly SP-convergent. Since cc is reductively valid, $p\tau$ is strongly SP-convergent. Hence $p\sigma$ is SP-convergent and we conclude that cc is sub-t-reductively valid. ❑

Together with Thm. 8.4 the preceding lemma implies that axioms of the following type build up ground confluent specifications (cf. Thm. 8.15).

Definition 8.13 (deterministic specification) Given a signature SIG as in 6.2, a conditional equation

$$F(t) \equiv u \;\Leftarrow\; t_1 \equiv u_1 \wedge \ldots \wedge t_n \equiv u_n$$

is **deterministic** if the following conditions hold true:

(1) F does not occur in t.

(2) For all $1 \leq i \leq n$, u_i is a SIG-normal form.

(3) For all $1 \leq i \leq n$, $var(t_i) \subseteq var(t) \cup \bigcup_{j=1}^{i-1} var(u_j)$.

(4) $var(u) \subseteq var(t) \cup \bigcup_{i=1}^{n} var(u_i)$.

A constructor-based specification $SP = (SIG, AX)$ is **deterministic** if all axioms of SP are deterministic and if

(5) all SP-critical clauses induced by two *different* axioms are reductively valid w.r.t. SP.

A strongly terminating and normal form complete deterministic specification is called **strongly functional**. ❑

[AL 93] call a constructor-based specification with only deterministic axioms *strongly deterministic.* 8.13(1) is often strengthened to the condition that F is a defined function and t consists of normal forms (cf. [Han 94]).

This would exclude equational axioms with only constructors and bottom constants on the left-hand side, which are needed for guaranteeing totalizedness (cf. 3.7). 8.13(1) implies that all SP-critical clauses are *overlays* in the sense of [DOS 87]. Hence 8.13(5) intuitively means that all "overlappings" among the axioms for the same defined function lead to joinable results. If there are no overlays, SP is usually called *orthogonal* or *non-ambiguous.*

Example 8.14 The following extension of SP_1 (cf. Ex. 2.5) is deterministic. The former `opns` list of operations is now splitted into a list `consts` of constructors and a list `defs` of defined functions.

```
DIV&REP
     base      SP_1
     sorts     tree  nat2  bool
     consts    leaf:nat->tree
               (_,_):nat*nat->nat2
               #:tree*tree->tree
               true,false:bool
     defs      div:nat*nat->nat2
               -:nat*nat->nat
               minOfTree:tree->nat
               replace:tree*nat->tree
               repByMin:tree->tree
               min:nat*nat->nat
     preds     <=:nat*nat
     infixes   -  #  <=
     vars      m,n,q,r:nat  T,T':tree
     axioms    div(m,n) = (0,m)  <==  m < n
               div(m,n) = (succ(q),r)  <==  n <= m /\ div(m-m,n) = (q,r)
               0-n = 0
               succ(m)-succ(n) = m-n
               minOfTree(leaf(m)) = m
               minOfTree(T#T') = min(m,n)  <==  minOfTree(T) = m /\ minOfTree(T') = n
               replace(leaf(m),n) = leaf(n)
               replace(T#T',n) = replace(T,n)#replace(T',n)
               repByMin(T) = T'  <==  minOfTree(T) = m /\ replace(T,m) = T'
               min(m,n) = m  <==  m <= n
               min(m,n) = n  <==  n <= m
               0 <= n
               succ(m) <= succ(n)  <==  m <= n
```

Note that the axioms

```
min(m,n) = m  <==  m <= n
min(m,n) = n  <==  n <= m
```

induce a critical clause, which can be shown to be reductively valid, so that 8.13(5) holds true. ❑

The Horn clause completion of $(Ini(SP_1), SP_2)$ (cf. Ex. 5.2) is deterministic if we regard 0, *succ*, *nil*, & and all operations with range sort *stream* as constructors. It is also easy to see that for any specification SP and reachable SP-model A, if the built-in model SP'-extension of A is strongly functional, then the Horn clause completion of (A, SP') is strongly functional as well (cf. Defs. 3.8, 5.1 and Ex. 9.12). A strongly functional specification SP is directly implementable in functional programming languages. The program

$$
\begin{aligned}
&F(t_1) \equiv u_1 \;\Leftarrow\; t_{11} \equiv u_{11} \wedge \ldots \wedge t_{1n} \equiv u_{1n} \\
&\ldots \\
&F(t_k) \equiv u_k \;\Leftarrow\; t_{k1} \equiv u_{k1} \wedge \ldots \wedge t_{kn} \equiv u_{kn}
\end{aligned}
$$

for a function F of SP (cf. 6.11) reads in Standard ML (cf. [Pau 91]) as the following function definition:

```
fun F(t1) = let val u11 = t11
                ...
                val u1n = t1n in u1 end
|   ...
|   F(tk) = let val uk1 = tk1
                ...
                val ukn = tkn in uk end
```

Programming language constructs for strongly functional action types have been proposed by [Hag 89] and [CF 92].

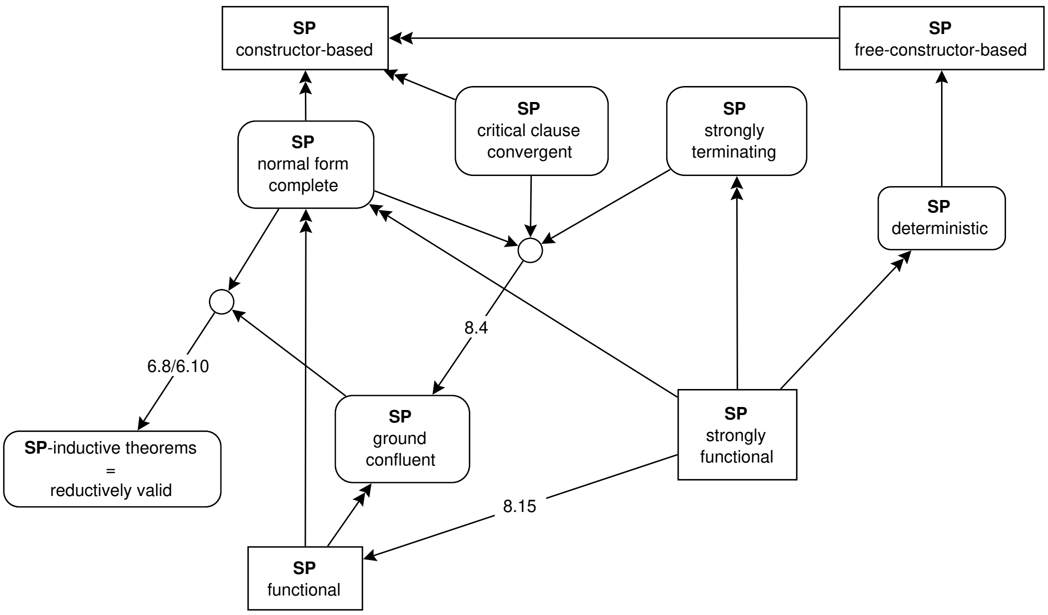


Figure 10. Variants of functionality[30]

Theorem 8.15 Strongly functional specifications are functional.

Proof. Let SP be strongly functional. Since SP is strongly terminating and normal form complete, Thm. 8.4 and Lemma 8.12 imply that $SP = (SIG, AX)$ is ground confluent iff all SP-critical clauses are reductively valid w.r.t. SP. By 8.13(1) and 8.13(5), each SP-critical clause, which might be reductively invalid, has the form $u\sigma \equiv u\tau \Leftarrow g\sigma \cup g\tau$ for some $t \equiv u \Leftarrow g \in AX$ and minimal substitutions σ, τ such that $t\sigma = t\tau$. It remains to show that $u\sigma \equiv u\tau \Leftarrow g\sigma \cup g\tau$ is reductively valid.

Let $h = \{t_1 \equiv u_1, \ldots, t_n \equiv u_n\}$ and $\rho : X \to NF_{SIG\bot}$ such that $h\sigma\rho \cup h\tau\rho$ is strongly SP-convergent. For all $0 \leq i \leq n$ we define a set V_i of variables and ground substitutions σ_i, τ_i (inductively) as follows:

- $V_0 =_{def} var(t)$ and $V_{i+1} =_{def} V_i \cup var(u_i)$,
- $\sigma_i =_{def} (\sigma\rho)_{V_i}$ and $\tau_i =_{def} (\tau\rho)_{V_i}$.

Let $0 \leq i \leq n$. Since σ and τ are minimal substitutions with $t\sigma = t\tau$, 8.13(2) implies that $\sigma_i(X)$ and $\tau_i(X)$ consist of SIG-normal forms. By 8.13(3), $u_i\sigma\rho = u_i\sigma_i$ and $u_i\tau\rho = u_i\tau_i$. Hence $u_i\sigma\rho$ and $u_i\tau\rho$ are SIG-normal forms, too.

[30]Double-headed arrows denote inclusion *by definition*, e.g., a normal form complete specification *is* constructor-based, a strongly functional specification *is* functional. Single-headed arrows stand for implications and point from a conjunction of premises to a conclusion. An edge label denotes the place where the implication is proved.

Next we show $\sigma_i = \tau_i$ by induction on i. $t\sigma = t\tau$ implies $\sigma_0 = \tau_0$. Suppose that $\sigma_i = \tau_i$. Then $t_i\sigma\rho = t_i\tau\rho$ because, by 8.13(4), $var(t_i) \subseteq V_i$. Since $g\sigma\rho$ and $g\tau\rho$ are SP-convergent, $t_i\sigma\rho \downarrow_{SP} u_i\sigma\rho$ and $t_i\tau\rho \downarrow_{SP} u_i\tau\rho$. Since $u_i\sigma\rho$ and $u_i\tau\rho$ are SIG-normal forms and SP is free-constructor-based, we obtain

$$u_i\sigma\rho \longleftarrow^*_{SP} t_i\sigma\rho \;=\; t_i\tau\rho \longrightarrow^*_{SP} u_i\tau\rho.$$

Hence $u_i\sigma\rho \equiv u_i\tau\rho$ is an SP-reduct of $t_i\tau\rho \equiv u_i\tau\rho$ and thus, since $h\sigma\rho$ and $h\tau\rho$ are *strongly* SP-convergent, $u_i\sigma\rho$ and $u_i\tau\rho$ are SP-joinable. Since these terms are SIG-normal forms and SP is free-constructor-based, they must be equal. Hence

$$\sigma_{i+1} = (\sigma\rho)_{V_{i+1}} = \sigma_i + (\sigma\rho)_{var(u_i)\setminus V_i} = \tau_i + (\tau\rho)_{var(u_i)\setminus V_i} = (\tau\rho)_{V_{i+1}} = \tau_{i+1}.$$

By 8.13(4), $var(u) \subseteq V_n$. Hence $u\sigma\rho = u\sigma_n = u\tau_n = u\tau\rho$, which concludes the proof that $u\sigma \equiv u\tau \Leftarrow h\sigma \cup h\tau$ is reductively valid. ❏

This result is close to [AL 93], Thm. 4.1. The difference lies in the termination property: [AL 93] requires *quasi-reductivity* where Thm. 8.15 assumes strong termination. We claim that the latter property is weaker than the former because quasi-reductivity replaces the assumption $h\sigma \vdash_{SP} \emptyset$ in the rewrite compatibility condition on $>_{SP}$ (cf. 8.1) by an assumption involving $>_{SP}$, which is definitely weaker than the SP-convergence of $h\sigma$, at least if SP is deterministic.

Unlike *DIV&REP* a deterministic specification SP is often *syntactically* deterministic, i.e. there are no SP-critical clauses induced by different axioms. In such a case Thm. 8.15 reduces functionality to strong termination, normal form completeness and the purely syntactical conditions 8.13(1) to (4).

Since a reduction ordering $>_{SP}$ is well-founded and rewrite compatible (cf. 8.1), strong termination implies that the SP-reduction relation $\longrightarrow_{SP}$ (cf. 6.5) is also well-founded and thus SP is *weakly terminating*, i.e. each ground term is SP-reducible into an SP-reduced term. Free-constructor-basedness (cf. 6.2) implies that $SIG\bot$-normal forms are SP-reduced. If SP is strongly terminating, then the inverse: all ground SP-reduced terms are $SIG\bot$-normal forms, is *equivalent* to normal form completeness. In other words, normal form completeness is equivalent to: each ground term is SP-reducible unless it is a $SIG\bot$-normal form. But this condition can be further reduced:

Lemma 8.16 (strong termination and normal form completeness) Let $SP = (SIG, AX)$ be strongly terminating. SP is normal form complete iff for all ground **innermost terms** t, i.e., terms $t = F(u) \in T_{SIG} \setminus NF_{SIG\bot}$ such that $u \in NF^*_{SIG\bot}$, there are $l \equiv r \Leftarrow h \in AX$ and $\sigma \in T^X_{SIG}$ such that $F(t) = l\sigma$ and $h\sigma$ is SP-convergent. ❏

Together with the above-mentioned method to show strong termination with the help of a path calculus Lemma 8.16 and Thm. 8.15 provide a both simple and powerful criterion for functionality. Let us now turn to criteria for consistency and inductive validity that employ functionality.

9 Consistency and Inductive Validity

Given a reachable SP-model A and an action extension SP' of SP, the (final) semantics of SP' is defined only if SP' is action consistent w.r.t. A (cf. 4.8). If SP and SP' are part of a swinging specification, then A is the initial model of a Horn clause completion of SP (cf. 5.21). For this case, Fact 3.6 and Assumption 6.1 yield the following characterization of consistency (cf. 3.7).

Fact 9.1 Given a Horn clause specification SP and an extension SP' of SP, $Ini(SP')$ is consistent w.r.t. $Ini(SP)$ iff for all ground atoms p over SP, $SP' \vdash_{cut} p$ implies $SP \vdash_{cut} p$. ❏

Theorem 9.2 (confluence implies consistency I) Let $SP' = (SIG', AX')$ be a functional extension of a specification $SP = (SIG, AX)$ such that the sorts of SP are the base sorts of SP' (cf. 6.2) and

(1) for all $l \equiv r \Leftarrow h \in AX' \setminus AX$, l includes a function of $SIG' \setminus SIG$,

(2) for all $c \in AX$ and $x \in fresh(c)$, $NF_{SIG',s} = NF_{SIG,s}$ where $s = sort(x)$.

Then $Ini(SP')$ is consistent w.r.t. $Ini(SP)$. Moreover,

(3) for all $t \in T_{SIG}$, $t \longrightarrow^*_{SP'} t'$ implies $t' \neq \bot$.

If, in addition, $NF_{SIG',s} = NF_{SIG,s}$ holds true for all sorts s of SP, then SP' is a totalized extension of SP (cf. 3.7).

Proof. Let $e = t \equiv u$ be a ground equation over SIG such that $SP' \vdash_{cut} e$. By Fact 9.1, we have to show $SP \vdash_{cut} e$. Since SP' is functional, Thm. 6.8 implies $t \downarrow_{SP'} u$.

We show $t \downarrow_{SP} u$ by induction on the minimal number n of SP'-rewriting steps leading from $t \equiv u$ to the empty goal. If $n = 0$, then $t = u$ and the proof is complete. Otherwise there is a ground SIG'-term v such that w.l.o.g. $t \longrightarrow_{SP'} v$ and the minimal number of rewriting steps from $v \equiv u$ to $\emptyset$ is less than n. By (1), the axiom c applied in the reduction step $t \longrightarrow_{SP'} v$ is an axiom of SP. Since $t \in T_{SIG}$, all subterms of v containing symbols of $SIG' \setminus SIG$ have been substituted for fresh variables of c. Hence these subterms are SIG'-normal and thus by (2), SIG-normal. Therefore, $t \longrightarrow_{SP'} v$ implies $t \longrightarrow_{SP} v$ and $v \in T_{SIG}$. By induction hypothesis, $v \downarrow_{SP} u$. But $t \longrightarrow_{SP} v$ and $v \downarrow_{SP} u$ yield $t \downarrow_{SP} u$.

Similarly, we show (3) by induction on the minimal number n of reduction steps leading from t to t'. If $n = 0$, then $t = t'$ and the proof is complete because $t \neq \bot$. Otherwise there is a ground SIG'-term u such that $t \longrightarrow_{SP'} u$ and the minimal number of reduction steps from u to the t' is less than n. By (1), the axiom c applied in the reduction step $t \longrightarrow_{SP'} u$ is an axiom of SP. Since $t \in T_{SIG}$, all subterms of u containing symbols of $SIG' \setminus SIG$ have been substituted for fresh variables of c. Hence these subterms are SIG'-normal and thus by (2), SIG-normal. Therefore, $t \longrightarrow_{SP'} u$ implies $t \longrightarrow_{SP} u$ and $u \in T_{SIG}$. By induction hypothesis, $t' \neq \bot$.

For totalizedness we have to show that for all sorts s of SP and $t \in T_{SIG',s}$, t is undefined iff $t \equiv_{SP'} \bot$. Suppose that

"$\Rightarrow$": Let t be undefined. Since $sort(t) \in SP$, SP' is normal form complete and the sorts of SP are the base sorts of SP', we have $t \equiv_{SP'} \bot$ or there is SIG'-normal form u such that $t \longrightarrow^*_{SP'} u$. In the second case the additional assumption would imply that u is SIG-normal. Hence $u \in T_{SIG}$ and thus t were defined so that only the first case can hold true.

"$\Leftarrow$": Let $t \equiv_{SP'} \bot$. Assume that t is defined. Then $t \equiv_{SP'} u$ for some $u \in T_{SIG}$ and thus $u \equiv_{SP'} \bot$. Since SP' is functional, Thm. 6.8 implies $u \downarrow_{SP'} \bot$ and thus $u \longrightarrow^*_{SP'} \bot$. But this contradicts (3). Hence t is undefined. ❏

Thm. 9.2 immediately implies:

Corollary 9.3 (confluence implies action consistency) Let $SP = (SIG, AX)$ be a specification, A be a reachable SP-model and SP' be an action extension of SP such that the sorts of SP are the base sorts of SP' and

(1) all functions of SP and all actions of SP' are regular (cf. 4.4),

(2) (A, SP') satisfies the coinduction principle (cf. 4.6),

(3) $BSP(A, SP') = (SIG', AX')$ is functional,

(4) for all $l \equiv r \Leftarrow h \in AX' \setminus AX$, l includes a symbol of $SIG' \setminus SIG$,

(5) for all sorts s of SP, $NF_{SIG',s} = NF_{SIG,s}$.

Then SP' is action consistent w.r.t. $Ini(SP)$. ❑

Consistency also provides a criterion for inductive validity. The proof of Horn clauses C by showing that the initial model of the extension of SP by C is conservative over $Ini(SP)$ is usually called a *proof by consistency* (cf. [KM 87]):

Lemma 9.4 (proof by consistency) Let $SP = (SIG, AX)$ be a Horn clause specification and C be a set of Horn clauses over SIG. Then $C \subseteq ITh(SP)$ iff $Ini(SIG, AX \cup C)$ is consistent w.r.t. $Ini(SP)$.

Proof. [Pad 92], Lemma 7.9. ❑

Thm. 9.2 is not appropriate for a proof by consistency because $SP \cup C$ augments the set of axioms, but not the signature of SP and thus 9.2(2) does not hold true. But we obtain a suitable criterion if the assumptions of 9.2 are replaced by strong termination and *ground reducibility* (cf. [JKo 86]):

Definition 9.5 Let $SP = (SIG, AX)$ be a constructor-based specification, C be a set of Horn clauses over SIG and $SP' = (SIG, AX \cup C)$. C is **ground reducible w.r.t.** SP if for all $l \equiv r \Leftarrow h \in C$ and $\sigma \in T^X_{SIG}$ such that $h\sigma$ is SP'-convergent, $l\sigma$ is not SP-reduced. ❑

Ground reducibility establishes the link between functionality and consistency. The following lemma generalizes [JKo 86], Theorem 1, from unconditional to conditional axioms.

Lemma 9.6 (confluence implies consistency II) Let $SP = (SIG, AX)$ be a constructor-based specification, C be a set of Horn clauses over SIG and $SP' = (SIG, AX \cup C)$ be strongly terminating.

(1) If SP' is functional and C is ground reducible w.r.t. SP, then $Ini(SP')$ is consistent w.r.t. $Ini(SP)$.

(2) If SP is functional and $Ini(SP')$ is consistent w.r.t. $Ini(SP)$, then C is ground reducible w.r.t. SP.

Proof. Let $>$ be the reduction ordering for SP'.

(1) Let SP' be functional, C be ground reducible w.r.t. SP and $e = t \equiv u$ be a ground equation over SIG such that $SP' \vdash_{cut} e$. By Fact 9.1, we have to show $SP \vdash_{cut} e$. Since SP' is functional, Thm. 6.8 implies $t \downarrow_{SP'} u$. We show $SP \vdash_{cut} e$ by Noetherian induction on e along $>$. If t and u are SP-reduced, then, by ground reducibility of C, t and u are SP'-reduced and thus equal so that the proof is complete. Otherwise there is a ground SIG-term v such that, w.l.o.g., $t \longrightarrow_{SP} v$ and thus $t > v$. Since $t \downarrow_{SP'} u$ and SP' is ground confluent, $t \longrightarrow_{SP} v$ implies $v \downarrow_{SP'} u$. Since $e' = v \equiv u$ is a ground equation and $e > e'$, the induction hypothesis yields $SP \vdash_{cut} e'$. Hence $t \longrightarrow_{SP} v$ implies $SP \vdash_{cut} e$.

(2) Let SP be functional and $Ini(SP')$ be consistent w.r.t. $Ini(SP)$. Assume that C is not ground reducible w.r.t. SP. Then there is $t \equiv u \Leftarrow h \in C$ and a ground substitution σ such that $h\sigma$ is SP'-convergent and $t\sigma$ is SP-reduced. $h\sigma \vdash_{SP'} \emptyset$ implies $SP' \vdash_{cut} h\sigma$ and thus $SP \vdash_{cut} h\sigma$ because $Ini(SP')$ is consistent w.r.t. $Ini(SP)$. By Lemma 9.4, $t \equiv u \Leftarrow h \in ITh(SP)$. Hence $SP \vdash_{cut} h\sigma$ yields $SP \vdash_{cut} t\sigma \equiv u\sigma$ and thus, by Thm. 6.8, $t\sigma \downarrow_{SP} u\sigma$ because SP is functional. Since $t\sigma$ is SP-reduced, $t\sigma \downarrow_{SP} u\sigma$ implies $u\sigma \longrightarrow^*_{SP} t\sigma$, while $h\sigma \vdash_{SP'} \emptyset$ implies $t\sigma \longrightarrow_{SP'} u\sigma$. Hence $t\sigma \longrightarrow^+_{SP'} t\sigma$ and thus $t\sigma > t\sigma$, which contradicts the well-foundedness of $>$. ❑

Thm. 8.8 tells us how to show the ground confluence of SP' by proving that (SP, C) is critical clause convergent. Ground reducibility of C w.r.t. SP allows us to restrict such a proof to critical clauses of C *on* AX (cf. 8.3):

Definition 9.7 Let $SP = (SIG, AX)$ be strongly terminating, C be a set of Horn clauses over SIG and $SP' = (SIG, AX \cup C)$. (SP, C) is **inductively convergent** if each SP'-critical clause $u \equiv v \Leftarrow h$ of some

$c \in C$ on some $c' \in AX$ induced at some t is sub-$(t \equiv u)$-reductively valid w.r.t. (SP, C) (cf. 8.5). ❑

In [Pad 92], Sect. 7, we have defined inductive convergence in a way that combines the above definition with ground reducibility. This follows the notions of *complete superposition occurrences* (cf. [Fri 89]), *inductively complete sets of term positions* (cf. [Küc 89]) and *term rewriting induction for orientable equations* (cf. [Red 90], Prop. 13), which all capture the same idea of reducing a proof of inductive validity to a minimal number of convergence proofs. However, the essence of the method becomes clearer if ground reducibility is not included into the definition of inductive convergence.

Proposition 9.8 Let $SP = (SIG, AX)$ be a constructor-based specification with reduction ordering $>$, C be a set of Horn clauses over SIG, $SP' = (SIG, AX \cup C)$ and e be a ground SIG-equation such that all SP'-convergent ground goals $g < e$ are SP-convergent. Then for all terms v, $l \equiv r \Leftarrow h \in AX$ and ground substitutions τ,

$$e \geq v(l\tau) \;\text{ and }\; h\tau \vdash_{SP'} \emptyset \;\text{ imply }\; v(l\tau) \longrightarrow_{SP} v(r\tau).$$

Proof. By assumption, $h\tau \vdash_{SP} \emptyset$ because $h\tau$ is an SP'-convergent ground goal with $h\tau < l\tau \leq v(l\tau) \leq e$. Hence $v(l\tau) \longrightarrow_{SP} v(r\tau)$. ❑

Lemma 9.9 (inductive and critical clause convergence) Let $SP = (SIG, AX)$ be functional, C be ground reducible w.r.t. SP and $SP' = (SIG, AX \cup C)$ be strongly terminating.

(SP, C) is inductively convergent iff (SP, C) is critical clause convergent.

Proof. Let $>$ be the reduction ordering for SP'.

"$\Rightarrow$": Let $c = (l \equiv r \Leftarrow g)$, $c' = (l' \equiv r' \Leftarrow h) \in AX \cup C$, $t(x)$ be a SIG-term and σ, τ be minimal substitutions such that $l\sigma = t(l'\tau)$ and $x = label(t, w)$ implies $label(l, w) \in AC \cup OP$. Let $e = (l\sigma \equiv r\sigma)$. By Thm. 8.8, it is sufficient to show that the SP'-critical clause

$$cc \quad = \quad r\sigma \equiv t(r'\tau) \;\Leftarrow\; g\sigma \cup h\tau$$

is sub-e-reductively valid w.r.t. (SP, C) if $c \in AX \cup C$ and $c' \in C$.

So let $c \in AX \cup C$, $c' \in C$ and ρ be a ground substitution such that $g\sigma\rho \cup h\tau\rho \vdash_{SP'} \emptyset$ and all SP'-convergent ground goals $g' < e\rho$ are strongly SP'-convergent and SP-convergent. Since C is ground reducible w.r.t. SP, there is a ground term t' such that $l'\tau\rho \longrightarrow_{SP} t'$. Either

(1) $r'\tau\rho \equiv t' \Leftarrow h\tau\rho$ is an instance of a critical clause of $c' \in C$ on some axiom of SP (*overlapping case*), or

(2) $t' = l'\eta$ and $z\tau\rho \longrightarrow_{SP} z\eta$ for some $z \in var(l')$ and $\eta =_{X \setminus \{z\}} \tau\rho$ (*non-overlapping case*).

Let (1) hold true. Since (SP, C) is inductively convergent, $r'\tau\rho \equiv t'$ and thus $t(r'\tau\rho) \equiv t(t')$ are SP'-convergent. Hence

$$t_1 =_{def} t(t') \longrightarrow^*_{SP'} t_2 \longleftarrow^*_{SP'} t(r'\tau\rho)$$

for some t_2.

Let $c \in AX$. Then $r\sigma\rho \longleftarrow_{SP'} l\sigma\rho = t(l'\tau\rho) \longrightarrow_{SP} t_1$ such that both reductions are induced by AX. Hence by Prop. 9.8, $g\sigma\rho \vdash_{SP'} \emptyset$ and $l\sigma\rho \leq e\rho$ imply $r\sigma\rho \longleftarrow_{SP} l\sigma\rho \longrightarrow_{SP} t_1$. Since SP is ground confluent, $r\sigma\rho \longrightarrow^*_{SP} t_3 \longleftarrow^*_{SP} t_1$ for some t_3. Hence $t_2 \longleftarrow^*_{SP'} t_1 \longrightarrow^*_{SP} t_3$. Since $e\rho \geq l\sigma\rho > t_1$, $t_2 \equiv t_1 \vdash_{SP'} \emptyset$ implies $t_2 \equiv t_3 \vdash_{SP'} \emptyset$, i.e. $t_2 \longrightarrow^*_{SP'} t_4 \longleftarrow^*_{SP'} t_3$ for some t_4. Hence

$$r\sigma\rho \longrightarrow^*_{SP'} t_3 \longrightarrow^*_{SP'} t_4 \longleftarrow^*_{SP'} t_2 \longleftarrow^*_{SP'} t(r'\tau\rho)$$

and we conclude that cc is sub-e-reductively valid w.r.t. (SP, C).

Let $c \in C$. Since C is ground reducible w.r.t. SP, there is a ground term t_0 such that $l\sigma\rho \longrightarrow_{SP} t_0$. Hence

$$t_0 \longleftarrow_{SP} l\sigma\rho = t(l'\tau\rho) \longrightarrow_{SP} t_1.$$

Since SP is ground confluent, $t_0 \longrightarrow^*_{SP} t_3 \longleftarrow^*_{SP} t_1$ for some t_3. Hence $t_2 \longleftarrow^*_{SP'} t_1 \longrightarrow^*_{SP} t_3$. Since $e\rho \geq l\sigma\rho > t_1$, $t_2 \equiv t_1 \vdash_{SP'} \emptyset$ implies $t_2 \equiv t_3 \vdash_{SP'} \emptyset$, i.e. $t_2 \longrightarrow^*_{SP'} t_4 \longleftarrow^*_{SP'} t_3$ for some t_4. Hence

$$t_0 \longrightarrow^*_{SP'} t_3 \longrightarrow^*_{SP'} t_4 \longleftarrow^*_{SP'} t_2 \longleftarrow^*_{SP'} t(r'\tau\rho).$$

Since $l\sigma\rho \longrightarrow_{SP} t_0$, we have either

(3) $r\sigma\rho \equiv t_0 \Leftarrow g\sigma\rho$ is an instance of a critical clause of $c \in C$ on some axiom of SP (*overlapping case*), or

(4) $t_0 = l\delta$ and $z\sigma\rho \longrightarrow_{SP} z\delta$ for some $z \in var(l)$ and $\delta =_{X\setminus\{z\}} \sigma\rho$ (*non-overlapping case*).

Let (3) hold true. Since (SP, C) is inductively convergent, $r\sigma\rho \equiv t_0$ is SP'-convergent. Hence $r\sigma\rho \longrightarrow^*_{SP'} t_5 \longleftarrow^*_{SP'} t_0$ for some t_5, and thus $t_5 \longleftarrow^*_{SP'} t_0 \longrightarrow^*_{SP'} t_4$. Since $e\rho \geq l\sigma\rho > t_0$, $t_5 \equiv t_0 \vdash_{SP'} \emptyset$ implies $t_5 \equiv t_4 \vdash_{SP'} \emptyset$, i.e. $t_5 \longrightarrow^*_{SP'} t_6 \longleftarrow^*_{SP'} t_4$ for some t_6. Hence

$$r\sigma\rho \longrightarrow^*_{SP'} t_5 \longrightarrow^*_{SP'} t_6 \longleftarrow^*_{SP'} t_4 \longleftarrow^*_{SP'} t(r'\tau\rho)$$

and we conclude that cc is sub-e-reductively valid w.r.t. (SP, C).

Let (4) hold true. Since $e\rho \geq l\sigma\rho > g\sigma\rho \vdash_{SP'} \emptyset$, $g\sigma\rho$ is strongly SP'-convergent and thus $g\delta$ is SP'-convergent. Hence

$$t_0 = l\delta \longrightarrow_{SP'} r\delta \longleftarrow_{SP} r\sigma\rho,$$

and the proof proceeds as in Case (3).

This finishes the proof of Case (1).

Let (2) hold true. Since $e\rho \geq l\sigma\rho > h\tau\rho \vdash_{SP'} \emptyset$, $h\tau\rho$ is strongly SP'-convergent and thus $h\eta$ is SP'-convergent. Hence

$$t_1 =_{def} t(t') = t(l'\eta) \longrightarrow_{SP'} t(r'\eta) \longleftarrow_{SP} t(r'\tau\rho).$$

With $t_2 =_{def} t(r'\eta)$, the proof proceeds as in Case (1).

"$\Leftarrow$": By definition, critical clause convergence implies inductive convergence. ❏

By combining several results of this and the previous section inductive and coinductive validity can be reduced to inductive convergence, provided that the conjectures are Horn clauses and there is a reduction ordering for both axioms and conjectures:

Theorem 9.10 (proof by convergence I)

(1) Let $SP = (SIG, AX)$ be a functional specification and C be a set of Horn clauses over SIG such that C is ground reducible w.r.t. SP, (SP, C) is inductively convergent and $SP' = (SIG, AX \cup C)$ is strongly terminating. Then $C \subseteq ITh(SP)$.

(2) Let SP and SP' be specifications, A be a reachable SP-model, $HSP = HSP(A, SP') = (SIG, AX)$ (cf. 5.1) be functional and C be a set of Horn clauses over SP' such that SP' is action consistent w.r.t. A (cf. 4.8), C_{beh} is ground reducible w.r.t. HSP, (HSP, C_{beh}) is inductively convergent and $HSP' = (SIG, AX \cup C_{beh})$ is strongly terminating. Then $C \subseteq CITh(A, SP')$ (cf. 5.7).

Proof. (1) follows from Thm. 8.8 and Lemmata 9.4, 9.6(1) and 9.9.

(2) is the special case of (1) where, in terms of (1), SP is given by the Horn clause completion HSP. Hence (1) implies $C_{beh} \subseteq ITh(HSP)$ and thus $C \subseteq CITh(A, SP')$. ❏

Thm. 9.10 has an inverse. If SP is functional and strongly terminating, then C can always be proved by inductive convergence:

Theorem 9.11 (completeness of proof by convergence) Given a functional and strongly terminating specification $SP = (SIG, AX)$ and a set C of Horn clauses over SIG, (SP, C) is inductively convergent if $C \subseteq ITh(SP)$.

Proof. Let C be set of inductive theorems of SP. By Lemma 9.4, the initial model of $SP' = (SIG, AX \cup C)$ is consistent w.r.t. $Ini(SP)$. Since SP is functional, Thm. 6.8 implies that SP' is reductively consistent w.r.t. SP. Hence by Prop. 8.7, SP' is ground confluent. It remains to show that (SP, C) is critical clause convergent. Let $cc = (u \equiv v) \Leftarrow h$ be an SP'-critical clause induced at a term t and σ be a ground substitution such that $h\sigma$ is SP'-convergent. Then $u\sigma \longleftarrow_{SP'} t\sigma \longrightarrow_{SP'} v\sigma$. Hence $u\sigma$ and $v\sigma$ are SP'-joinable because SP' is ground confluent. ❏

Sample proofs by convergence are carried out below and in the next chapter. The theorems proved there stem from different proof obligations: the correctness of a stack implementation by maps (Ex. 9.12); coinduction principle for a set specification (10.2); the coinduction principle for a specification of basic process algebra (10.4.1). The second and third example use a variant of Thm. 9.10, which does not assume that the reduction ordering for SP is rewrite compatible upon C (cf. Thm. 9.19). If neither Lemma 8.9 nor Thm. 8.11 are sufficient to show inductive convergence, one may try to prove subreductive validity by constructing *subreductive expansions* in the sense of [Pad 95].

Example 9.12 (partially correct implementation of stacks by maps) The following example is a popular benchmark for formal refinement approaches (cf. e.g. [GHM 76], Sect. 4.4; [Pad 92], Ex. 7.20; [MG 94], Sect. 4.1). A pair consisting of a finite array, i.e. a map with finite domain (cf. 9.1), and a pointer implements a stack (or pushdown list) and its operations as follows. *bottom* denotes a bottom constant. Recall that bottom constants for base sorts are added implicitly when a specification is extended by partial functions (cf. 2.3).

```
STACK_AS_MAP
     base     ENTRY (cf. 10.1)
     sorts    map  stack
     acts     get:map*nat->entry
              top:stack->entry
              pop:stack->stack
     consts   new:map
              put:nat*entry*map->map
              mkStack:map*nat->stack
     defs     empty:stack
              push:entry*stack->stack
     vars     i:nat  x:entry  f:map
     axioms   (1) get(new,i) = bottom
              (2) get(put(i,x,f),i) = x
              (3) get(put(i,x,f),j) = get(f,j) <== eq(i,j) = false
              (4) top(mkStack(f,i)) = get(f,i)
              (5) pop(mkStack(f,i)) = mkStack(f,pred(i))
              (6) empty = mkStack(new,0)
              (7) push(x,mkStack(f,i)) = mkStack(put(succ(i),x,f),succ(i))
```

mkStack is the **abstraction function** of the implementation. The last four axioms provide the implementation of the actions *top* and *pop* and the operations *empty* and *push*, respectively. Let $SP' = STACK\_AS\_MAP$ and $A = Ini(BASE)$.

First we apply Cor. 9.3 in order to show that SP' is action consistent w.r.t. A (cf. 4.8). We leave it to the reader to show that $BSP(A,SP')$ is strongly functional and thus, by Thm. 8.15, functional. Moreover, (A,SP') satisfies the coinduction principle w.r.t. the *map*-head normal forms *new* and $put(i,x,f)$ and the *stack*-head normal form $mkStack(f,i)$ (cf. 4.6). 9.3(3) and (4) are trivially satisfied. Hence SP' is action consistent w.r.t. A and thus the semantics of SP' is given by the final SP'-model over A, $Fin(A,SP')$, whose theory agrees with the coinductive theory of (A,SP') (cf. 4.3 and 5.8). $Fin(A,SP')$ is a **partially correct implementation** of stacks if the **abstract stack axioms**:

```
 (8) top(empty) = bottom
 (9) pop(empty) = empty
(10) top(push(x,st)) = x
(11) pop(push(x,st)) = st
```

are coinductive theorems of (A,SP'). (9) and (11) are the behaviour variants (cf. 5.7) of

```
(12) pop(empty) ~ empty
(13) pop(push(x,st)) ~ st
```

By definition, (8) to (11) are coinductive theorems of (A,SP') if (8),(10),(12) and (13) are inductive theorems of the Horn clause completion HSP of (A,SP') (cf. 5.1):

```
HSP
     base      ENTRY  STACK_AS_MAP
     preds     ~i:map*nat*map
               ~:map*map
               ~i:stack*nat*stack
               ~:stack*stack
     infixes   ~i  ~
     vars      i,x:nat  f,f':map  st,st':stack
     axioms    f ~ f'  <==  /\{x}(get(f,x) = get(f',x))
               st ~0 st'
               st ~succ(i) st'  <==  top(st) = top(st') /\ pop(st) ~i pop(st')
               st ~ st'  <==  /\{i}(st ~i st')
```

Let $C = \{(8),(10),(12),(13)\}$. By Thm. 9.10(2), $C \subseteq ITh(HSP)$ if HSP is strongly functional, $HSP \cup C$ is strongly terminating, C is ground reducible w.r.t. HSP and (HSP,C) is inductively convergent. Let us check these conditions.

HSP is deterministic and normal form complete with respect to the constructors *new*, *put* and *mkStack* and the defined functions *get*, *top*, *pop*, *empty* and *push*. Critical clauses induced by two different axioms of HSP do not exist. A reduction ordering for $HSP \cup C$ can be constructed along the lines of [Pad 92], Sect. 6.6, or [Pad 95], Sect. 7. Hence HSP is strongly functional and $HSP \cup C$ is strongly terminating. The ground reducibility of C is verified by a short glance at the axioms of HSP. There are four critical clauses of C on axioms of HSP:

```
(cc1) top(mkStack(new,0)) = bottom
(cc2) pop(mkStack(new,0)) ~ empty
(cc3) top(mkStack(put(succ(i),x,f),succ(i))) = x
(cc4) pop(mkStack(put(succ(i),x,f),succ(i))) ~ mkStack(f,i)
```

In order to prove that (cc1) to (cc4) are subreductively valid w.r.t. (HSP,C) we apply Lemma 8.9 and show that the behaviour variants of these equations are coinductive theorems of (A,SP'). The proof consists of term replacements upon axioms of SP' (cf. 7.2) and an application of the lemma

(13) $mkStack(put(i,x,f),j) \equiv mkStack(f,j) \;\Leftarrow\; i > j.$

Proof of cc1:

$top(mkStack(new,0)) \equiv \bot \;\vdash\; get(new,0) \equiv \bot \;\vdash\; \bot \equiv \bot \;\vdash\; TRUE$

Proof of cc2:

$pop(mkStack(new,0)) \equiv empty \;\vdash\; mkStack(new,pred(0)) \equiv empty$

$\vdash\; mkStack(new,0) \equiv empty \;\vdash\; mkStack(new,0) \equiv mkStack(new,0) \;\vdash\; TRUE$

Proof of cc3:

$top(mkStack(put(succ(i),x,f),succ(i))) \equiv x \;\vdash\; get(put(succ(i),x,f),succ(i)) \equiv x \;\vdash\; x \equiv x \;\vdash\; TRUE$

Proof of cc4:

$pop(mkStack(put(succ(i),x,f),succ(i))) \equiv mkStack(f,i)$

$\vdash\; mkStack(put(succ(i),x,f),pred(succ(i))) \equiv mkStack(f,i)$

$\vdash\; mkStack(put(succ(i),x,f),i) \equiv mkStack(f,i)$

deductive expansion upon (12)

$\vdash\; succ(i) > i \;\vdash\; TRUE$

Proof of (12):

$mkStack(put(i,x,f),j) \equiv mkStack(f,j) \;\Leftarrow\; i > j$

variable introduction (cf. Section 7)

$\vdash\; st \equiv st' \;\Leftarrow\; st \equiv mkStack(put(i,x,f),j) \wedge st' \equiv mkStack(f,j) \wedge i > j$

fixpoint coinduction on $\equiv_{stack}$ (cf. Section 7)

$\vdash\; top(st) \equiv top(st') \wedge Q(pop(st),pop(st')) \;\Leftarrow\; Q(st,st'),$

$Q(mkStack(put(i,x,f),j),mkStack(f,j)) \;\Leftarrow\; i > j$ $\qquad prog(Q)$

only-if-completion of Q upon $prog(Q)$

$\vdash\; \exists\{i',x',f',j'\}(top(st) \equiv top(st')$
$\wedge\; pop(st) \equiv mkStack(put(i',x',f'),j') \wedge pop(st') \equiv mkStack(f',j') \wedge i' > j')$
$\Leftarrow\; st \equiv mkStack(put(i,x,f),j) \wedge st' \equiv mkStack(f,j) \wedge i > j$

variable introduction

$\vdash\; \exists\{i',x',f',j'\}(top(mkStack(put(i,x,f),j)) \equiv top(mkStack(f,j))$
$\wedge\; pop(mkStack(put(i,x,f),j)) \equiv mkStack(put(i',x',f'),j')$
$\wedge\; pop(mkStack(f,j)) \equiv mkStack(f',j') \wedge i' > j') \;\Leftarrow\; i > j$

term replacement upon Axioms 4 and 5

$\vdash\; \exists\{i',x',f',j'\}(get(put(i,x,f),j) \equiv get(f,j) \wedge mkStack(put(i,x,f),pred(j)) \equiv mkStack(put(i',x',f'),j')$
$\wedge\; mkStack(f,pred(j)) \equiv mkStack(f',j') \wedge i' > j') \;\Leftarrow\; i > j$

unification

$\vdash\; get(put(i,x,f),j) \equiv get(f,j) \wedge i > pred(j) \;\Leftarrow\; i > j$

deductive expansion upon $i > pred(j) \Leftarrow i > j$

$\vdash\; get(put(i,x,f),j) \equiv get(f,j) \wedge i > j \;\Leftarrow\; i > j$

only-if-completion of get and constructor clash

$\vdash\; (x \equiv get(f,j) \wedge i \equiv j \wedge i > j) \vee (get(f,j) \equiv get(f,j) \wedge eq(i,j) \equiv false \wedge i > j) \;\Leftarrow\; i > j$

$\neg$-rule upon $\neg(i \equiv j \wedge i > j)$

$\vdash\; get(f,j) \equiv get(f,j) \wedge eq(i,j) \equiv false \wedge i > j \;\Leftarrow\; i > j$

unification

$\vdash\; eq(i,j) \equiv false \wedge i > j \;\Leftarrow\; i > j$

deductive expansion upon $eq(i,j) \equiv false \Leftarrow i > j$

$\vdash\ i > j \ \Leftarrow\ i > j$
subsumption
$\vdash\ TRUE$ ❑

Showing inductive validity with the help of Thm. 9.10 (*proof by convergence I*) and Lemma 8.9 (*Criterion I for subreductive validity*) results in proofs that are quite similar to classical proofs by explicit induction. Indeed, the above derivations of cc1 to cc4 and Lemma (13) would also occur in a direct proof of C, which would split into four cases corresponding to the above four critical clauses of C on axioms of HSP. The situation would be different if the direct proof of C would use C as an induction hypothesis. Then the corresponding proof by convergence can no longer apply Lemma 8.9. Instead, induction steps would turn into reductions upon C, i.e., the proof by convergence would refer to Thm. 8.11 (*Criterion II for subreductive validity*). Section 10.2 contains an example of such a proof. Nevertheless, it is always worthwhile to try a direct proof before starting a proof by convergence because the latter must be based on a reduction ordering, not only for the original specification SP, but also for the set C of conjectures. Moreover, all lemmas applied in the proof must be e-bounded (cf. Thm. 8.11). These additional proof obligations often do not pay off if compared to a classical proof involving explicit induction steps (cf. [Pad 92], Sect. 7.4).

The correctness proof of Ex. 9.12 is simpler than previous initial-semantics versions (cf. [GHM 76], [Pad 92]). This is due to the smaller set of axioms that are necessary to present $SP' = STACK\_AS\_MAP$ with respect to final semantics. We have shown that Lemma (13) is a coinductive theorem of SP'. It becomes an *inductive* theorem only if the abstract stack axiom (11) were added to SP' (cf. [Pad 95], Example 7.4). [GHM 76] calls Lemma (13) the **equality interpretation** of the implementation. Initial semantics enforces explicit axiomatizations of equality interpretations, while in final semantics, they are just behavioural equivalences.

SP' satisfies the abstract stack axioms, which made us claim that SP' is a partially correct implementation. Why only partial? Because there is the converse question: does SP' preserve the (abstract) stack theory? Or does SP' entail "new" stack properties? In general, total correctness is more delicate than partial correctness. Even the final-semantics treatment of *STACK_AS_MAP* in [MG 94], Sect. 4.1, lacks a proof of total correctness. What needs to be done is a proof that the final model of *STACK_AS_MAP* is consistent w.r.t. the final model of the abstract stack axioms. But since *STACK_AS_MAP* is an action type, we can only expect *behavioural consistency*:

Definition 9.13 (final model implementation) Let SP be a specification, A be an SP-model and ABS and SP' be action extensions of SP such that the signature of ABS, $sig(ABS)$, is contained in the signature of SP'[31] and ABS and SP' are action consistent w.r.t. A.

$Fin(A, SP')$ is a **totally correct implementation of** $Fin(A, ABS)$ if the axioms of ABS are $sig(ABS)$-theorems of $Fin(A, SP')$ (*partial correctness*) and $Fin(A, ABS \cup SP')$ is consistent w.r.t. $Fin(A, ABS)$. ❑

Lemma 9.4 provides a criterion for the consistency property:

Lemma 9.14 Given the assumptions of Def. 9.13, $Fin(A, ABS \cup SP')$ is consistent w.r.t. $Fin(A, ABS)$ iff there is a set C of Horn clause axioms of SP' such that $C \subseteq CITh(A, ABS \cup SP' \setminus C)$ and $Fin(A, ABS \cup SP' \setminus C)$ is consistent w.r.t. $Fin(A, ABS)$. ❑

Example 9.15 (*STACK_AS_MAP* is totally correct) First the abstract stack axioms are completed to a specification ABS (cf. 9.12):

```
ABS
      base     ENTRY
      sorts    stack
      acts     top:stack->entry
```

[31] Some operations of ABS may be actions of SP'.

```
               pop:stack->stack
    consts     empty:stack
               push:entry*stack->stack
    vars       x:entry  st:stack
    axioms     (8) top(empty) = bottom
               (9) pop(empty) = empty
              (10) top(push(x,st)) = x
              (11) pop(push(x,st)) = st
```

Let $A = Ini(ENTRY)$ and $SP' = STACK\_AS\_MAP$. In Ex. 9.12 we have shown that the axioms of ABS are coinductive theorems of (A, SP'). For the proof that $(A, SP' \cup ABS)$ is behaviourally consistent w.r.t. (A, ABS) we apply Lemma 9.14 and extract from $SP' \cup ABS$ a specification SP'' that has the same signature as SP', but only Axioms (1) to (5) of SP':[32]

```
SP"
    base       ABS
    sorts      map
    acts       get:map*nat->entry
    consts     new:map
               put:nat*entry*map->map
               mkStack:map*nat->stack
    vars       i:nat  x:entry  f:map
    axioms     (1) get(new,i) = bottom
               (2) get(put(i,x,f),i) = x
               (3) get(put(i,x,f),j) = get(f,j) <== eq(i,j) = false
               (4) top(mkStack(f,i)) = get(f,i)
               (5) pop(mkStack(f,i)) = mkStack(f,pred(i))
```

Note that the defined functions *empty* and *push* of SP' became constructors of ABS and thus of SP''. By Lemma 9.14 and Thm. 5.8, it remains to show:

- Axioms (6) and (7) of SP' are coinductive theorems of (A, SP''),
- $Ini(HSP(A, SP''))$ is consistent w.r.t. $Ini(HSP(A, ABS))$,

Proof of Axiom (6):

$empty \equiv mkStack(new, 0)$

variable introduction

$\vdash \; st \equiv st' \;\Leftarrow\; st \equiv empty \wedge st' \equiv mkStack(new, 0)$

fixpoint coinduction on $\equiv_{stack}$

$\vdash \; top(st) \equiv top(st') \wedge pop(st) \equiv empty \wedge pop(st') \equiv mkStack(new, 0) \;\Leftarrow\; st \equiv empty \wedge st' \equiv mkStack(new, 0)$

variable introduction

$\vdash \; top(empty) \equiv top(mkStack(new, 0)) \wedge pop(empty) \equiv empty \wedge pop(mkStack(new, 0)) \equiv mkStack(new, 0)$

term replacement upon Axioms 4, 5, 8 and 9

$\vdash \; 0 \equiv get(new, 0) \wedge empty \equiv empty \wedge mkStack(new, pred(0)) \equiv mkStack(new, 0)$

term replacement upon Axiom 1, only-if-completion of *pred* and constructor clash

$\vdash \; 0 \equiv 0 \wedge empty \equiv empty \wedge mkStack(new, 0) \equiv mkStack(new, 0)$

unification

$\vdash \; TRUE$

Proof of Axiom (7):

$\vdash \; push(x, mkStack(f, i)) \equiv mkStack(put(succ(i), x, f), succ(i))$

[32] In terms of Lemma 9.14, C consists of Axioms (6) and (7).

if-completion of $\equiv_{stack}$ (cf. 7.2)

$\vdash$ $top(push(x, mkStack(f, i))) \equiv top(mkStack(put(succ(i), x, f), succ(i)))$
$\quad\wedge pop(push(x, mkStack(f, i))) \equiv pop(mkStack(put(succ(i), x, f), succ(i)))$

term replacement upon Axioms 4, 5, 10 and 11

$\vdash$ $x \equiv get(put(succ(i), x, f), succ(i)) \wedge mkStack(f, i) \equiv mkStack(put(succ(i), x, f), pred(succ(i)))$

term replacement upon Axiom 2, only-if-completion of *pred* and constructor clash

$\vdash$ $x \equiv x \wedge mkStack(f, i) \equiv mkStack(put(succ(i), x, f), i)$

unification

$\vdash$ $mkStack(f, i) \equiv mkStack(put(succ(i), x, f), i)$

deductive expansion upon Lemma (13)

$\vdash$ $succ(i) > i$

deductive expansion upon $succ(i) > i$

$\vdash$ $TRUE$

Since Lemma (13) was derived in Ex. 9.12 without applying (6) or (7) (cf.), we may use it here in the proof of (7).

For showing that $Ini(HSP(A, SP''))$ is consistent w.r.t. $Ini(HSP(A, ABS))$ we apply Thm. 9.2. It is easy to see that $Ini(HSP(A, SP''))$ is consistent w.r.t. $Ini(HSP(A, ABS))$ iff the initial model of $BSP' = BSP(A, SP'')$ is consistent w.r.t. the initial model of $BSP = BSP(A, ABS)$. BSP' is deterministic and normal form complete with respect to the constructors new, put, $mkStack$, $empty$ and $push$ and the defined functions get, top and pop. Critical clauses induced by two different axioms of BSP' do not exist. A reduction ordering for BSP' can be constructed along the lines of [Pad 92], Sect. 6.6, or [Pad 95], Sect. 7. Hence BSP is strongly functional and thus functional. 9.2(1) is satisfied: the left-hand side of each of Axioms (1) to (5) contains a function of $BSP' \setminus BSP$. 9.2(2) holds true trivially because (1) to (5) have no fresh variables. There is one condition left, which should not be overlooked. The sorts of BSP are supposed to be the base sorts of BSP' and thus all constructors of $BSP' \setminus BSP$ mapping into a base sort may have only those *non*-base-sorted arguments whose constructors do not refer to base sorts (cf. 6.2(2)). The only constructor of $BSP' \setminus BSP$ mapping into a base sort is $mkStack$, which is the actual *implementing constructor* and hence satisfies this condition naturally: since implementing constructors map implementing (= non-base) sorts to implemented (= base) sorts, the constructors for their arguments never refer to their range sorts. ❑

The obligation to show that SP' is a correct implementation of ABS forced us to specify stacks as an action type, although stacks are finite free-constructor-based objects and thus suggest a constructor type presentation in the first place. This were almost the same as a specification of lists (cf. 2.5 or 10.1): *empty* becomes *nil*, *push* becomes &, etc. We could also build a specification of "dynamic" stacks on top of the list specification:

```
STACK
     base      LIST (cf. 10.1)
     sorts     stack
     acts      top:stack->entry
               pop:stack->stack
     consts    empty:stack
               push:entry*stack->stack
     defs      cont:stack->list
               no_el:stack->nat
     vars      x:entry  st:stack
     axioms    top(empty) = bottom
               pop(empty) = empty
               top(push(x,st)) = x
               pop(push(x,st)) = st
               cont(empty) = nil
               cont(push(x,st)) = x&cont(st)
```

```
no_el(empty) = 0
no_el(push(x,st)) = succ(no_el(st))
```

This simple example backs up our claim that we do not need separate formalisms for treating static, constructor-based types on the one hand and dynamic, state-based, *object-oriented* types on the other hand. If at all they only occupy different components of a single hierarchical swinging specification.

Critical clauses of conjectures on axioms constructed in a proof by convergence such as in 9.12 correspond to cases a proof by explicit induction usually divides into. Taking this observation into account, [Red 90], [KR 90] and [BKR 92] replaced the confluence-proof-oriented notion of a critical clause by that of a *cover set* or *test set* of terms, which covers all ground solutions of the premise of the conjecture to be proved. The proof itself is performed by induction on the test set along a reduction ordering for SP.

The main advantage of induction on test sets is that – in contrast to 9.10(1) – the reduction ordering for SP need not be rewrite compatible upon C, the conjectures to be proved. That is already suggested by the *completeness* of proof by convergence (cf. 9.10(2)) and by the fact that the assumptions of 9.10(1) imply more than inductive validity of C, namely ground confluence and reductive consistency of SP' w.r.t. SP (cf. 8.8 and 9.9). So we introduce a condition, which is weaker than reductive consistency of SP', but strong enough to conclude the validity of C from inductive convergence of (SP, C), and which captures the idea of test set induction insofar as the substitutions $\sigma_1, \ldots, \sigma_n$ constitute a test set in the above sense.

Let $SP = (SIG, AX)$ be a Horn clause specification, C be a set of Horn clauses over SIG and $SP' = (SIG, AX \cup C)$.

Definition 9.16 Let $c = (l \equiv r \Leftarrow h)$ be a Horn clause over SIG and

$$\begin{array}{ll} (cc_1) & r\sigma_1 \equiv t_1 \Leftarrow h\sigma_1 \cup h_1 \\ \ldots & \ldots \\ (cc_n) & r\sigma_n \equiv t_n \Leftarrow h\sigma_n \cup h_n \end{array}$$

be the SP'-critical clauses of c on AX. For all $1 \le i \le n$, let $X_i =_{def} var(cc_i) \setminus var(c)$. Then

$$crit(c) \quad =_{def} \quad \exists X_1(h_1 \cup EQ(\sigma_1)) \ \vee \ \ldots \ \vee \ \exists X_n(h_n \cup EQ(\sigma_n)) \ \Leftarrow \ h$$

is the **critical test clause for** c. $crit(C) =_{def} \{crit(c) \mid c \in C\}$. ❑

Proposition 9.17 Let SP be functional and free-constructor-based (cf. 6.2) and C be a set of Horn clauses over SIG. If C is ground reducible w.r.t. SP (cf. 9.5), then $crit(C) \subseteq ITh(SP)$.

Proof. Let $c = (l \equiv r \Leftarrow h) \in C$. We show that $crit(c) = gs \Leftarrow h$ is an inductive theorem of SP. So let σ be a ground substitution such that $SP \vdash_{cut} h\sigma$. Since SP is functional, σ has an SP-reduct $\tau : X \rightarrow NF_{SIG}$ such that $h\tau$ is SP- and thus SP'-convergent. Since C is ground reducible w.r.t. SP, $l\tau$ is not SP-reduced. For all $x \in var(l)$, $x\tau$ is SP-reduced because SP is free-constructor-based. Hence there are an SP'-critical clause $cc = (r\delta \equiv t \Leftarrow h\delta \cup g)$ of c on AX and a substitution ϕ such that $\tau = \delta\phi$, $l\tau \longrightarrow_{SP} t\phi$, $g\phi \vdash_{SP} \emptyset$ and $dom(\delta) \subseteq var(c)$. Hence $\exists Y(g \cup EQ(\delta)) \in gs$ where $Y = var(cc) \setminus var(c)$. Let $\psi = \sigma_{var(c)} + \phi_{X \setminus var(c)}$. $SP \vdash_{cut} g\phi$ implies $SP \vdash_{cut} g\psi$. For all $x \in dom(\delta)$, $x\psi = x\sigma$ and $x\delta\psi = x\delta\phi = x\tau$. Hence $SP \vdash_{cut} x\sigma \equiv x\tau$ implies $SP \vdash_{cut} EQ(\delta)\psi$, and we obtain $SP \vdash_{cut} (g \cup EQ(\delta))\psi$. ❑

In many cases, the validity of $crit(c)$ can be shown directly so that we need not prove ground reducibility.

For all goals g, let $Sol(g)$ be the set of all ground substitutions σ such that $SP \vdash_{cut} g\sigma$.

Lemma 9.18 Suppose that SP is functional and strongly terminating and for all $e \Leftarrow g \in C$, C is e-bounded w.r.t. $>$ (cf. 8.10). For each $e \Leftarrow g \in C$ and $\sigma \in Sol(g)$ such that

(1) for all $e' \Leftarrow h \in C$ and $\tau \in Sol(h)$, $e'\tau < e\sigma$ implies $SP \vdash_{cut} e'\tau$,

each SP'-convergent ground equation $(t \equiv u) < e\sigma$ is SP-convergent and strongly SP'-convergent.

Proof. Let $>$ be the reduction ordering for SP. We show the conjecture by induction on $e\sigma$ along $>$ simultaneously over all $e \Leftarrow g \in C$ and $\sigma \in Sol(g)$ satisfying (1). So let $e \Leftarrow g \in C$, $\sigma \in Sol(g)$ and $t \equiv u$ be an SP'-convergent ground equation with $(t \equiv u) < e\sigma$ such that (1) holds true. Then there is a ground term v with $t \longrightarrow^*_{SP'} v \longleftarrow^*_{SP'} u$. Suppose that

(2) $t \downarrow_{SP} v$ and $v \downarrow_{SP} u$.

Then $t \downarrow_{SP} u$ because SP is ground confluent. Hence $t \equiv u$ is SP-convergent. We must also show that $t \equiv u$ is strongly SP'-convergent. W.l.o.g. let $t \longrightarrow^*_{SP'} t'$. Suppose that

(3) $t \downarrow_{SP} t'$.

Then $t' \equiv u$ is SP-convergent because SP is ground confluent. Hence $t \equiv u$ is strongly SP'-convergent. It remains to show (2) and (3). Both conditions follow from (4):

(4) $e\sigma > t \longrightarrow^*_{SP'} t'$ implies $t \downarrow_{SP} t' < e\sigma$.

So let $e\sigma > t \longrightarrow^*_{SP'} t'$. (3) is shown by induction on the number of rewrite steps leading from t to t'. If $t = t'$, the proof is complete. Otherwise there are terms u, v such that $t \longrightarrow^*_{SP'} u \longrightarrow_{SP'} v \longrightarrow^*_{SP} t'$ and the step from u to v is performed by applying some $(l \equiv r \Leftarrow h) \in C$. By the inner induction hypothesis, $t \downarrow_{SP} u < e\sigma$. We have $u = u'(l\tau)$, $u = u'(r\tau)$ and $h\tau \vdash_{SP'} \emptyset$ for some u', τ. Since $(l \equiv r)\tau \geq l\tau > h\tau$, the outer induction hypothesis implies that $h\tau$ is SP-convergent. Hence $e\sigma > v$ because c is e-bounded. By (1), $e\sigma > (u \equiv v) \geq (l \equiv r)\tau$ and $\tau \in Sol(h)$ imply $SP \vdash_{cut} (l \equiv r)\tau$. and thus $SP \vdash_{cut} u \equiv v$. Hence $u \downarrow_{SP} v$ and thus $t \downarrow_{SP} t'$ because SP is ground confluent. Finally, $e\sigma > v \longrightarrow^*_{SP} t'$ implies $e\sigma > t'$. ❑

Theorem 9.19 (proof by convergence II) Suppose that SP is functional and strongly terminating, (SP, C) is inductively convergent, $crit(C) \subseteq ITh(SP)$ and C is C-bounded w.r.t. $>$.[33] Then $C \subseteq ITh(SP)$.

Proof. Let $>$ be the reduction ordering for SP. We show $SP \vdash_{cut} e\sigma$ by induction on $e\sigma$ along $>$ over all $e \Leftarrow g \in C$ and $\sigma \in Sol(g)$.

So let $c = (e \Leftarrow g) = (l \equiv r \Leftarrow g) \in C$ and $\sigma \in Sol(g)$. Since $crit(c) \in ITh(SP)$, there are an SP'-critical clause $cc = (r\eta \equiv t \Leftarrow g\eta \cup h)$ of c on AX and a ground substitution $\tau =_{var(c)} \sigma$ such that $SP \vdash_{cut} h\tau \cup EQ(\eta)\tau$. Hence $SP \vdash_{cut} g\tau$ and thus $SP \vdash_{cut} g\eta\tau$. Since (SP, C) is inductively convergent, cc is sub-$e\eta$-reductively valid w.r.t. (SP, C). Since SP is ground confluent, $SP \vdash_{cut} g\eta\tau \cup h\tau$ implies $g\eta\tau \cup h\tau \vdash_{SP} \emptyset$. By Lemma 9.18, all SP'-convergent ground goals $g' < e\sigma = e\tau = e\eta\tau$ are SP-convergent and strongly SP'-convergent. Hence $(r\tau \equiv t\tau) = (r\eta\tau \equiv t\tau)$ is SP'-convergent. $h\tau \vdash_{SP} \emptyset$ implies $l\sigma = l\tau = l\eta\tau \longrightarrow_{SP} t\tau$ by the construction of cc (cf. 8.3). Hence $e\sigma > (r\sigma \equiv t\tau)$, and we conclude that $(r\sigma \equiv t\tau) = (r\tau \equiv t\tau)$ is SP-convergent. Finally, $l\sigma \longrightarrow_{SP} t\tau$ implies $r\sigma \equiv l\sigma \vdash_{SP} \emptyset$ and thus $SP \vdash_{cut} e\sigma$. ❑

Moreover, Thms. 9.11 and 9.19 yield a *characterization* of inductive validity by inductive convergence:

Theorem 9.20 (inductive convergence and inductive validity) Given a functional and strongly terminating specification $SP = (SIG, AX)$ and a set C of Horn clauses over SIG such that $crit(C) \subseteq ITh(SP)$ and C is C-bounded w.r.t. $>$. (SP, C) is inductively convergent iff $C \subseteq ITh(SP)$. ❑

[33] Actually, only clauses of C used in the proof that (SP, C) is inductively convergent must be C-bounded.

[34] The arrows stand for implications and point from a conjunction of premises to a conclusion. An edge label denotes the place where the implication is proved. Boldface paths show the way how the soundness of proof by convergence is obtained from the assumptions of Thms. 9.10 and 9.19, respectively.

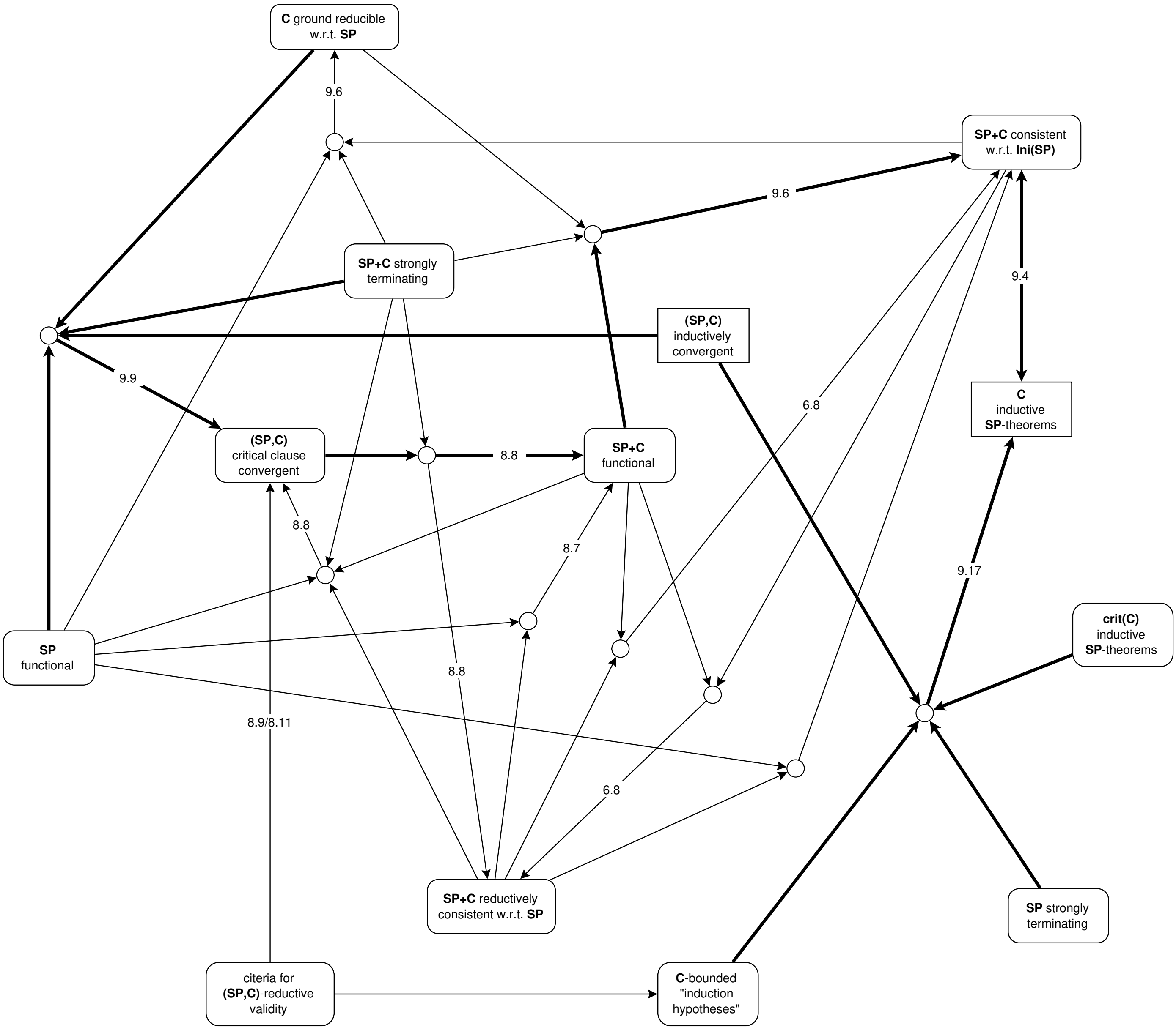


Figure 11. Inductive convergence and inductive validity[34]

10 Applications

Swing your own thing.
Parrish Smith, *PMD Music*

This chapter provides plenty of material for investigating and further developing well-known constructor and action types. We do not claim neither that this is a complete overview of all application areas nor that everything has already been thoroughly investigated. The chapter is supposed to give some starting points for future research, which should focus on exploring various application areas in detail, developing tactics and strategies for guiding proofs about swinging types and comparing the results with established design and verification practice employing, e.g., built-in theories for permutative types (cf. 1.1), cpos for partial functions and streams (cf. 1.3), process algebra and modal, dynamic or temporal logic for processes (cf. 1.5) and pure functional programming for compilers, interpreters and other *pattern-driven* problems. Readers who are familiar with one or the other of these approaches are challenged to contrast it with the swinging approach when reasoning about particular applications.

10.1 Parametric types

Polymorphic types with universally-quantified type variables usually have non-inhabited specifications. For incorporating polymorphism we distinguish between sorts and sort variables as in the following example:

```
ENTRY
     sorts     bool  nat
     sort vars entry  index
     consts    true,false:bool
               0:nat
               succ:nat->nat
     defs      not:bool->bool
               or,and:bool*bool->bool
               eq:bool*bool->bool
               eq:nat*nat->bool
               eq:entry*entry->bool
               eq:index*index->bool
               even:nat->bool
               pred:nat->nat
               +,-,*:nat*nat->nat
               ite:bool*entry*entry->entry
     preds     <,<=,>,>=:nat*nat
     infixes   or  and  +  <  <=  >  >=
     vars      b:bool  m,n:nat  x,x':entry
     axioms    not(true) = false
               not(false) = true
               true or b = true
               false or b = b
               true and b' = b
               false and b = false
               eq(true,b) = b
               eq(false,b) = false
               eq(0,0) = true
               eq(0,succ(n)) = false
               eq(succ(m),0) = false
               eq(succ(m),succ(n)) = eq(m,n)
               even(0) = true
               even(succ(n)) = not(even(n))
               pred(0) = 0
               pred(succ(n)) = n
```

```
0+n = n
succ(m)+n = succ(m+n)
0-n = 0
m-0 = m
succ(m)-succ(n) = m-n
0*n = 0
succ(m)*n = n+succ(m*n)
ite(true,x,x') = x
ite(false,x,x') = x'
0 < succ(n)
succ(m) < succ(n)  <==  m < n
0 <= n
succ(m) <= succ(n)  <==  m <= n
m > n  <==  m < n
m >= n  <==  m <= n
```

Given a *nat*-term t, we usually write $t+1$ instead of $succ(t)$.

A specification PSP is called **parametric** or **parameterized** if it contains sort variables or *input sorts* as *ENTRY* does. Given that In is the set of input sorts of PSP, we extract from PSP the greatest sets of operations OP, predicates PR and axioms AX, respectively, such that $SP = ((In, \emptyset, \emptyset, OP, PR), AX)$ is a subspecification of PSP. The initial semantics of PSP is given by the class **Ini(PSP)** of initial PSP-models over reachable SP-models. Accordingly, the **inductive theory of** PSP, **ITh(PSP)**, is defined as the intersection of the inductive theories of all built-in model PSP-extension of reachable SP-models. If all these specifications are inhabited, we conclude from Thm. 3.6 that for all Gentzen clauses c over PSP,

$$Ini(PSP) \models c \quad \text{iff} \quad c \in ITh(PSP).$$

Consequently, proofs of conjectures about PSP do not refer to properties of individual SP-models A. The only properties of A to be taken into account in such proofs are the axioms of SP.

Next we generalize the specifications SP_1 and SP_2 of Example 2.7 to parametric specifications that extend *ENTRY*. As usually, the function sort constructor $\rightarrow$ is supposed to be right-associative. $*$ binds stronger than $\rightarrow$.

```
LIST
     base      ENTRY
     sorts     list = list(entry)
     sort vars entry'
     consts    nil:list
               &:entry*list->list
     defs      is_nil:list->bool
               in:list->bool
               [_]:entry->list
               @:list*list->list
               length:list->nat
               prefix:list*list
               first:list->list
               last:list->entry
               map:(entry->entry')->list->list(entry')
               fold:(entry*entry'->entry')->list->entry'->entry'
               forall,exists:(entry->bool)->list->bool
               sublists:list->list(list)
               add:entry->list->list
     infixes   &  in  @
     vars      x,y:entry  z:entry'  f:entry->entry'  g:entry*entry'->entry'  h:entry->bool
               L,L':list  SL:list(list)
     axioms    is_nil(nil) = true
               is_nil(x&L) = false
```

```
                x in nil = false
                x in y&L = eq(x,y) or (x in L)
                [x] = x&nil
                nil@L = L
                L@nil = L
                (x&L)@L' = x&(L@L')
                length(nil) = 0
                length(x&L) = succ(length(L))
                prefix(nil,L)
                prefix(x&L,x&L')  <==  prefix(L,L')
                first(nil) = nil
                first(x&nil) = nil
                first(x&(y&L)) = x&first(y&L)
                last(nil) = bottom
                last(x&nil) = x
                last(x&(y&L)) = last(y&L)
                map(f)(nil) = nil
                map(f)(x&L) = f(x)&map(f)(L)
                fold(g)(nil)(z) = z
                fold(g)(x&L)(z) = f(x,fold(g)(L)(z))
                forall(h)(nil) = true
                forall(h)(x&L) = g(x) and forall(g)(L)
                exists(h)(nil) = false
                exists(h)(x&L) = h(x) or exists(h)(L)
                sublists(nil) = [nil]
                sublists(x&L) = sublists(SL)@map(add(x))(SL)  <==  sublists(L) = SL
                add(x)(L) = x&L

STREAM
     base       LIST
     sorts      stream(entry)
     acts       head:stream(entry)->entry
                tail:stream(entry)->stream(entry)
     opns       &:entry*stream(entry)->stream(entry)
                01,10:stream(nat)
                nats:nat->stream(nat)
                @:list*stream(entry)->stream(entry)
                merge:stream(entry)*stream(entry)->stream(entry)
     preds      fair:stream(nat)
     infixes    & @ merge
     vars       s,s':stream(entry)  x:entry
     axioms     head(x&s) = x
                tail(x&s) = s
                01 = 0&10
                10 = 1&01
                nats(n) = n&nats(n+1)
                nil@s = s
                (x&L)@s = x&(L@s)
                merge(x&s,s') = x&merge(s',s)
                fair(s)  ==>  \/{L,s'}{s = L@(0&s'), fair(s')}
```

By using structured sort names such as $stream(entry)$ and $stream(nat)$ we reduce the number of components of a parametric specification PSP in an obvious way. If PSP contains an operation or predicate F of type t, then, implicitly, PSP includes a synonymous operation or predicate of any type that matches t with respect to the sort variables of t. Correspondingly "actualized" axioms are also assumed to be part of PSP. For instance, the stream concatenation @ of $STREAM$ actually has as all types of the form

$$stream(t) * stream(t) \rightarrow stream(t)$$

where t is a type expression constructed from sorts of $STREAM$ and the type constructors $list, stream, *$ and $\rightarrow$. Concerning the function space constructor $\rightarrow$ it must be noted that we do not use λ-abstraction here. For capturing the equivalence of higher-order terms it is sufficient to replace the third congruence axiom of the cut calculus (cf. 3.3) by the following clauses. Let $F, G : s_1 \dots s_n \rightarrow s$ be two actions or operations of SP.

- $F(x_1, \dots, x_n) \equiv_s G(y_1, \dots, y_n) \Leftarrow \{F \equiv G, x_1 \equiv_{s_1} y_1, \dots, x_n \equiv_{s_n} y_n\}$
- $F \equiv G \Leftarrow \forall\{x_1, \dots, x_n\} F(x_1, \dots, x_n) \equiv_s G(x_1, \dots, x_n)$

10.1.1 Proofs by fixpoint induction

In Section 7 we have formulated fixpoint induction and coinduction as Gentzen rules and given two proofs employing fixpoint *coinduction* (Ex. 7.8(2) and (4)). Since we have just introduced the most widely used constructor type *LIST*, let us take the opportunity and prove the correctness conditions of some list functions by fixpoint *induction*.

The first example shows the correctness of a predicate *part* that holds true iff its second argument is a partition of its first argument.

```
PARTITION
     base      LIST
     opns      flatten:list(list)->list
     preds     part:list*list(list)
     vars      x,y:entry  L,L':list  P:list(list)
     axioms    (1) part([x],[[x]])
               (2) part(x&y&L,[x]&P)  <==  part(y&L,P)
               (3) part(x&y&L,(x&L')&P)  <==  part(y&L,L'&P)
               (4) flatten(nil) = nil
               (5) flatten(L&P) = L@flatten(P)
     conjects  (1) L = flatten(P)  <==  part(L,P)
```

The first argument of *part* is a nonempty list L. The second argument yields a partition of L into sublists. Axiom 1 defines the case where L consists of a single element x. Then the only partition of L has L as its single element. Axioms 2 and 3 deal with lists starting with two elements x and y. Here L denotes the rest of the list. Both clauses involve the same recursive call: remove x from the list and get a partition (P resp. $L'\&P$) of the remaining list $y\&L$. Axiom 2 extends P to a partition of the entire list $x\&y\&L$ by adding the sublist $[x]$ to P. Axiom 3 extends $L'\&P$ to a partition of $x\&y\&L$ by adding the sublist $x\&L'$ to P. Here L' stands for the first element of the partition of $y\&L$ that returns from the recursive call.

Proof of Conjecture 1:

$$L \equiv flatten(P) \Leftarrow part(L, P)$$

fixpoint induction on *part*

$$\begin{array}{ll} \vdash & [x] \equiv flatten([[x]]), \\ & x\&y\&L \equiv flatten([x]\&P) \Leftarrow y\&L \equiv flatten(P), \\ & x\&y\&L \equiv flatten((x\&L')\&P) \Leftarrow y\&L \equiv flatten(L'\&P) \end{array}$$

only-if-completion of $flatten$ and constructor clash

$$\begin{array}{ll} \vdash & \exists\{L, P\}([[x]] \equiv L\&P \wedge [x] \equiv L@flatten(P)), \\ & \exists\{L', P'\}([x]\&P \equiv L'\&P' \wedge x\&y\&L \equiv L'@flatten(P')) \Leftarrow y\&L \equiv flatten(P), \\ & \exists\{L'', P'\}((x\&L')\&P \equiv L''\&P' \wedge x\&y\&L \equiv L''@flatten(P')) \Leftarrow y\&L \equiv flatten(L'\&P) \end{array}$$

unification

$$\begin{array}{ll} \vdash & [x] \equiv [x]@flatten(nil), \\ & x\&y\&L \equiv [x]@flatten(P) \Leftarrow y\&L \equiv flatten(P), \end{array}$$

$$x\&y\&L \equiv (x\&L')@flatten(P) \;\Leftarrow\; y\&L \equiv flatten(L'\&P)$$

only-if-completion of @ and constructor clash

$$\begin{aligned}\vdash \quad & x\&nil \equiv x\&flatten(nil),\\ & x\&y\&L \equiv x\&flatten(P) \;\Leftarrow\; y\&L \equiv flatten(P),\\ & x\&y\&L \equiv x\&(L'@flatten(P)) \;\Leftarrow\; y\&L \equiv flatten(L'\&P)\end{aligned}$$

only-if-completion of $flatten$ and constructor clash

$$\begin{aligned}\vdash \quad & x\&nil \equiv x\&nil,\\ & x\&y\&L \equiv x\&flatten(P) \;\Leftarrow\; y\&L \equiv flatten(P),\\ & x\&y\&L \equiv x\&(L'@flatten(P)) \;\Leftarrow\; y\&L \equiv L'@flatten(P)\end{aligned}$$

constructor decomposition

$$\begin{aligned}\vdash \quad & y\&L \equiv flatten(P) \;\Leftarrow\; y\&L \equiv flatten(P),\\ & y\&L \equiv L'@flatten(P) \;\Leftarrow\; y\&L \equiv L'@flatten(P)\end{aligned}$$

subsumption

$$\vdash \quad TRUE$$

The second example is a specification of *merge sort*. Conjecture 1 below states the correctness of the algorithm. We use the action type of bags or multisets that will be given in Section 10.2.

```
MSORT
     base      LIST  BAG (cf. 10.2)
     defs      mergesort:list->list
               merge:list*list->list
               split:list->(list*list)
               mkBag:list->bag
     preds     sorted:list
     vars      x,y:entry  L,L',L1,L2,L3,L4:list
     axioms     (1) mergesort(nil) = nil
                (2) mergesort([x]) = [x]
                (3) mergesort(x&y&L) = merge(mergesort(x&L1),mergesort(y&L2))  <==  split(L) = (L1,L2)
                (4) merge(nil,L) = L }
                (5) merge(L,nil) = L }
                (6) merge(x&L1,y&L2) = x&merge(L1,y&L2)  <==  x <= y
                (7) merge(x&L1,y&L2) = y&merge(x&L1,L2)  <==  x > y
                (8) split(nil) = (nil,nil)
                (9) split([x]) = ([x],nil)
               (10) split(x&y&L) = (x&L1,y&L2)  <==  split(L) = (L1,L2)
               (11) mkBag(nil) = empty
               (12) mkBag(x&L) = insert(x,mkBag(L))
               (13) sorted(nil)
               (14) sorted([x])
               (15) sorted(x&y&L)  <==  x <= y /\ sorted(y&L)
     theorems   (1) sorted(merge(L1,L2))  <==  sorted(L1) /\ sorted(L2)
                (2) mkBag(merge(L1,L2)) = mkBag(L1@L2)
                (3) mkBag(L1@L2) = mkBag(L3@L4)  <==  mkBag(L1) = mkBag(L3) /\ mkBag(L2) = mkBag(L4)
                (4) mkBag(L1@L2) = mkBag(L)  <==  split(L) = (L1,L2)
                (5) mkBag(L1@(x&L2)) = mkBag(x&(L1@L2))
                (6) mkBag(x&L1) = mkBag(x&L2)  <==  mkBag(L1) = mkBag(L2)
                (7) sorted(x&L)  <==  sorted(L) /\ mkBag(L) = mkBag(y&(L1@L2))
                                      /\ sorted(x&L1) /\ sorted(y&L2) /\ x <= y
                (8) sorted(x&L)  <==  sorted(L) /\ mkBag(L) = mkBag(y&(L1@L2))
                                      /\ sorted(y&L1) /\ sorted(x&L2) /\ x <= y
     conjects   (1) sorted(L') /\ mkBag(L) = mkBag(L')  <==  mergesort(L) = L'
                (2) sorted(L)  <==  sorted(L1) /\ sorted(L2) /\ merge(L1,L2) = L
```

Theorems 1 to 4 are the lemmas used in the following proof of Conjecture 2. As this clause states the correctness of *mergesort* in terms of *sorted* and *mkBag*, Theorems 1 to 4 do so for the auxiliary functions *merge*, *split* and @. We write $L \sim L'$ for the equation $mkBag(L) \equiv mkBag(L')$.

Proof of Conjecture1:

$$sorted(L') \wedge L' \sim L \;\Leftarrow\; mergesort(L) \equiv L'$$

fixpoint induction on the graph of *mergesort* (cf. 6.13)

$$\begin{array}{ll} \vdash & sorted(nil) \wedge nil \sim nil, \\ & sorted([x]) \wedge [x] \sim [x], \\ & sorted(merge(L3, L4)) \wedge merge(L3, L4) \sim x\&y\&L \\ & \quad \Leftarrow\; split(L) \equiv (L1, L2) \wedge sorted(L3) \wedge L3 \sim x\&L1 \wedge sorted(L4) \wedge L4 \sim y\&L2 \end{array}$$

only-if-completion of *sorted* and *mkBag* and constructor clash

$$\begin{array}{ll} \vdash & empty \sim empty, \\ & insert(x, empty) \sim insert(x, empty), \\ & sorted(merge(L3, L4)) \wedge merge(L3, L4) \sim x\&y\&L \\ & \quad \Leftarrow\; split(L) \equiv (L1, L2) \wedge sorted(L3) \wedge L3 \sim x\&L1 \wedge sorted(L4) \wedge L4 \sim y\&L2 \end{array}$$

constructor decomposition

$$\begin{array}{ll} \vdash & sorted(merge(L3, L4)) \wedge merge(L3, L4) \sim x\&y\&L \\ & \quad \Leftarrow\; split(L) \equiv (L1, L2) \wedge sorted(L3) \wedge L3 \sim x\&L1 \wedge sorted(L4) \wedge L4 \sim y\&L2 \end{array}$$

deductive expansion upon Theorem 1

$$\begin{array}{ll} \vdash & sorted(L3) \wedge sorted(L4) \wedge merge(L3, L4) \sim x\&y\&L \\ & \quad \Leftarrow\; split(L) \equiv (L1, L2) \wedge sorted(L3) \wedge L3 \sim x\&L1 \wedge sorted(L4) \wedge L4 \sim y\&L2 \end{array}$$

term replacement upon Theorem 2

$$\begin{array}{ll} \vdash & sorted(L3) \wedge sorted(L4) \wedge L3@L4 \sim x\&y\&L \\ & \quad \Leftarrow\; split(L) \equiv (L1, L2) \wedge sorted(L3) \wedge L3 \sim x\&L1 \wedge sorted(L4) \wedge L4 \sim y\&L2 \end{array}$$

deductive expansion upon Theorem 3

$$\begin{array}{ll} \vdash & \exists\{L1, L2\}(sorted(L3) \wedge sorted(L4) \wedge L1@L2 \sim x\&y\&L \wedge L3 \sim L1 \wedge L4 \sim L2) \\ & \quad \Leftarrow\; split(L) \equiv (L1, L2) \wedge sorted(L3) \wedge L3 \sim x\&L1 \wedge sorted(L4) \wedge L4 \sim y\&L2 \end{array}$$

deductive expansion upon Theorem 4

$$\begin{array}{ll} \vdash & \exists\{L1, L2\}(sorted(L3) \wedge sorted(L4) \wedge split(x\&y\&L) \equiv (L1, L2) \wedge L3 \sim L1 \wedge L4 \sim L2) \\ & \quad \Leftarrow\; split(L) \equiv (L1, L2) \wedge sorted(L3) \wedge L3 \sim x\&L1 \wedge sorted(L4) \wedge L4 \sim y\&L2 \end{array}$$

only-if-completion of *split* and constructor clash

$$\begin{array}{ll} \vdash & \exists\{L1, L2, L1', L2'\}(sorted(L3) \wedge sorted(L4) \wedge (x\&L1', y\&L2') \equiv (L1, L2) \\ & \qquad\qquad \wedge\; split(L) \equiv (L1', L2') \wedge L3 \sim L1 \wedge L4 \sim L2) \\ & \quad \Leftarrow\; split(L) \equiv (L1, L2) \wedge sorted(L3) \wedge L3 \sim x\&L1 \wedge sorted(L4) \wedge L4 \sim y\&L2 \end{array}$$

constructor decomposition

$$\begin{array}{ll} \vdash & \exists\{L1, L2, L1', L2'\}(sorted(L3) \wedge sorted(L4) \wedge x\&L1' \equiv L1 \wedge y\&L2' \equiv L2 \\ & \qquad\qquad \wedge\; split(L) \equiv (L1', L2') \wedge L3 \sim L1 \wedge L4 \sim L2) \\ & \quad \Leftarrow\; split(L) \equiv (L1, L2) \wedge sorted(L3) \wedge L3 \sim x\&L1 \wedge sorted(L4) \wedge L4 \sim y\&L2 \end{array}$$

variable elimination

$$\begin{array}{ll} \vdash & \exists\{L1', L2'\}(sorted(L3) \wedge sorted(L4) \wedge split(L) \equiv (L1', L2') \wedge L3 \sim x\&L1' \wedge L4 \sim y\&L2') \\ & \quad \Leftarrow\; split(L) \equiv (L1, L2) \wedge sorted(L3) \wedge L3 \sim x\&L1 \wedge sorted(L4) \wedge L4 \sim y\&L2 \end{array}$$

subsumption

$$\vdash \quad TRUE$$

Conjecture 2, which is equivalent to Theorem 1, does not match the antecedent of fixpoint induction as defined in Section 7. While Conjectures 1 and 2 state the correctness of a predicate resp. function without any preconditions, Conjecture 3 demands a property of function values only under certain preconditions on the arguments, namely $sorted(L1)$ and $sorted(L2)$. For coping with an arbitrary precondition *pre* we derive a special induction rule.[35] First note that a Gentzen clause $gs \Leftarrow g \wedge h$ is equivalent to its "curried" version

[35] If P is the graph of a function F, then the precondition often describes an *invariant.* i.e. the conjecture reads as $inv(y) \Leftarrow inv(x) \wedge F(x) \equiv y$ (cf. 6.13).

$(gs \Leftarrow g) \Leftarrow h$. Hence the following rule is an instance of fixpoint induction:

$$\frac{gs \;\Leftarrow\; pre(x) \wedge P(x)}{\exists Q:\; prog(P)[Q/P],\quad gs \;\Leftarrow\; pre(x) \wedge Q(x)} \Updownarrow \tag{1}$$

The general version of fixpoint induction:

$$\frac{gs \;\Leftarrow\; P(x)}{\exists Q:\; prog(P)[Q/P],\quad gs \;\Leftarrow\; Q(x)} \Updownarrow \tag{2}$$

allows us to define $Q(x)$ by gs. We might need a generalization of gs, but, at first, one will always try to prove gs directly. Accordingly, (1) should allow us to define $Q(x)$ by $gs \Leftarrow pre(x)$. However, this is not possible in Horn logic unless the negation of pre can be axiomatized separately and thus $Q(x) \Leftarrow gs \vee \neg pre(x)$ is a Horn clause. pre describes a precondition, for instance a function domain. Hence the negation of pre is often unspecified. For these cases we develop a special fixpoint induction rule that is a simplification like the original one, provided that pre is compatible with $prog(P)$ in a certain sense described below.

When constructing $prog(P)[Q/P]$ we substitute $Q(u)$, i.e. $gs \vee \neg pre(u)$, for all atoms $P(u)$ that may occur not only in the conclusion, but also in the premise of some $c \in prog(P)$. Let $c = P(t) \Leftarrow g$. c can be written as

$$P(t) \Leftarrow P(u_1) \wedge \ldots \wedge P(u_n) \wedge h$$

where h consists of all atoms of g whose predicate is different from P. Then $c[Q/P]$ is equivalent to

$$gs(t) \;\Leftarrow\; pre(t) \wedge (gs(u_1) \vee \neg pre(u_1)) \wedge \ldots \wedge (gs(u_n) \vee \neg pre(u_n)) \wedge h, \tag{3}$$

which is not a Gentzen clause. However, if

$$pre(u_1) \wedge \ldots \wedge pre(u_n) \;\Leftarrow\; pre(t) \wedge h \tag{4}$$

holds true, then (3) follows from (5):

$$gs(t) \;\Leftarrow\; pre(t) \wedge gs(u_1) \wedge \ldots \wedge gs(u_n) \wedge h. \tag{5}$$

Since (4) is a sort of inversion of $c[pre/P]$ we denote (4) by $inv(c, pre)$ and define **inv(pre,P)** as the set of all Horn clauses $inv(c, pre)$ with $c \in prog(P)$. Moreover, let **prog(pre,P)** be the set of all clauses $P(t) \Leftarrow pre(t) \wedge g$ such that $P(t) \Leftarrow g \in prog(P)$. $prog(pre, P)$ is called the ***SP*-program for *P* with precondition** pre. For instance, the above clause c has a counterpart c' in $prog(pre, P)$ that is given by

$$P(t) \Leftarrow pre(t) \wedge P(u_1) \wedge \ldots \wedge P(u_n) \wedge h.$$

If $inv(c, pre)$ (= (4)) holds true, then $c'[Q/P]$ is equivalent to

$$Q(t) \;\Leftarrow\; pre(t) \wedge (Q(u_1) \vee \neg pre(u_1)) \wedge \ldots \wedge (Q(u_n) \vee \neg pre(u_n)) \wedge h$$

and thus equivalent to $c[Q'/P]$ where $Q'(x)$ is defined by $Q(x) \Leftarrow pre(x)$. With this definition of Q' the following rule is an instance of the expansion part of fixpoint induction:

$$\frac{gs \;\Leftarrow\; pre(x) \wedge P(x)}{\exists Q:\; prog(P)[Q'/P],\quad gs \;\Leftarrow\; pre(x) \wedge (Q(x) \Leftarrow pre(x))} \Uparrow \tag{6}$$

Since $gs \Leftarrow pre(x) \wedge (Q(x) \Leftarrow pre(x))$ is equivalent to $gs \Leftarrow pre(x) \wedge Q(x)$ and since – as illustrated above at c and c' – $prog(P)[Q'/P]$ is equivalent to $prog(pre, P)[Q/P]$, provided that $inv(pre, P)$ consists of inductive theorems, we conclude that under this assumption (6) can be reformulated as follows:

fixpoint induction on (pre, P)
$$\frac{gs \;\Leftarrow\; pre(x) \wedge P(x)}{\exists Q:\; prog(pre, P)[Q/P],\quad gs \;\Leftarrow\; pre(x) \wedge Q(x)} \Updownarrow$$

if P is a safety predicate or the graph of a defined function (cf. 6.13) and $inv(pre, P) \subseteq ITh(SP)$

As in (2), the contraction property of this rule follows from substituting P for all occurrences of Q in the succedent. $prog(pre, P)$ is implied by $prog(P)$. Moreover, with this rule we do not run into the problem involved in (1). As when using (2), one may always try to apply fixpoint induction on (pre, P) with $Q(x) \Leftrightarrow gs$.

Back to Conjecture 2 of *MSORT*:

$$sorted(L) \ \Leftarrow\ sorted(L1) \wedge sorted(L2) \wedge merge(L1, L2) \equiv L.$$

For applying fixpoint induction we add the predicate $pre : list \times list$ and the axiom

$$pre(L1, L2) \ \Leftarrow\ sorted(L1) \wedge sorted(L2)$$

to *MSORT*. P is the graph of *merge* and thus $inv(pre, P)$ consists of the clauses:

$$pre(L1, y\&L2) \ \Leftarrow\ pre(x\&L1, y\&L2) \wedge x \leq y, \qquad inv(pre,Axiom\ 6)$$
$$pre(x\&L1, L2) \ \Leftarrow\ pre(x\&L1, y\&L2) \wedge x > y, \qquad inv(pre,Axiom\ 7)$$

Proof of $inv(pre, P)$*:*

$inv(pre, P)$

only-if-completion of *pre*

$\vdash$ $sorted(L1) \wedge sorted(y\&L2) \ \Leftarrow\ sorted(x\&L1) \wedge sorted(y\&L2) \wedge x \leq y,$
$sorted(x\&L1) \wedge sorted(L2) \ \Leftarrow\ sorted(x\&L1) \wedge sorted(y\&L2) \wedge x > y$

deductive expansion upon Theorem 5

$\vdash$ $sorted(x\&L1) \wedge sorted(y\&L2) \ \Leftarrow\ sorted(x\&L1) \wedge sorted(y\&L2) \wedge x \leq y,$
$sorted(x\&L1) \wedge sorted(y\&L2) \ \Leftarrow\ sorted(x\&L1) \wedge sorted(y\&L2) \wedge x > y$

subsumption

$\vdash$ $TRUE$

Proof of Conjecture 2:

$sorted(L) \ \Leftarrow\ pre(L1, L2) \wedge merge(L1, L2) \equiv L$

fixpoint induction on (pre, P)

$\vdash$ $Q(nil, L, L) \ \Leftarrow\ pre(nil, L),$
$Q(L, nil, L) \ \Leftarrow\ pre(L, nil),$
$Q(x\&L1, y\&L2, x\&L) \ \Leftarrow\ pre(x\&L1, y\&L2) \wedge Q(L1, y\&L2, L) \wedge x \leq y,$
$Q(x\&L1, y\&L2, y\&L) \ \Leftarrow\ pre(x\&L1, y\&L2) \wedge Q(x\&L1, L2, L) \wedge x > y,$
$sorted(L) \ \Leftarrow\ pre(L1, L2) \wedge Q(L, L1, L2)$ *generalization*

only-if-completion of *pre* and Q upon $prog(Q) = \{Q(L1, L2, L) \Leftarrow sorted(L) \wedge L \sim L1@L2\}$

$\vdash$ $sorted(L) \wedge L \sim nil@L \ \Leftarrow\ sorted(nil) \wedge sorted(L),$
$sorted(L) \wedge L \sim L@nil \ \Leftarrow\ sorted(L) \wedge sorted(nil),$
$sorted(x\&L) \wedge x\&L \sim (x\&L1)@(y\&L2)$
$\Leftarrow\ sorted(x\&L1) \wedge sorted(y\&L2) \wedge sorted(L) \wedge L \sim L1@(y\&L2) \wedge x \leq y,$
$sorted(y\&L) \wedge y\&L \sim (x\&L1)@(y\&L2)$
$\Leftarrow\ sorted(x\&L1) \wedge sorted(y\&L2) \wedge sorted(L) \wedge L \sim (x\&L1)@L2 \wedge x > y,$
$sorted(L) \ \Leftarrow\ sorted(L1) \wedge sorted(L2) \wedge sorted(L) \wedge L \sim L1@L2$

only-if-completion of *sorted* and @, constructor clash and subsumption

$\vdash$ $sorted(L) \wedge L \sim L \ \Leftarrow\ sorted(L),$
$sorted(L) \wedge L \sim L \ \Leftarrow\ sorted(L),$

$$\begin{aligned}
&sorted(x\&L) \land x\&L \sim x\&(L1@(y\&L2)) \\
&\qquad \Leftarrow\ sorted(x\&L1) \land sorted(y\&L2) \land sorted(L) \land L \sim L1@(y\&L2) \land x \leq y, \\
&sorted(y\&L) \land y\&L \sim (x\&L1)@(y\&L2) \\
&\qquad \Leftarrow\ sorted(x\&L1) \land sorted(y\&L2) \land sorted(L) \land L \sim x\&(L1@L2) \land x > y
\end{aligned}$$

deductive expansion upon $L \sim L$ and term replacement upon Theorem 5

$$\begin{aligned}
\vdash\quad &sorted(L)\ \Leftarrow\ sorted(L), \\
&sorted(L)\ \Leftarrow\ sorted(L), \\
&sorted(x\&L) \land x\&L \sim x\&(y\&(L1@L2)) \\
&\qquad \Leftarrow\ sorted(x\&L1) \land sorted(y\&L2) \land sorted(L) \land L \sim L1@(y\&L2) \land x \leq y, \\
&sorted(y\&L) \land y\&L \sim y\&((x\&L1)@L2) \\
&\qquad \Leftarrow\ sorted(x\&L1) \land sorted(y\&L2) \land sorted(L) \land L \sim x\&(L1@L2) \land x > y
\end{aligned}$$

subsumption

$$\begin{aligned}
\vdash\quad &sorted(x\&L) \land x\&L \sim x\&(y\&(L1@L2)) \\
&\qquad \Leftarrow\ sorted(x\&L1) \land sorted(y\&L2) \land sorted(L) \land L \sim L1@(y\&L2) \land x \leq y, \\
&sorted(y\&L) \land y\&L \sim y\&((x\&L1)@L2) \\
&\qquad \Leftarrow\ sorted(x\&L1) \land sorted(y\&L2) \land sorted(L) \land L \sim x\&(L1@L2) \land x > y
\end{aligned}$$

deductive expansion upon Theorem 6

$$\begin{aligned}
\vdash\quad &sorted(x\&L) \land L \sim y\&(L1@L2) \\
&\qquad \Leftarrow\ sorted(x\&L1) \land sorted(y\&L2) \land sorted(L) \land L \sim L1@(y\&L2) \land x \leq y, \\
&sorted(y\&L) \land L \sim (x\&L1)@L2 \\
&\qquad \Leftarrow\ sorted(x\&L1) \land sorted(y\&L2) \land sorted(L) \land L \sim x\&(L1@L2) \land x > y
\end{aligned}$$

only-if-completion of @ and constructor clash

$$\begin{aligned}
\vdash\quad &sorted(x\&L) \land L \sim y\&(L1@L2) \\
&\qquad \Leftarrow\ sorted(x\&L1) \land sorted(y\&L2) \land sorted(L) \land L \sim L1@(y\&L2) \land x \leq y, \\
&sorted(y\&L) \land L \sim x\&(L1@L2) \\
&\qquad \Leftarrow\ sorted(x\&L1) \land sorted(y\&L2) \land sorted(L) \land L \sim x\&(L1@L2) \land x > y
\end{aligned}$$

deductive expansion upon Theorems 7 and 8

$$\begin{aligned}
\vdash\quad &sorted(L) \land L \sim y\&(L1@L2) \land sorted(x\&L1) \land sorted(y\&L2) \land x \leq y \\
&\qquad \Leftarrow\ sorted(x\&L1) \land sorted(y\&L2) \land sorted(L) \land L \sim L1@(y\&L2) \land x \leq y, \\
&sorted(L) \land L \sim x\&(L1@L2) \land sorted(x\&L1) \land sorted(y\&L2) \land y \leq x \\
&\qquad \Leftarrow\ sorted(x\&L1) \land sorted(y\&L2) \land sorted(L) \land L \sim x\&(L1@L2) \land x > y
\end{aligned}$$

term replacement upon Theorem 5 and deductive expansion upon $y \leq x \Leftarrow x > y$

$$\begin{aligned}
\vdash\quad &sorted(L) \land L \sim y\&(L1@L2) \land sorted(x\&L1) \land sorted(y\&L2) \land x \leq y \\
&\qquad \Leftarrow\ sorted(x\&L1) \land sorted(y\&L2) \land sorted(L) \land L \sim y\&(L1@L2) \land x \leq y, \\
&sorted(L) \land L \sim x\&(L1@L2) \land sorted(x\&L1) \land sorted(y\&L2) \land x > y \\
&\qquad \Leftarrow\ sorted(x\&L1) \land sorted(y\&L2) \land sorted(L) \land L \sim x\&(L1@L2) \land x > y
\end{aligned}$$

subsumption

$$\vdash\quad TRUE$$

10.2 Permutative types

If bags, maps and sets with complements are specified as action extensions of *ENTRY*, the "non-free constructors" *put* and *insert* (cf. 1.1) become defined functions.

```
MAP
     base      ENTRY
     sorts     map
     acts      get:map*index->entry
     consts    new:map
```

```
            put:index*entry*map->map
     vars   x:entry  f:map  i,j:index
     axioms get(new) = bottom
            get(put(i,x,f),i) = x
            get(put(i,x,f),j) = get(f,j)  <==  eq(i,j) = false

BAG
     base    ENTRY
     sorts   bag
     acts    card:bag*entry->nat
     consts  empty:bag
             insert,delete:entry*bag->bag
     vars    x,y:entry  b,b':bag
     axioms  card(empty,x) = 0
             card(insert(x,b),x) = succ(card(b,x))
             card(insert(x,b),y) = card(b,y)  <==  eq(x,y) = false
             card(delete(x,b),x) = pred(card(b,x))
             card(delete(x,b),y) = card(b,y)  <==  eq(x,y) = false

SET
     base     ENTRY
     sorts    set
     acts     contains:set*entry->bool
     consts   empty,all,evennats:set
              insert,delete:entry*set->set
              compl:set->set
              union,inter:set*set->set
              filter:(entry->bool)*set->set
     defs     diff:set*set->set
     preds    sub:set*set
     infixes  sub
     vars     x,y:entry  n:nat  s,s':set  f:entry->bool
     axioms   contains(empty,x) = false
              contains(all,x) = true
              contains(evennats,n) = even(n)
              contains(compl(s),x) = not(contains(s,x))
              contains(insert(x,s),y) = eq(x,y) or contains(s,y)
              contains(delete(x,s),y) = not(eq(x,y)) and contains(s,y)
              contains(union(s,s'),x) = contains(s,x) or contains(s',x)
              contains(inter(s,s'),x) = contains(s,x) and contains(s',x)
              contains(filter(f,s),x) = contains(s,x) and f(x)
              diff(s,s') = inter(s,compl(s'))
              s sub s'  ==>  /\{x}(not(contains(s,x)) or contains(s',x) = true)
```

Note that *sub* cannot be axiomatized by the clause

```
s sub s'  ==>  contains(s,x) = false \/ contains(s',x) = true
```

because this is not a liveness axiom: all universal variables of the clause must occur in the premise.

With *map*-, *bag*- and *set*-head normal forms being just variables, *MAP*, *BAG* and *SET* satisfy the coinduction principle trivially (cf. 4.6). Let $A = Ini(ENTRY)$. By Lemma 9.2, the Horn clause completions of (A,MAP), (A,BAG) and (A,SET) are consistent w.r.t. A. Moreover, using proof rules of Section 7 one may show that the "constructor equations"

```
vars    x,y:entry  b:bag  s:set  f:map  i,j:index
axioms  insert(x,insert(y,b)) = insert(y,insert(x,b))
        insert(x,insert(x,s)) = insert(x,s)
        insert(x,insert(y,s)) = insert(y,insert(x,s))
        put(i,x,put(i,y,f)) = put(i,x,f)
        put(i,x,put(j,y,f)) = put(j,y,put(i,x,f))  <==  eq(i,j) = false
```

(cf. Section 1.1) as well as commutatitivity, associativity and idempotence of *union* and *inter* are coinductive theorems of (A,*MAP*), (A,*BAG*) and (A,*SET*), respectively. *SET* specifies finite sets of elements of sort *entry*, complements of such sets with respect to the set *all* of all elements of sort *entry*, and the individual set *evennats* of all even natural numbers. Finite sets are represented by terms of the form $insert(t_1,\dots,insert(t_n,empty)\dots)$, while $delete(t_1,\dots,delete(t_n,all)\dots)$ stands for the complement of a finite set.

The following constructor-based specification SET_2 defines only finite sets of elements of sort *entry*:

```
SET_2
     base      ENTRY
     sorts     set
     acts      contains:set*entry->bool
     consts    empty:set
               insert:entry*set->set
     defs      delete:entry*set->set
               union,inter,diff:set*set->set
               sub:set*set->bool
     infixes   sub
     vars      x,y:entry  s,s':set
     axioms    contains(empty,x) = false
               contains(insert(x,s),y) = eq(x,y) or contains(s,y)
               delete(x,empty) = empty
               delete(x,insert(x,s)) = ite(eq(x,y),delete(x,s),insert(y,delete(x,s)))
               union(empty,s) = s
               union(insert(x,s),s') = insert(x,union(s,s'))
               inter(empty,s) = empty
               inter(insert(x,s),s') = ite(contains(s',x),insert(x,inter(s,s')),inter(s,s'))
               diff(s,empty) = s
               diff(s,insert(x,s')) = delete(x,diff(s,s'))
               (empty sub s) = true
               (insert(x,s) sub s') = contains(x,s') and (s sub s')
```

The choice of constructors is determined by the requirement that the specification becomes normal form complete (cf. 6.6). In contrast to *SET* where theorems satisfying the coinduction condition 4.6(3) came already as *axioms* we must *derive* them from SET_2, except those defining the constructors *empty* and *insert* because they are also axioms of SET_2. For instance, let us show that the *SET*-axiom

$(e)\quad contains(delete(x,s),z) \equiv not(eq(x,z))\ and\ contains(s,z)$

is an inductive theorem of (A,SET_2) and thus 4.6(3) holds true for $F = delete$. Let $t,u \in NF_{SIG,set}$ such that $SET_2 \vdash_{cut} \forall x : contains(t,x) \equiv contains(u,x)$. Then

$$\begin{aligned} SET_2 \vdash_{cut} contains(delete(x,t),y) &\equiv not(eq(x,y))\ and\ contains(t,y) \\ &\equiv not(eq(x,y))\ and\ contains(u,y) \equiv contains(delete(x,u),y) \end{aligned}$$

and the proof is complete. Let us prove $e = e(s)$ by convergence. For applying Thm. 9.19, we mainly have to show that the critical test clause for e (cf. 9.16) is an inductive SET_2-theorem and each critical clause of e on SET_2-axioms is subreductively valid w.r.t. (SET_2, e) (cf. 8.5). There are two such clauses e_1 and e_2:

$(e_1)\quad not(eq(x,z))\ and\ contains(empty,z) \equiv contains(empty,z)$ and

$(e_2)\quad not(eq(x,z))\ and\ contains(insert(y,s),z) \equiv contains(ite(eq(x,y),delete(x,s),insert(y,delete(x,s))),z).$

Hence $crit(e) = \{\{s \equiv empty\},\{s \equiv insert(y,s')\}\}$. Since SET_2 is normal form complete, $crit(e)$ is an inductive SET_2-theorem. Moreover, e_1 and e_2 must be sub-$e(empty)$-reductively valid and sub-$e(insert(y,s'))$-reductively valid, respectively. For this purpose we may apply Thm. 8.11 because SET_2 is functional and strongly terminating.

$(e_3)\quad not(eq(x,z))\ and\ false \equiv false$

is a $e(empty)$-bounded SET_2-theorem. Hence we may use e_3 in the following proof that e_1 is convergent:

$$\begin{array}{l} not(eq(x,z))\ and\ contains(empty,z) \longrightarrow_{SET_2} not(eq(x,z))\ and\ false \longrightarrow_{e_3} false \\ \longleftarrow_{SET_2} contains(empty,z) \end{array}$$

We leave it to the reader to identify the set E of $e(insert(y,s'))$-bounded SET_2-theorems used in the following proof that e_2 is convergent:

$$\begin{array}{ll} \multicolumn{2}{l}{contains(ite(eq(x,y),delete(x,s),insert(y,delete(x,s))),z)} \\ \longrightarrow_E & (eq(x,y)\ and\ contains(delete(x,s),z)) \\ & or\ (not(eq(x,y))\ and\ contains(insert(y,delete(x,s)),z)) \\ \longrightarrow_{SET_2} & (eq(x,y)\ and\ not(eq(x,z))\ and\ contains(s,z)) \\ & or\ (not(eq(x,y))\ and\ (eq(y,z)\ or\ contains(delete(x,s),z))) \\ \longrightarrow_e & (eq(x,y)\ and\ not(eq(x,z))\ and\ contains(s,z)) \\ & or\ (not(eq(x,y))\ and\ (eq(y,z)\ or\ (not(eq(x,z))\ and\ contains(s,z)))) \\ \longrightarrow_E & (eq(x,y)\ and\ not(eq(x,z))\ and\ contains(s,z)) \\ & or\ (not(eq(x,y))\ and\ eq(y,z))\ or\ (not(eq(x,y))\ and\ not(eq(x,z))\ and\ contains(s,z)) \\ \longleftarrow_E & (not(eq(x,z))\ and\ eq(y,z))\ or\ (not(eq(x,z))\ and\ contains(s,z)) \\ \longleftarrow_E & not(eq(x,z))\ and\ (eq(y,z)\ or\ contains(s,z)) \\ \longleftarrow_{SET_2} & not(eq(x,z))\ and\ contains(insert(y,s),z) \end{array}$$

Note that these two reductions correspond to the two cases a direct proof of e would fall into (see the remark following Ex. 9.12).

Analogously, it can be shown that the *SET*-axioms for *union*, *inter* and $diff$ are inductive theorems of (A, SET_2). For *sub*, one may turn the liveness axiom

```
s sub s'  ==>  /\{x}(not(contains(s,x)) or contains(s',x) = true)
```

of *SET* into equivalent Horn clauses over SET_2:

```
s sub s' = true  <==  /\{x}(not(contains(s,x)) or contains(s',x) = true})
s sub s' = false  <==  contains(s,x) = true /\ contains(s',x) = false
```

In fact, if these two clauses are inductive theorems of (A, SET_2), Condition 4.6(3) holds true because their premise equations are strict and cover all possible cases.

Infinite sets can be specified by guarded recursion, analogously to infinite streams (cf. 10.3). For instance, the set of all natural numbers $k \geq n$ is defined by the following extension of SET_2:

```
NATS
     base      SET_2
     consts    nats:nat->set(nat)
     vars      n
     axioms    nats(n) = insert(n,nats(n+1))
```

A further specification of finite sets employs *empty*, *single* (singleton set) and *union* as set constructors and all other operations of SET_2 as defined functions:

```
SET_3
```

```
     base      ENTRY
     sorts     set
     acts      contains:set*entry->bool
     consts    empty:set
               single:entry->set
               union:set*set->set
     defs      insert,delete:entry*set->set
               diff,inter:set*set->set
               sub:set*set->bool
     infixes   sub
     vars      x,y:entry  s,s',s":set
     axioms    contains(empty,x) = false
               contains(single(x),y) = eq(x,y)
               contains(union(s,s'),x) = contains(s,x) or contains(s',x)
               insert(x,s) = union(single(x),s)
               delete(x,empty) = empty
               delete(x,single(y)) = ite(eq(x,y),empty,single(y))
               delete(x,union(s,s')) = delete(x,delete(x,s),s')
               diff(s,empty) = s
               diff(s,single(x)) = delete(x,s)
               diff(s,union(s',s")) = diff(diff(s,s'),s")
               inter(s,empty) = empty
               inter(s,single(x)) = ite(contains(s,x),single(x),empty)
               inter(s,union(s',s")) = union(inter(s,s'),inter(s,s"))
               (empty sub s) = true
               (single(x) sub s) = contains(s,x)
               (union(s,s') sub s") = (s sub s") and (s' sub s")
```

Action consistency of *SET*, SET_2 and SET_3 w.r.t. *Ini*(*ENTRY*) can be shown easily with the help of Thm. 9.3.

Directed graphs with nodes of sort *entry* represent finite binary relations on *entry*. Hence we specify them analogously to SET_2:

```
RELATION
     base      SET_2
     sorts     rel
     acts      contains:rel*entry*entry->bool
     consts    empty:relation
               insert:entry*entry*rel->rel
     defs      +,*:rel*rel->rel
               succs:entry*rel->set
               cone:entry*set->rel
               depthfirst1:entry*rel*set->set
               depthfirst:set*rel*set->set
     preds     path:entry*entry*rel
     infixes   + *
     vars      x,y,z,x',y':entry  r,r':rel  s,s':set
     axioms    contains(empty,x,y) = false
               contains(insert(x,y,r),x',y') = (eq(x,x') and eq(y,y')) or contains(r,x',y')
               empty+r = r
               insert(x,y,r)+r' = insert(x,y,r+r')
               empty*r = r
               insert(x,y,r)*r' = (r*r')+cone(x,succs(y,r'))
               succs(x,empty) = empty
               succs(x,insert(y,z,r)) = ite(eq(x,y),insert(z,succs(x,r)),succs(x,r))
               cone(x,empty) = empty
               cone(x,insert(y,s)) = insert(x,y,cone(x,s))
               depthfirst1(x,r,s) = ite(contains(s,x),s,depthfirst(succs(x,r),r,insert(x,s)))
               depthfirst(empty,r,s) = s
               depthfirst(insert(x,s),r,s') = depthfirst(s,r,depthfirst1(x,r,s'))
```

```
path(x,y,r)  ==>  x = y \/ \/{z}(contains(r,x,z) /\ path(z,y,r))
```

RELATION satisfies the coinduction principle if the head normal forms for *rel* coincide with the ground normal forms of sort *rel*.

Exercise. Show that

$$path(x, y, r) \quad \Leftrightarrow \quad contains(depthfirst1(x, r, empty), y) \equiv true$$

is a coinductive theorem of *RELATION*.

10.3 Streams

Streams are certainly the most extensively used infinite data structures. They occur in functional languages like Miranda (cf. [Tur 90]) and Haskell (cf. [Dav 92]), build up the *trace semantics* of concurrent languages (cf. [Hoa 85]) and provide the basic data type in specifications of distributed systems (cf. [BDDFGW 93]). Stream specifications have already been given in Ex. 2.5 (SP_2) and Sect. 10.1 (*STREAM*). The "natural" *stream*-head normal form is given by the term $x\&s$ (cf. Ex. 4.7). For making *STREAM* normal form complete (cf. 6.6), 01, 10, *nats* and *merge* must be axiomatized as in Ex. 6.7.

The *stream*-actions could be chosen differently. For example, the cpo approach to streams (cf. 1.3) is implicitly based on the action $prelist : stream \times nat \rightarrow list$ instead of *head* and *tail*. For instance, & could be specified in terms of *prelist* as follows:

```
prelist(x&s,0) = nil
prelist(x&s,n+1) = x&prelist(s,n)
```

$prelist(s, n)$ maps s to the list of the first n elements of s. Conversely, one may keep to the actions *head* and *tail*, regard *prelist* as an operation specified by the above equations and prove that the so-called take lemma

$$s \equiv s' \ \Leftarrow\ \forall\{n\}(prelist(s, n) \equiv prelist(s', n))$$

is a coinductive theorem (cf. [BW 88], 7.5.1).

A third way of achieving stream equivalence is provided by the action $prefix : stream \rightarrow (list \rightarrow bool)$ in terms of which & could have the following axioms:

```
prefix(x&s,nil) = true
prefix(x&s,y&L) = eq(x,y) and prefix(x&s,L)
```

$prefix(s)(L)$ returns *true* iff L is a prefix of s. Of course, *prefix* could also be specified as a liveness predicate as in the following extension of *STREAM*:

```
STREAM_2
     base      STREAM (cf. 10.1)
     sort vars entry'
     opns      repeat:entry->stream(entry)
               iterate:(entry->entry)->entry->stream(entry)
               map:(entry->entry')->stream(entry)->stream(entry')
     preds     regular:stream(entry)
               prefix:list*stream(entry)
               exists:(entry->bool)*stream(entry)
     vars      x:entry  s,s':stream(entry)  f:entry->entry'  g:entry->bool  L,L':list
     axioms    repeat(L) = L@repeat(L)
```

```
iterate(f)(x) = x&iterate(f)(f(x))
map(f)(x&s) = f(x)&map(f)(s)
regular(s)  ==>  \/{L,L'}(s = L@repeat(L'))
prefix(L,s)  ==>  L = nil \/ \/{x,L',s'}(L = x&L' /\ s = x&s' /\ prefix(L',s'))
exists(g,s)  ==>  \/{L}(prefix(L,s) /\ exists(g)(L) = true)
```

$repeat(x)$ generates the stream of infinitely many occurrences of x. $iterate(f)(x)$ repeatedly applies f and thus produces the stream $(x, f(x), f(f(x)), \ldots)$. $map(f)$ maps a stream of numbers to the stream of their images under f. $regular$ recognizes regular streams (streams that eventually become periodic). Compare the above liveness axiom for $prefix$ with the Horn clause axioms for the corersponding relation on lists (cf. 10.1). $exists(g, s)$ tests whether s contains an element satisfying g.

Exercise. Construct the if-completions of $regular$, $prefix$ and $exists$ (cf. 5.10).

Bird and Wadler employ the above take lemma in a proof of the equation

$$iterate(f)(f(x)) \equiv map(f)(iterate(f)(x)) \tag{1}$$

(cf. [BW 88], p.184). This in fact shows that (1) is a coinductive theorem with *prelist* as the underlying *stream*-action. [Gor 95], 4.1, proves the same equation set-theoretically by coinduction (cf. 1.5). We leave the reader to prove (1) w.r.t. the actions *head* and *tail* and employ rules of Section 7, in particular fixpoint coinduction (cf. Ex. 7.8(2)). Show also the equation

$$map(f \circ g)(s) \equiv map(f)(map(g)(s))$$

taken from [Gor 95].

An operation on streams that filters all elements satisfying a given Boolean function cannot be specified in terms of *STREAM* because its range must comprise finite as well as infinite lists. *This* problem could be solved easily by adding (a copy of) the empty list *nil* to the constants of sort *stream*. *nil* becomes also a further *stream*-head normal form. The *stream*-actions *head* and *tail* or *prelist* or $prefix$, respectively, are extended to *nil* as follows:

```
head(nil) = bottom
tail(nil) = nil
prelist(nil,n) = nil
prefix(nil,L) = is_nil(L)
```

Nevertheless, if we specify the filter operation by:

```
filter(g)(x&s) = ite(g(x),x&filter(g)(s),filter(g)(s))
```

the coinduction principle is violated because $filter$ is not defined by *guarded recursion* (cf. 4.7): each *stream*-sorted ground term t must be equivalent to an instance of the head normal form $n\&s$. Then for all *stream*-contexts c, $c(t)$ is equivalent to a term over *LIST* and the coinduction principle is satisfied. Any stream constant such as 01 and 10, any stream generating function such as *nats* or *repeat* and even any stream processing function can be added to *STREAM* without violating the coinduction principle, provided that it is defined by guarded recursion.

The situation can be remedied by changing the representation of streams, in particular the finite elements among them. We abstract from all occurrences of bottom constants in a stream. To this end we define new *stream*-actions, which ensure that two streams are behaviourally equivalent if they are equal up to occurrences of $\perp$.

```
ABS_STREAM
     base      LIST
     sorts     stream(entry)
     sort vars entry'
     acts      head:stream(entry)->entry
               tail:stream(entry)->stream(entry)
     opns      head':nat*stream(entry)->entry
               &:entry*stream(entry)->stream(entry)
               filter:(entry->bool)->stream(entry)->stream(entry)
               map:(entry->entry')->stream(entry)->stream(entry')
     vars      s:stream(entry)  x:entry  i:nat  f:entry->entry'  g:entry->bool
     axioms    head(s) = x  <==  head'(i,s) = x /\ eq(x,bottom) = false
               head(s) = bottom  <==  /\{i}(head'(i,s) = bottom)
               head'(0,s) = bottom
               head'(i+1,x&s) = ite(eq(x,bottom),head'(i,s),x)
               tail(x&s) = s
               filter(g)(x&s) = ite(g(x),x&filter(g)(s),bottom&filter(g)(s))
               map(f)(x&s) = f(x)&map(f)(s)  <==  f(bottom) = bottom
```

head$'(i,s)$ is the first element x of s that is different from $\perp$ if x is among the first i elements of s. Otherwise *head*$'(i,s) = \perp$. Now *each* recursive call of $filter$ has a guard, namely x or $\perp$. Note that *ABS_STREAM* is not *entry*-complete (cf. 3.7) because *head*$'$ is defined only partially. The premise of the *map* axiom ensures that behavioural equivalence is compatible with *map*. Analogous abstractions can be performed on the *stream*-actions *prelist* and $prefix$ (see above):

```
ABS_STREAM2
     base      LIST
     sorts     stream(entry)
     sort vars entry'
     acts      prelist:stream(entry)*nat->list
     opns      prelist':nat*stream(entry)->list->bool
     vars      s:stream(entry)  x:entry  i:nat  L,L':list
     axioms    prelist(s,n) = L  <==  prelist'(i,s,n) = x&L'
               prelist(s,L,n) = nil  <==  /\{i}(prelist'(i,s,n) = nil)
               prelist'(i,s,0) = nil
               prelist'(0,x&s,n+1) = prelist'(0,s,n)
               prelist'(i+1,x&s,n+1)
                 = ite(eq(x,bottom), prelist'(i,s,n+1), prelist'(i,s,n))

ABS_STREAM3
     base      LIST
     sorts     stream(entry)
     sort vars entry'
     acts      prefix:stream(entry)*list->bool
     opns      prefix':nat*stream(entry)*list->bool
     vars      s:stream(entry)  x:entry  i:nat  L:list
     axioms    prefix(s,L) = true  <==  prefix'(i,s,L) = true
               prefix(s,L) = false  <==  /\{i}(prefix'(i,s,L) = false)
               prefix'(i,s,nil) = true
               prefix'(0,x&s,y&L) = eq(x,y) and prefix'(0,s,L)
               prefix'(i+1,x&s,y&L)
                 = ite(eq(x,bottom), eq(x,y) and prefix'(i,s,y&L), prefix'(i,s,L))
```

prelist$'(i,s,n)$ is the list of the first $k < n$ elements of s that are different from $\perp$ among the first i elements of s. $prefix'(i,s,L)$ is true if the list of the first n elements of s that are different from $\perp$ is among the first i elements of s.

In terms of process algebra, the behavioural equivalences of the above "abstract" stream specifications are *weak bisimilarities*. Similar abstractions from stronger equivalences are specified in Sections 10.4.3 and 10.5.2.

We claim that the following equation taken from [Gor 95] is a coinductive theorem of all three abstract stream specifications:

$$filter(g)(map(f)(s)) \equiv map(f)(filter(g \circ f)(s)).$$

Here are further examples of stream operations:

```
STREAM_3
     base      ABS_STREAM
     opns      fib:nat->stream(nat)
               filter2:stream(entry)*stream(entry)->stream(entry)
               switch:stream(nat)*stream(entry)*stream(entry)->stream(entry)
     vars      n:nat  c:stream(nat)  x,y:entry  s,s':stream(entry)
     axioms    fib(0) = 0&fib(1)
               fib(1) = 1&fib(2)
               fib(succ(n+1)) = (head(fib(n))+head(fib(n+1)))&fib(succ(succ(n+1)))
               filter2(n&c,x&s) = ite(eq(n,0),x&filter2(c,s),bottom&filter2(c,s))
               switch(n&c,x&s,y&s') = ite(eq(n,0),x&switch(c,s,y&s'),y&switch(c,x&s,s'))
```

$fib(n)$ generates a stream of consecutive Fibonacci numbers, starting from the n-th one. The stream processing functions $filter2$ and $switch$ are taken from [BB 84]. Their first stream argument determines how the other arguments are processed.

Note that the above specifications are not functional (cf. 6.6), but can be turned easily into functional ones by modifying them analogously to SP_2 (cf. Ex. 6.7).

As explained in Section 1.3, the specification SP of an action type need not present all objects whose behaviour is definable on the visible subtype. More precisely, not all of the following functions will have ground term representations in SP.

Definition (s-behaviour) Let Assumption 4.1 hold true. Then for all $s \in S' \setminus S$, let $Cont_s$ be the set of s-contexts. The product

$$Beh_s \quad = \prod_{c:sw \to s' \in Cont_s} [A_w \to A_{s'}]$$

is called the set of **s-behaviours of** (A, SP'). ❏

For example, let A be the initial $LIST$-model over a reachable $ENTRY$-model and $s = stream(nat)$ (cf. 10.1). Since each s-context of $STREAM$ has the form $head(tail(\ldots tail(x)\ldots)) : s \to nat$, we have $Cont_s \cong I\!N$. Since $A_{nat} \cong I\!N$, Beh_s is isomorphic to the usual representation $[I\!N \to I\!N]$ of the set of *all* streams of natural numbers.

In the sequel, we use the notation $[t_1, \ldots, t_n]$ for a list term of the form $t_1\& \ldots \&t_n\&nil$.

10.3.1 Distributed systems

Distributed systems are often specifiable as networks of stream processing functions. For instance, the famous *alternating bit protocol* (cf. [BSW 69]) can be presented as a ring of message stream processing functions *send* and *receive* and two instances of the operation *channel*, which simulates the unsafe transmission of messages.

```
ABP
     base      STREAM
     sorts     entry2
     opns      pair:entry*bool->entry2
               send:stream(entry)*stream(bool)*bool->stream(entry2)
               receive:stream(entry2)*bool->stream(bool)*stream(entry)
               channel:stream(entry)*stream(bool)->stream(entry)
```

```
preds    fairbs:stream(bool)
vars     x:entry  b,b':bool  xs:stream(entry)  bs,bs':stream(bool)
         ps:stream(entry2)  bL:list(bool)
axioms   send(x&xs,b'&bs,b)
           = ite(eq(b,b'),pair(x,b)&send(xs,bs,not(b)),pair(x,b)&send(x&xs,bs,b))
         send(x&xs,bottom&bs,b) = pair(x,b)&send(x&xs,bs,b)
         receive(pair(x,b)&ps,b) = (b&bs,x&xs)  <==  receive(ps,not(b)) = (bs,xs)
         receive(pair(x,b)&ps,b') = (not(b')&bs,xs)
            <==  eq(b,b') = false /\ receive(ps,b') = (bs,xs)
         receive(bottom&ps,b) = (not(b)&bs,xs)  <==  receive(ps,b) = (bs,xs)
         channel(x&xs,true&bs) = x&channel(xs,bs)
         channel(x&xs,false&bs) = bottom&channel(xs,bs)
         fairbs(bs)  ==>  \/{bL,bs'}(bs = bL@(true&bs') /\ fairbs(bs'))
```

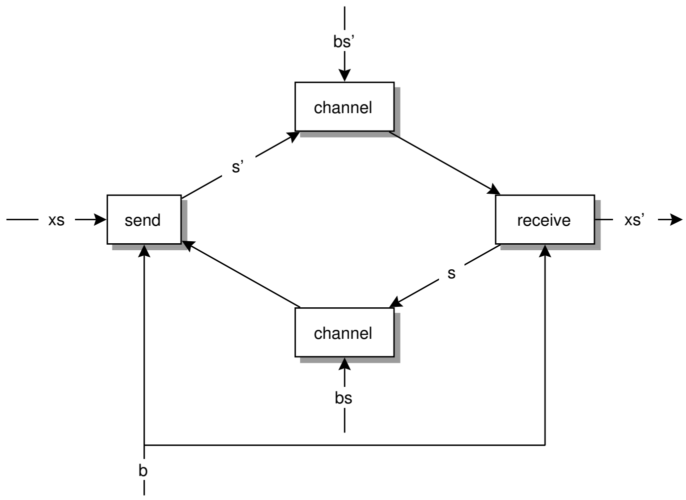


Figure 12. An alternating bit protocol

The edge labels in Fig. 12 correspond to the variables of the following goal that describes the network:

$$send(xs, channel(s, bs), b) \equiv s' \wedge receive(channel(s', bs'), b) \equiv (s, xs') \qquad (NET)$$

ABP agrees with the specification given in [Pad 92], Sect. 8.7, except that finite lists were here replaced by streams. The "oracle" streams bs and bs' determine which elements are transmitted correctly between the sender and the receiver. bs and bs' are fair if only finitely many consecutive elements are damaged. Under this assumption the message stream xs consumed by the sender is required to be equivalent to the stream xs' returned by the receiver. Hence the Horn clause

$$xs \equiv xs' \Leftarrow NET \wedge fairbs(bs) \wedge fairbs(bs')$$

expresses the correctness of *ABP*. The proof of this clause given in [Pad 92] only applies to finite message lists and thus proceeds by induction on their length. But we claim that it can be transformed easily into a corresponding proof for streams, based on the Horn clause completion of *ABP*.

10.3.2 Process networks

A network consisting of a variable number n of processes $p_1, \ldots, p_n$ and a process manager pm controlling a queue of free process identifiers is to be specified as an extension of *STREAM*. The network is shaped as a ring with all processes attached as shown in Fig. 13. The following specification *RING* has been designed along the lines of a functional program, presented in Section 3 of [JS 89].

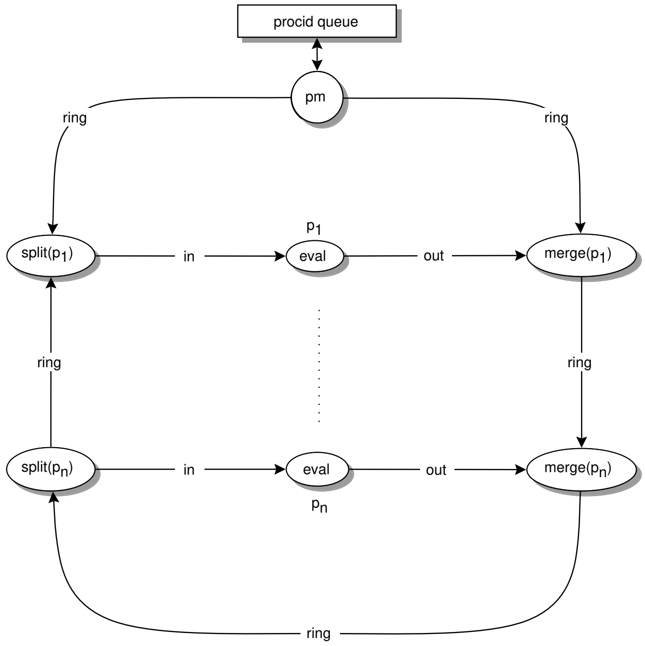


Figure 13. A network of processes

```
RING
     base      STREAM
     sorts     procid  agent  message
               inout = stream(procid*message)
               ringmsgs = stream(procid*message*procid)
     opns      pmid,p1,...,pn:procid
               create:message
               created,kill:procid->message
               execute:procid*agent->message
               pm:list(procid)*ringmsgs->list(procid)*ringmsgs
               split:procid->ringmsgs->inout*ringmsgs
               merge:procid->ringmsgs*inout->ringmsgs
               eval:inout*agent->inout
     vars      m,m':message  pid,src,dest,dest',new:procid  ring,ring',ring":ringmsgs
               in,out:inout  queue,queue',queue":list(procid)
     axioms    pm(queue,(src,create,pmid)&ring) = (queue",(pmid,created(new),src)&ring')
                   <==  is_nil(queue) = false /\ last(queue) = new /\ first(queue) = queue'
                        /\ pm(queue',ring) = (queue",ring')
               pm(nil,(src,create,pmid)&ring) = pm(queue',(src,create,pmid)&ring')
                   <==  pm(nil,ring) = (queue',ring')
               pm(queue,(src,execute(pid,a),pmid)&ring) = pm(queue,ring")
                   <==  split(pid)(ring) = (in,ring') /\ merge(pid)(ring',eval(in,a)) = ring"
               pm(queue,(src,kill(pid),pmid)&ring) = pm(pid&queue,ring)
               pm(queue,(src,m,dest)&ring) = (queue',(src,m,dest)&ring')
                   <==  eq(dest,pmid) = false /\ pm(queue,ring) = (queue',ring')
               split(pid)((src,m,dest)&ring) = (ite(eq(pid,dest),(src,m)&in,in),ring')
                   <==  split(pid)(ring) = (in,ring')
               merge(pid)((src,m,dest)&ring,(dest',m')&out)
                 = (src,m,dest)&(pid,dest',m')&merge(pid)(ring,out)
```

agent stands for concurrent programs of a still unspecified language that processes *inout* streams. Such a language has been defined and investigated in, e.g., [BPW 84], Sect. 6, under the name DSPL[36]. Section 10.3.3

[36]deterministic stream processing language

presents a similar language whose (abstract) syntax and semantics is given by *agent* constructors and by axioms defining the interpreter function *eval* that is associated with each of the processes $p_1, \ldots, p_n$ in the ring network (cf. Fig. 13). *RING* treats *eval* as an unspecified parameter.

split(p) adds messages for p to p's input stream. *merge*(p) merges a message from p with the ring stream. *merge*(p) Alternates between letting a ring message pass by p's output stream and merging a message from p with the ring stream.

pm executes *create* and *kill* messages by (virtually) inserting resp. deleting processes. For creating a process, *pm* takes the first element, say *new*, of the process identifier queue and sends it to the sender, say s, of the *create* message. If the queue is empty, the *create* command is delivered to the ring and *pm* will handle it when the queue has become non-empty. After creation of, say, process p, q receives the message *created*(p) from *pm*, which may cause q to send an *execute* message to the process manager. *pm* reacts to this message by opening p's input stream, evaluating an agent and merging p's output stream with the ring stream. For killing a process p, *pm* adds it(s identifier) to the queue. *create*, *execute* and *kill* messages are the only ones handled by the process manager. Other messages pass *pm* unseen.

10.3.3 Concurrent languages

We first specify a concurrent programming language called CSP0, which is a mixture of DSPL (cf. 10.3.2) and Hoare's CSP (cf. [Hoa 85]). In contrast to DSPL, CSP0 forbids recursion than tail recursion, but admits parallelism (||) as well as communication via the send operator "!" and the receive operator "?". From DSPL, CSP0 adopts the substitution operator for agents, which may replace agent names by process identifiers. For the sake of comprehension, mixfix notation is used for *agent* constructors.

```
CSP0_ABS
     sort vars id  agentid  procid
     sorts     exp  boolexp
     consts    parbegin _ parend :list(agent)->agent
               #:procid->agent
               _:=_;_ :id*exp*agent->agent
               _?_;_ :agentid*id*agent->agent
               _!_;_ :agentid*exp*agent->agent
               skip:agent
               if:list(agent)->agent
               do:list(agent)->agent
               _->_ :agent*agent->agent
               _->_ :boolexp*agent->agent
               _::_ :agentid*agent->agent
               var:id->exp
     defs      ids:list(agent)->list(agentid)
               bodies:list(agent)->list(agent)
               substLL:list(procid)*list(agentid)*list(agent)->list(agent)
               substL:procid*agentid*list(agent)->list(agent)
               substaid:procid*agentid*agent->agent
               substid:exp*id*agent->agent
               substid:exp*id*exp->exp
     infixes   ->  ::
     vars      a:agent  aL:list(agent)  aid,aid':agentid  aidL:list(agentid)  x,y:id
     axioms    ids(nil) = nil
               ids((p::a)&aL) = p&ids(aL)
               bodies(nil) = nil
               bodies((p::a)&aL) = a&bodies(aL)
               substLL(nil,nil,aL) = nil
               substLL(pid&pidL,aid&aidL,aL) = substL(pid,aid,substLL(pidL,aidL,aL))
               substL(pid,aid,nil) = nil
```

```
substL(pid,aid,a&aL) = substaid(pid,aid,substL(pid,aid,aL))
substaid(pid,aid,aid'?x;a) = ite(eq(aid,aid'),#pid?x,aid'?x);substaid(pid,aid,a)
substaid(pid,aid,aid'?x->a) = ite(eq(aid,aid'),#pid?x,aid'?x)->substaid(pid,aid,a)
substid(e,x,var(y)) = ite(eq(x,y),x,var(y))
... "frame" axioms for substaid and substid ...
```

Let us implement the factorial function in CSP0. From a concurrency point of view, the factorial pertains to the class of producer/consumer problems for which [BB 84] present a schematic translation into CSP. Our CSP0 program for the factorial follows that schema:

```
parbegin    producer :: x:=n; do x > 0 -> (x:=x-1; buffer!x; skip) od
            consumer :: x:=n; y:=1; do x > 0 -> (y:=x*y; buffer?x; skip) od
            buffer :: b:=nil; do producer?x -> b:=b@[x]
                                 b <> nil -> (consumer!head(b); b:=tail(b); skip)
                              od
parend
```

As in CSP, the agents defined by a CSP0 program communicate via handshaking and not via shared variables. Hence each variable (element of sort *id*) is local with respect to a scope that is given by the "body" a of a named agent $p :: a$. Variable declarations are omitted in CSP0.

The semantics of CSP0 shall be based on the ring network of 10.3.2. To this end we extend *CSP0_ABS* by a specification of the interpreter function *eval* that is associated with each process identifier of *RING*. This complies with the interpretation of concurrent programs as functions from input to output streams, which is also adopted from DSPL.

```
CSP0_EVAL
    base    RING  CSP0_ABS
    opns    getpids:inout*nat*list(procid)*list(agent)->inout
            exec:inout*list(procid)*list(procid)*list(agent)->inout
            replyfrom:procid*inout->(message*inout)
            evalcond,evalloop:inout*list(agent)->inout
            evalbool:boolexp->bool
            val:exp->message
    vars    a:agent  aL:list(agent)  aid,aid':agentid  aidL:list(agentid)  in,in':inout
            e:exp  be:boolexp  x,y:id  n:nat
    axioms  eval(in,parbegin aL parend)) = getpids(in,length(aL),nil,aL)
            getpids(in,0,pidL,aL) = exec(in,pidL,nil,substLL(pidL,ids(aL),bodies(aL)))
            getpids(in,n+1,pidL,aL) = (pmid,create)&getpids(in',n,pidL@[new],aL)
                <==  replyfrom(pmid,in) = (created(new),in')
            exec(in,nil,nil,aL) = in
            exec(in,nil,pid&pidL,aL) = (pmid,kill(pid))&exec(in,nil,pidL,aL)
            exec(in,pid&pidL,pidL',a&aL) = (pmid,execute(a))&exec(in,pidL,pid&pidL',aL)

            eval(in,x:=e;a) = eval(in,substid(e,x,a))
            eval(in,#pid?x;a) = eval(in',substid(e,x,a))
                <==  replyfrom(pid,in) = (val(e),in')
            eval(in,#pid!e;a) = (pid,val(e))&eval(in,a)
            eval(in,skip) = in

            eval(in,if(aL)) = evalcond(in,aL)
            evalcond(in,nil) = in
            evalcond((src,x)&in,(#pid?v->a)&aL)
              = ite(eq(src,pid),eval(in,substid(x,v,a)),evalcond(in,aL))
            evalcond(in,(be->a)&aL) = ite(evalbool(be),eval(in,a),evalcond(in,aL))

            eval(in,do(aL)) = evalloop(in,aL)
            evalloop(in,nil) = in
```

```
evalloop((src,val(e))&in,(#pid?x->a)&aL)
  = ite(eq(src,pid),eval(in,substid(e,x,a);do(aL)),evalloop(in,aL))
evalloop(in,(be->a)&aL) = ite(evalbool(be),eval(in,a;do(aL)),evalloop(in,aL))

replyfrom(pid,(src,m)&in) = ite(eq(pid,src),(m,in),(m',(src,m)&in'))
    <==  replyfrom(pid,in) = (m',in')
```

Constructors for *exp* and *boolexp* as well as the interpreter *evalbool* for Boolean expressions have been left unspecified. Some axioms for *eval* violate the normal form completeness of *CSP0_EVAL*. However, this property can be enforced by totalizing *eval* along the lines of the remark following Def. 4.2.

10.4 Process algebra

The *stream*-actions *head* and *tail* suggest to define in an analogous way infinite trees, which arise, for instance, from unfolding process graphs. *head* and *tail* would be replaced by $root : proc \to entry$[37] and $subprocs : proc \to procs$. However, this would entail multiple representations of the same process. Processes are usually modelled as elements of an initial structure with non-free constructors. The corresponding set of constructor equations is known as *basic process algebra* (BPA) (cf. [BW 90]). For modelling BPA correctly the sort *procs* must be interpreted as the set of sets and not of lists of processes, which would follow from the above choice of actions. Hence *head* and *tail* might be replaced by a membership function $containsproc : procs \times proc \to bool$, analogously to *contains* of *SET* (cf. 10.2). However, *containsproc* is not an action because both *procs* and *proc* are action sorts, which mutually depend on each other and thus cannot be introduced one after the other.

Therefore, *head* and *tail* do not provide an appropriate starting point for the actions of process algebra. In Sect. 10.3, we mentioned two alternative *stream*-actions: $prelist : stream \times nat \to list$, mapping a stream s to the list of the first n elements of s where n is the second parameter of *prelist*, and $prefix : stream \to (list \to bool)$, which checks a given list for being a prefix of s. Analogously to *prelist*, an action $pretree : inftree \times nat \to tree$ could map a process tree to its prefix trees of finite depth. Indeed, the *projection functions* of process algebra (cf. [BW 90]) are instances of *pretree* and equip processes with a terminal semantics. Moreover, the *Approximation Induction Principle (AIP)* of [BW 90] identifies the processes that are behaviourally equivalent w.r.t. *pretree*, and the *projection algebras* of [EPBRDG 90] are models of processes that satisfy AIP. But processes are not trees so that BPA needs constructor equations in order to capture the set structure of processes. Hence we take the third alternative and turn the *stream*-action *prefix* into a *proc*-action by replacing the *list*-parameter with a *tree*-parameter where *tree* is the sort of finite trees with arbitrary finite outdegree. $prefix : proc \to (list \to bool)$ captures both the tree and the set structure of processes.

10.4.1 Basic process algebra

Starting out from a constructor extension *TREE* of *LIST* (cf. 10.1) we "swing" to the action extension *PROCESS* and define the operations of BPA on terms of the action *prefix*. In the sequel, we use s as an abbreviation for the sort $s(entry)$.

```
TREE
     base      LIST
     sorts     tree
     consts    *:entry*list(tree)->tree
     defs      <=,eq:tree*tree->bool
               <=,eq:list(tree)*list(tree)->bool
               conc:tree*tree*tree->bool
               conc:list(tree)*tree*list(tree)->bool
```

[37] Node entries of a process graph denote atomic actions.

```
    vars      x,y:entry  t,u,v:tree  tL,uL:list(tree)
    infixes   * <=
    axioms    (x*tL <= y*uL) = eq(x,y) and (tL <= uL)
              (nil <= tL) = true
              (t&tL <= nil) = false
              (t&tL <= u&uL) = (t <= u) and (tL <= uL)
              eq(t,u) = (t <= u) and (u <= t)
              eq(tL,uL) = (tL <= uL) and (uL <= tL)
              conc(x*tL,v,y*uL) = eq(x,y) and conc(tL,v,uL)
              conc(nil,v,uL) = eq([v],uL)
              conc(t&tL,v,u&uL) = conc(t,v,u) and conc(tL,v,uL)
```

$t \leq u$ holds true if t is a prefix of u. eq decides the equality of trees. $conc(t, v, u)$ is true iff u is obtained from t by appending v to t at all leaves of t.

```
PROCESS
    base      TREE
    sorts     proc
    acts      prefix:proc->tree->bool
    consts    empty,deadlock:proc
              *:entry*proc->proc
              +:proc*proc->proc
    defs      o:tree*proc->proc
              add:proc->list(tree)*proc->proc
              leaf:entry->proc
    infixes   * + o
    vars      p,p':proc
    axioms    prefix(empty)(t) = false                              (A1)
              prefix(deadlock)(t) = false                           (A2)
              prefix(x*p)(y*tL) = eq(x,y) and forall(prefix(p))(tL)     (A3)
              prefix(p+p')(t) = prefix(p)(t) or prefix(p')(t)        (A4)
              (x*tL) o p = x*fold(add(p))(tL)(empty)
              add(p)(t,p') = (t o p)+p'
              leaf(x) = x*empty
```

BPA's alternative composition of processes ($+$) became a constructor, while sequential composition ($\circ$) turned out to be a defined function whose defining axioms imply that BPA's constructor equations become behavioural equivalences. As the concatenation of streams (cf. 10.1), sequential composition makes sense only if its first argument is a *finite* process, represented here as a finite tree. $\circ$ transforms such a tree into the corresponding process. Note that $empty + p \equiv p$ is a coinductive theorem of *PROCESS*. Hence *empty* does not coincide with the process ϵ of process algebra (cf. [BW 90], 2.2.6).

The statement that each process has a head normal form (cf. [BW 90], Sect. 2.4) follows directly from the above axioms for $\circ$. In terms of a graph notation, a *proc*-term represents a node-labelled graph instead of a usually edge-labelled process graph. If the latter has n root outarcs with labels $a_1, \ldots, a_n$, it is given here by n *proc*-terms with root labels $a_1, \ldots, a_n$, respectively.

Let $A = Ini(TREE)$. Analogously to the *set*-head normal forms of 10.2, the *proc*-head normal forms must be chosen as variables because processes are also permutative. Again the crucial requirement of the coinduction principle are inductive theorems satisfying Condition 4.6(3). For the constructors *empty*, *deadlock*, $*$ and $+$, we take the axioms defining these operations. The axiom for $\circ$, however, does not fulfil 4.6(3) for $F = \circ$. But the following two clauses do so, provided they are inductive theorems of $BSP = BSP(A, PROCESS)$:

$$prefix(t \circ p)(u) \equiv true \;\Leftarrow\; (u \leq t) \textit{ or } (conc(t, v, u) \textit{ and } prefix(p)(v)) \equiv true \qquad (c_1)$$
$$prefix(t \circ p)(u) \equiv false \;\Leftarrow\; \forall v : \; (u \leq t) \textit{ or } (conc(t, v, u) \textit{ and } prefix(p)(v)) \equiv false \qquad (c_2)$$

We show c_1. c_2 can be proved analogously. Since t and u satisfy the premise of c_1 only if the roots of t and u have the same outdegree, say n, c_1 follows from $c = e \Leftarrow h$ where

$$\begin{array}{rcl} e &=& prefix((x * [t_1,\dots,t_n]) \circ p)(y * [u_1,\dots,u_n]) \equiv true, \\ h &=& (y * [u_1,\dots,u_n] \le x * [t_1,\dots,t_n])\ or\ (conc(x * [t_1,\dots,t_n], v, y * [u_1,\dots,u_n])\ and\ prefix(p)(v)) \equiv true. \end{array}$$

Let us prove c by convergence. For applying Thm. 9.19 we mainly have to show that the critical test clause for c (cf. 9.16) is an inductive theorem of BSP and each critical clause of c on *PROCESS*-axioms is subreductively valid w.r.t. (*PROCESS*,*C*) (cf. 8.5). There is one such clause:

$$true \equiv prefix(x * fold(add(p))([t_1,\dots,t_n])(empty))(y * [u_1,\dots,u_n]) \ \Leftarrow\ h. \tag{cc}$$

Hence $crit(c) = (h \Leftarrow h)$, and cc must be sub-e-reductively valid. We may apply Thm. 8.11 because *PROCESS* is functional and strongly terminating. It is left to the reader to identify the set D of e-bounded theorems of BSP used in the following goal reduction of the conclusion of cc into $\emptyset$. The premise of some of these theorems is equal to h. According to Thm. 8.11, only the conclusion is applied in these cases.

$$\begin{array}{ll}
 & true \equiv prefix(x * fold(add(p))([t_1,\dots,t_n])(empty))(y * [u_1,\dots,u_n]) \\
\vdash_{PROCESS} & true \equiv eq(x,y)\ and\ forall(prefix(fold(add(p))([t_1,\dots,t_n])(empty))([u_1,\dots,u_n]) \\
\vdash_D & true \equiv eq(x,y)\ and\ forall(prefix(t_1 \circ p + \dots + t_n \circ p +\ empty))([u_1,\dots,u_n]) \\
\vdash_{PROCESS} & true \equiv eq(x,y)\ and\ forall(prefix(t_1 \circ p + \dots + t_n \circ p))([u_1,\dots,u_n]) \\
\vdash_D & true \equiv eq(x,y)\ and\ prefix(t_1 \circ p + \dots + t_n \circ p)(u_1)\ and \dots \\
 & \qquad prefix(t_1 \circ p + \dots + t_n \circ p)(u_n) \\
\vdash_{PROCESS} & true \equiv eq(x,y)\ and\ (prefix(t_1 \circ p)(u_1)\ or \dots or\ prefix(t_n \circ p)(u_1))\ and \dots \\
 & \qquad (prefix(t_1 \circ p)(u_1)\ or \dots or\ prefix(t_n \circ p)(u_n)) \\
\vdash_c & true \equiv eq(x,y)\ and\ (true\ or \dots or\ prefix(t_n \circ p)(u_1))\ and \dots \\
 & \qquad (prefix(t_1 \circ p)(u_1)\ or \dots or\ true) \\
 & \wedge\ (u_1 \le t_1)\ or\ (conc(t_1, v, u_1)\ and\ prefix(p)(v)) \equiv true\ \wedge\ \dots \\
 & \wedge\ (u_n \le t_n)\ or\ (conc(t_n, v, u_n)\ and\ prefix(p)(v)) \equiv true \\
\vdash_D & true \equiv eq(x,y)\ \wedge\ (u_1 \le t_1)\ or\ (conc(t_1, v, u_1)\ and\ prefix(p)(v)) \equiv true\ \wedge\ \dots \\
 & \wedge\ (u_n \le t_n)\ or\ (conc(t_n, v, u_n)\ and\ prefix(p)(v)) \equiv true \\
\vdash_D & true \equiv eq(x,y)\ \wedge\ true \equiv true\ \wedge\ \dots\ \wedge\ true \equiv true \\
 & \wedge\ ([u_1,\dots,u_n] \le [t_1,\dots,t_n])\ or\ (conc([t_1,\dots,t_n], v, [u_1,\dots,u_n])\ and\ prefix(p)(v)) \equiv true \\
\vdash_D & true \equiv true \\
 & \wedge\ ([u_1,\dots,u_n] \le [t_1,\dots,t_n])\ or\ (conc([t_1,\dots,t_n], v, [u_1,\dots,u_n])\ and\ prefix(p)(v)) \equiv true \\
\vdash_D & true \equiv true \\
\vdash_D & \emptyset
\end{array}$$

All constructor equations of BPA are coinductive theorems of *PROCESS*. Associativity and right distributivity of $\circ$ over $+$ follows from the axioms for $\circ$. These equations as well as the idempotency, commutativity and associativity of $+$ can be derived using proof rules of Section 7. The behavioural equivalence of *proc*-terms agrees with **strong bisimilarity**, the natural equality relation for processes running synchronously (cf. [Mil 83]; [BW 90], Def. 2.7.6). For instance, the process $x * (leaf(y) + leaf(y))$ strongly bisimulates the process $(x * leaf(y)) + (x * leaf(y))$, while $x * (leaf(y) + leaf(z))$ does not bisimulate $(x * leaf(y)) + (x * leaf(z))$. Consequently,

$$x * (leaf(y) + leaf(y)) \ \equiv\ (x * leaf(y)) + (x * leaf(y))$$

is a coinductive theorem of *PROCESS*, while

$$x * (leaf(y) + leaf(z)) \ \equiv\ (x * leaf(y)) + (x * leaf(z))$$

is not valid. The latter equivalence would be valid as well if $prefix$ would relate a process to its path prefixes instead of its tree prefixes.

BPA is sometimes specified as a free-constructor type by introducing a transition relation of the form $\xrightarrow{x}$: $proc \times entry \times proc \rightarrow bool$ (cf. e.g. the type CCSO of [AW 89]). Neglecting the fact that $\xrightarrow{x}$ has two *proc*-parameters and thus cannot be regarded as an action, CCSO can be turned into an action type such that strong bisimilarity agrees with the interpretation of a suitable liveness predicate in the terminal CCSO-model (cf. [AW 89], Sect. 3). Although this **relational** view of BPA has induced several generalizations of process algebra (cf. [CR 91], [Mes 93]), we do not adopt it here because it handles process equality in a rather implicit way. Instead, we propagate the more **functional** flavor of *PROCESS* and its direct realization of process equivalence as strong bisimilarity (see above). The relationship between $\xrightarrow{x}$ and the action $prefix$ of *PROCESS* reads formally as follows:

$$prefix(p)(x * nil) \equiv true \quad \Leftrightarrow \quad \exists p' : p \xrightarrow{x} p'.$$

So far *PROCESS* specifies only finite processes, which are given by ground terms over *empty*, $*$ and $+$. Analogously to infinite streams, infinite processes are defined in extensions of *PROCESS*. They are defined by **guarded recursion** (cf. [BW 90] and [EPBRDG 90], Ex. 3.11). A stream definition by guarded recursion is an axiom of the form $s \equiv x\&s'$ (cf. 10.3; x is the guard). Analogously, a process definition by guarded recursion is an axiom of the form

$$p \;\equiv\; x_1 * p_1 + \ldots + x_n * p_n \tag{1}$$

($x_1, \ldots, x_n$ are the guards.) All recursively defined operations $F : s_1 \ldots s_n \rightarrow proc$ with $s_i \neq proc$ for all $1 \leq i \leq n$ become additional *proc*-constructors and the set of *proc*-head normal forms becomes the set of ground normal forms, i.e. ground terms built up of arbitrary constructors. For example, the two one-bit buffers of [BW 90], Ex. 4.6.2, is specified as follows:

```
BUFFERS
     base      PROCESS
     sorts     bit
     consts    0,1:bit
               r1,s2,r2,s3:bit->entry
               B12,B23:proc
     axioms    B12 = r1(0)*s2(0)*B12 + r1(1)*s2(1)*B12
               B23 = r2(0)*s3(0)*B23 + r2(1)*s3(1)*B23
```

BUFFERS is not normal form complete unless B_{12} and B_{23} are specified in terms of the action $prefix$. Hence (1) should be regarded as a short notation for

$$prefix(p)(t) \;\equiv\; prefix(x_1 * p_1 + \ldots + x_n * p_n)(t)$$

as a stream axiom $s \equiv x\&s'$ actually stands for $head(s) \equiv x \wedge tail(s) \equiv s'$ (cf. 6.7). It is easy to see that the restriction to guarded stream or process definitions guarantees the coinduction principle, in particular Condition 4.6(3).

10.4.2 Concurrency and communication

The next step in process algebra is to augment BPA with operations for establishing concurrent processes, i.e. with *free merge* ($||$) and *left merge* ($\lfloor\!\lfloor$; cf. [BK 84]; [BW 90], Chapt. 3). Both operations can be defined in accordance with the coinduction principle for *PROCESS*:

```
FREE_MERGE
     base      PROCESS
```

```
        defs      ||,lmerge:proc*proc->proc
        infixes   ||
        axioms    p||p' = lmerge(p,p')+lmerge(p',p)
                  lmerge(empty,p) = empty
                  lmerge(deadlock,p) = deadlock
                  lmerge(x*p),p') = x*(p||p')
                  lmerge(p+p',p") = lmerge(p,p")+lmerge(p',p")
```

For instance, the following recursive process definition with entries *plus* and *minus* describes the behaviour of a counter whose current number of decrements (calls of *minus*) never exceeds the current number of increments (calls of *plus*) (cf. [BW 90], 3.5.6):

$$counter \;\equiv\; plus * (counter||minus).$$

Then BPA is extended to ACP, the *algebra of communicating processes*, for modelling synchronous communication between concurrent processes. ACP replaces *free merge* by *communication merge* (|; cf. [BK 84]; [BW 90], Chapt. 4) and depends on a commutative and associative partial function *com*, which maps two communicating actions to a single communication action (cf. [BW 90], 4.1.1). Axioms for *com* are part of the actual parameter to be substituted for *ENTRY* such as *C_BUFFERS* (see below).

```
MERGE
     base      PROCESS
     defs      com:entry*entry->entry
               ||,lmerge,|:proc*proc->proc
     infixes   ||  |
     axioms    p||p' = lmerge(p,p')+lmerge(p',p)+(p|p')          (A)
               lmerge(empty,p) = empty
               lmerge(deadlock,p) = deadlock
               lmerge(x*p,p') = x*(p||p')
               lmerge(p+p',p") = lmerge(p,p")+lmerge(p',p")
               empty|p = empty
               p|empty = empty
               deadlock|p = deadlock
               p|deadlock = deadlock
               (x*p)|(y*p') = com(x,y)*(p||p')  <==  Def(com(x,y))
               (x*p)|(y*p') = deadlock  <==  com(x,y) = bottom
               (p+p')|p" = (p|p")+(p'|p")
               p|(p'+p") = (p|p')+(p|p")
```

All subprocesses of $p||p'$, which communicate with each other, are deleted from the interleavings $lmerge(p,p')$ and $lmerge(p',p)$ of $p||p'$ by applying the following *encapsulation operator enc* to $p||p'$. The actual communication paths $com(x,y)$ are built up in the third summand $p|p'$ of $p||p'$ (cf. Axiom A).

```
ENCAPS
     base      MERGE
     defs      enc:proc->proc
     axioms    enc(empty) = empty
               enc(deadlock) = deadlock
               enc(x*p) = deadlock  <==  Def(com(x,y))
               enc(x*p) = x*enc(p)  <==  /\{y}(com(x,y) = bottom)
               enc(p+p') = enc(p)+enc(p')
```

enc realizes ACP's encapsulation operator δ_H where H is the set of all x such that for some y, $com(x,y)$ is defined (cf. [BW 90], 3.6.4 and 4.2.1).

For example, the buffers B_{12} and B_{23} (cf. *BUFFERS*) may communicate over a channel from B_{12}'s sender s_2 to B_{23}'s receiver r_2 such that the coupled process $B_{12}||B_{23}$ is based on the following definition of *com*.

```
C_BUFFERS
     base      BUFFERS  ENCAPS
     consts    c2:bit->entry
     defs      channel:entry*entry*bit->bool
               C13:proc
     axioms    channel(x,y,b) = (eq(x,s2(b)) and eq(y,r2(b)) or (eq(y,s2(b)) and eq(x,r2(b))
               com(x,y) = c2(b)  <==  channel(x,y,b) = true
               com(x,y) = bottom  <==  channel(x,y,0) = false /\ channel(x,y,1) = false
               C13 = enc(B12||B23)
```

[BW 90], 4.6.2, presents a proof that C_{13} is behaviourally equivalent to the process C defined by the following recursive specification:

```
C = r1(0)*c2(0)*G(0) + r1(1)*c2(1)*G(1)
G(b) = s3(b)*C + H(b,0) + H(b,1)
H(b,b') = r1(b')*s3(b)*c2(b')*G(b')
```

The question arises whether the rules of Section 7 permit more straightforward proofs of process equivalences such as $C_{13} \sim C$ than process algebra itself where the right choice among many axioms, lemmas and proof principles seems to be a difficult task.

10.4.3 Abstraction and refinement

ACP$^\tau$ (cf. [BK 85]; [BW 90], p. 123), the variant of ACP that involves *abstraction by silent τ-steps* can be specified in a similar, though more complicated way. The *τ-law*

$$x * ((\tau * (p + p')) + p') = x * (p + p')$$

of ACP$^\tau$ is in fact a constructor equation, which can be derived as a coinductive theorem of the following modification *ABS_PROC* of *PROCESS*. We identify τ with $\perp$.

```
ABS_PROC
     base      PROCESS without Axioms A1 to A4 (cf. 10.5.1)
     defs      prefix':nat*proc->tree->bool
               blow:list(tree)->list(list(tree))
               add_tau:list(tree)->list(tree)->list(tree)
     vars      i:nat  x,y:entry  p:proc  t:tree  tL,tL':list(tree)  L:list
     axioms    prefix(p)(t) = true  <==  prefix'(i,p)(t) = true
               prefix(p)(t) = false  <==  /\{i}(prefix'(i,p)(t) = false)
               prefix'(i,empty)(t) = false
               prefix'(i,deadlock)(t) = false
               prefix'(0,x*p)(y*tL) = eq(x,y) and forall(prefix'(0,p))(tL)
               prefix'(i+1,x*p)(y*tL) = eq(x,y) and exists(forall(prefix'(i,p)))(blow(tL))
               prefix'(i,p+p')(t) = prefix'(i,p)(t) or prefix'(i,p')(t)
               blow(tL) = tL&map(add_tau(tL))(sublists(tL))
               add_tau(tL)(tL') = (bottom*tL)&tL'
ABS_ENCAPS
     base      ABS_PROC
               ENCAPS without Axioms A1 to A4 (cf. 10.5.1)
     defs      abs:proc*list->proc
     axioms    abs(empty,L) = empty
               abs(deadlock,L) = deadlock
               abs(x*p,L) = ite(x in L, bottom*abs(p,L), x*abs(p,L))
               abs(p+p',L) = abs(p,L)+abs(p',L)
```

$prefix'(i,p)(t)$ holds true if some tree t' is a prefix of p such that t' is obtained from t by inserting at most i τ-steps into the same path of t. $prefix'$ is specified similarly to the synonymous Boolean function of *ABS_STREAM*$_3$

(cf. 10.3). τ-steps are deleted implicitly from a process by "blowing up" its prefixes with such steps. Precisely, $blow(tL)$ returns all trees of the form $(\tau * tL)\&tL'$ where tL' is a sublist of tL. $abs(p, L)$ replaces each entry of L occurring in p by a τ-step. The rest of *ABS_ENCAPS* is taken over from *ENCAPS*.

The behavioural equivalence of *proc*-terms induced by *ABS_PROC* agrees with **weak bisimilarity**, the natural equality for processes running asynchronously (cf. [Mil 83]).

Exercise. Show that

$$B_{13} \;\sim\; abs(C_{13}, [c_2(0), c_2(1)])$$

is a coinductive theorem of *C_BUFFERS'* where

```
C_BUFFERS'
    base      ABS_PROC
              C_BUFFERS without ENCAPS
    consts    B13:proc
    axioms    B13 = r1(0)*s3(0)*B13 + r1(1)*s3(1)*B13
```

(cf. [BW 90], Thm. 6.2.9). An equivalence of the form $B \sim abs(C, L)$ tells us that C *refines* or *implements* B: B is equivalent to C if one abstracts from the entries of L occurring in C.

10.5 Temporal logic of actions

In Section 10.3 we have given examples of concurrent systems that are specified as networks of stream-generating and -processing functions based on the action type *STREAM* given in 10.1. Section 10.5 revised process algebra as the action type underlying the recursive specification of processes. The terminal *STREAM*-model is given by streams of data. The terminal *PROCESS*-model is given by sets of infinite tree structures whose node entries denote atomic actions. A further model for concurrent programs stems from temporal logic and consists of streams of program states. Lamport combined temporal logic with the logic of program assertions (cf. [Flo 67], [Hoa 69]) to the *temporal logic of actions* (TLA; cf. [Lam 94]). At first, a concurrent program P is presented as a formula Φ_P, which includes an initial state description $Init$ and a first-order formula $Act_1 \vee \ldots \vee Act_n$ where each Act_i is a set of equations describing the state transition performed by an atomic command of P. As known from assertion logic, Act_i uses primed and unprimed program variables (identifiers) for presenting the effect of P on individual state components. For instance, the following CSP0-program *MUTEX* (cf. Sect. 10.3.3) guarantees mutually exclusive access to some device D:

```
x:=0; y:=0;
parbegin  (A) do y = 0 -> (B) (x:=1; access D; x:=0; skip) od
          (A) do x = 0 -> (B) (y:=1; access D; y:=0; skip) od
parend
```

MUTEX splits into two subprograms running concurrently. Hence we have two program counter variables pc_1 and pc_2. Further program variables of *MUTEX* are x and y. Each of them achieves the value A or B when the control flow passes the label (A) resp. (B) in the program. For all lists L of program variables let $unchanged(L) =_{def} \wedge_{x \in V}(x \equiv x')$. Then a TLA formula describing *MUTEX* reads as

$$\Phi_{MUTEX} \;=\; Init \wedge \Box(Act_1 \vee \ldots \vee Act_4 \vee unchanged[x, y, pc_1, pc_2])$$

where

$$\begin{aligned}
Init &= (x \equiv 0 \wedge y \equiv 0 \wedge pc_1 \equiv A \wedge pc_2 \equiv A),\\
Act_1 &= (pc_1 \equiv A \wedge y \equiv 0 \wedge x' \equiv 1 \wedge pc_1' \equiv B \wedge unchanged[y, pc_2]),\\
Act_2 &= (pc_1 \equiv B \wedge x' \equiv 0 \wedge pc_1' \equiv A \wedge unchanged[y, pc_2]),\\
Act_3 &= (pc_2 \equiv A \wedge x \equiv 0 \wedge y' \equiv 1 \wedge pc_2' \equiv B \wedge unchanged[x, pc_1]),\\
Act_4 &= (pc_2 \equiv B \wedge y' \equiv 0 \wedge pc_2' \equiv A \wedge unchanged[x, pc_1]).
\end{aligned}$$

As in assertion logic, each program variable x has a primed counterpart x', which takes the new value of x after an assignment in the program has been executed. As in temporal logic, TLA formulas are interpreted in the domain of state streams[38] and thus may contain the *always* operator $\Box$, the *sometimes* operator $\Diamond$ and other operators like *enabled*, which can be interpreted in that domain. This allows us to express not only concurrency and synchronization (via program counter variables), but also liveness conditions such as *weak fairness* and *strong fairness* (cf. [Lam 94]). A requirement to the program P is given by a further TLA formula, say F. P satisfies F if $\Phi_P \Rightarrow F$ is valid in the model of state streams. If F starts with the *always* operator, Lamport calls F an *invariance property.* In particular, if F has the form $\Box(G \Rightarrow \Diamond H)$, then F is called an *eventuality property.*

10.5.1 Simple TLA

The states are functions from a finite domain of identifiers to values. Hence a specification of states can be given by (a constructor extension of) an instance of *MAP* (cf. 10.2).[39]

```
STATE
     base      MAP[id/index,val/entry,state/map]
     defs      eq:state*state->bool
     vars      st,st':state
     axioms    eq(st,st) = true
               eq(st,st') = false  <==  st =/= st'
```

$Ini(STATE) \not\models true \equiv false$ is guaranteed because, by Thm. 5.18, $Fin(MAP)$ has inequality predicates. In particular, $\not\equiv_{state}$ is indeed the negation of the equality predicate for *state*.

The first part of TLA, called Simple TLA (cf. [Lam 94]), can then be formulated as an extension of *STREAM* and *STATE*:

```
SIMPLE_TLA
     base      STREAM (cf. 10.1)  STATE
     sorts     exp  form
     acts      eval:exp*state*state->nat
               evalbool:boolexp*state*state->bool
     consts    var:id->exp
               prime:id->exp
               Nat:nat->exp
               plus:exp*exp->exp
               Eq:exp*exp->boolexp
               ...
               True,False:boolexp
               Not:boolexp->boolexp
               And,Or:boolexp*boolexp->boolexp
               enabled:boolexp->boolexp
```

[38]called behaviours in [Lam 94]

[39]Instances of specifications are defined via *signature morphisms*, which rename symbols of a signature such that the renaming of functions and predicates is compatible with the renaming of sorts (see the basic literature on algebraic specification).

```
                disabled:boolexp*list(exp)->boolexp
                changed,unchanged:list(exp)->boolexp
                mkform:boolexp->form
                And,Or:form*form->form
                box,dia:form->form
                live:boolexp->form
                WF,SF:boolexp*list(exp)->form
      defs      prime:exp->exp
      preds     Eval:form*stream(state)
      infixes   And  Or
      vars      x,y:id  v:val  e,e':exp  A,B:boolexp  F,G:form  st,st',st1:state  sts:stream(state)
                eL:list(exp)
      axioms    eval(var(x),st,st') = n  <==  get(st,x) = Nat(n)
                eval(prime(x),st,st') = n  <==  get(st',x) = Nat(n)
                eval(Nat(n),st,st') = n
                eval(plus(e,e'),st,st') = eval(e,st,st')+eval(e',st,st')
                evalbool(Eq(e,e'),st,st') = eq(eval(e,st,st'),eval(e',st,st'))
                ...
                evalbool(Not(A),st,st') = not(evalbool(A,st,st'))
                evalbool(A And B,st,st') = evalbool(A,st,st') and evalbool(B,st,st')
                evalbool(A Or B,st,st') = evalbool(A,st,st') or evalbool(B,st,st')
                evalbool(enabled(A),st,st') = true  <==  evalbool(A,st,st1) = true
                evalbool(enabled(A),st,st') = false  <==  /\{st1}(evalbool(A,st,st1) = false)
                evalbool(disabled(A,eL),st,st') = true
                  <==  /\{st1}(evalbool(A And changed(eL),st,st1) = false)
                evalbool(disabled(A,eL),st,st') = false
                  <==  evalbool(A And changed(eL),st,st1) = true
                changed(nil) = False
                changed(e&eL) = Not(Eq(e,e')) Or changed(eL)
                unchanged(eL) = Not(changed(eL))
                Eval(mkform(A),st&st'&sts)  ==>  evalbool(st,st',A) = true
                Eval(F And G,sts)  ==>  Eval(F,sts) /\ Eval(G,sts)
                Eval(F Or G,sts)  ==>  Eval(F,sts) \/ Eval(G,sts)
                Eval(box(F),s&sts)  ==>  Eval(F,s&sts) /\ Eval(box(F),sts)
                Eval(dia(F),s&sts)  ==>  Eval(F,s&sts) \/ Eval(dia(F),sts)
                Eval(live1(A),sts)  ==>  Eval(box(dia(mkform(A))),sts)
                Eval(live2(A),sts)  ==>  Eval(dia(box(mkform(A))),sts)
                Eval(WF(A,eL),sts)
                  ==>  Eval(live1(A And changed(eL)),sts) \/ Eval(live1(disabled(A,eL)),sts)
                Eval(SF(A,eL),sts)
                  ==>  Eval(live1(A And changed(eL)),sts) \/ Eval(live2(disabled(A,eL)),sts)
                prime(var(x)) = prime(x)
                prime(plus(t,u)) = plus(prime(t),prime(u))
                ...
```

For avoiding the specification of a substitution operator we do not distinguish between TLA actions and TLA predicates. Terms of sort *exp* are the *state functions* of TLA. The liveness predicate *Eval* interprets TLA formulas (terms of sort $form$) w.r.t. state streams. WF and SF are Lamport's weak resp. strong fairness conditions, which often occur in TLA formulas Φ_P describing a program P (see above).

10.5.2 Abstraction and refinement

The description of programs by TLA formulas admits a simple notion of refinement. A program P is said to *implement* a program Q if Φ_P implies Φ_Q. For two reasons this notion is too restrictive. At first, Lamport points out that Q may allow "stuttering" steps, which reflect actions that change program variables of P, but not of Q. Stuttering steps are the equivalent of τ-steps in process algebras, which also entered the theory when it came to a reasonable notion of refinement (cf. 10.5.3). As we had to adapt the process defining actions for incorporating τ-steps we must change the stream defining actions for taking into account stuttering steps. In process algebra, the goal was to obtain a behavioural equivalence that satisfies the *τ-law*

$$(\tau * (p + p')) + p' = p + p'.$$

In TLA, we need a behavioural equivalence that identifies successive identical states:

$$e\&e\&es = e\&es.$$

This cannot be achieved with the actions *head* and *tail* of *STREAM*. Instead, we proceed analogously to the construction of *ABS_STREAM* (cf. 10.3). There we abstracted from all occurrences of ⊥ in a stream. Here we abstract from all successive duplicates of stream elements:

```
ABS_STREAM4
     base      LIST
     sorts     stream(entry)
     acts      head:stream(entry)->entry
               tail:stream(entry)->stream(entry)
     opns      head':nat*stream(entry)->entry
               bottom:entry'
               &:entry*stream(entry)->stream(entry)
     vars      s:stream(entry)  x,y:entry  i:nat
     axioms    head(s) = x  <==  head'(i,s) = x /\ eq(x,bottom) = false
               head(s) = bottom  <==  /\{i}(head'(i,s) = bottom)
               head'(0,s) = bottom
               head'(i+1,x&y&s) = ite(eq(x,y),head'(i,y&s),x)
               tail(x&s) = s
```

head′(i, s) is the first element x of s that is different from its successor in s if x is among the first i elements of s. Otherwise *head′*$(i, s) = \perp$. TLA with stuttering steps can then be presented by an extension *ABS_TLA* of *ABS_STREAM*, which agrees with *SIMPLE_TLA* except for the base specification.

Even if we take *ABS_TLA* as the basis for proving that Program P implements Program Q, the validity $\Phi_P \Rightarrow \Phi_Q$ is still too strong as a general requirement for correct refinements. It would assume that Φ_P and Φ_Q use the same program variables, which is not only unrealistic, but also contradicts the idea that Q is a more "abstract" than P. Any refinement has a constructive part where each abstract symbol is defined in terms of "concrete" symbols. In a functional language, abstract constructors are translated into functions using concrete operations. In a procedural language whose programs are represented by TLA formulas, abstract program variables are translated into terms over concrete program variables. Supposed that X and Y are the sets of program variables of P and Q, respectively, the actual implementation of Q by P is given by a substitution $\sigma : Y \to T_{ABS\_TLA,exp}$, and we might say that P *refines* Q *via* σ if $\Phi_P \Rightarrow \Phi_Q\sigma$ holds true (as a coinductive theorem of *ABS_TLA*).

10.6 The SECD machine

If interpreters like the function *eval* of *CSP0_EVAL* (cf. 10.3.3) or *SIMPLE_TLA* (cf. 10.5.1) are presented as actions, then program equivalence becomes behavioural equivalence. The classical interpreter of a functional programming language based on the λ-calculus is the SECD[40] machine, introduced in [Lan 64] and decomposed by [Rea 89] into a compiler and an interpreter of its target language. The following specification of both parts also includes the interpretation of the μ- or *LABEL*-operator for writing recursive programs.

```
SECD_COMP
     base      LIST (cf. 10.1)
```

[40] S = stack, E = environment (set of states), C = control, D = dump.

```
     sorts     exp  com
     sort vars id  fun
     consts    Nat:nat->exp                (* lambda-expressions = source language *)
               Bool:bool->exp
               Fun:fun->exp
               var:id->exp
               Abs,Rec:id*exp->exp
               App:exp*exp->exp
               Tup:list(exp)->exp
               Ite:exp*exp*exp->exp
               plus,minus,times,Eq:fun
               LDN:nat->com                (* target language *)
               LDB:bool->com
               LDV:id->com
               LDS:fun->com
               LDF,LAB:id*list(com)->com
               APP,END:com
               TUP:nat->com
               SEL:com*com->com
     defs      compile:exp->list(com)
     vars      n:nat  b:bool  f:fun  x:id  c,e,e':exp  eL:list(exp)
     axioms    compile(Nat(n)) = [LDN(n)]
               compile(Bool(b)) = [LDB(b)]
               compile(Fun(f)) = [LDS(f)]
               compile(var(x)) = [LDV(x)]
               compile(Abs(x,e)) = [LDF(x,compile(e))]
               compile(Rec(x,e)) = [LAB(x,compile(e))]
               compile(App(e,e')) = compile(e')@compile(e)@[APP]
               compile(Tup(eL)) = map(compile)(eL)@[TUP(length(eL))]
               compile(Ite(c,e,e')) = compile(c)@[SEL(compile(e),compile(e'))]

SECD_EVAL
     base      SECD_COMP  STATE (cf. 10.5.1)
               secd = list(val)*state*list(com)*dump
               dump = list(list(val)*state*list(com))
     consts    Nat:nat->val
               Bool:bool->val
               Fun:fun->val
               Tup:list(val)->val
               Closure:state*id*list(com)->val
               Cycle:state*id*id*list(com)->val
     defs      eval:secd->val
               Apply:fun*val->val
               mktup:nat*list(val)*list(val)->val
     vars      x,y:id  v:val  S,S':list(val)  E,E':state  C,C',C":list(com)  D:dump
               n:nat  b:bool  f:fun
     axioms    eval(S,E,LDN(n)&C,D) = eval(Nat(n)&S,E,C,D)
               eval(S,E,LDB(b)&C,D) = eval(Bool(b)&S,E,C,D)
               eval(S,E,LDS(f)&C,D) = eval(Fun(f)&S,E,C,D)
               eval(S,E,LDV(x)&C,D) = eval(get(E,x)&S,E,C,D)
               eval(S,E,LDF(x,C)&C',D) = eval(Closure(E,x,C)&S,E,C',D)
     (1)       eval(S,E,LAB(x,[LDF(y,C)])&C',D) = eval(Cycle(E,x,y,C)&S,E,C',D)
     (2)       eval(Cycle(E,x,y,C)&S,E',C',D) = eval(Closure((x,Cycle(E,x,y,C))&E,y,C)&S,E',C',D)
               eval(Fun(f)&v&S,E,APP&C,D) = eval(Apply(f,v)&S,E,C,D)
               eval(Closure(E,x,C)&v&S,E',APP&C',D) = eval(nil,(x,v)&E,C@[END],(S,E',C')&D)
               eval(S,E,TUP(n)&C,D) = eval(mktup(n,nil,S),E,C,D)
               eval(Bool(true)&S,E,SEL(C,C')&C",D) = eval(S,E,C@C",D)
               eval(Bool(false)&S,E,SEL(C,C')&C",D) = eval(S,E,C'@C",D)
               eval(v&S,E,[END],(S',E',C)&D) = eval(v&S',E',C,D)
               eval(v&S,E,[END],nil) = v
               Apply(plus,tup(Nat(m)&[Nat(n)])) = Nat(m+n)
               Apply(minus,tup(Nat(m)&[Nat(n)])) = Nat(m-n)
```

```
Apply(times,tup(Nat(m)&[Nat(n)])) = Nat(m*n)
Apply(Eq,tup(Nat(m)&[Nat(n)])) = Bool(eq(m,n))
mktup(0,vL,S) = tup(vL)&S
mktup(n+1,vL,v&S) = mktup(n,v&vL,S)
```

The reader is invited to experiment with this specification. For instance, the factorial function reads as follows if it is translated into an *SECD_COMP*-term of sort *exp*:

```
Rec(fact,Abs(N,Ite(App(Fun(Eq),Tup(var(N)&[Nat(0)])),
                   Nat(1),
                   App(Fun(times),Tup(var(N)&[App(var(fact),App(Fun(minus),Tup(var(N)&[Nat(1)])))])))))
```

The *compile* function transforms this term into the following target program, given by a term t of sort $list(com)$:

```
[LAB(fact,[LDF(N,LDV(N)&LDN(0)&TUP(2)&LDS(Eq)&APP&
                SEL([LDN(1)],LDV(N)&LDV(N)&LDN(1)&TUP(2)&LDS(minus)&APP&LDV(fact)&APP&
                             TUP(2)&LDS(times)&[APP]))])]
```

Expander (cf. Sect. 7) has applied the interpreter *eval* to the initial state

$$(nil, nil, LDN(3)\&t\&APP\&[END], nil)$$

and obtained the result $Nat(6)$ after 127 rewriting steps (cf. 6.4).

Axioms (1) and (2) of *SECD_EVAL* are particularly interesting. They demonstrate how circular structures can be expressed in a purely applicative way. (1) and (2) stem from a single axiom where the subterm $Cycle(E, x, y, C)$ is a pointer to the closure on the right-hand side of Axiom (2):

$$eval(S, E, LAB(x, [LDF(y, C)])\&C', D) \;=\; eval(Closure((x, \bullet)\&E, y, C)\&S, E, C', D)$$

Like *CSP0_EVAL* (cf. 10.3.3), *SECD_EVAL* is not normal form complete unless *eval* is totalized along the lines of the remark following Def. 4.2. For instance, the lefthand side of the above equation reduces to a normal form only if, intuitively, the function $\mu x.\lambda y.C$ terminates when called in the state (S, E, C', D).

10.7 Object-oriented modules

Many action extensions can be turned directly into modules in the sense of the both functional-logic and object-oriented specification language FOOPS (cf. [GM 87]) or even in the sense of the language Maude (cf. [Mes 93]) whose execution model realizes associative-commutative rewriting. In general, any specification or implementation language that refers to a state chart model (cf. [Win 90]) and thus favors the "automaton metaphor" (cf. 1.1) actually takes swinging specifications and equips them with a modular structure. Instead of comparing the syntax and semantics of, say, FOOPS with the syntax and semantics of swinging specifications formally let us just turn the bank account example of [GM 87] and [Mes 93] into an action extension $ACCOUNT$ of a constructor extension *ACCOUNT_BASE* of *LIST* (cf. 10.1):

```
ACCOUNT_BASE
     base      LIST
     sorts     transaction
     consts    plus,minus,overdraw:nat->transaction
               from,to:entry*nat->transaction
     defs      leq:nat*nat->bool
```

```
      axioms    leq(m,n) = true  <==  m <= n
                leq(m,n) = false  <==  m > n

  ACCOUNT
      base      ACCOUNT_BASE
      sorts     account  account2
      acts      name:account->entry
                bal:account->nat
                hist:account->list(transaction)
                fst,snd:account2->account
      consts    new:entry->account
                credit,debit:account*nat->account
                transfer:account*nat*account->account2
      vars      a,b:account  h:list(transaction)  k:nat
      axioms    name(new(x)) = x
                bal(new(x)) = 0
                hist(new(x)) = nil
                name(credit(a,n))  = name(a)
                bal(credit(a,n)) = bal(a)+n
                hist(credit(a,n)) = plus(n)&hist(a)
                name(debit(a,n)) = name(a)
                bal(debit(a,n)) = ite(leq(m,n),m-n,m)  <==  bal(a) = m
                hist(debit(a,n)) = ite(leq(m,n),minus(n)&h,overdraw(n)&h)
                   <==  bal(a) = m /\ hist(a) = h
                name(fst(transfer(a,n,b))) = name(a)
                name(snd(transfer(a,n,b))) = name(b)
                bal(fst(transfer(a,n,b))) = ite(leq(m,n),m-n,m)  <==  bal(a) = m
                bal(snd(transfer(a,n,b))) = ite(leq(m,n),k+n,k)
                   <==  bal(a) = m /\ bal(b) = k
                hist(fst(transfer(a,n,b))) = ite(leq(m,n), to(b,n)&h, overdraw(n)&h)
                   <==  bal(a) = m /\ hist(a) = h
                hist(snd(transfer(a,n,b))) = ite(leq(m,n), from(a,n)&h, h)
                   <==  bal(a) = m /\ hist(b) = h
```

The sort *account*2 models the "communication channel" that $transfer$ establishes between two account objects. In general, product action sorts like *account*2 are needed for specifying operations that change several objects simultaneously.

10.8 Dynamic logic

Dynamic logic is the instance of modal logic where the underlying actions (cf. 1.5) are program fragments of an imperative language that are interpreted as binary relations on *State* (cf. [Pra 76], [Wan 82]). As in *SIMPLE_TLA* (cf. 10.5.1) and *SECD_EVAL* (cf. 10.6), the states are functions from a finite domain of identifiers to values. Hence we may use the specification *STATE* for presenting states (cf. 10.5.1). If the imperative language admits block-structured programs with local variables and nested procedure declarations, then *STATE* must be augmented accordingly, e.g. with a symboltable specification as given in [GHM 76] or with a more advanced interpreter specification as presented in [Pad 91a].

Both the cpo approach and dynamic logic provide interpretations of the logic of program assertions (cf. [Flo 67], [Hoa 69]). The cpo approach interprets imperative programs as continuous functions (cf. [LS 87]). Dynamic logic gives a relational semantics, which does not only free us from cpo theory, but also permits the incorporation of *non-deterministic* language constructs (cf. [AO 91], [Best 95]). The inference rules of assertion logic are the most widely used and quite powerful "tools" for verifying imperative programs. Let us specify an imperative language with procedure declarations such that assertion rules can be derived from the specification's (co)inductive theory. This allows us to handle implementations of functions by procedures or procedures by procedures within the algebraic refinement approach (cf. Def. 9.13) while using traditional methods when proving the correctness of such implementations.

At first sight, specifying programs as relations means presenting them as predicates. But imperative language constructs are program *operators*. Hence we had to equip our specification language with second-order predicates, which would entail a revision of the whole theory. This is avoided by introducing an action sort *com* ("command") that corresponds to the sort *rel* of *RELATION* (cf. 10.2). Accordingly, an action *trans* replaces the action *contains* and the regular operators on state relations imperative-program logics are usually based upon replace the constructors *empty* and *insert* of *RELATION* (cf. [KT 90]).

```
COMMAND
     base      STATE
     sorts     com  exp  boolexp
     acts      trans:com*state*state->bool
               trans':nat*com*state*state->bool
               eval:exp*state->val
               evalbool:boolexp*state->bool
     consts    _:=_:id*exp->com
               _;_:com*com->com
               +:com*com->com
               ?:boolexp->com
               var:id->exp
               True,False:boolexp
               neg:boolexp->boolexp
     defs      star:com->com
               star':nat*com->com
               skip,fail:com
               if _ then _ else _ :boolexp*com*com->com
               while _ do _ :boolexp*com->com
               repeat _ until _ :com*boolexp->com
     infixes   :=  ;
     vars      x:id  e:exp  be:boolexp  c,c':com  st,st',st1:state
     axioms    trans(c,st,st') = true  <==  trans'(i,c,st,st') = true
               trans(c,st,st') = false  <==  /\{i} trans'(i,c,st,st') = false
               trans'(i,(x:=e),st,st') = eq(put(x,eval(st,e),st),st')
               trans'(i,(c;c'),st,st') = trans'(i,c',st1,st')
                  <==  trans'(i,c,st,st1) = true
               trans'(i,(c;c'),st,st') = false
                  <==  /\{st1} trans'(i,c,st,st1) = false
               trans'(i,c+c',st,st') = trans'(i,c,st,st') or trans'(i,c',st,st')
               trans'(i,?be,st,st') = eq(st,st') and evalbool(st,be)
               trans'(0,star(c),st,st') = eq(st,st')
               trans'(i+1,star(c),st,st') = trans'(i,c;star(c),st,st')

               skip = ?True
               fail = ?False
               if be then c else c' = (?be; c) + (?neg(be); c')
               while be do c = star(?be; c); ?neg(be)
               repeat c until be = c; while neg(be) do c

               eval(var(x),st) = get(st,x)
               evalbool(True,st) = true
               evalbool(False,st) = false
               evalbool(neg(be),st) = not(evalbool(be,st))
```

Procedures with input parameters (terms of sort *exp*) and output parameters (terms of sort *id*) are introduced as additional constructors.[41] As a simple example, suppose we want to implement the factorial function $fact : nat \rightarrow nat$ as a procedure with input parameter x and output parameter z. The implementation can be given as an extension of *COMMAND*:

[41] The sorts *id* and *val* are usually splitted into subsorts for presenting actual program doamins.

```
FACT
     base      COMMAND
     consts    Nat:nat->exp
               minus,times:exp*exp->exp
               Eq:exp*exp->boolexp
               Fact:exp*id->com
     defs      fact:nat->nat
     vars      x,z:id  e,e':exp
     axioms    fact(0) = 1
               fact(n+1) = fact(n)*(n+1)
     (1)       Fact(e,z) = if Eq(e,Nat(0)) then z:=Nat(1)
                                           else (Fact(minus(e,Nat(1)),z); z:=times(var(z),e))
               eval(Nat(n),st) = Nat(n)
               eval(minus(e,e'),st) = m-n  <==  eval(e,st) = Nat(m) /\ eval(e',st) = Nat(n)
               eval(times(e,e'),st) = m*n  <==  eval(e,st) = Nat(m) /\ eval(e',st) = Nat(n)
               evalbool(Eq(e,e'),st) = eq(m,n)  <==  eval(e,st) = Nat(m) /\ eval(e',st) = Nat(n)
```

Axiom (1) is a short notation for the following equations that extend the specification of $trans$:[42]

```
trans'(0,Fact(e,z),st,st') = false
trans'(i+1,Fact(e,z),st,st')
   = trans'(i, if Eq(e,Nat(0)) then z:=Nat(1)
                               else (Fact(minus(e,Nat(1)),z); z:=times(var(z),e)),
            st,st')
```

The correctness of $Fact$ is described by the assertion

$$\{eval(e,st) \equiv n\}\ Fact(e,z)\ \{get(st,z) \equiv fact(n)\},$$

which is captured by the Horn clause:

$$get(st',z) \equiv fact(n) \ \Leftarrow\ eval(e,st) \equiv n \wedge trans(Fact(e,z),st,st') \equiv true.$$

This example suggests a general definition of assertions as particular Horn clauses over extensions of *COMMAND*:

Definition (assertion over EXT) Let EXT be an extension of *COMMAND*, c be a command term (term of sort *com*) and ϕ and ψ be **predicative** goal sets, i.e. goal sets without command terms, but with a unique state variable st. Then the formula

$$\{\phi\}\ c\ \{\psi\}$$

is an **assertion over** EXT and stands for the clause

$$\phi \wedge trans(c,st,st') \equiv true \ \Rightarrow\ \psi[st'/st].\ ❑$$

Classical Hoare-style assertions do not refer to explicit states such as st. If used in pre- or postconditions, an identifier x (term of sort *id*) or an expression e (term of sort *exp* or *boolexp*) actually denotes the value of x in the current state. Hence we write $get(st,x)$ instead of x and $eval(e,st)$ instead of e in the goal sets of an assertion over EXT. If we wanted to "hide" st from ϕ and ψ, we had to present ϕ and ψ as Boolean expressions and interpret them by an evaluation function with state parameters. In Section 10.5.1 we proceeded in this way when specifying Simple TLA. But there formulas had to be interpreted with respect to single states, pairs of states and even streams of states. For dealing with assertion logic, however, "internalizing" first-order formulas by presenting them as Boolean expressions is certainly more complicated than introducing the state variable st.

[42]The attentive reader will notice that the usual evaluation of procedure calls is not yet fully captured by the above axioms. We'll make this precise in Section 10.8.1.

The *COMMAND* axioms for $trans$ guarantee that all ground terms of the form $trans(c, st, st')$ can be reduced to terms without command subterms. This property is preserved by EXT if additional command operators such as procedures are specified in the way illustrated above. Consequently, EXT is **expressive** in the sense of assertion logic, i.e. for all ground command terms c and predicative goal sets ψ there is a predicative goal set ϕ such that $Fin(EXT)$ satisfies

$$\phi \;\Leftrightarrow\; \forall st' : trans(c, st, st') \equiv false \vee \psi[st'/st].$$

ϕ is usually called the **weakest liberal precondition** $wlp(c, \psi)$ of c and ψ (cf., e.g., [LS 87]). Dynamic logic employs the box operator of modal logic (cf. 1.5) for presenting $wlp(c, \psi)$: $[c]\psi$ is interpreted as the set of states satisfying $wlp(c, \psi)$. If EXT is expressive, Hoare's assertion rules for iterative programs are equivalence transformations w.r.t. $Fin(EXT)$:

consequence $$\frac{\{\phi\}\ c\ \{\psi\}}{\exists \phi', \psi' :\ \phi\ \Rightarrow\ \phi',\ \ \{\phi'\}\ c\ \{\psi'\},\ \ \psi'\ \Rightarrow\ \psi} \Updownarrow$$

skip $$\frac{\{\phi\}\ skip\ \{\psi\}}{\phi\ \Rightarrow\ \psi} \Updownarrow$$

fail $$\frac{\{\phi\}\ fail\ \{\psi\}}{\neg\phi} \Updownarrow$$

assignment $$\frac{\{\phi\}\ x := e\ \{\psi\}}{\phi\ \Rightarrow\ \psi[e/var(x), eval(e, st)/get(st, x)]} \Updownarrow$$

sequence $$\frac{\{\phi\}\ c; c'\ \{\psi\}}{\exists \phi' :\ \{\phi\}\ c\ \{\phi'\},\ \ \{\phi'\}\ c'\ \{\psi\}} \Updownarrow$$

conditional $$\frac{\{\phi\}\ if\ be\ then\ c\ else\ c'\ \{\psi\}}{\{\phi \wedge eval(be, st) \equiv true\}\ c\ \{\psi\},\ \ \{\phi \wedge eval(be, st) \equiv false\}\ c'\ \{\psi\}} \Updownarrow$$

while $$\frac{\{\phi\}\ while\ be\ do\ c\ \{\psi\}}{\exists \phi' :\ \phi\ \Rightarrow\ \phi',\ \ \{\phi' \wedge eval(be, st) \equiv true\}\ c\ \{\phi'\},\ \ (\phi' \wedge eval(be, st) \equiv false)\ \Rightarrow\ \psi} \Updownarrow$$

repeat $$\frac{\{\phi\}\ repeat\ c\ until\ be\ \{\psi\}}{\exists \phi' :\ \{\phi\}\ c\ \{\phi'\},\ \ \{\phi' \wedge eval(be, st) \equiv false\}\ c\ \{\phi'\}\ \ (\phi' \wedge eval(be, st) \equiv true)\ \Rightarrow\ \psi} \Updownarrow$$

That these rules are contractions and not only expansions is trivial for *consequence, skip, fail, assignment* and *conditional*. For *sequence*, ϕ' must be chosen as $wlp(c', \psi)$. With the help of the behavioural equivalences

```
while be do c = if be then (c; while(be,c)) else skip
repeat c until be = c; while neg(be) do c
```

while and *repeat* can be derived from other rules. For the contraction property of *while* and *repeat*, ϕ' must be chosen as $wlp(while\ be\ do\ c, \psi)$ and $wlp(while\ neg(be)\ do\ c, \psi)$, respectively. Since each rule application decreases the command structure of the redex, we can conclude that the rules comprise a complete calculus for assertions about iterative programs (cf. [LS 87], Thm. 8.11).

10.8.1 A complete assertion rule for recursive procedures

For verifying recursive programs like $Fact$ we need further, preferably complete, assertion rules. Rules for procedures can be found in [Apt 81], Sect. 3, [Cou 90], Sect. 8.2, and [Fra 92], Chapt. 6. But they are less general and elegant as those for iterative programs and - if at all - only *relatively* complete, i.e. complete under the assumption that weakest liberal preconditions are presentable as predicative formulas.[43] The general approach propagated in this paper, namely to orient the specification and verification of programs at a fixed model, should also provide complete assertion rules for recursive procedures. For recursive *functional-logic* programs we have developed the fixpoint induction rules of Sections 7 and 10.1.1. Since $Fin(EXT)$ interprets procedures as binary (state) relations, fixpoint induction should also be able to cope with *imperative* recursion. Can our previous induction rules even be used directly? No. Why not?

Given a Horn clause specification SP, we defined in the proof of Lemma 5.14 a monotone function Ψ whose least fixpoint agrees with the initial model of SP (cf. Lemma 5.15). We note by the way that the combination of Lemma 5.15 with the Knaster-Tarski fixpoint theorem (cf. Section 1.5) leads to a further characterization of $Ini(SP)$, namely:

$$Ini(SP) \;=\; inf\{B \in PreIni \mid \Psi(B) \leq B\},$$

i.e. for all logical predicates $P \in SP$,

$$P^{Ini} \;=\; \cap_{B\in PreIni}\{P^B \mid P^{\Psi(B)} \subseteq P^B\}.$$

Now recall the definition of Ψ (cf. Lemma 5.14):

$$P^{\Psi(B)} \;=\; \{b(x) \mid b : X \to B,\; B \models_b gs \text{ where } P(x) \Rightarrow gs = ONLY(P)\}.$$

Since the carriers of B are those of $Ini(SP)$, this can be rewritten as:

$$P^{\Psi(B)} \;= \{[t] \in Ini(SP) \mid B \models gs[t/x] \text{ where } P(x) \Rightarrow gs = ONLY(P)\}.$$

Here we see how Ψ acts as an *operator* on relations. Roughly said, gs^B is mapped to $P^{\Psi(B)}$, or: gs is applied to P. The monotonicity of Ψ does not depend on the syntactical assumption that gs is a goal set. As in modal logic, any first-order formula may be used here as long as it is $\neg$-free (cf. Section 1.5).

To sum up, our fixpoint rules work because Ψ realizes a derived operator on relations and this operator is presented by $\neg$-free first-order formulas. Hence it is built up from union and intersection. However, relations representing imperative programs are built up from the regular operators union (+), composition (;) and iteration (*star*). Moreover, all these relations are binary relations on a set $State$ of states. Hence parameterized procedures should be interpreted as second-order relations or functions into the powerset of $State^2$. We may still employ Kleene's fixpoint theorem for deriving induction rules, but must start out from other monotonic functions than those that account for the fixpoint rules of Sections 7 and 10.1.1.

If we do not want to restrict the recursion schema of procedure definitions to tail-recursion, which is known to be equivalent to iteration, then $State$ can no longer be interpreted simply as a space of functions from identifiers to values. As every compiler builder knows, non-tail-recursive calls require a stack of environments for keeping several values of the same identifier, depending on the nesting of scopes that arise from the execution of such calls. Denotational semantics employs quite complex function spaces in order to formalize the evaluation of recursive procedure calls (cf. [Gor 79], Sect. 8). Final semantics, however, allows us to refer to a simple modification of MAP (cf. 10.2), called $SYMTAB$ ("symboltable"; cf. [GHM 76], Sect. 2.3) that serves the same purpose:

```
SYMTAB
     base      ENTRY
     sorts     symtab
     acts      get:symtab*index->entry
     consts    new:symtab
               enter:symtab->symtab
               put:index*entry*symtab->symtab
```

[43] The low value put on (non-tail) recursion in imperative programming seems to be the reason for the marginal treatment of recursion in assertion logic. At least, a couple of textbooks on program verification do not even mention recursion.

```
defs      leave:symtab->symtab
vars      x:entry  st:symtab  i,j:index
axioms    get(new) = bottom
          get(enter(st)) = get(st)
          get(put(i,x,st),i) = x
          get(put(i,x,st),j) = get(st,j)  <==  eq(i,j) = false
          leave(new) = new
          leave(enter(st)) = st
          leave(put(i,x,st)) = leave(st)
```

We change the state specification accordingly (cf. 10.5.1):

```
STATE
     base      SYMTAB[id/index,val/entry,state/symtab]
     defs      eq:state*state->bool
     vars      st,st':state
     axioms    eq(st,st) = true
               eq(st,st') = false  <==  st =/= st'
```

Now we want to extend *COMMAND* (based on the new state specification) by a procedure declaration such as (1). For this purpose we first add to *COMMAND* a *begin*-command for establishing a local scope and a *return*-command for removing a local scope while saving one of its assignments:

```
BLOCK
     base      COMMAND
     consts    begin _ return _ :com*id->com
     vars      x,z:id  e:exp  c:com  st,st',st1:state
     axioms    trans'(i,begin c return(z),st,st') = eq(st',put(z,get(st1,z),leave(st1)))
                  <==  trans'(i,c,enter(st),st1) = true
               trans'(i,begin c return(z),st,st') = false
                  <==  /\{st1} trans'(i,c,enter(st),st1) = false
```

The action $put(z, get(st1, z), leave(st1))$ removes the local scope of c and transmits the local value of z to the global scope by reassigning it to z. Supposed that the identifier x does not occur in the command term c, the procedure declaration $F(e, z) \equiv c$ is just an abbreviation for the following extension of *BLOCK*:

```
EXT
     base      BLOCK
     defs      F:exp*id->com
     vars      x,z:id  e:exp  c:com  st,st',st1:state
               trans'(0,F(e,z),st,st') = false
               trans'(i+1,F(e,z),st,st')
                  = trans'(i,begin x:=e; c[var(x)/e] return(z),st,st')
```

For instance, Equation (1) stands for

```
trans'(0,Fact(e,z),st,st') = false
trans'(i+1,Fact(e,z),st,st')
   = trans'(i, begin x:=e;
                   if Eq(var(x),Nat(0)) then z:=Nat(1)
                   else (Fact(minus(var(x),Nat(1)),z); z:=times(var(z),var(x)))
                   return(z),
               st,st')
```

For simplicity we assume that F is the only procedure of EXT. Of course, this schema of a procedure specification can be generalized straightforwardly to a set $F_1, \ldots, F_n$ of mutually recursive procedures as well as to procedures with several input or output parameters. But note that procedures are defined applicatively. They cannot be nested or used as expressions, in other words, higher-order procedures cannot be modelled here.

Let $A = Fin(EXT)$ and $Proc = [A_{exp} \times A_{id} \to A_{com}]$. Then

$$A_{state} \cong [A_{id} \to A_{val}]^*, \quad A_{exp} \cong [A_{state} \to A_{val}], \quad A_{com} \cong \wp(A^2_{state}),$$

and c induces the monotonic function

$$\Theta_F : Proc \;\to\; Proc$$

such that for all $g \in Proc$, $(\varepsilon, \delta) \in A_{exp} \times A_{id}$ and $b : X \to A$ with $b(e) = \varepsilon$ and $b(z) = \delta$,

$$\Theta_F(g)(\varepsilon, \delta) \;=_{def}\; b^*(begin\ x := e;\ c_{x/e}[G/F]\ return(z))$$

where $c_{x/e} = c[var(x)/e]$, $G : exp \times id \to com$ is an auxiliary procedure symbol and $G^A =_{def} g$. Since e and z are the only variables of c, $\Theta_F(g)(\varepsilon, \delta)$ is a unique element of A_{com}. Moreover, F^A is a fixpoint of Θ_F: for all $(\varepsilon, \delta) \in A_{exp} \times A_{id}$,

$$\Theta_F(F^A)(\varepsilon, \delta) \;=\; b^*(begin\ x := e;\ c_{x/e}\ return(z)) \;=\; b^*(F(e, z)) \;=\; F^A(\varepsilon, \delta).$$

Let $\Omega \in Proc$ assign the empty set to all $(\varepsilon, \delta) \in A_{exp} \times A_{id}$. For all $f, g \in Proc$,

$$f \le g \iff_{def} \forall (\varepsilon, \delta) \in A_{exp} \times A_{id} : f(\varepsilon, \delta) \subseteq g(\varepsilon, \delta).$$

For all $fs \subseteq Proc$, $sup(fs)(\varepsilon, \delta) =_{def} \cup\{f(\varepsilon, \delta) \mid f \in fs\}$ and $inf(fs)(\varepsilon, \delta) =_{def} \cap\{f(\varepsilon, \delta) \mid f \in fs\}$. The fixpoint theorems of Kleene and Knaster-Tarski tell us that Θ_F has the least fixpoint

$$lfp(\Theta_F) \;=\; sup\{\Theta^i_F(\Omega) \mid i \in I\!N\} \;=\; inf\{f \in Proc \mid \Theta_F(f) \le f\}. \tag{2}$$

Hence fixpoint induction in this setting means employing the equivalence transformation

$$\frac{lfp(\Theta_F) \le f}{\exists g : \Theta_F(g) \le g \le f} \Updownarrow \tag{3}$$

Can we extract from (3) a complete (= simplification) rule for a procedure assertion $\{\phi\}F(e, z)\{\psi\}$? For this purpose we transform $\{\phi\}F(e, z)\{\psi\}$ into the antecedent of (3). We may assume that e and z are the only variables of sort exp resp. id that occur in ϕ or ψ. At first, ϕ and ψ are turned into a function $<\phi, \psi> \in Proc$:

$$<\phi, \psi> (\varepsilon, \delta) \;=_{def}\; \{(b(st), b(st')) \mid b : X \to A,\ b(e) = \varepsilon,\ b(z) = \delta,\ A\ \models_b\ \phi \Rightarrow \psi[st'/st]\}.$$

Suppose that

$$F^A \;=\; lfp(\Theta_F). \tag{4}$$

With $f =_{def} <\phi, \psi>$ the antecedent of (3) is indeed equivalent to the assertion $\{\phi\}F(e, z)\{\psi\}$:

$$\begin{array}{ll}
 & A \;\models\; \{\phi\}\ F(e, z)\ \{\psi\} \\
\Leftrightarrow & A \;\models\; \phi \wedge trans(F(e, z), st, st') \equiv true \;\Rightarrow\; \psi[st'/st] \\
\Leftrightarrow & A \;\models\; trans(F(e, z), st, st') \equiv true \;\Rightarrow\; (\phi \;\Rightarrow\; \psi[st'/st]) \\
\Leftrightarrow & \forall b : X \to A : A\ \models_b\ trans(F(e, z), st, st') \equiv true \ \text{ implies }\ A\ \models_b\ \phi \Rightarrow \psi[st'/st] \\
\Leftrightarrow & \forall b : X \to A : (b(st), b(st')) \in F^A(b(e), b(z)) \ \text{ implies }\ (b(st), b(st')) \in <\phi, \psi> (b(e), b(z)) \\
\Leftrightarrow & \forall b : X \to A : F^A(b(e), b(z)) \;\subseteq <\phi, \psi> (b(e), b(z)) \\
\Leftrightarrow & \forall (\varepsilon, \delta) \in A_{exp} \times A_{id} : F^A(\varepsilon, \delta) \;\subseteq <\phi, \psi> (\varepsilon, \delta) \\
\Leftrightarrow & F^A \;\le <\phi, \psi> \\
\overset{(4)}{\Leftrightarrow} & lfp(\Theta_F) \;\le <\phi, \psi> .
\end{array}$$

It remains to show (4). For all $i \in I\!N$ let $F_i, G_i : exp \times id \to com$ be auxiliary procedure symbols such that for all $(\varepsilon,\delta) \in A_{exp} \times A_{id}$,

$$F_i^A(\varepsilon,\delta) \ =_{def} \ \{(\rho,\rho') \in A^2_{state} \mid trans'^A(i, F^A(\varepsilon,\delta),\rho,\rho') = true\}$$

and $G_i^A =_{def} \Theta_F^i(\Omega)$. Since F^A is a fixpoint of Θ_F, (4) follows from (2) and

$$F_i^A \ \le \ G_i^A. \tag{5}$$

We show (5) by induction on i. By definition, (5) holds true for $i = 0$. Let $i > 1$ and $b : X \to A$ such that $b(e) = \varepsilon$ and $b(z) = \delta$. Then

$$\begin{array}{ll}
 & (\rho,\rho') \in F_i^A(\varepsilon,\delta) \\
\Leftrightarrow & trans'^A(i, F^A(\varepsilon,\delta),\rho,\rho') = true \\
\Leftrightarrow & trans'^A(i, b^*(F(e,z)),\rho,\rho') = true \\
\Leftrightarrow & trans'^A(i, b^*(begin\ x := e;\ c_{x/e};\ return(z)),\rho,\rho') = true \\
\Leftrightarrow & \exists i_1,\dots,i_n < i : trans'^A(i, b^*(begin;\ x := e;\ c_{x/e}[F_{i_1}/F,\dots,F_{i_n}/F]\ return(z)),\rho,\rho') = true \\
 & \text{where each } F_{i_j} \text{ replaces a single occurrence of } F \text{ in } c \\
\overset{ind.hyp.}{\Rightarrow} & \exists i_1,\dots,i_n < i : trans'^A(i, b^*(begin\ x := e;\ c_{x/e}[G_{i_1}/F,\dots,G_{i_n}/F]\ return(z)),\rho,\rho') = true \\
 & \text{where each } G_{i_j} \text{ replaces a single occurrence of } F \text{ in } c \\
\Rightarrow & trans'^A(i, b^*(begin\ x := e;\ c_{x/e}[G_{i-1}/F]\ return(z)),\rho,\rho') = true \\
 & \text{where } G_{i-1} \text{ replaces all occurrences of } F \text{ in } c \\
\Rightarrow & trans^A(b^*(begin\ x := e;\ c_{x/e}[G_{i-1}/F]\ return(z)),\rho,\rho') = true \\
\Leftrightarrow & (\rho,\rho') \in b^*(begin\ x := e;\ c_{x/e}[G_{i-1}/F]\ return(z)) \\
\Leftrightarrow & (\rho,\rho') \in \Theta_F(G^A_{i-1})(\varepsilon,\delta) \\
\Leftrightarrow & (\rho,\rho') \in \Theta_F(\Theta_F^{i-1}(\Omega))(\varepsilon,\delta) \\
\Leftrightarrow & (\rho,\rho') \in \Theta_F^{i}(\Omega)(\varepsilon,\delta).
\end{array}$$

To sum up, we may reformulate Rule (3) as follows:

$$\frac{\{\phi\}\ F(e,z)\ \{\psi\}}{\exists g : \Theta_F(g) \le g \le \ <\phi,\psi>} \Updownarrow \tag{6}$$

How can we bring the succedent of (6) into assertion form? $\Theta_F(g) \le g$ follows from:

$$\forall h : h \le g \ \Rightarrow \ \Theta_F(h) \le g. \tag{7}$$

Suppose that g can be expressed in terms of two predicative goal sets ϕ' and ψ' such that $g =<\phi',\psi'>$. Then (7) is equivalent to:

$$\forall h : h \le \ <\phi',\psi'> \ \Rightarrow \ \Theta_F(h) \le \ <\phi',\psi'>, \tag{8}$$

which, as can be shown easily, is equivalent to:

$$\forall h :\ (A \ \models \ \{\phi'\}\ H(e,z)\ \{\psi'\}) \ \Rightarrow \ (A \ \models \ \{\phi'\}\ begin\ x := e;\ c_{x/e}[H/F]\ return(z)\ \{\psi'\}) \tag{9}$$

where $H^A =_{def} h$. (9) is equivalent to:

$$\forall h :\ A \ \ \models \ \ (\forall X : \{\phi'\}\ H(e,z)\ \{\psi'\}) \ \Rightarrow \ \{\phi'\}\ begin\ x := e;\ c_{x/e}[H/F]\ return(z)\ \{\psi'\}. \tag{10}$$

The second part of the succedent of (6) can also be reduced to an implication.

$$<\phi',\psi'> \ \le \ <\phi,\psi>$$

holds true iff

$$A \models (\phi \wedge \forall X \setminus \{e,z,st,st'\} : (\phi' \Rightarrow \psi'[st'/st])) \Rightarrow \psi[st'/st]. \tag{11}$$

From (10) and (11) we obtain an assertion form of Rule (6):

procedure rule

$$\frac{\{\phi\}\ F(e,z)\ \{\psi\}}{\begin{array}{c}\exists\phi',\psi'\ \forall F:\ (\forall X : \{\phi'\}\ F(e,z)\ \{\psi'\}) \ \Rightarrow\ \{\phi'\}\ begin\ x := e;\ c_{x/e}\ return(z)\ \{\psi'\},\\ (\forall X \setminus \{e,z,st,st'\} :\ \phi' \Rightarrow \psi'[st'/st]) \ \wedge\ \phi \ \Rightarrow\ \psi[st'/st]\end{array}} \Uparrow$$

For reducing the succedent of the procedure rule we must first resolve the block structure:

block rule
$$\frac{\{\phi \wedge \phi'\}\ begin\ c\ return(z)\ \{\psi\}}{\{\phi \wedge \phi'\}\ c\ \{get(st,z) \equiv t\},\quad get(st,z) \equiv t \wedge \phi' \ \Rightarrow\ \psi} \Updownarrow$$

where $\phi' = \bigwedge_{i=1}^{n} get(st,x_i) \equiv t_i$, $x_1,\ldots,x_n \in T_{SIG,id} \setminus \{z\}$ and $t, t_1, \ldots, t_n$ are SIG-terms over $X \setminus \{st\}$

Example. We show the correctness of $Fact$ by deriving the assertion

$$\{eval(e,st) \equiv n\}\ Fact(e,z)\ \{get(st,z) \equiv fact(n)\} \tag{12}$$

where the underlying extension of $BLOCK$ is given by $FACT$. The pre- and postconditions must be generalized to

$$\phi' =_{def} eval(e,st) \equiv n \wedge get(st,x) \equiv k,$$
$$\psi' =_{def} get(st,z) \equiv fact(n) \wedge get(st,x) \equiv k.$$

The procedure rule expands (12) into

```
/\ X {phi'} F(e,z) {psi'}
==>  {phi'} begin x:=e;
            if Eq(var(x),Nat(0)) then z:=Nat(1)
            else (F(minus(var(x),Nat(1)),z); z:=times(var(z),var(x)))
            return(z) {psi'},
/\{k}(phi' ==> psi') /\ eval(e,st) = n  ==>  get(st,x) = k
```

The last clause holds true trivially. The remaining proof employs rules of Section 7 for simplifying the goal sets of assertions as well as Hoare's rules for simplifying the entire assertions. First the block rule yields

```
/\ X {phi'} F(e,z) {psi'}
==>  \/{t} {phi'} x:=e;
                 if Eq(var(x),Nat(0)) then z:=Nat(1)
                 else (F(minus(var(x),Nat(1)),z); z:=times(var(z),var(x))) {get(st,z) = t}
           /\   get(st,z) = t /\ get(st,x) = k  ==>  psi'
```

The sequence rule leads to

```
/\ X {phi'} F(e,z) {psi'}
==>  \/{t,P} {phi'} x:=e {P(st,x,n)}
             /\   {P(st,x,n)} if Eq(var(x),Nat(0)) then z:=Nat(1)
                                else (F(minus(var(x),Nat(1)),z); z:=times(var(z),var(x))) {get(st,z) = t}
             /\   get(st,z) = t /\ get(st,x) = k  ==>  psi'
```

The conditional rule yields

```
    /\ X {phi'} F(e,z) {psi'}
    ==>  \/{t,P} {phi'} x:=e {P(st,x,n)}
                /\   {P(st,x,n) /\ eval(Eq(var(x),Nat(0)),st) = true} z:=Nat(1) {P(st,x,n,z)}
                /\   {P(st,x,n) /\ eval(Eq(var(x),Nat(0)),st) = false}
                        F(minus(var(x),Nat(1)),z); z:=times(var(z),var(x)) {get(st,z) = t}
                /\   get(st,z) = t /\ get(st,x) = k  ==>  psi'
```

Again, the sequence rule leads to

```
    /\ X {phi'} F(e,z) {psi'}
    ==>  \/{t,P,Q} {phi'} x:=e {P(st,x,n)}
                  /\   {P(st,x,n) /\ get(st,x) = 0} z:=Nat(1) {P(st,n,z)}
                  /\   {P(st,x,n) /\ get(st,x) > 0} F(minus(var(x),Nat(1)),z) {Q(st,x,n,z)}
                  /\   {Q(st,x,n,z)} z:=times(var(z),var(x)) {get(st,z) = t}
                  /\   get(st,z) = t /\ get(st,x) = k  ==>  psi'
```

Next we instantiate the existentially quantified variables t, P and Q by tracing the remaining implications and assignments from right to left and prepare the induction step (*) by an application of the consequence rule:

```
<==
    /\ X {phi'} F(e,z) {psi'}
    ==>  \/{P,Q} {phi'} x:=e {P(st,x,n)}
                /\   {P(st,x,n) /\ get(st,x) = 0} z:=Nat(1) {get(st,z) = fact(n)}
                /\   {P(st,x,n) /\ get(st,x) > 0} F(minus(var(x),Nat(1)),z) {Q(st,x,n,z)}
                /\   {Q(st,x,n,z)} z:=times(var(z),var(x)) {get(st,z) = fact(n)}
                /\   get(st,z) = fact(n) /\ get(st,x) = k  ==>  psi'
<==>
    /\ X {phi'} F(e,z) {psi'}
    ==>  \/{P,Q} {phi'} x:=e {P(st,x,n)}
                /\   P(st,x,n) /\ get(st,x) = 0  ==>  eval(Nat(1),st) = fact(n)
                /\   {P(st,x,n) /\ get(st,x) > 0} F(minus(var(x),Nat(1)),z) {Q(st,x,n,z)}
                /\   Q(st,x,n,z)  ==>  eval(times(var(z),var(x)),st) = fact(n)
<==
    /\ X {phi'} F(e,z) {psi'}
    ==>  \/{P} {phi'} x:=e {P(st,x,n)}
              /\   P(st,x,n) /\ get(st,x) = 0  ==>  fact(n) = 1
              /\   {P(st,x,n) /\ get(st,x) > 0} F(minus(var(x),Nat(1)),z) {get(st,z)*get(st,x) = fact(n)}
              /\   get(st,z)*get(st,x) = fact(n)  ==>  get(st,z)*get(st,x) = fact(n)
<==>
    /\ X {phi'} F(e,z) {psi'}
    ==>  \/{P,Q,R} {phi'} x:=e {P(st,x,n)}
                  /\   P(st,x,n) /\ get(st,x) = 0  ==>  fact(n) = 1
                  /\   P(st,x,n) /\ get(st,x) > 0  ==>  Q(st,x,n)
                  /\   {Q(st,x,n)} F(minus(var(x),Nat(1)),z) {R(st,x,n,z)}
                  /\   R(st,x,n,z)  ==>  get(st,z)*get(st,x) = fact(n)
(*)
<==
     \/{P} {phi'} x:=e {P(st,x,n)}
          /\   P(st,x,n) /\ get(st,x) = 0  ==>  fact(n) = 1
          /\   P(st,x,n) /\ get(st,x) > 0  ==>  eval(minus(var(x),Nat(1)),st) = n-1 /\ get(st,x) = n
          /\   get(st,z) = fact(n-1) /\ get(st,x) = n  ==>  get(st,z)*get(st,x) = fact(n)
<==>
     \/{P} {phi'} x:=e {P(st,x,n)}
          /\   P(st,x,n) /\ get(st,x) = 0  ==>  fact(n) = 1
          /\   P(st,x,n) /\ get(st,x) > 0  ==>  get(st,x)-1 = n-1 /\ get(st,x) = n
<==
          {phi'} x:=e {get(st,x) = n}
          /\   get(st,x) = n /\ get(st,x) = 0  ==>  fact(n) = 1
          /\   get(st,x) = n /\ get(st,x) > 0  ==>  get(st,x)-1 = n-1
<==>
```

```
            phi'  ==>  eval(e,st) = n
            /\   get(st,x) = n /\ get(st,x) = 0  ==>  fact(n) = 1
            /\   get(st,x) = n /\ get(st,x) > 0  ==>  get(st,x)-1 = n-1
<==>
            TRUE
```

❑

Finally, we want to figure out the conditions under which the procedure rule is not only an expansion, but also a contraction rule so that antecedent and succedent are equivalent. As the contraction property of (3) follows from defining g as $lfp(\Theta_F)$, we need two predicative goal sets ϕ' and ψ' such that $<\phi',\psi'>= lfp(\Theta_F)$ and the succedent of the procedure rule holds true for this choice of ϕ' and ψ', i.e.

$$\forall F:\ (\forall X:\{\phi'\}\ F(e,z)\ \{\psi'\})\ \Rightarrow\ \{\phi'\}\ begin\ x:=e;\ c_{x/e}\ return(z)\ \{\psi'\} \tag{13}$$

and

$$(\forall X\setminus\{e,z,st,st'\}:\ \phi'\Rightarrow\psi'[st'/st])\ \wedge\ \phi\quad\Rightarrow\quad\psi[st'/st] \tag{14}$$

are valid. With $g =_{def} <\phi',\psi'>= lfp(\Theta_F)$ we follow the above equivalences backwards and conclude that (13) is equivalent to:

$$\forall h: h\le lfp(\Theta_F)\ \Rightarrow\ \Theta_F(h)\le lfp(\Theta_F). \tag{15}$$

But (15) is equivalent to the monotonicity of Θ_F because $\Theta_F(lfp(\Theta_F)) = lfp(\Theta_F)$. Moreover, we have shown that (14) is equivalent to

$$lfp(\Theta_F)\ \le <\phi,\psi>,$$

which in turn can be reduced to $\{\phi\}F(e,z)\{\psi\}$, the antecedent of the procedure rule (see above). Hence it remains to define ϕ' and ψ' such that

$$lfp(\Theta_F)\ = <\phi',\psi'>. \tag{16}$$

As weakest liberal preconditions can be expressed as predicative goal sets (see above), we may also construct a predicative goal set $sp(\phi,c)$ that presents the **strongest postcondition** of a ground command term c and a predicative goal set ϕ, i.e. $Fin(EXT)$ satisfies

$$sp(\phi,c)\ \Leftrightarrow\ \exists st':\phi[st'/st]\wedge trans(c,st',st)\equiv true.$$

Hence a predicate $P_F: exp\times val\times id\times state$ is specified by the Horn clause

$$P_F(e,v,z,st)\ \Leftarrow\ sp(eval(e,st)\equiv v, F(e,z))$$

and interpreted in A as follows:

$$P_F^A\ =\ \{(\varepsilon, eval^A(\varepsilon,\rho),\delta,\rho')\mid(\rho,\rho')\in F^A(\varepsilon,\delta)\}.$$

Then we define:

$$\begin{aligned}\phi' &=_{def}\ eval(st,e)\equiv v\wedge \textstyle\bigwedge_{i=1}^n get(st,x_i)\equiv v_i,\\ \psi' &=_{def}\ P_F(e,v,z,st)\wedge \textstyle\bigwedge_{i=1}^n (get(st,x_i)\equiv v_i\vee x_i\equiv z)\end{aligned}$$

and for all $(\varepsilon,\delta)\in A_{exp}\times A_{id}$,

$$h(\varepsilon,\delta)\ =_{def}\ \{(\rho,\rho')\in A_{state}^2\mid(\varepsilon,eval^A(\varepsilon,\rho),\delta,\rho')\in P_F^A,\ \forall y\in A_{id}\setminus\{\delta\}, get^A(\rho',y)=get^A(\rho,y)\}$$

where $\{x_1,\dots,x_n\} = T_{SIG,id}$ and $v,v_1,\dots,v_n$ are variables of sort val. By (4), (16) is valid if for all $(\varepsilon,\delta)\in A_{exp}\times A_{id}$,

$$F^A(\varepsilon,\delta)\ =\ \{(b(st),b(st'))\mid b:X\to A,\ b(e)=\varepsilon,\ b(z)=\delta,\ A\ \models_b\ \phi'\Rightarrow\psi'[st'/st]\},$$

or, equivalently, if $F^A = h$.

Let $(\rho,\rho') \in F^A(\varepsilon,\delta)$. Then $(\varepsilon, eval^A(\varepsilon,\rho),\delta,\rho') \in P_F^A$ and

$$(\rho,\rho') \;\in\; \Theta_F(F^A)(\varepsilon,\delta) \;=\; b^*(begin\ x := e;\ c_{x/e}\ return(z))$$

where $b : X \to A$ such that $b(e) = \varepsilon$ and $b(z) = \delta$. Hence

$$(put^A(x^A, eval^A(\varepsilon,\rho), enter^A(\rho)),\rho'') \in b^*(c_{x/e}) \quad \text{and} \quad \rho' = put^A(\delta, get^A(\rho'',\delta), leave^A(\rho''))$$

for some ρ''. Since $leave^A(\rho'') = \rho$, we have $get^A(\rho',y) = get^A(\rho,y)$ for all $y \in A_{id} \setminus \{\delta\}$. Hence $F^A \le h$.

Conversely, let $(\rho,\rho') \in h(\varepsilon,\delta)$. Then there is $\sigma \in A_{state}$ such that $(\sigma,\rho') \in F^A(\varepsilon,\delta)$, $eval^A(\varepsilon,\rho) = eval^A(\varepsilon,\sigma)$ and for all $y \in A_{id} \setminus \{\delta\}$, $get^A(\rho',y) = get^A(\rho,y)$. Hence for $b : X \to A$ with $b(e) = \varepsilon$ and $b(z) = \delta$,

$$(\sigma,\rho') \;\in\; \Theta_F(F^A)(\varepsilon,\delta) \;=\; b^*(begin\ x := e;\ c_{x/e}\ return(z))$$

and thus $(put^A(x^A, eval^A(\varepsilon,\sigma), enter^A(\sigma)),\sigma'') \in b^*(c_{x/e})$ and

$$\rho' \;=\; put^A(\delta, get^A(\sigma'',\delta), leave^A(\sigma'')) \tag{17}$$

for some σ''. Since $eval^A(\varepsilon,\rho) = eval^A(\varepsilon,\sigma)$,

$$(put^A(x^A, eval^A(\varepsilon,\rho), enter^A(\sigma)),\sigma'') \in b^*(c_{x/e}). \tag{18}$$

Provided that

(*) no identifier occurring in c is used before its value is changed, in other words, for all occurrences o of a term $var(y)$ in c there is an occurrence of y that precedes o,

(17) and (18) imply for some ρ'' and all $y \in A_{id} \setminus \{\delta\}$, $(put^A(x^A, eval^A(\varepsilon,\rho), enter^A(\rho)),\rho'') \in b^*(c_{x/e})$, $get^A(\rho',\delta) = get^A(\sigma'',\delta) = get^A(\rho'',\delta)$ and $get^A(\rho',y) = get^A(\rho,y) = get^A(leave^A(\rho''),y)$. Hence $\rho' = put^A(\delta, get^A(\rho'',\delta), leave^A(\rho''))$ and thus

$$(\rho,\rho') \;\in\; b^*(begin\ x := e;\ c_{x/e}\ return(z)) \;=\; \Theta_F(F^A)(\varepsilon,\delta) \;=\; F^A(\varepsilon,\delta).$$

This completes the proof of $h \le F^A$. Assumption (*) makes the procedure rule into a contraction rule:

procedure rule

$$\frac{\{\phi\}\ F(e,z)\ \{\psi\}}{\begin{array}{c}\exists\phi',\psi'\ \forall F :\ (\forall X : \{\phi'\}\ F(e,z)\ \{\psi'\}) \;\Rightarrow\; \{\phi'\}\ begin\ x := e;\ c_{x/e}\ return(z)\ \{\psi'\},\\ (\forall X \setminus \{e,z,st,st'\} :\ \phi' \Rightarrow \psi'[st'/st]) \;\wedge\; \phi \quad\Rightarrow\quad \psi[st'/st]\end{array}} \;\Updownarrow$$

if (*) holds true

Reasoning about imperative programs within the final model $A = Fin(EXT)$ allows us to employ both the rules for (co)inductive theories developed in Section 7 and the assertion rules presented in this section. Assertions over EXT are particular Gentzen clauses. The procedure rule generates non-Gentzen clauses, but since these are implications, we may use at least the basic Gentzen rules *expansion* and *contraction* for further expanding the succedent of the procedure rule.

The results of this section can probably be modified easily for capturing other *state-oriented* language constructs and incorporating them into the uniform framework developed in this paper. The relational interpretation of commands already admits non-deterministic constructs (cf. [Apt 84], [AO 91]). Distribution and synchronization features may further extend the model and its proof system. Most challenging is the question how this approach extends the possibilities of *specification refinement* (cf. Exs. 9.12 and 9.15). For instance, can we use it for verifying implementations of functions or relations by procedures, implementations of "static" types by "dynamic" objects?

10.9 Iteration

The previous section dealt with verifying proper recursive programs within an imperative language. Let us now head for the inverse goal: verifying iterative (also called tail-recursive or repetitively recursive) programs within our functional language of Horn logic. Each iterative program can be presented as a set of Horn clauses for two functions $F : A \to C$ and $G : B \to C$ that matches a particular axiom schema:

$$\begin{aligned}
&F(x) \equiv G(in(x))\\
&G(y) \equiv G(loop_1(y)) \;\Leftarrow\; g_1(y)\\
&\ldots\\
&G(z) \equiv G(loop_n(y)) \;\Leftarrow\; g_n(y)\\
&G(y) \equiv out(y) \;\Leftarrow\; g_{n+1}(y)
\end{aligned}$$

where $in : A \to B$, $loop_1, \ldots, loop_n : B \to B$ und $out : B \to C$ are auxiliary functions and $g_1 \vee \ldots \vee g_{n+1}$ is supposed to be an inductive theorem.

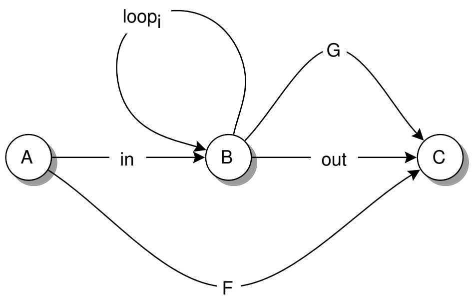


Figure 14. The type schema of an iterative program

Since F is defined in terms of the *loop function* G, a requirement P to F can only be derived from a suitable condition Q that generalizes P to a property of G. In Section 7 we mentioned that fixpoint induction can also be applied to several predicates simultaneously. Indeed, given programs for the predicates P_1, P_2 and given two further predicates Q_1, Q_2, we may use the following rule:

fixpoint induction on (P_1, P_2)

$$\frac{gs \;\Leftarrow\; P_i(x)}{\exists Q_1, Q_2 :\; (prog(P_1) \cup prog(P_2))[Q_1/P_1, Q_2/P_2],\quad gs \;\Leftarrow\; Q_i(x)} \Updownarrow$$

if $i = 1, 2$, P_1 and P_2 are safety predicates or graphs of defined functions F_1 resp. F_2 where, in the latter case, $prog(P_j) = prog_{F_1,F_2}(F_j)$, $j = 1, 2$ (cf. 6.13).

Similarly to fixpoint induction on P, the expansion property of the new rule is due to an intermediate cut step, while the contraction property follows from choosing Q_1 as P_1 and Q_2 as P_2.

Provided that the assumptions of Lemma 6.13 hold true, the following rule is a sound instance of the previous one:

fixpoint induction on (F, G)

$$\frac{gs \;\Leftarrow\; F(x) \equiv z}{\exists P \subseteq A \times C, Q \subseteq B \times C :\; (prog_{F,G}(F) \cup prog_{F,G}(G))[P/graph_F, Q/graph_G],\quad gs \;\Leftarrow\; P(x, z)} \Updownarrow$$

Given the above functional programs for F and G, the succedent of this rule splits into the following clauses:

$$\begin{aligned}
&P(x,z) \Leftarrow Q(in(x),z)\\
&Q(y,z) \Leftarrow Q(loop_1(y),z) \wedge g_1(y)\\
&\dots\\
&Q(y,z) \Leftarrow Q(loop_n(y),z) \wedge g_n(y)\\
&Q(y,out(y)) \Leftarrow g_{n+1}(y)\\
&gs \Leftarrow P(x,z)
\end{aligned}$$

A predicate Q that satisfies the first n+2 conditions is usually called a **subgoal invariant** of F w.r.t. P (cf. [LS 87]). $Q \subseteq B \times C$ relates the values of the *loop variable(s)* y to the *values* of F. But is often easier to find a **Hoare invariant** $H \subseteq A \times B$ of F w.r.t. P. Dually to Q, H relates the *arguments* of F to the loop variables.

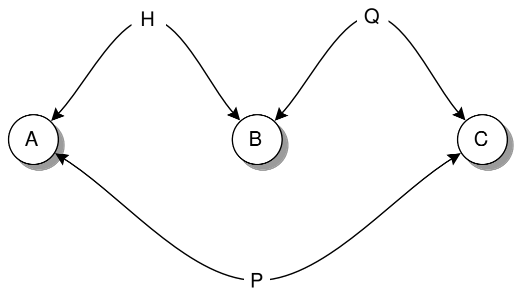


Figure 15. Hoare and subgoal invariant

H is a Hoare invariant if the following clauses are inductive theorems:

$$\begin{aligned}
&H(x,in(x))\\
&H(x,loop_1(y)) \Leftarrow H(x,y) \wedge g_1(y)\\
&\dots\\
&H(x,loop_n(y)) \Leftarrow H(x,y) \wedge g_n(y)\\
&P(x,out(y)) \Leftarrow H(x,y) \wedge g_{n+1}(y)
\end{aligned}$$

A subgoal invariant can be derived from a Hoare invariant as well as a Hoare invariant is definable in terms of a subgoal invariant:

$$\begin{aligned}
Q(y,z) &\iff_{def} \forall x : P(x,z) \Leftarrow H(x,y),\\
H(x,y) &\iff_{def} \forall z : P(x,z) \Leftarrow Q(y,z).
\end{aligned}$$

Hence an iterative program can always be proved correct by establishing a Hoare invariant:

Hoare induction on (F,G)

$$\frac{gs \Leftarrow F(x) \equiv z}{\exists P \subseteq A \times C, H \subseteq A \times B : \begin{array}{l} H(x,in(x)),\ \ \forall 1 \le i \le n : H(x,loop_i(y)) \Leftarrow H(x,y) \wedge g_i(y),\\ P(x,out(y)) \Leftarrow H(x,y) \wedge g_{n+1}(y),\ \ gs \Leftarrow P(x,z)\end{array}} \Updownarrow$$

Exercise. Let

$$\begin{aligned}
P_1(n,z) &\iff_{def} \textstyle\sum_{i=1}^{n} i = z\\
P_2(m,n,q,r) &\iff_{def} m = n * q + r \wedge r < n\\
P_3(n,z) &\iff_{def} z \text{ is the n-th Fibonacci number.}
\end{aligned}$$

Given the iterative programs

```
F1(n) = G1(n,0)
G1(x,y) = G1(x,y+x)  <==  x > 0
G1(0,y) = y

F2(m,n) = G2(n,0,m)  <==  n > 0
F2(m,0) = bottom
G2(n,q,r) = G2(n,succ(q),r-n)  <==  r >= n
G2(n,q,r) = (q,r)  <==  r < n

F3(n) = G3(n,0,1)
G3(n,x,y) = G3(n-1,y,x+y)  <==  n > 0
G3(0,x,y) = x
```

show that

$$
\begin{aligned}
H_1(n,x,y) &\iff_{def} && \textstyle\sum_{i=1}^{n} i = y + \sum_{i=1}^{n} x \\
H_2(m,n,q,r) &\iff_{def} && m = n * q + r \\
H_3(n,x,y,z) &\iff_{def} && y \text{ and } z \text{ are the (n-x)-th resp. (n-x+1)-th Fibonacci numbers}
\end{aligned}
$$

are Hoare invariants of F_i w.r.t. P_i, $i = 1,2,3$. With $gs =_{def} P(x,z)$ we conclude by Hoare induction on (F_i, G_i) that $P_1(n,z) \Leftarrow F_1(n) \equiv z$, $P_2(m,n,q,r) \Leftarrow F_2(m,n) \equiv (q,r)$ and $P_3(n,z) \Leftarrow F_3(n) \equiv z$ are inductive theorems.

We have hidden the axioms that specify P_i and H_i. These predicates are often defined in terms of other functions. If the overall goal is to show that F is equivalent to some non-iterative "abstract" program F', then $P(x,z)$ is defined as $F'(x) \equiv z$ so that a Hoare invariant of F w.r.t. P proves $\forall x : F(x) \equiv F'(x)$. The question remains whether the iterative program, i.e. the axioms for F, may replace those for F'. To work this out formally one needs a notion of *inductive equivalent* specifications, which in turn relies on a notion of *signature morphisms* that identify corresponding symbols of two specifications (cf. [Pad 96]). Roughly said, both the original specification and the new one where F is "redefined" by the axioms for F' must be consistent w.r.t. the initial (or final) model of their common subspecification (cf. 3.7).

References

[AL 93a] M. Abadi, L. Lamport, *Composing Specifications*, ACM Transactions of Progr. Lang. and Syst. 15 (1993) 73-132

[Abr 90] S. Abramsky, *The Lazy λ-Calculus*, in: D.A. Turner, ed., Research Topics in Functional Programming, Addison-Wesley (1990) 65-116

[AO 93] S. Abramsky, C.-H. Luke Ong, *Full Abstraction in the Lazy λ-Calculus*, Information and Computation 105 (1993) 159-267

[AS 87] B. Alpern, F.B. Schneider, *Recognizing Safety and Liveness*, Distributed Computing 2 (1987) 117-126

[Apt 81] K.R. Apt, *Ten Years of Hoare's Logic: A Survey - Part I*, ACM TOPLAS 5 (1981) 431-483

[Apt 84] K.R. Apt, *Ten Years of Hoare's Logic: A Survey - Part II: Nondeterminism*, Theoretical Computer Science 28 (1984) 83-109

[Apt 90] K.R. Apt, *Logic Programming*, in: J. van Leeuwen, ed., Handbook of Theoretical Computer Science, Elsevier (1990) 493-574

[AvE 82] K.R. Apt, M.H. van Emden, *Contributions to the Theory of Logic Programming*, J. ACM 29 (1982) 841-862

[AO 91] K.R. Apt, E.-R. Olderog, *Verification of Sequential and Concurrent Programs*, Springer 1991

[AW 89] E. Astesiano, M. Wirsing, *Bisimulation in Algebraic Specifications*, in: H. Ait-Kaci, M. Nivat, eds., Resolution of Equations in Algebraic Structures 1, Academic Press (1989) 1-31

[AB 92] J. Avenhaus, K. Becker, *Conditional Rewriting modulo a Built-in Algebra*, SEKI Report SR-92-11, Universität Kaiserslautern (1992)

[AL 93] J. Avenhaus, C. Loría-Sáenz, *Canonical Conditional Rewrite Systems Containing Extra Variables*, SEKI Report SR-93-03, FB Informatik, Universität Kaiserslautern 1993

[BW 90] J.C.M. Baeten, W.P. Weijland, *Process Algebra* Cambridge University Press 1990

[Bar 93] M. Barr, *Terminal Coalgebras in Well-founded Set Theory*, Theoretical Computer Science 114 (1993) 299-315

[BSW 69] K.A. Bartlet, R.A. Scantlebury, P.T. Wilkinson, *A Note on Reliable Full-Duplex Transmission over Half-Duplex Links*, Communications ACM 12 (1969) 260-261

[Bat 79] G. Bateson, *Mind and Nature: A Necessary Unity*, Dutton 1979

[BK 84] J.A. Bergstra, J.W. Klop, *Process Algebra for Synchronous Communication*, Information and Control 60 (1984) 109-137

[BK 85] J.A. Bergstra, J.W. Klop, *Algebra of Communicating Processes with Abstraction*, Theoretical Computer Science 37 (1985) 77-121

[BT 85] J.A. Bergstra, J.V. Tucker, *Top-down Design and the Algebra of Communicating Processes*, Science of Computer Programming 5 (1985) 171-199

[BG 89] H. Bertling, H. Ganzinger, *Completion-Time Optimization of Rewrite-Time Goal Solving*, Proc. RTA '89, Springer LNCS 355 (1989) 45-58

[Best 95] E. Best, *Semantik: Theorie sequentieller und paralleler Programmierung*, Vieweg 1995

[BH 95] M. Bidoit, R. Hennicker, *Behavioural Theories and the Proof of Behavioural Properties*, Report LIENS-95-5, DMI, LIENS Paris 1995

[BH 95a] M. Bidoit, R. Hennicker, *Proving the Correctness of Behavioural Implementations*, Proc. AMAST '95, Springer LNCS 936 (1995) 152-168

[BHW 95] M. Bidoit, R. Hennicker, M. Wirsing, *Behavioural and Abstractor Specifications*, Science of Computer Programming 25 (1995) 149-186

[BW 88] R. Bird, Ph. Wadler, *Introduction to Functional Programming*, Prentice Hall 1988

[Blo 76] S.L. Bloom, *Varieties of Ordered Algebras*, J. Computer and System Sciences 13 (1976) 200-212

[BGM 88] P.G. Bosco, E. Giovannetti, C. Moiso, *Narrowing versus SLD-Resolution*, Theoretical Computer Science 59 (1988) 3-23

[BKR 92] A. Bouhoula, E. Kounalis, M. Rusinowitch, *Automated Mathematical Induction*, Report INRIA Lorraine No. 1636 (1992)

[BB 84] M. Broy, F.L. Bauer, *A Systematic Approach to Language Constructs for Concurrent Programs*, Science of Computer Programming 4 (1984) 103-139

[BDDFGW 93] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, Th. Gritzner, R. Weber, *The Design of Distributed Systems: An Introduction to FOCUS*, Report, Institut für Informatik, TU München 1993

[BPW 84] M. Broy, C. Pair, M. Wirsing, *A Systematic Study of Models of Abstract Data Types*, Theoretical Computer Science 33 (1984) 139-174

[BW 82] M. Broy, M. Wirsing, *Partial Abstract Types*, Acta Informatica 18 (1982) 47-64

[Bun 88] W. Buntine, *Generalized Subsumption and Its Applications to Induction and Redundancy*, Artificial Intelligence 36 (1988) 149-176

[CO 90] Y. Chen, M.J. O'Donnell, *Infinite Terms and Infinite Rewritings*, Proc. CTRS '90, Springer LNCS 516 (1990) 115-126

[CF 92] R. Cockett, T. Fukushima, *About Charity*, Report, University of Calgary 1992

[CR 91] G. Costa, G. Reggio, *Abstract Dynamic Types: A Temporal Logic Approach*, Proc. MFCS '91, Springer LNCS 520 (1991) 103-112

[Cou 90] P. Cousot, *Methods and Logics for Proving Programs*, in: J. van Leeuwen, ed., Handbook of Theoretical Computer Science, Elsevier (1990) 841-993

[Dav 92] A.J.T. Davie, *An Introduction to Functional Programming Systems using Haskell*, Cambridge University Press 1992

[DW 90] F. Dederichs, R. Weber *Safety and Liveness from a Methodological Point of View*, Information Processing Letters 36 (1990) 25-30

[DJ 90] N. Dershowitz, J.-P. Jouannaud, *Rewrite Systems*, in: J. van Leeuwen, ed., Handbook of Theoretical Computer Science, Elsevier (1990) 243-320

[DOS 87] N. Dershowitz, M. Okada, G. Sivakumar, *Confluence of Conditional Rewrite Systems*, Proc. CTRS '87, Springer LNCS 308 (1988) 31-44

[DOS 88] N. Dershowitz, M. Okada, G. Sivakumar, *Canonical Conditional Rewrite Systems*, Proc. CADE '88, Springer LNCS 310 (1988) 538-549

[DO 90] N. Dershowitz, M. Okada, *A Rationale for Conditional Equational Programming*, Theoretical Computer Science 75 (1990) 111-138

[EM 85] H. Ehrig, B. Mahr, *Fundamentals of Algebraic Specification 1*, Springer 1985

[EPBRDG 90] H. Ehrig, F. Parisi-Presicce, P. Boehm, C. Riekhoff, C. Dimitrovici, M. Grosse-Rhode, *Combining Data Type and Recursive Process Specifications Using Projection Algebras*, Theoretical Computer Science 71 (1990) 347-380

[Flo 67] R.W. Floyd, *Assigning Meanings to Programs*, in: J. T. Schwartz, ed., Mathematical Aspects of Computer Science, American Mathematical Society (1967) 19-32

[Fra 92] N. Francez, *Program Verification*, Addison-Wesley 1992

[Fri 85] L. Fribourg, *Handling Function Definitions through Innermost Superposition and Rewriting*, Proc. RTA '85, Springer LNCS 202 (1985) 325-344

[Fri 89] L. Fribourg, *A Strong Restriction of the Inductive Completion Procedure*, J. Symbolic Computation 8 (1989) 253-276

[GW 93] H. Ganzinger, U. Waldmann, *Termination Proofs of Well-Moded Logic Programs via Conditional Rewrite Systems*, Proc. CTRS '92, Springer LNCS 656 (1993) 430-437

[GGM 76] V. Giarratana, F. Gimona, U. Montanari, *Observability Concepts in Abstract Data Type Specifications*, Proc. MFCS '76, Springer LNCS 45 (1976) 576-587

[GD 94] J.A. Goguen, R. Diaconescu, *Towards an Algebraic Semantics for the Object Paradigm*, Proc. 9th ADT Workshop, Springer LNCS 785 (1994) 1-29

[GM 86] J.A. Goguen, J. Meseguer, *Eqlog: Equality, Types, and Generic Modules for Logic Programming*, in: D. DeGroot, G. Lindstrom, Logic Programming: Functions, Relations, and Equations, Prentice-Hall (1986) 295-363

[GM 87] J.A. Goguen, J. Meseguer, *Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics*, in: B. Shriver, P. Wegner, eds., Research Directions in Object-Oriented Programming, MIT Press (1987) 417-477

[GTW 78] J.A. Goguen, J.W. Thatcher, E.G. Wagner, *An Initial Algebra Approach to the Specification,Correctness and Implementation of Abstract Data Types*, in: R. Yeh, ed., Current Trends in Programming Methodology 4, Prentice-Hall (1978) 80-149

[GTWW 77] J.A. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright, *Initial Algebra Semantics and Continuous Algebras*, J. ACM 24 (1977) 68-95

[Gor 95] A.D. Gordon, *A Tutorial on Co-induction and Functional Programming*, Proc. Functional Programming Glasgow 1994, Springer (1995) 78-95

[Gor 79] M.J.C. Gordon, *The Denotational Description of Programming Languages*, Springer 1979

[Gur 91] Y. Gurevich, *Evolving Algebras: A Tutorial Introduction*, EATCS Bulletin 43 (1991) 264-284

[GHM 76] J. Guttag, E. Horowitz, D.R. Musser *Abstract Data Types and Software Validation*, Report ISI/RR-76-48, University of Southern California 1976

[Hag 89] T. Hagino, *Codatatypes in ML*, J. Symbolic Computation 8 (1989) 629-650

[Han 94] M. Hanus, *The Integration of Functions into Logic Programming: From Theory to Practice*, J. Logic Programming 19/20 (1994) 583-628

[HM 85] M. Hennessy, R. Milner, *Algebraic Laws for Nondeterminism and Concurrency*, J. Assoc. for Computing Machinery (1985) 137-161

[Hen 91] R. Hennicker, *Context Induction: A Proof Principle for Behavioural Abstractions*, Formal Aspects of Computing (1991) 326-345

[Hoa 69] C.A.R. Hoare, *An Axiomatic Basis for Computer Programming*, Communications ACM 12 (1969) 576-580

[Hoa 85] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall 1985

[Höl 89] S. Hölldobler, *Foundations of Equational Logic Programming*, Springer (1989)

[JLM 84] J. Jaffar, J.-L. Lassez, M.J. Maher, *A Theory of Complete Logic Programs with Equality*, J. Logic Programming 3 (1984) 211-223

[JL 87] J. Jaffar, J.-L. Lassez, *Constraint Logic Programming*, Proc. POPL '87 (1987) 111-119

[JS 89] S.B. Jones, A.F. Sinclair, *Functional Programming and Operating Systems*, The Computer Journal 32 (1989) 162-174

[JKi 86] J.-P. Jouannaud, H. Kirchner, *Completion of a Set of Rules Modulo a Set of Equations*, SIAM J. Computing 15 (1986) 1155-1194

[JKo 86] J.-P. Jouannaud, H. Kounalis, *Automatic Proofs by Induction in Equational Theories without Constructors*, Proc. LICS '86 (1986) 358-366

[JW 86] J-P. Jouannaud, B. Waldmann, *Reductive Conditional Term Rewriting Systems*, Proc. Conf. Formal Description of Programming Concepts III, North-Holland (1986) 223-244

[Kam 83] S. Kamin, *Final Data Type Specifications: A New Data Type Specification Method*, ACM TOPLAS 5 (1983) 97-123

[Kap 87] S. Kaplan, *Simplifying Conditional Term Rewriting Systems: Unification, Termination and Confluence*, J. Symbolic Computation 4 (1987) 295-334

[KM 87] D. Kapur, D.R. Musser, *Proof by Consistency*, Artificial Intelligence 31 (1987) 125-157

[Kin 94] E. Kindler, *Safety and Liveness Properties: A Survey*, EATCS Bulletin 53 (1994) 268-272

[KR 90] E. Kounalis, M. Rusinowitch, *Mechanizing Inductive Reasoning*, EATCS Bulletin 41 (1990) 216-226

[KT 90] D. Kozen, J. Tiuryn, *Logics of Programs*, in: J. van Leeuwen, ed., Handbook of Theoretical Computer Science, Elsevier (1990) 789-840

[Küc 89] W. Küchlin, *Inductive Completion by Ground Proof Transformation*, in: H. Ait-Kaci, M. Nivat, eds., Resolution of Equations in Algebraic Structures, Vol. 2, Academic Press (1989) 211-244

[Kro 87] F. Kröger, *Temporal Logic of Programs*, Springer 1987

[Lam 94] L. Lamport, *The Temporal Logic of Actions*, Transactions of Progr. Lang. and Syst. 16 (1994) 308-320

[Lan 64] P.J. Landin, *The Mechanical Evaluation of Expressions*, BCS Computing Journal 6 (1964) 872-923

[LNS 82] J.-L. Lassez, V.L. Nguyen, E.A. Sonenberg, *Fixed Point Theorems and Semantics: A Folk Tale*, Information Processing Letters 14 (1982) 112-116

[LS 81] D.J. Lehmann, M.B. Smyth, *Algebraic Specification of Data Types: A Synthetic Approach*, Mathematical Systems Theory 14 (1981) 97-139

[LM 82] M.R. Levy, T.S.E. Maibaum, *Continuous Data Types*, SIAM J. Computing 11 (1982) 201-216

[Llo 87] J. W. Lloyd, *Foundations of Logic Programming*, 2nd ed., Springer 1987

[LS 87] J. Loeckx, K. Sieber, *The Foundations of Program Verification*, 2nd ed., Teubner 1987

[MPS 86] D. MacQueen, G. Plotkin, R. Sethi, *An Ideal Model for Recursive Polymorphic Types*, Information and Control 71 (1986) 95-130

[MG 94] G. Malcolm, J.A. Goguen, *Proving Correctness of Refinement and Implementation*, Technical Monograph PRG-114, Oxford University Computing Lab 1994

[Manna 74] Z. Manna, *Mathematical Theory of Computation*, McGraw-Hill 1974

[MP 92] Z. Manna, A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer 1992

[MM 82] A. Martelli, U. Montanari, *An Efficient Unification Algorithm*, ACM TOPLAS 4 (1982) 258-282

[MG 85] J. Meseguer, J.A. Goguen, *Initiality, Induction and Computability*, in: M. Nivat, J. Reynolds, eds., Algebraic Methods in Semantics, Cambridge University Press (1985) 459-541

[Mes 93] J. Meseguer, *A Logical theory of Concurrent Objects and its Realization in the Maude Language*, in: G. Agha, P. Wegner, A. Yonezawa, eds., Research Directions in Object-Based Concurrency, MIT Press (1993) 313-389

[Mil 77] R. Milner, *Fully Abstract Models of Typed λ-Calculi*, Theoretical Computer Science 4 (1977) 1-22

[Mil 83] R. Milner, *Calculi for Synchrony and Asychrony*, Theoretical Computer Science 25 (1983) 267-310

[Mil 89] R. Milner, *Communication and Concurrency*, Prentice-Hall 1989

[Möl 84] B. Möller, *On the Algebraic Specification of Infinite Objects: Ordered and Continuous Models of Algebraic Types*, Acta Informatica 22 (1984) 537-578

[MTW 88] B. Möller, A. Tarlecki, M. Wirsing, *Algebraic Specifications of Reachable Higher-Order Algebras*, Proc. 5th ADT Workshop, Springer LNCS 332 (1988) 154-169

[Mor 71] J.H. Morris, *Another Recursion Induction Principle*, Communications ACM 14 (1971) 351-354

[MR 94] S. Muggleton, L. de Raedt, *Inductive Logic Programming: Theory and Methods*, J. Logic Programming 19/20 (1994) 629-679

[NO 88] P. Nivela, F. Orejas, *Initial Behaviour Semantics for Algebraic Specifications*, Proc. 5th ADT Workshop, Springer LNCS 332 (1988) 184-207

[Niv 75] M. Nivat, *On the Interpretation of Recursive Polyadic Program Schemes*, Symposia Mathematica 15 (1975) 255-281

[Pad 83] P. Padawitz, *Equational Data Type Specifications and Recursive Program Schemes*, Proc. Formal Description of Programming Concepts II, North-Holland (1983) 305-328

[Pad 88] P. Padawitz, *Computing in Horn Clause Theories*, Springer 1988

[Pad 91] P. Padawitz, *Inductive Expansion: A Calculus for Verifying and Synthesizing Functional and Logic Programs*, J. Automated Reasoning 7 (1991) 27-103

[Pad 91a] P. Padawitz, *A Functional Compiler Specification and its Direct Implementation into SML*, Report MIP-9105, FB Mathematik und Informatik, Universität Passau 1991

[Pad 92] P. Padawitz, *Deduction and Declarative Programming*, Cambridge University Press 1992

[Pad 94] P. Padawitz, *Expander: A System for Testing and Verifying Functional-Logic Programs*, Report No. 522/1994, FB Informatik, Universität Dortmund 1994

[Pad 95] P. Padawitz, *Inductive Theorem Proving for Design Specifications*, J. Symbolic Computation 21 (1996) 41-99

[Pad 96] P. Padawitz, *Specification and Deduction*, Report, FB Informatik, Universität Dortmund 1996

[Par 69] D. Park, *Fixpoint Induction and Proofs of Program Properties*, in: B. Meltzer, D. Michie, eds., Machine Intelligence 5, Elsevier (1969) 59-78

[Pau 91] L.C. Paulson, *ML for the Working Programmer*, Cambridge University Press 1991, 2nd ed. 1996

[Pau 94] L.C. Paulson, *A Fixedpoint Approach to Implementing (Co)Inductive Definitions*, Proc. CADE 12, Springer LNAI 814 (1994) 148-161

[Plo 70] G.D. Plotkin, *A Note on Inductive Generalization*, in: B. Meltzer, D. Michie, eds., Machine Intelligence 5, Elsevier (1970) 153-163

[Pra 76] V. Pratt, *Semantical Considerations of Floyd-Hoare Logic*, Proc. IEEE FOCS '76 (1976) 109-121

[Rea 89] C. Reade, *Elements of Functional Programming*, Addison-Wesley 1989

[Red 85] U. Reddy, *Narrowing as the Operational Semantics of Functional Languages*, Proc. IEEE Symp. on Logic Programming (1985) 138-151

[Red 90] U. Reddy, *Term Rewriting Induction*, Proc. CADE 10, Springer LNCS 449 (1990) 162-177

[Rei 95] H. Reichel, *An Approach to Object Semantics based on Terminal Coalgebras*, Math. Structures in Comp. Sci. 5 (1995) 129-152

[Rei 78] R. Reiter, *On Closed World Data Bases*, Proc. Logic and Data Bases, Plenum Press (1978)

[Rut 93] J. Rutten, *A Structural Coinduction Theorem*, Proc. MFPS '93, Springer LNCS 802 (1993) 83-102

[ST 87] D. Sannella, A. Tarlecki, *On Observational Equivalence and Algebraic Specification*, J. Computer and System Sciences 34 (1987) 150-178

[Sco 70] D. Scott, *Outline of a Mathematical Theory of Computation*, Proc. Information Sciences and Systems, Princeton (1970) 169-176

[Sla 74] J.R. Slagle, Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity, Journal ACM 21 (1974) 622-642

[Sti 85] M. Stickel, *Automated Deduction by Theory Resolution*, J. Automated Reasoning 1 (1985) 333-356

[Sti 92] C. Stirling, *Modal and Temporal Logics*, in: S. Abramsky et al., eds., Handbook of Logic in Computer Science, Clarendon Press (1992) 477-563

[SW 89] C. Stirling, D. Walker *Local Model Checking in the Modal μ-Calculus*, Proc. TAPSOFT '89, Springer LNCS 351 (1989) 369-383

[TW 86] A. Tarlecki, M. Wirsing, *Continuous Abstract Data Types*, Fundamenta Informaticae 9 (1986) 95-126

[Tur 90] D.A. Turner, *An Overview of Miranda*, in: D.A. Turner, ed., Research Topics in Functional Programming, Addison-Wesley 1990

[Tur 91] R. Turner, *Constructive Foundations for Functional Languages*, McGraw-Hill 1991

[Wan 79] M. Wand, *Final Algebra Semantics and Data Type Extensions*, J. Computer and System Sciences 19 (1979) 27-44

[Wan 82] M. Wand, *Specifications, Models, and Implementations of Data Abstractions*, Theoretical Computer Science 20 (1982) 3-32

[Win 90] J. Wing, *A Specifier's Introduction to Formal Methods*, Computer 9/1990 (1990) 8-24

[WPPDB 83] M. Wirsing, P. Pepper, H. Partsch, W. Dosch, M. Broy, *On Hierarchies of Abstract Data Types*, Acta Informatica 20 (1983) 1-33

[Wir 90] M. Wirsing, *Algebraic Specification*, in: J. van Leeuwen, ed., Handbook of Theoretical Computer Science, Elsevier (1990) 675-788