Refinements in HOLCF: Implementation of Interactive Systems

Oscar Slotosch

Fakultät für Informatik
der Technischen Universität München

Refinements in HOLCF: Implementation of Interactive Systems

Oscar Slotosch

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. rer. nat. habil Jürgen Eickel

Prüfer der Dissertation:

1. Univ.-Prof. Dr. rer. nat. Dr. rer. nat. habil Manfred Broy
2. Univ.-Prof. Tobias Nipkow, Ph. D.

Die Dissertation wurde am 23.1.1997 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 18.3.1997 angenommen.

Abstract

In this thesis refinement relations for the logic HOLCF are defined. We compare refinement relations defined by theory interpretations and by model inclusion. We use these refinements to implement abstract data types (ADTs) with LCF domains and continuous functions. Therefore, the implementation of ADTs may be applied to the implementation of interactive and distributed systems specified in HOLCF.

The implementation of interactive systems is embedded into the deductive software development process. Every development step corresponds to a refinement in HOLCF. A detailed classification of different situations in the development of interactive and distributed systems is given. For all possible development situations concrete refinement methods are described. This allows us to prove the correctness of development steps by verifying the refinement relation in HOLCF. The refinement relation is compositional and in some situations the methods improve the known methods for interaction refinement by requiring less, and more concrete, proof obligations and by a stronger compositionality result.

For the implementation of abstract data types in HOLCF two type constructors are added to the Isabelle proof system. The `subdom` type constructor conservatively introduces a subdomain of a LCF domain that is again a LCF domain. The `quot` type constructor conservatively defines flat quotient domains with respect to an arbitrary partial equivalence relation (PER). PERs are also the basis for a higher order predicate allowing us to express observability of (higher order) functions in HOLCF to characterize congruences. Both type constructors are supported with methods and examples for the introduction of continuous functions.

A standard library of ADTs is defined and the implementation of a WWW server is taken as an example and some critical development steps are verified with the Isabelle proof system.

Acknowledgments

Four years ago, when I started to work at the group of Manfred Broy and Tobias Nipkow, with the plan to write a thesis, I did not know how much work this would be and how much support would be necessary. Now I try to thank all who helped me in doing this thesis.

First of all I like to thank my supervisors Manfred Broy and Tobias Nipkow. Manfred Broy gave me the opportunity to work in the field of formal methods and he directed me towards the concrete topic of my thesis. Tobias Nipkow encouraged me to work with the Isabelle system, and he improved the quality of the realization with his technical advice. For the scientific and constructive discussions I thank my supervisors.

Special thanks go to Franz Regensburger for inventing the logic HOLCF and for introducing me into it. For discussions about HOLCF, domains, and fixed points I thank David von Oheimb. I thank Birgit Schieder for carefully reading the proofs which I did not carried out in Isabelle.

Particular thanks go to all my colleagues from our department. For comments on many parts of previous versions of my thesis I would like to thank Manfred Broy, Tobias Nipkow, Bernhard Schätz, Ursula Hinkel, Ingolf Krüger, Rudolf Hettler, Konrad Slind, Katharina Spies, Birgit Schieder, Bernhard Rumpe, Franz Regensburger, David von Oheimb, Eva Geisberger, and Cornel Klein. For help with LaTeX I also thank Franz Regensburger, Bernhard Rumpe and Katharina Spies.

I furthermore want to thank Konrad Slind, Rosemary Monahan, Susen Werner, Ingolf Krüger and Bernhard Schätz for polishing up my sometimes rough English with their constructive comments.

Personally, I am grateful to my family. Especially I would like to thank my father who gave me the energy for this work and Susanne for her love and patience.

Contents

Chapter 1

Introduction and Concepts

In this chapter the motivation for the implementation of interactive systems by refinement in the logic HOLCF is presented. The deductive software development process is sketched and a classification of different situations in the development of interactive and distributed systems is presented in Section 1.2. We call the concrete methods for formally developing these situations *implementation of interactive systems*, because we extend the methods for the implementation of abstract data types (ADTs) to interactive systems. In Section 1.3 we look at the ideas of the implementation of ADTs and collect the requirements for the refinement relation in Section 1.4. The use of HOLCF is explained in Section 1.4.4, goals and structure of this thesis are given at the end of this chapter.

The main structure of this work is influenced by the implementation of ADTs. There are two different implementation steps (quotient step in Chapter 4), restriction step in Chapter 3) for ADTs and we develop two refinements techniques (theory interpretation and model inclusion) for each of them and compare them by the criteria from the deductive software development process. The proposed method (Chapter 5) for the implementation of ADTs is a combination of these two refinements and is applied (and extended) to the implementation of interactive systems in Chapter 6. Chapter 7 defines a standard library of ADTs and uses it for case studies on a WWW server.

1.1 Motivation

Hardware becomes smaller and *systems* are growing. This trend is caused by the miniaturization and integration in the development of computer chips. Having smaller and more efficient hardware allows us to build more complex systems. An additional source of system complexity is *communication*. The development of fiber optics and satellite channels are examples for the improvement in communication technology. This allows us to connect computers worldwide to build efficiently interacting systems. The internet is an example

of this development. The number of WWW servers in the internet doubles approximately every year [Pax94].

In the expanding market of software systems, product *quality* is an important marketing strategy. Quality has many aspects ranging from a friendly interface, good service and documentation to adequate problem solutions. *Correctness* of the system is the basis for quality, since for example a wrong documentation, an aborting system or an inadequate software program are not acceptable in a market with competition. Apart from the marketing aspects of correctness there are *critical systems* which have to be correct, since errors are very intolerable. For example avionic systems, medical systems, and access control systems in confidential environments have to fulfil their requirements.

In traditional software development reviews and tests are the methods to increase the correctness and the quality of systems. The software development process starts with an specification specification of the behaviour of the system. The development process *refines* these documents by incorporating design decisions. For fixed input values *tests* are used to show that a sequential program fulfils its requirements. Due to the high complexity of distributed systems, tests can only cover small parts of the system and therefore, they cannot ensure correctness, in general.

Formal methods give the possibility to prove the correctness of systems. Using them requires to formalize the system requirements into a specification and to give formal semantics to the system. Apart from the possibility to prove the correctness of a system such a formalization may detect errors and inconsistencies of the requirement specification. A calculus allows us to prove the correctness of the system by showing that the specification of the system *refines* the requirement specification. There are two notions of refinement. In the software development it means the design of the system, in formal methods it denotes a logical relation, which can be verified by a deduction in a calculus. Formal methods apply the logical refinement to verify the refinement of systems.

The advantage of the formal methods is that they are abstract and hence applicable to every kind of systems. However, the drawback is that the calculus does not provide any methods for structuring the correctness proof. One possibility to structure this proof is to give a concrete formal method that develops the proof step by step together with the stepwise realization of the system. *Deductive software development* is the result of integrating formal methods into the software development process.

Large systems are composed of *components*. For the development of interactive systems it is important that the components can be developed independently in a modular way. The correctness of the system should also be provable in a modular way. A refinement relation is called *compositional*, if the correctness of the system can be derived from the correctness of the components. Hence, the most important basis for a modular development of correct interactive systems is a compositional refinement relation, which is integrated into the software development process.

1.2 Deductive Software Development

Deductive software development includes formal methods into the process of software development. It aims at a software development process that allows us to deduce the correctness of software with respect to the specification, that describes the required behaviour of the system.

For deducing the correctness of a program it is necessary to have formal semantics for the specification and the program. Methods and tools are needed to show that the program is correct with respect to its specification. In Section 1.2.1 a sketch of a traditional software development process, as it is state of the art (in industry) is given. In Section 1.2.2 it is briefly described, how the software development process of interactive systems could look like, if formal methods were integrated. In this work we concentrate on the foundations for the deductive software development process by developing a refinement relation, which allows us to prove the correctness of all steps in the development of interactive systems.

1.2.1 Traditional Software Development

There are many traditional methods for the development of systems. Many of them are influenced by a certain programming paradigm or are tailored to a special application domain. One programming paradigm are the structured methods [GS77, Dem78]. They use data flow diagrams to model the system. More recently the object oriented design methods appeared [RBP+91, Jac92, Boo93]. There are basically two application domains for which the given methods were specialized. One are technical systems with real time aspects. See [WM85, HP87] for extensions of the structured techniques and [SGW94] for an object oriented method for technical systems. The other application domain are database systems with a large amount of data modelling [Che76, DCC92, AG90, Den91].

All traditional methods use mainly testing to ensure the quality and the correctness of the system. The following coarse scheme describes traditional software development. It emerges from the waterfall model [Roy70], and graphically displayed it looks like a V, and therefore, it is called V model in [Dav90].

1. The development starts with a specification of the *requirements* of the system. This specification may be informal and coarse, but it is, in general, easy to understand. Usually these specifications contain text and diagrams, describing the behaviour of the system. In the first sketch these requirements are loose and have to be worked out. They fix only the requirements of the system, but not the structure or other details of the system.

2. During the design phase this requirement specification is developed step by step by incorporating decisions concerning the architecture of the system. Splitting the system into components requires a splitting of the requirements, with each component

having its own requirements. The combination of the components must preserve the system specification.

3. The functions are implemented by programming some algorithms or by choosing pieces of hardware for their realization. This is done until all parts are completely designed.

4. Testing shows whether the system fulfils the requirements or not. First the basic components are tested, then their integration is checked. If a test contradicts the requirements, some wrong design or implementation steps will be revised and the development will continue at this point.

During the specification and the design of the system some behaviours of the system are fixed, and during the integration phase it can be tested whether the system has these behaviours. In this scheme, and, therefore, in every traditional software development method errors can only be detected by testing[1]. For small components such a try and error development can be manageable, but in large systems, where the first design decisions are justified by the final integration test, early errors have fatal consequences: the development has to be redone, starting from the point where the wrong step was made[2]. Such a process is very expensive and difficult to control for the management. Sources of early errors are misunderstandings due to ambiguity of the informal requirement specification.

Graphical descriptions are an important part of traditional software development methods (for example E/R-diagrams, SDL, statecharts, SSADM), since they help us to visualize and communicate complex structures. Their disadvantage is that they often lack precise semantics and, therefore, they are an additional source of errors.

1.2.2 Formal Methods in Software Development

The use of formal methods allows us to deduce the correctness of programs with respect to their specifications. See [GM85, Bro96, Hoa96] for overviews of formal methods, [Dil94] for Z or [AI91] for VDM, some very popular formal methods and [BDD+92, Bro93, SS95] for a functional method for developing distributed systems. Formal methods are the basis of deductive software development. They provide precise syntax and formal semantics, based on mathematics and logic, for specifications and programs, a definition of refinement between specifications and programs, and deduction methods and tools to prove this refinement relation.

[1] To reduce the number of design and implementation errors reviews and code inspections are practiced.
[2] The V model [Dav90] is such a process model describing the development of large systems.

Deductive software development is the result of integrating formal methods into software development. The basic concepts of deductive software development are:

- to use the traditional development process (requirement, design, implementation),
- to use the same logic in all phases of the software development to specify components,
- to use implication as basis for the refinement relation, since there are a lot of theorem provers which support implication proofs, and
- to use graphical description techniques with formal semantics.

A transitive refinement relation allows us to divide the deductive software development process into several development steps. Every development step corresponds to a refinement step on the semantic level .

Having the same logic in all phases requires the formal language to include abstract and understandable requirement specifications as well as executable specifications. The advantage of using one logic in all phases is that there are no transitions between different logics and that the developer has to use only one logical formalism.

Executable specifications can be executed by an interpreter (*prototyping*) or can be translated into a programming language. If this generation (or interpretation) is correct, there is no need of coding and testing in the development process, since the programs are generated correctly out of executable specifications. If executable specifications correspond directly to programs, as in [HR94, Hot95], code generation will be only a syntactic translation and no formal correctness proof will be needed. There are other formal methods allowing us to extend the definition of executable specifications, for example to define an interpreter and to prove its correctness [BHN+94]. However, since functional programming languages are quite expressive we define executability with respect to these languages and do not worry about the correctness of their realization.

We use the following notations for the deductive software development process:

Definition 1.2.1 ***Deductive Software Development Basis***

A *deductive software development basis* DSWDB $= (\mathcal{L}, M, \models\!\!\!\approx\!\!\!>, \vdash\!\!\!\rightarrow)$ is a quadruple where:

- $\mathcal{L}$ is a specification language and
- M is the semantic domain of the language, containing the (set of) formal models, and
  $M[\![]\!] : \mathcal{L} \longrightarrow M$ assigns the semantics to a specification. All executable specifications have (at least) one model.

- $\mid\!\approx\!\Rightarrow \subseteq M \times M$ is the semantic refinement relation between models such that for $\mathtt{A}, \mathtt{C} \in M$: $\mathtt{A} \mid\!\approx\!\Rightarrow \mathtt{C}$ denotes that $\mathtt{C}$ (concrete) is a refinement of $\mathtt{A}$ (abstract), we require $\mid\!\approx\!\Rightarrow$ to be transitive, and
- $\mid\!\rightsquigarrow \subseteq \mathcal{L} \times \mathcal{L}$ is a syntactic refinement relation for which a calculus and tools should exist. For the correctness of the calculus it is required that for all $\mathtt{A}, \mathtt{C} \in \mathcal{L}$: $\mathtt{A} \mid\!\rightsquigarrow \mathtt{C}$ implies $M[\![\mathtt{A}]\!] \mid\!\approx\!\Rightarrow M[\![\mathtt{C}]\!]$

We will use the term *refinement* polymorphically. If we consider formal models, we mean $\mid\!\approx\!\Rightarrow$, if we are talking about specifications $\mathtt{A}, \mathtt{C} \in \mathcal{L}$, we mean $\mathtt{A} \mid\!\rightsquigarrow \mathtt{C}$ or $M[\![\mathtt{A}]\!] \mid\!\approx\!\Rightarrow M[\![\mathtt{C}]\!]$. We will even use the term refinement for components or functions, and will mean the refinement of the specifications describing the functions.

The intention is that we develop the abstract requirement specification into a concrete, and executable specification and require that we can prove this development step by step with our deductive software development basis. Although these notations will be filled with concrete definitions later on, they suffice for example to define consistency:

Definition 1.2.2 ***Consistency***

Let ($\mathcal{L}$, M, $\mid\!\approx\!\Rightarrow$, $\mid\!\rightsquigarrow$) be a deductive software development basis, then a specification $\mathtt{A} \in \mathcal{L}$ is called *consistent*, if

- $M[\![\mathtt{A}]\!] \neq \emptyset$.

Otherwise $\mathtt{A}$ is called *inconsistent*.

In this introduction we collect the requirements for the method for the implementation of interactive systems (the result is described in Section 1.4). Every refinement step from an abstract specification $\mathtt{A}$ to a concrete specification $\mathtt{C}$ will be correct, if $M[\![\mathtt{A}]\!] \mid\!\approx\!\Rightarrow M[\![\mathtt{C}]\!]$ holds. In the deductive software development process the resulting proof obligation is $\mathtt{A} \mid\!\rightsquigarrow \mathtt{C}$. Since the refinement relation is transitive, a correctly developed executable specification will be a refinement of the requirement specification.

There have been many projects in the field of deductive software development, providing semantics, calculi and refinement relations for the deductive software development process, especially for sequential software systems [BBB+85, BJ95]. For interactive and distributed systems the methods are in a more theoretical form, i.e. their description is less detailed (see Section 1.5).

There are two main approaches to formal software development: *transformational* and *deductive* software development. Both have a deductive software development basis, but the process of the formal development is different: In transformational software development (for example in the CIP-Project [BBB+85]) correct transformation rules are applied to a requirement specification until the development is finished. The correctness of the rules

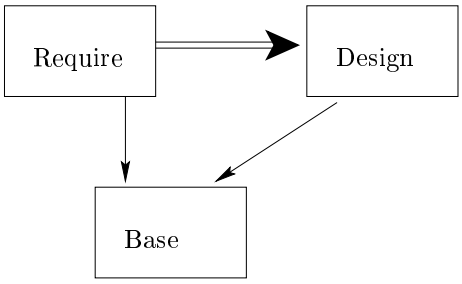

Figure 1.1: KorSo Development Graph

has to be proved before they are applied to a development situation. In deductive software development (as in many parts of the KorSo project [BJ95]) the developer proves the correctness after each development step. In order to simplify these proofs, it is allowed, and desired to reuse some general theorems. Deductive developments allow us to delay correctness proofs[3]. We will view transformational software development as a special case of deductive software development and we will not use transformation rules in our notations.

1.2.3 Graphical Description Techniques

As in traditional software development graphical description techniques are an important part of the deductive software development process. In the deductive software development graphical description techniques have a fixed formal semantics, i.e. they are only nice abbreviations for complex formulas. In this work we will use only two different graphical description techniques:

- (KorSo) development graphs, and
- (Focus) system diagrams.

Development graphs are used to represent the logical structure of the specifications and their development. System diagrams show the structure (architecture) of the system.

Development Graphs

The development graphs we use here were introduced in the KorSo project [PW95, BW95]. One result of the KorSo project was the method for software development. A graphical representation for specifications was used to visualize the development. This representation was called (KorSo) *development graphs*. These graphs have specifications as nodes (depicted as boxes), and two important relations:

[3]Proofs should not be delayed too long, since they may reveal errors.

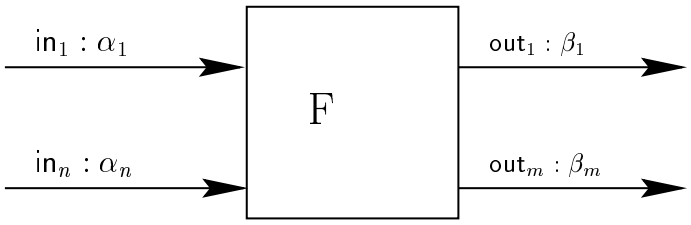

Figure 1.2: Black Box System Diagram

- The semantic refinement relation $\models\!\!\!\approx\!\!\!>$, called `is_refinement_of` in KorSo, and depicted as double arrows (reverse implication arrows[4]), and
- a syntactic structure relation, called `is_based_on` and depicted as simple arrows[5].

A simple example is given in Figure 1.1. It shows the refinement from a requirement specification `Require`, based on a module `Base` by a design specification `Design`, which is also based on `Base`. The specifications `Require` and `Design` are both syntactic extensions of the specification `Base`.

System Diagrams

In the Focus method [BDD+92, Bro93] system diagrams are used to visualize the structure of distributed systems, or components of a system. System diagrams are hierarchical, i.e. every component of the system may be described by another system diagram. An distributed, interactive system (or a component), receives messages on input channels and sends messages on output channels. Focus system diagrams may be used in two different specification styles:

- as *black box specifications*, and
- as *glass box specifications*

The black box specifications graphically describe the interface of a system. Black box specifications contain the name of the system, the names of the input and output channels, and the types of the messages. The specifications of the system describes the behaviour of system. The system diagram in Figure 1.2 specifies the interface of a system `F`. It has n

[4] In KorSo $M[\![A]\!] \models\!\!\!\approx\!\!\!> M[\![C]\!]$ has been model inclusion $M[\![C]\!] \subseteq M[\![A]\!]$ and $A \vdash\!\!\!\rightarrow C$ has been logical implication: $C \Longrightarrow A$

[5] In the KorSo specification language Spectrum [BFG+93a] `is_based_on` was used to denote the syntactic enrichment of specifications.

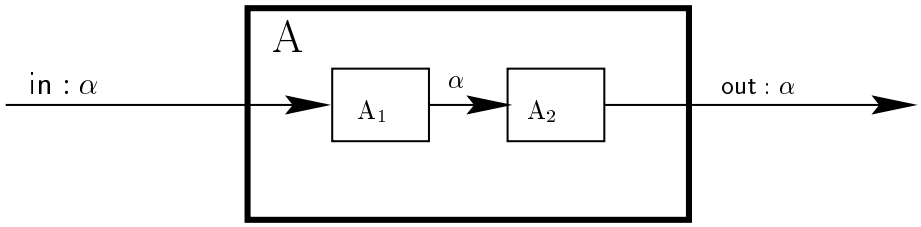

Figure 1.3: Glass Box System Diagram

input channels, named $\mathtt{in}_i$ of type α_i and m output channels named $\mathtt{out}_j$ of type β_j[6].

The glass box specifications contain the same information as the black box specification and in addition they show of which components a system consists and how they communicate. This is a special form to describe the behaviour of a system. The system diagram in Figure 1.3 specifies the interfaces and the structure of system $\mathtt{A}$. It consists of two subsystems $\mathtt{A}_1$ and $\mathtt{A}_2$ communicating over an internal channel of type α. Because we have directed (and typed) channels, a restriction of the communication within a glass box specification is that the types of the channels have to fit and that no input channel is connected with another input channel and no output channel is connected with another output channel. Glass box specifications allow us to describe recursive systems.

We use only system diagrams, development graphs and textual specifications in our development process. There are many other useful graphical description techniques (see Section 1.2.1), but since we focus on a refinement relation these diagrams suffice[7].

1.2.4 Development of Interactive and Distributed Systems

In interactive and distributed systems communication is an important aspect. We decided to work with an asynchronous communication model that models every communication with a channel. Interactive systems communicate by sending messages over channels.

The term *interactive system* refers to the fact that the system has interaction with its environment. A *distributed system* is a system which may consist of several components, which can be at different places. The components of a distributed system communicate by sending messages over channels. A distributed system with communication between the components is therefore also an interactive system and hence interactive systems enclose distributed systems. Since both kinds of systems communicate over channels we will not

[6] To simplify formulas, we omit the ranges from indices if they are clear from the context. In this case they are: $1 \leq i \leq n$ and $1 \leq j \leq m$.

[7] Our refinement relations can be used as basis for defining graphical refinement steps for further graphical description techniques of interactive systems that have a semantics in HOLCF.

further differentiate between these systems and use the terms interactive and distributed synonymously.

The development of distributed systems encloses the development of sequential systems. Therefore, the refinement requirements for the refinement relation $\models\!\!\approx\!\!>$ should also support development of sequential programs. The requirements for the development of sequential systems are mainly those, arising from the implementation of ADTs (see Section 1.3). This section describes the different situations which are typical for the development of interactive systems. The refinement relation $\models\!\!\approx\!\!>$ has to be able to express all these steps as refinements. Concrete methods are needed to support the refinement of these situations with the calculus $\vdash\!\rightsquigarrow$.

We assume that we are working with a deductive development basis ($\mathcal{L}$, M, $\models\!\!\approx\!\!>$, $\vdash\!\rightsquigarrow$). We call the specification of the abstract system $\mathtt{A} \in \mathcal{L}$ and the specification of the designed system $\mathtt{C} \in \mathcal{L}$. The requirements for the refinement relation are that it has to be possible to support every development situation. Of course whether $M[\![\mathtt{A}]\!] \models\!\!\approx\!\!> M[\![\mathtt{C}]\!]$ holds or not in the concrete development depends on the specifications A and C. In Chapter 6 we will give concrete methods for refining the specifications, arising in the development situations of this section.

We give a short overview of the different types of development situations. A similar classification is in [Bro93]. As was mentioned in Section 1.2.3 we use two forms of system diagrams to specify the structure of interactive systems: black box specifications and glass box specifications. Therefore, we have two groups of development situations in the development of distributed systems.

Behavioural Development

In the development of interactive systems one typical situation is that an arbitrary system A, (possibly specified as a black box) is developed into another system C with the same interface by giving a more concrete specification of the system. The specification of the refined system C has all behaviours specified for the abstract system A. This situation is called behavioural development. One example is the development of a component with a sorting algorithm out of its requirement specification. Many development steps are behavioural[8]. Behavioural development is quite well known. There exists a lot of functional refinement techniques to support behavioural developments[9]. We will focus more on other development steps and try to express them as special behavioural developments.

Communication Channel Development

In the development of distributed systems one typical situation is that we develop one single message channel. This situation is independent from the structure of components

[8] And a lot of different development steps will be reduced to behavioural development.

[9] Therefore we call behavioural development sometimes functional development.

and hence also a black box development situation. The messages on the two channels[10] are isomorphic in this situation and therefore, they may be easily translated into each other. Looking closer at these translations we can distinguish two kinds of translations:

- schematic translations, and
- individual translations.

In schematic translations every message of one channel corresponds exactly to one message of the other channel. The translations between the two channels can be constructed by applying element-wise translations to all messages of the channels. With individual translations we may transform every single message on the abstract channel into many concrete messages, possibly on many concrete channels.

An example for a schematic communication channel development situation is the development of a character channel into a byte channel. The messages in the channels are isomorphic. An example for an individual communication channel development is the development of one byte channel by eight parallel bit channels.

Restricted Communication Channel Development

A more general situation is that we have two non-isomorphic channels. A possible reason is that not all values in one channel are used. Therefore, we need to formulate restrictions on the values of a channel.

Again we have schematic and individual translations. In the schematic restricted communication channel development the restriction for the channel is just a conjunction of the restrictions of all single messages of the channel. In individual restricted communication channel development arbitrary restrictions are possible. An example of an individual restricted communication channel development is the development of a byte channel into a sequential bit channel. In this case the restriction for the bit channel is that the number of elements in the channel is divideable by eight (see page 193).

Interface Simulation

Another situation is the development of an arbitrary black box specification with simulations translating the inputs from the abstract specification into inputs for the designed system and translating the output of the designed system back[11]. The idea is that the translations and the concrete system simulate the abstract system.

[10] These two channels refer to two specifications of one channel: one abstract requirement specification and one concrete design specification.

[11] This corresponds to U simulation in [Bro93], and Section 6.3.

Again we have schematic and individual interface simulations. An example for an individual interface simulation is a protocol where strings are translated into packets, then the packets are sent over a packet channel and then the packets are put together to a string. This protocol simulates a string communication on the abstract level (see page 213).

Structural Development

Structural development is the border between black box development situations and glass box situations. It develops a black box specification into a glass box specification by defining the internal structure of the system (sometimes also called architecture). An example is a development where a component is defined as a network.

State Development

In glass bock specifications states are an important technique to specify systems, especially in abstract specifications. Many software development methods use graphical description techniques to specify state dependent components. We will call state dependent components *automata*. Automata have with state transition diagrams nice graphical representations (for example in [LT89, Har87]).

In the development of state-based systems it is often required to add or to remove state transitions from an automaton. Adding and removing of *state transitions* can be treated with behavioural development techniques, since it does not change the interface of the automaton[12].

Adding or removing *states* of automata changes the signature of the automata[13], and, therefore, corresponds to the development of the ADTs of the states. These situations can be developed by our method for the implementation of ADTs.

An interesting question is to relate two state-based specifications of the same components with different state spaces, and to find out whether they specify the same behaviour.

State Elimination

In sequential programs states may be explicitly used as data types. We regard states in interactive systems as abstract views of input histories, i.e. those messages on a channel, which arrived until now. In the development we have a situation where we need to eliminate

[12] Adding a new state transitions, adds new behaviour to the systems, which is a trivial refinement step, but for the elimination of state transitions the context has to be analyzed and it has to be proved that the removed state transition was superfluous.

[13] We will use functions from states to components to specify of state-based components.

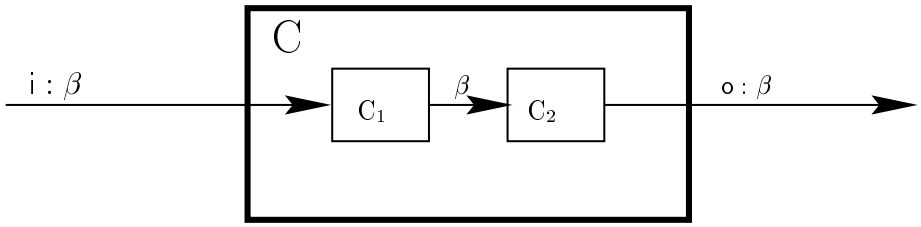

Figure 1.4: Developed System Diagram

states[14]. If we use executable simulations for state elimination we are able to execute the abstract automaton.

State elimination removes the states from the specification of the system. It is a development step between two different description styles of system, and, therefore it is different from removing a single state from the state space of a description, which is described in the previous section.

The introduction of states will not be treated separately, since it can be done by defining states as predicates on the input histories (see for example [Spi94]).

Dialog Development

Dialog development is a situation, where (at least) two components, communicating over (at least) one channel are developed together. This can only be done if the abstract system and the concrete system are specified by glass box specifications. Consider for example the system $\mathtt{A}$ in Figure 1.3 as abstract requirement specification and the system $\mathtt{C}$ in Figure 1.4 as designed specification where $\mathtt{C}_i$ is a refinement of $\mathtt{A}_i$. The general dialog development may also be regarded as interface simulation, but the relation between $\mathtt{A}_i$ and $\mathtt{C}_i$ is more concrete: $\mathtt{C}_i$ is called downward simulation[15] of $\mathtt{A}_i$ if the result of applying the downward translation before $\mathtt{C}_i$ is a development of the composition of first applying $\mathtt{A}_i$ and then doing the downward transformation.

Again we have schematic and individual dialog developments, depending on the form of the translations.

Dialog development is the behavioural development of glass box specifications. It allows us to use restrictions arising from the environment of a component. In our example we may develop the internal channel between $\mathtt{A}_1$ and $\mathtt{A}_2$ into the internal channel between $\mathtt{C}_1$ and $\mathtt{C}_2$ with a restricted communication channel development, since we know that all values of the internal channel are sent by $\mathtt{C}_1$.

[14]The elimination of states comes from our FOCUS view of interactive systems. We regard systems as (stateless) functions processing streams of messages (see Chapter 6).

[15]See page 181 for a formal definition of downward simulation.

An important requirement for the translations, used in many development situations, is their executability. For a logical embedding (for example an isomorphism) no executable translations are needed. With executable translations we can extend our (executable) system model from stream processing functions (see Chapter 6) to more abstract models. For example an executable elimination of states allows us to simulate automata on the basis of messages in a system.

Our goal is to have a refinement relation which supports all these development situations. In Chapter 6 we will define concrete methods for every development situation presented in this section.

1.3 Implementation of Abstract Data Types

This section gives an idea why the implementation of ADTs is an important part of the implementation of interactive systems. If we are able to implement ADTs in a deductive development basis which is adequate for the development of interactive systems, then we will be able to give concrete methods for the development situations of Section 1.2.4. Therefore, the implementation of ADTs is a central topic in this work.

First the concept of ADTs is presented together with two different refinement steps: functional refinement and the implementation step. The implementation step is informally presented in Section 1.3.2. Section 1.3.3 illuminates this presentation by presenting an implementation step of sets by sequences in a simple equational logic.

1.3.1 Abstract Data Types

ADTs are a well known concept [Hoa69, EM85, EM90, EGL89, Wir90] and an accepted basis for the deductive software development process. The basic idea of abstract data types is to describe data types *abstractly* (without referring to the later implementation) by specifying the basic functions operating on them. In first approaches initial semantics [EM85] were chosen, but since loose semantics [BW88b] leave more room for implementations they fit better to the deductive software development process.

A loose specification characterizes the behaviour of the functions, without determining the programs for their realization. Therefore, an ADT may have different *implementations*. Algebraic specifications [EGL89, Wir90] use axioms, written in a certain logic to specify ADTs. The most popular logic for ADTs is equational logic. It allows us to write universally quantified, positive conditional axioms to specify the functions of an ADT. A notion of refinement is needed to make ADTs suitable for the deductive software development process.

There are two different development steps for ADTs. Both should be supported by the refinement relation $\models\!\!\approx\!\!>$.

- *functional refinement* (also called property refinement or behavioural refinement) and the
- *implementation step* (also called data refinement).

Functional refinement does not change the signature of a function. In functional refinement the specification of a function is developed into a more concrete specification. With this refinement step functions, described by some axioms that characterize their behaviour quite abstractly, may be refined by more concrete functions. This refinement step can be repeated until the functions are executable in a certain programming language. Since refinement is transitive, the executable function is a refinement of the first required function. This topic is very well understood in formal methods literature, and will not be elaborated further in this work.

The other step in the development is the implementation step. Of course this refinement should also be compatible with functional refinement. Compatibility is ensured if the refinement relation is transitive and includes also the implementation step. More details on the implementation step are presented in the following section.

1.3.2 Implementation Step

In the development of ADTs the implementation step allows us to replace one type in the ADT by another. The implementation provides methods to define the functions working on the replaced type by functions working with the new type. The goal of the implementation of ADTs is to hide the representation of the data type and to characterize the type only by axioms for the functions working on it.

Depending on the relation between the replaced type and the new type we consider two different forms of implementation, which of course may also be combined.

- a restriction step, which builds a subtype and
- a quotient step, which allows us to identify different elements.

Implementing an abstract ADT with type A with a restriction step by a concrete ADT with type C means that A is isomorphic to a subtype of C (a subset of values of type C). The subtype is characterized by a restriction predicate.

Implementing an abstract ADT with type A with a quotient step by a concrete ADT with type C means that A is isomorphic to a quotient of C (a set of equivalent values of type C). The quotient is characterized by an observable equivalence relation.

The implementation step is split into these two steps, because they may occur independently in the development process and because it is easier to treat them one by one. In the literature many implementations cover both steps (see for example [EKMP82]). The concepts of the implementation of ADTs go back to [Hoa72].

To implement an abstract ADT we will define a sort implementation which relates the abstract sort to the concrete sort. A constant implementation is defined to relate the operations of the abstract ADT to the corresponding operations. These terms become clear in the following example.

1.3.3 Example: Sets by Sequences

In this section the implementation is presented in an equational logic framework on the example of implementing sets by sequences. Equational logic is chosen for two reasons:

- it is easier to get into the topic (compared to a higher order logic),
- the idea of using an explicit abstraction function is not new in the equational framework (see eg. [PBDD95]), but has not been totally worked out, especially the explicit proof obligations to ensure the correctness were missing.

Refining sets by sequences is a good example since it is small and contains all relevant details like subtypes and quotients. The refinement proofs of the example and the general description of the method are in [Slo95]. The method is integrated into the CSDM system [Sta94].

The chosen representation of sets by sequences is the following:

- Every sequence without duplicates represents a set.
- Two sequences represent the same set if they contain the same elements.

For the notation of signatures and axioms the syntax of the specification language SPECTRUM [BFG+93a] is used since it supports the description of axioms in a convenient way. For example the definedness of a function δ`(x)`$\Longrightarrow\delta$`(f(x))` is abbreviated by "`f total;`". The strictness of some functions (`f(`$\perp$`)`$=\perp$) is a requirement for specific functional programming languages (like ML, see [Pau92]).

Sets:

The abstract sort `Set` is specified polymorphically[16]. Axioms {`set4`} and {`set5`} require some sets to be identified. Sets are generated by `empty` and `add`.

[16] Sort classes to control the polymorphism are omitted here for readability.

```
SET = { strict;                              -- all functions are strict
   sort Set α;                               -- sort declaration
   empty: Set α;                             -- constant signature
   add: α × Set α → Set α;                   -- function signatures
   has: α × Set α → Bool;
   has total; add total;                     -- attributes
   Set α generated by empty, add;            -- generation constraint
   axioms ∀ x,y:α, s:Set α in                -- SET axioms:
      {set1}    ¬(has(x,empty));
      {set2}    has(x,add(x,s));
      {set3}    ¬(x=y) ⟹ has(x,add(y,s)) = has(x,s);
      {set4}    add(x,add(x,s)) = add(x,s);
      {set5}    add(x,add(y,s)) = add(y,add(x,s));
   endaxioms;
}
```

Sequences:

The concrete sort is `Seq`. The `data` construct defines the sequences together with constructor functions and some implicit axioms which make the definition equivalent to the `datatype` declaration of ML.

```
SEQ = { strict;
        data Seq α = eseq                    -- executable sort
                   | cons(α,Seq α);          -- with induction rule
        isin: α × Seq α → Bool;              -- function signatures
        subset: Seq α × Seq α → Bool;
        axioms ∀ x,y:α, p,q:Seq α in         -- SEQ axioms:
        {isin1} isin(x,eseq) = false;
        {isin2} isin(x,cons(y,q)) =
                if x=y then true else isin(x,q) endif;
        {subs1} subset(eseq,q) = true;
        {subs2} subset(cons(x,p),q) =
                if isin(x,q) then subset(p,q) else false endif;
        endaxioms;
}
```

After the selection of `SET` and `SEQ` the following specifications are used to represent the development of the implementation step (see Figure 1.5). The combination of an `is_refinement_of` arrow with an `is_based_on` arrow is the graphical form in which proof obligations are depicted in the development graph. In SPECTRUM the semantic refinement relation $M[\![A]\!] \models\!\!\Rightarrow M[\![C]\!]$ is model inclusion $M[\![C]\!] \subseteq M[\![A]\!]$ and $A \vdash\!\!\rightarrow C$ is logical

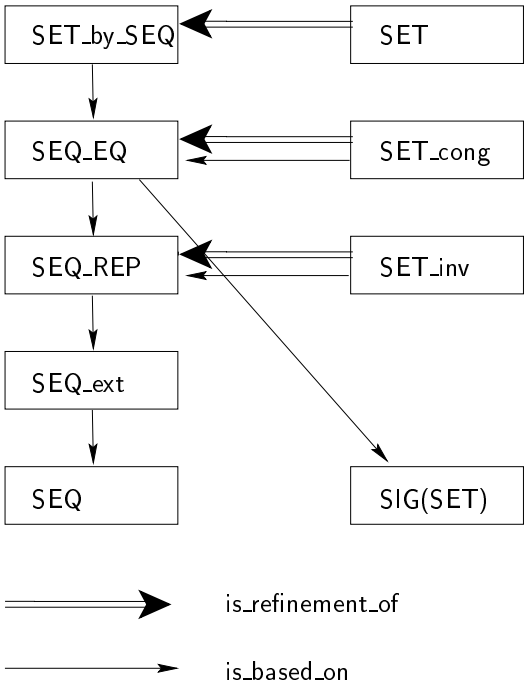

Figure 1.5: Development Graph for Sets by Sequences

implication ($C \Longrightarrow A$). For example, on the basis of `SEQ_EQ` the axioms of `SET_cong` have to be derived (`SEQ_EQ` is a refinement of `SET_cong`).

In the following, the contents of these specifications are listed, together with the necessary user[17] inputs for the definitions of the corresponding functions, the restriction predicate, and the congruence.

Extension of Sequences for Sets:

In this specification the corresponding functions are specified.

```
SEQ_ext = { enriches SEQ;
-- Step 0 (scheme): Define corresponding functions:
        empty_x: Seq α;
        add_x: α × Seq α → Seq α;      add_x strict;
        has_x: α × Seq α → Bool;       has_x strict;
-- Step 0 (user): Specify the functions:
        axioms ∀ x:α, q:Seq α in
```

[17] The term *user* refers to the user of the method.

```
        {construct_1}   empty_x = eseq;
        {construct_2}   add_x(x,q) =
                        if isin(x,q) then q else cons(x,q) endif;
        {function_1}    has_x(x,q) = isin(x,q);
        endaxioms;
}
```

Restriction of Sequences for Sets:

The representation predicate is modelled by a function `is_Set:Seq` α $\rightarrow$ `Bool;`

```
SEQ_REP = { enriches SEQ_ext;
-- Step 1 (scheme): Add the representation predicate:
        is_Set:Seq α → Bool;          is_Set strict total;
-- Step 2 (user): Specify the representation predicate::
        axioms ∀ x:α, q:Seq α in
        {rep1}  is_Set(eseq);
        {rep2}  is_Set(cons(x,q)) = (¬isin(x,q) ∧ is_Set(q));
        endaxioms;
}
```

Homomorphism and Equality:

The homomorphism is modelled by a partial function `abs:Seq` α $\rightarrow$ `Set` α, whereas the congruence relation for the equality on sets is defined as a partial function `eq_Set:Seq` α $\times$ `Seq` α $\rightarrow$ `Bool` . Both have to be defined only for elements that fulfil the restriction predicate `is_Set`.

```
SEQ_EQ = { enriches SEQ_REP + SIG(SET);
-- Step 3 (scheme): Define the (partial) homomorphism:
        abs: Seq α → Set α; abs strict;
-- Step 4 (scheme):Give the source of the partial homomorphism abs:
        axioms ∀ q: Seq α in
        {partial}       δ(abs(q))=is_Set(q);
        endaxioms;
-- Step 5 (scheme):Define the homomorphism:
        axioms ∀ x:α, q:Seq α in
        {hom1}                  empty = abs(empty_x);
        {hom2}  is_Set(q) ⟹    add(x,abs(q)) = abs(add_x(x,q));
        {hom3}  is_Set(q) ⟹    has(x,abs(q)) = has_x(x,q);
        endaxioms;
```

```
-- Step 6 (scheme): Add a congruence predicate:
        eq_Set: Seq α × Seq α → Bool; eq_Set strict;
-- Step 7 (user): Specify the congruence predicate:
        axioms ∀ p,q:Seq α in
        {eq} is_Set(p) ∧ is_Set(q) ⟹
             eq_Set(p,q)=(subset(p,q) ∧ subset(q,p));
        endaxioms;
}
```

Generate Sets by Sequences:

This specification is the most important one: it instantiates the equality to the congruence and adds the new generation principle.

```
SET_by_SEQ = { enriches SEQ_EQ;
-- Step 8 (scheme): axiom to instantiate equality to the congruence:
        axioms ∀ p,q:Seq α in
        {instant}       is_Set(p) ∧ is_Set(q) ⟹
                        (abs(p)=abs(q)) = eq_Set(p,q);
        endaxioms;
-- Step 9 (scheme): Add the generation principle for sets:
        Set α generated by abs;
}
```

Invariance of the Restriction Predicate:

This specification contains all proof obligations for the invariance of the restriction predicate. It is required that all corresponding functions are preserving the predicate `is_Set`.

```
SET_inv = { enriches SEQ_REP;
-- invariance of the corresponding functions (proof obligation 2)
        axioms ∀ x:α, q:Seq α in
        {invar1}        is_Set(empty_x);
        {invar2}        is_Set(q) ⟹ is_Set(add_x(x,q));
        endaxioms;
}
```

Congruence of the Equality for the Corresponding Functions:

This specification contains all proof obligations for the congruence of the equality for the corresponding functions.

```
SET_cong = { enriches SEQ_EQ;
-- the axioms for a congruence (proof obligation 3)
        axioms ∀ x:α, p,q,r:Seq α in
-- equivalence relation
        {refl}          is_Set(p) ⟹ eq_Set(p,p);
        {sym}           is_Set(p) ∧ is_Set(q) ⟹
                           eq_Set(p,q) = eq_Set(q,p);
        {trans}         is_Set(p) ∧ is_Set(q) ∧ is_Set(r) ∧
                        eq_Set(p,q) ∧ eq_Set(q,r) ⟹
                           eq_Set(p,r);
-- substitutivity for observable functions
        {cong_add_x}    is_Set(p) ∧ is_Set(q) ∧ eq_Set(p,q) ⟹
                           eq_Set(add_x(x,p),add_x(x,q));
        {cong_has_x}    is_Set(p) ∧ is_Set(q) ∧ eq_Set(p,q) ⟹
                           has_x(x,p) = has_x(x,q);
        endaxioms;
}
```

Here we use an explicit axiomatization of the congruence `eq_Set`. We will define a predicate `is_Cobs` on page 128 to express this in a more elegant way.

Proofs:

All `is_refinement_of` relations of the development graph have to be proved by theory inclusion of all axioms.

1. `SET` $\hookrightarrow$ `SET_by_SEQ`
2. `SET_inv` $\hookrightarrow$ `SEQ_REP`
3. `SET_cong` $\hookrightarrow$ `SEQ_EQ`

The consistency of `is_Set` and the executability of `eq_Set` can (automatically) be seen from its syntactic form. The proofs of this example have been carried out with the Isabelle proof system [Pau94b] in [Slo95]. All axioms of the three proof obligations were proved. The structure of the specifications provides a structure for the proofs. Most proofs are simple rewrite proofs. Some proofs were done by induction on sets. Therefore, it has been helpful to first deduce an induction rule for sets (`Set` α `generated by empty, add;`) from the axiom `Set` α `generated by abs;` which allows induction on the constructors `empty` and `add` instead of `abs`.

In the example the implementation is fixed by:

- A sort implementation: `Set` $\leadsto$ `Seq`
- Constant implementations:
  - `empty` $\leadsto$ `eseq`
  - `add` $\leadsto$ `add_x`
  - `has` $\leadsto$ `isin`
- A restriction predicate: `is_Set`
- A congruence: `eq_Set`

The equivalence relation has to be a congruence. This means that it has to be substitutive for all observable functions. In the example `abs` is not an observable function, but `add` and `has` are observable. For code generation, this means that `abs` must not be exported. Hidden functions allow multiple concrete representations for the same abstract element.

Since equational logic (with arbitrary types) does not suffice for the development of interactive systems[18] we need a more expressive logic. HOLCF is expressive enough, but has no appropriate notion of refinement to cover the implementation step.

1.4 Refinement Requirements

This section defines the requirements for the refinement relation $\mid\!\approx\!>$ of our deductive software development basis (see Definition 1.2.1). Furthermore, it is explained why we use HOLCF as logic and how the requirements may be specialized for HOLCF.

The method for the implementation of interactive systems is a collection of different methods, one for every development situation. All methods are based on the refinement relation $\mid\!\approx\!>$ and have to fulfil the requirements.

1.4.1 Consistency

Consistency (see Definition 1.2.2) is a very important aspect in the deductive software development process. Therefore, one requirement to the refinement relation $\mid\!\approx\!>$ is that it should be consistency preserving.

[18] Of course we may also specify the systems in HOLCF only with equations, but we have types with *cpo* structures and continuous functions in HOLCF, which ensure the existence of least fixed points for the adequate description of components with feedback.

Given an arbitrary requirement specification of a system, we do not know whether it is consistent. If it is inconsistent, the system specification is wrong and the system cannot be implemented. It is difficult to ensure the existence of a model at the beginning of the development[19].

Definition 1.4.1 ***Consistency Preserving***

If ($\mathcal{L}$, M, $\mid\!\approx\!\!>$, $\mid\!\sim$) is a deductive software development basis, then $\mid\!\approx\!\!>$ is called *consistency preserving*, iff

- for all $\mathtt{A},\mathtt{C} \in \mathcal{L}$ with $\mathtt{A} \mid\!\approx\!\!> \mathtt{C}$ and $\mathtt{C}$ is consistent implies that $\mathtt{A}$ is also consistent.

Our way to ensure the consistency of the requirement specification is to require $\mid\!\approx\!\!>$ to be consistency preserving.

Note that this property does not prevent us from refining a consistent specification by an inconsistent one (for example by adding a wrong axiom), but the property ensures that if we arrive, during the deductive development, at an executable specification, then our requirement specification will also be consistent. To summarize transitivity of $\mid\!\approx\!\!>$ ensures that the executable specification is a refinement of the requirement specification and a consistency preserving $\mid\!\approx\!\!>$ ensures that the requirement specification is consistent.

Of course, we prefer refinements that cannot introduce inconsistencies (like conservative extensions in Section 2.1.4) and we require from our methods that they do not introduce inconsistencies.

1.4.2 Modularity

For the development of large systems a modular development has to be supported, such that many teams can work simultaneously on different parts of the system. This leads to an additional requirement for our refinement relation $\mid\!\approx\!\!>$.

The idea of modularity is to allow the refinement of arbitrary modules of a specification and to require that the specification with the refined module is a refinement of the original specification. The modules are syntactic parts of the specification and do not depend on the system structure. A module has to be a specification. The module structure is not necessarily related with the structure of the system. For example two (distributed) components can both base on the same specification module. The module of strings is used in many specifications.

[19] The initial approach to specification assigns a model to every specification, but in the case that $true = false$ this model is useless.

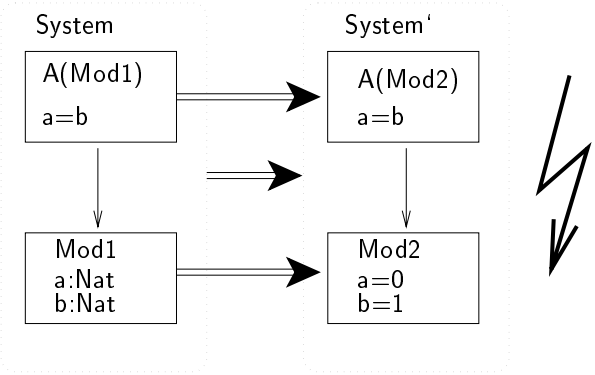

Figure 1.6: Development graph with Modules

Definition 1.4.2 ***Modularity***

Let $(\mathcal{L}, M, \models\!\!\approx\!\!\gg, \vdash\!\!\sim)$ be a deductive software development basis, then $\models\!\!\approx\!\!\gg$ is called *modular*, if

- for all $S, T \in \mathcal{L}$ with $M[\![S]\!] \models\!\!\approx\!\!\gg M[\![T]\!]$ and for all modular specifications $A(x)$[20] $\in \mathcal{L}$ the following holds: $M[\![A(S)]\!] \models\!\!\approx\!\!\gg M[\![A(T)]\!]$.
- If $M[\![A(S)]\!]$ and $M[\![T]\!]$ are consistent, then $M[\![A(T)]\!]$ is consistent.

A modular specification $A(x)$ is a specification based on x. It may also be regarded as a parametrized specification (see [EGL89]).

The development graphs of Section 1.2.3 are used to present the module structure of the specification.

Modularity is a very strong requirement, since it requires to split a specification into arbitrary modules, to develop them independently and then the result of the combination of the modules should be a refinement of the splitted specification. Modularity includes consistency. This may lead to troubles. Assuming we have the possibility to express equations in $\mathcal{L}$, then we may look at the example in Figure 1.6. It shows a system `System` consisting of a modular specification `A(Mod1)` using the module `Mod1`. This module is refined by the module `Mod2`, and the system `A(Mod2)` is a refinement of `A(Mod1)`, The problem in this example is that even if `Mod1`, `Mod2` and `A(Mod1)` are consistent, `A(Mod2)` is not.

[20] We chose the notation $A(x)$ as generalization of all constructs, which allows us to structure specifications. For example in SPECTRUM [BFG+93a] one possibility to express that the specification A textually includes the specification $x \in \mathcal{L}$ is `A = { enriches x; ..}`. In Isabelle we have the + operation to structure specifications.

There are two possibilities to weaken the requirements of modularity:

- Restrict the specification language $\mathcal{L}$, for example by excluding equations.
- Restrict the form of the Modules $A(x)$, for example by allowing only specific composition operators.

Conservative extensions (see Section 2.1.4) are a restricted specification language and ensure conservativity for arbitrary modules and, therefore, support simultaneous development of large systems.

Modularity is called horizontal compositionality in [CZdR92], whereas vertical compositionality means transitivity. If a refinement relation is horizontally compositional, the development of the modules may be done independently and in parallel. In FOCUS the emphasis lies on the composition of interactive systems from components. The composition operators are sequential (;), parallel (||) and feedback (μ) composition. Therefore, the form of modularity (from Definition 1.4.2) is not needed in FOCUS and compositionality with respect to the main composition operators suffices. We call compositionality with respect to arbitrary structuring operators modularity and compositionality with respect to ;, || and μ compositionality (see Definition 6.1.7).

1.4.3 Development

In addition to consistency and modularity (or compositionality) we require that $\mid\!\approx\!>$ supports the development of interactive systems in all possible development situations. This includes the possibility to express the following development steps as refinements:

- all development situations in Section 1.2.4,
- implementation of ADTs,
- functional refinement, and

for our methods we require that we have

- executable constructions or code generation.

Of course we require $\mid\!\approx\!>$ to cover all development situations. This includes the implementation of ADTs (see Section 1.2.4) and functional refinement. The requirement of executability or code generation is practically relevant. Here we see the influence of practical requirements on theory: Theoretically a semantic embedding would suffice, for example the semantic definition of automata in terms of stream processing functions is a sufficient foundation for proving some properties about automata, but an executable translation, and an executable translation back from stream processing functions representing automata to automata allows us to construct tools that simulate automata on distributed systems.

1.4.4 Logic and Language

This section describes the requirements for the logic used and the specification language and it shortly explains why we decided to use HOLCF.

The requirements for the logic M and the language $\mathcal{L}$ we use in the deductive software development process are simply that it should be possible to express all development situations as refinements and to formulate all proof obligations in the language. Therefore, the requirements are:

- Express requirement, design and executable specification in the same logic,
- specify interactive systems with feedback channels, and
- support an adequate characterization of executability.

Having the same logic and language for requirements, design and executable specification preserves us from using and learning several languages and from worrying about the correctness of the changes between the languages and logics.

In the deductive software development we need to be able to characterize requirement specifications of interactive systems. If we would restrict us to operational semantics, then we would have a very concrete view of systems. Using denotational semantics gives us more flexibility by allowing abstract behavioural specifications of systems [Hoa96].

The second requirement sounds trivial, but our general model of interactive systems requires to express components and communication channels, messages and their types in the language (and to have the corresponding semantic concepts). Since we allowed almost arbitrary composition of components to systems (see Section 1.2.3), we need semantics (and syntax) for feedback composition (components that use their own output channels as input). To model interactive systems with recursive structures functionally, requires to assign semantics to feedback compositions. Fixed points are used to denote their semantics and the underlaying domains should have an order to express that the denoted fixed points is the least one. HOLCF provides fixed points and *cpo* structures. Therefore, we model interactive systems with HOLCF.

Executability needs an adequate characterization in the logic, otherwise our results are not of practical relevance (see Section 1.4.3).

HOLCF is described in Section 2.1. The reader not interested in the technical argumentation why we selected HOLCF may accept our choice and skip the rest of this section.

HOLCF is a higher order logic and offers the possibility to express abstract requirement specifications, more concrete specifications and executable specifications. HOLCF allows us to give denotational semantics to systems. HOLCF also includes executable specifications,

therefore, HOLCF is one logic for abstract requirement specifications and for executable systems. With a powerful refinement relation HOLCF is a good basis for the deductive software development process.

The specification of interactive systems is supported with the lazy data type of streams to represent communication histories and stream processing functions for the components (see Definition 6.1.2 or [SS95]).

The feedback rules are defined with a fixed point construction. To ensure the existence of fixed points the functions are required to be monotone with respect to an order on the domain of streams. HOLCF provides ordered domains.

HOLCF contains LCF, the logic of computable functions and it is therefore, adequate to characterize executable functions, including aspects like partial functions and underspecification. HOLCF has continuous functions, that work on *cpo* structured domains. Fixed points of continuous functions may be approximated by computing Kleene-chains. The domain construct of HOLCF allows us to conservatively introduce free data types with *cpo* structures.

The refinement requirement of executability may be formulated in HOLCF as:

- continuous functions have to be implemented by continuous functions and
- *cpo* structured domains have to be implemented by *cpo* structured domains.

The only disadvantage of HOLCF is that it has no support for introducing subtypes and quotients (needed in the implementation of ADTs and interactive systems) with *cpo* structures and continuous functions on it. Therefore, the main technical part of this work is a conservative definition of such constructs in HOLCF.

1.5 Related Work

A different approach to the correctness of programs is the transformational approach [Bro82, BBB+85, HKB93, KB94]. Its aim is a transformation of the requirement specification to an executable specification by applying correctness preserving transformation rules instead of showing the correctness of every development step deductively. However, transformational development restricts us to use a set of correctness preserving rules. A theorem prover is needed to derive some application conditions of some rules; furthermore, the correctness of every new rule has to be established deductively. On the other hand, we want the deductive software development to have some schemes that may be applied without excessive use of theorem provers, for example for the refinement of automata. The approaches are approaching each other and the more interesting aspect is the logic they use.

For both approaches there exist many specializations for certain classes of systems or programming languages. An important criterion is whether the specification is state-based, which corresponds to imperative programming languages, or whether the system specification is not state based and realized by functional languages. Since imperative programming languages (Algol [Rut67], Pascal [Wir72]) have a longer tradition than functional language (ML[Pau92], Haskell [HJW92]) the first specification languages where developed for those state oriented languages [Hoa69, Hoa72, Bac78, Bac88, MV94] together with appropriate refinement relations even for distributed systems and nondeterminism [Nip86, Nip87, Han80, WB94]. The notion of state is quite complicated in distributed systems and is usually refined by simulation and bisimulation techniques [Mil83]. Even the first papers of refinement in sequential settings deal with variables and states and tackle the initialization of variables for refined systems.

Interactive systems may be described with different semantics. Operational, algebraic and denotational semantics are the most used ones. See [Hoa96] for an overview. Operational semantics are used if the emphasis is on the execution of programs, for example for automata or CCS [Mil83]. If we are interested in algebraic manipulations as transformations of distributed programs, we may use process algebras with algebraic semantics, like CSP [Hoa84], to describe our systems. For the specification of interactive systems denotational semantics are best suited, since they give semantics to recursive predicates describing the observable behaviour of the system (see [SS71] for the foundations). For simple programming languages these semantics can be shown to be equivalent, but for the abstract specifications of requirements denotational semantics are preferable.

Functional languages allow us to model states explicitly. An imperative procedure may be seen as functions on the state with the state as input and output parameter. The refinement calculus [MV94] describes procedures together with read and write access to the state in order to simplify proofs for the non affected variables. This is quite close to the functional view. The possibility of abstracting from states makes functional descriptions an elegant way in modelling distributed systems abstractly. Since in the software development process the most important criterion is to have a compositional refinement relation, some models (for example traces) are enriched with additional structures just to make them compositional. For functional system descriptions those tricks are not necessary. This makes them quite appropriate for the deductive software development process.

Abstract specifications may be quite understandable especially when graphical description techniques are used. This requires them to have formal semantics in terms of HOLCF or HOL (which is a part of HOLCF). HOLCF builds a basis for many description techniques: State automata are formally founded in [NS95], E/R diagrams are defined in [Het95] and structuring diagrams for distributed systems are embedded into HOLCF in [SS95]. Even for Statecharts, a graphical specification widely used in praxis [Har87], there exist formal semantics [NRS96]. With these user oriented description techniques the specification of systems is mathematically founded and user friendly. For the development with such graphical specifications techniques the basic refinement steps have to be applied. They

can either be applied on the level of HOLCF or, more user friendly, there should be some graphical refinements defined, based on the refinements of HOLCF.

[BDD+92] defines several methods for structural refinement that allow us to decompose the system. In [BDD+92, Bro93] the refinement relations of FOCUS are defined in a quite abstract way. Abstraction and representation functions are used, and conditions are given for compositionality. In the deductive software development process these conditions will be put to the developer as *proof obligations*. However, these conditions are very general and hard to prove without a supporting method. This work refines these conditions by some necessary conditions (for example invariance), which guide the developer to design implementations and the proof obligations are helpful lemmata for the abstract proof obligations.

To summarize the related work we can say that there has been a lot of work in the area of the specification of interactive systems and also a lot of work in the area of the implementation of ADTs, but no work combines the advantages of both approaches. In this work the implementation of ADTs provides concrete methods for the implementation of interactive systems, which improve the general methods from FOCUS by some specific results.

1.6 Goals

The technical goals of this work stem from the motivation of giving a practicable method for the implementation of interactive systems on the basis of ADTs in HOLCF. This requires to solve the following tasks:

- Conservative implementation of subtypes with cpo structures and continuous functions.
- Conservative implementation of quotients with cpo structures and continuous functions.
- Give concrete methods for different development situations in the development of interactive systems.

To solve these tasks in a practicable way we decided to work with HOLCF and we have to ensure that:

- *simple* proof obligations are generated to guarantee the correctness of the implementation,
- all proof obligations are expressible in HOLCF,
- methodical help for the proof is provided,

- the methods are constructive, such that code generation is possible,
- restriction predicates and congruences are formulated in HOLCF, and that
- the methods are integrated into the deductive development process.

To ensure that the proof obligations are simple, two different techniques for the definition of refinement relations (theory interpretation and model inclusion with conservative extension) are compared. Technically the most interesting results are:

- A conservative definition of subtypes in LCF and its implementation in Isabelle's logic HOLCF.
- A constructive method for defining coercion functions (between the abstract elements and the corresponding terms) for polymorphic and higher order terms.
- A higher order theory interpretation that preserves continuous functions.
- A conservative definition of quotients in HOLCF, including equivalence classes and methods for the definition of continuous functions.
- A definition of observability in a higher order logic on the basis of partial equivalence relations (PERs), and a flexible class `eq` for the specification of observable congruences.
- Refining FOCUS using ADTs.
- Improving compositionality of downward simulation for invariant functions.
- A simulation of an automaton, based on stream processing functions.

Besides many small and medium sized examples in this work some critical aspects of a WWW server are taken as case study in Chapter 7.

1.7 Structure of the Thesis

The structure of this thesis is influenced by the different implementation steps and by the two applied techniques to define the refinement relation. We decided to compare two techniques, since model inclusion did not fit to our definition of executability, whereas theory interpretation, which is the more general technique seamed to generate more complicated proof obligations. Having compared both methods for the implementation of ADTs explicitly, we know that our method is the best solution for our requirements.

The development of interactive systems requires to have a refinement relation that supports all development situations of Section 1.2.4. These situations include the implementation of the ADTs of messages in the system. The implementation (of ADTs) is studied since [Hoa72] in many, different logics (see [ONS96] for an overview). The implementation consists of two important steps:

- The restriction step which builds a subtype and
- the quotient step which allows us to identify equivalent elements.

Since we decided to use HOLCF for the implementation, we start with the presentation of the parts of HOLCF, which are relevant for this work. The two refinement techniques theory interpretation, and model inclusion with conservative extension are presented in Chapter 2. In Chapter 3 two methods, based on the two different refinement relations (one for model inclusion and one for theory interpretation) for the restriction step are compared, where Chapter 4 does the same comparison for the quotient step.

The reader not interested in a higher order theory interpretation that preserves continuous functions may skip Section 3.2. The credulous reader, believing in the deficiencies of model inclusion in the treatment of multiple representations may skip Section 4.3.

Chapter 5 combines the conservative extension of Section 3.3 with the theory interpretation of Section 4.4 to a compositional method, which is well suited for the implementation of interactive systems. As we mentioned in the previous sections the implementation of interactive systems uses the implementation of ADTs in two different ways: in schematic translations and in individual translations. In schematic translations interactive systems are implemented by applying the implementation of ADTs to every single message. In individual translations the methods for the implementation of ADTs are extended to implement arbitrary systems, but they still use the same concepts of abstraction and representation functions which were already introduced by Hoare.

In Chapter 6 we show that this method for the implementation of ADTs is well suited for the schematic implementation of interactive systems and we extend it to a method for the individual implementations of interactive systems. These methods are applied in the case study of implementing some critical aspects of a WWW server in Chapter 7. This work is the basis for many future research activities, which are described in the concluding Chapter 8. At the end an index and a list of definitions are added to find occurrences of important terms.

Chapter 2

Refinements in HOLCF

In the software development process a coarse requirement specification is developed stepwise towards a realization of the system. The deductive software development process models each correct development step as refinement and requires to prove the correctness of every development step. This avoids incorrect development steps. For this process it is important to have an appropriate refinement relation, which supports the verification of all desired development steps.

We decided to compare two techniques, since model inclusion did not fit to our definition of executability, whereas theory interpretation, which is the more general technique seamed to generate more complicated proof obligations. Having compared both methods for the implementation of ADTs explicitly, we know that our method is the best solution for our requirements. Theory interpretations are needed for the translation of equivalence classes into representing elements. Theory interpretations are described in Section 2.3, model inclusion for HOLCF is defined in Section 2.2.

First, a short introduction to the used parts of HOLCF is given (see [Reg95] for a short overview of HOLCF), and the the specifications of ADTs in HOLCF are presented.

2.1 HOLCF

The logic HOLCF [Reg94] is a conservative extension of the higher order logic HOL [GM93] by the concepts from the logic of computable functions LCF [Pau87]. It is well suited for the specification of interactive systems, since it has fixed points and *cpo* structured domains. Furthermore, its supports the development from abstract requirement specifications to concrete designs. It has a higher order logic for formulating abstract and expressive requirement specifications, but also a continuous function space and a `domain` construct, which are adequate for formulating executable recursive functions. This is the basis for code generation.

This section gives some definitions of HOLCF from [Reg94]. In order to cut down terminology some definitions are simplified at points where the complete formal definitions are not important for this work, for example type classes and type inference. The notions of data type, abstract data type, free data type, and executable data type are also defined for HOLCF. An interesting aspect in the following chapters is the executable implementation of subtypes and quotients, which are non-free data types.

2.1.1 HOLCF Terms

HOLCF terms are HOL terms. In HOLCF there are a lot of additional constants, types, classes and arities, but since they are introduced conservatively it suffices to focus on HOL terms. HOL is a higher order logic and, therefore, we have to signatures (Ω and Σ), one for the type terms, and the other for the terms.

HOL is strongly typed, i.e. every term has a type. Types are type terms over a type signature with type constructors and type variables.

Definition 2.1.1 ***Type Term***

The set of type terms $T_{\Omega(\Xi)}$ over a type signature $\Omega := \{tc_j\}$, a set of type constructors with type variables $\alpha \in \Xi$, is defined by:

- TYP_VAR: $\alpha \in T_{\Omega(\Xi)}$ for $\alpha \in \Xi$
- TYP_APP: $t^i \in T_{\Omega(\Xi)}$ for $1 \leq i \leq n \Rightarrow tc_n(t^1, .., t^n) \in T_{\Omega(\Xi)}$ for $tc_n \in \Omega$

Type constructors have an arity, which we sometimes write as an index. tc_0 are type constants, and n-tuples as $(t_1, .., t_n)$ are abbreviated by $\overline{t_i}$. In Isabelle the arity is declared together with the definition of the type by the `types` statement (see for example page 41)[1].

Some examples of type terms in HOL are:

- `bool, nat` (type constants)
- α `list, nat set` (type terms with postfix constructors of arity 1)
- `nat` $\Rightarrow$ `bool` (type term with infix constructor of arity 2)

The well typed terms in HOLCF are defined as the set of all untyped λ-terms (which in [Reg94] are called raw terms), which are type correct.

[1] In Isabelle there is also an `arities` statement. It is used to declare the type classes of a type constructor or type constants.

Definition 2.1.2 ***Raw Term***

Let Ω be a type signature and let $C = \{c, d, ..\}$ be a set of constants, then (for $\Sigma = (\Omega, C)$) the set $RT_{\Sigma(\Psi)}$ of raw terms with variables from Ψ is the set of untyped λ-terms over Ψ, defined by:

- RTerm_Con: $c \in RT_{\Sigma(\Psi)}$ for $c \in C$
- RTerm_Var: $x \in RT_{\Sigma(\Psi)}$ for $x \in \Psi$
- RTerm_App: $t_1, t_2 \in RT_{\Sigma(\Psi)} \Rightarrow (t_1\ t_2) \in RT_{\Sigma(\Psi)}$
- RTerm_Abs: $t \in RT_{\Sigma(\Psi)} \Rightarrow \lambda x.t \in RT_{\Sigma(\Psi)}$ for $x \in \Psi$

Every raw term t may be annotated by a type term $\tau \in T_{\Omega(\Xi)}$ by $t::\tau$[2].

The set of raw terms is reduced to type correct terms by the following definition.

Definition 2.1.3 ***Term***

The set of terms $T_{\Sigma(\Psi)}$ over a signature $\Sigma = (\Omega, C)$ with variables from Ψ is the type correct subset of $RT_{\Sigma(\Psi)}$, which is defined by:

- for every term $t \in T_{\Sigma(\Psi)}$ there exists a type context Γ and a type term τ with $\Gamma \triangleright t::\tau$.

See [Reg94, Section 2.3] for the definitions of type context and a calculus for the type inference relation $\triangleright$.

For this work an intuitive understanding of type correctness suffices. A classification of polymorphic type systems can be found in [Naz95]. We assume type correctness of the terms, since the implementation of abstract data types makes only sense for type correct ADTs.

Some examples of constants in HOLCF are:

- $\bot$, `TT`, `FF` ::`tr`
- `not`::`bool`$\Rightarrow$`bool`,$\forall$
- $\vee$::`bool`$\Rightarrow$`bool`

`x`=`TT` and `x`=`FF` are abbreviated in the following by $\lceil$`x`$\rceil$ and $\lfloor$`x`$\rfloor$ respectively. In HOLCF the distinction between the types `bool` and `tr` is important. `bool` is two-valued and used in the predicate level to express properties of specifications and operations. `tr` is three-valued and represents the truth values of operations in the specification, which may not

[2] Even typed λ-abstraction is used: $\lambda x::\tau.t$.

terminate (expressed by $\bot$::tr). With $\bot$ partial functions may be modelled (see [MS96] for an overview).

In addition to these real HOLCF constants we use some schemes as abbreviations for concrete constants. This allows us to give more readable definitions for the treatment of data types with arbitrary types and constructors. Instead of writing n-tuples $(x_1, .., x_n)$ or $x_1 \to .. \to x_n$ we also use $\overline{x_i}$ in Isabelle formulas as abbreviation. The following abbreviations are also used:

- $\mathtt{and}_n :: \overline{\mathtt{tr}} \to \mathtt{tr}$ with $\mathtt{and}_n \equiv \Lambda\ \overline{x_i}.\mathtt{and}(x_1, \mathtt{and}(.., x_n)..)$[3]
- $\mathtt{or}_n :: \overline{\mathtt{tr}} \to \mathtt{tr}$ with $\mathtt{or}_n \equiv \Lambda\ \overline{x_i}.\mathtt{or}(x_1, \mathtt{or}(.., x_n)..)$
- $\mathtt{cases}_n :: \overline{\mathtt{tr} \to \alpha} \to \mathtt{tr} \to \alpha$ with $\mathtt{cases}_n \equiv \Lambda\ \overline{\mathtt{cond}_i\ x_i}$.
  If $\mathtt{cond}_1$ then x_1 else If ..
  .. If $\mathtt{cond}_{n-1}$ then x_{n-1} else x_n fi .. fi

2.1.2 HOLCF Type Classes

This section informally describes the type classes of HOLCF and explains their application. For a formal definition of type classes see [Reg94, Section 2].

Type classes are used to control polymorphism. In ML [Pau92] there only exist two type classes (α and $\alpha_=$) to distinguish polymorphic functions that do not permit equality tests from those that do.

As in other type systems (Haskell/Gofer [HJW92, Jon93] or Isabelle [Pau94b]) HOLCF allows type classes to be defined with a subclass hierarchy. Type classes consist of polymorphic characteristic functions and constants, available on every monomorphic type that "belongs" to the class and *characteristic axioms* describing properties of the *characteristic constants*.

The characteristic constants are defined polymorphically, but are only available on types that belong to the class. Therefore, before applying them to a value of a concrete type, it must be assured that this type belongs to the class. This is called *instantiation*. It is done by showing that some terms (mostly constants) on the concrete type satisfy the characteristic axioms of the class. Such proofs are called *witnesses* for the fact that the concrete type belongs to a class. If such a witness exists, the characteristic constants may be instantiated to the concrete constants by defining them to be equal to the term, satisfying the characteristic axioms. The type checker needs an arity declaration to check the new instance correctly[4]. Instantiating a type into a class requires to instantiate it first into all superclasses of the class (see Section 3.3.1 for an example).

[3] Since we have in HOLCF two function spaces, we need two different lambda for the abstractions. We use λ for the full function space and Λ for the continuous function space.

[4] The introduction of new arities is fully treated in [Reg94].

class	subclass of	constants	axioms	
po	term	$\sqsubseteq$	refl_less	x$\sqsubseteq$x
			antisym_less	$[\![$ x$\sqsubseteq$y; y$\sqsubseteq$x $]\!] \Longrightarrow$ x=y
			trans_less	$[\![$ x$\sqsubseteq$y; y$\sqsubseteq$z $]\!] \Longrightarrow$ x$\sqsubseteq$z
pcpo	po	$\bot$	minimal	$\bot\sqsubseteq$x
			cpo	is_chain S$\Longrightarrow\exists$x.range S<<|x

Figure 2.1: Type Classes of HOLCF

Figure 2.1 shows the type classes used in HOLCF. If we declare the type of a polymorphic constant in a type class, we may append the type class, separated by "::" to the type of the constant. For example the type of $\bot$ is $\bot::\alpha::$pcpo. The type class pcpo describes *cpo* structures, and especially the proof of the axiom cpo for subdomains is an interesting aspect in Section 3.3.

Axiomatic type classes [Wen94] allow us to declare the characteristic axioms for a type class and axiomatic type classes provide a syntax for a safe instantiation into type classes. This safe instantiation rule checks the witnesses for the characteristic axioms before it declares the arity to the type checker. Because axiomatic type classes formulate axioms over arbitrary constants, they do not allow us to introduce a characteristic constant for a type class in one step.

Introducing an axiomatic type class with a characteristic constant, available only on this class requires two steps: The first step is to define a general constant, which is available on all polymorphic types of the class term (or any other superclass of the defined class). With this general constant the new type class is axiomatized. The second step is to define the characteristic constant only available on the new class and to define it to be equal to the general constant. With this two step method the type checker tests whether the characteristic constant is applied to a term which resides in the new class.

Example 2.1.1 ***Axiomatic Type Class*** per

Consider the theory PER0 as an example for such a two step definition.

```
PER0 = Set +        (* axclass per with characteristic constant ∼ *)
consts              (* general constant *)
       "∼∼"   :: α::term ⇒ α ⇒ bool (infixl 55)
axclass per < term
                    (* characteristic axioms for per *)
       ax_sym_per       x ∼∼ y ⟹ y ∼∼ x
       ax_trans_per     ⟦x ∼∼ y; y ∼∼ z⟧ ⟹ x ∼∼ z
consts              (* characteristic constant for per *)
       "∼"    :: α::per ⇒ α ⇒ bool (infixl 55)
```

```
defs    ax_per_def      (op ∼):: [α::per,α]⇒bool) ≡ (op ∼∼)
end
```

Since this is our first Isabelle specification, we explain the used syntax briefly: `PER0` is the name of the specification, it bases on the specification `Set` from HOL. `consts` introduces constants and declares their types. `(infixl 55)` assigns a priority to an infix operator. `axclass` is the keyword for the introduction of axiomatic type classes, `per` is the name of the defined class. It is a subclass of the general class `term`. The axiomatic type class is introduced with characteristic axioms, which start with a name and are followed by the rule. `defs` can be used to define operations in Isabelle. `defs` requires to use the definition symbol ≡.

The first step after the definition is to derive the characteristic axioms for the characteristic constant (by a simple simplification). We get the following theorems:

```
sym_per         x ∼ y ⟹ y ∼ x
trans_per       ⟦x ∼ y; y ∼ z⟧ ⟹ x ∼ z
```

From now on all further theorems for `per` use ∼. Only for the definition of PERs on other types we need the general constant ∼∼. For example we define ∼∼ on bool to be the identity by:

```
bool_per        ((op ∼∼):: [bool,bool]⇒bool) ≡ (op =)
```

The safe instantiation of `bool` into the class `per` is performed by:

```
PER = PER0 +
instance            (* proofs of the characteristic axioms *)
        bool :: per     (bool_sym_per,bool_trans_per)
end
```

The `instance` syntax allows us to provide theorems, which are witnesses of the characteristic axioms. Before the above instantiation we proved the following witnesses.

```
bool_sym_per    (x::bool)∼∼y ⟹ y∼∼x
bool_trans_per  ⟦(x::bool)∼∼y;y∼∼z⟧ ⟹ x∼∼z
```

This example shows the conservative definition and instantiation of type classes in the Isabelle system. All proofs are simple and are only mentioned here to show that the witnesses for the instantiation into the class have been proved.

Using axiomatic type classes without this two step declaration has the disadvantage, that type inference sometimes infers a too general type. For example the types of `x` and `y` in `y∼∼x⟹y∼∼x` are inferred to α`::term`. Therefore we can not prove the property, which

we could have proved if `x` and `y` are of type `bool`. Sometimes this might be confusing. Even more confusing is the fact that `-#1=(#0::dnat)` is type correct[5]. Using the two step declaration of characteristic constants would allow the type checker to reject those strange terms. Therefore we use the two step instantiations in this work[6].

The formal semantics of type classes are not used in this work. The informal treatment of type classes leads to a simplification of the semantics, since type classes and resulting restrictions as regularity, coregularity, downward completeness and some others are not mentioned here. See [Reg94] or [Nip91] for the treatment of these aspects.

2.1.3 HOLCF Models

The models of HOLCF in [Reg94] are defined in two steps: first, the models of HOLC (HOL with *c*lasses) are defined; then, HOLCF is syntactically constructed on top of HOLC by introducing new constants, types, classes and arities. Since these introductions are conservative, HOLCF models are constructed along this syntactic extension. This method is called *conservative extension method*. Some examples of conservative extensions can be found in Section 2.1.4.

Thus the models of HOLCF are based on the models of HOLC. A difficult problem in HOLC is the semantic modelling of type classes with characteristic constants. Complex models that include a class structure of models for the characteristic constants are used to define the semantics for HOLC in [Reg94, Section 2.4]. We may omit type classes from the structure of the HOLCF models, since for this work an informal understanding of type classes suffices. And besides it would only be a repetition of [Reg94], and notations are simpler without them. Therefore, this definition of models is based only on the simple type models for type terms.

Definition 2.1.4 ***Simple Type Model***

A (simple) type model $TM = (PU, TC)$ for a type signature Ω consists only of:

- a set PU of nonempty carriers, closed under nonempty subsets, products and total functions ($\mapsto$) and with a choice function ch that chooses an arbitrary element of any carrier (for $X \in PU : ch(X) \in X$).
- The set $TC = \{tc_n^{TM}\}$ of interpretations for all type constructors $tc_n \in \Omega$.

[5] Here $-$ denotes an operation of a class `minus`, in which `dnat` is not instantiated.

[6] A further advantage of the definition with two constants is that it allows us to adopt theories, which are defined without axiomatic type classes (like HOLCF in its current version) easily to axiomatic type classes. The method is to introduce a second constant for every characteristic constant and to derive the old characteristic axioms in the first step. Then all other theorems can be adopted without further change.

Some examples of simple type models in HOLCF are:

- $(PU, \{\mathtt{bool}_0^{TM}\})$ with $\{\mathbb{T}, \mathbb{F}\} \in PU$
- $(PU, \{\mathtt{tr}_0^{TM}\})$ with $\{\mathtt{TT}, \mathtt{FF}, \bot\} \in PU$
- $(PU, \{\Rightarrow_2^{TM}\})$ with for all $A, B \in PU$: $A \mapsto B \in PU$ since PU is closed under $\mapsto$.

To define an interpretation of type terms a type variable assignment $\nu : \Xi \longrightarrow PU$ is needed. With this mapping the interpretation can be defined:

Definition 2.1.5 ***Type Interpretation***

Let TM be a type model for a type signature Ω with type variables Ξ, then the interpretation $TM[\![]\!]_\nu^\Omega$ of a type term $\tau \in T_{\Omega(\Xi)}$ is simply defined by:

- $TM[\![\alpha]\!]_\nu^\Omega = TM[\![\nu(\alpha)]\!]_\nu^\Omega$ for $\alpha \in \Xi$, where $\nu :: \Xi \longrightarrow T_{\Omega(\emptyset)}$ is a variable assignment
- $TM[\![tc_n(\overline{t_i})]\!]_\nu^\Omega = tc_n^{TM}(\overline{TM[\![t_i]\!]_\nu^\Omega})$ for $tc_n \in \Omega$.

Compared to the semantics of type terms in [Reg94] this is a much simpler semantics of type models. In the following we omit the type variable assignment ν in the index, since it is not needed because we treat type classes and polymorphism informally.

With these semantics of type terms, models for terms may be defined:

Definition 2.1.6 ***Model***

A model $M = (TM, I)$ for a signature $\Sigma = (\Omega, C)$ consists of:

- a simple type model TM for Ω and
- a set $I = \{c^M\}$ of interpretations for all constants $c \in C$.

The interpretation of terms depends on a variable assignment $\eta : \Psi \longrightarrow X$ where $X \in PU$ and Ψ is a set of variables.

Definition 2.1.7 ***Term Interpretation***

The interpretation in a model $M = (TM, I)$ and $I = \{c^M\}$ of a Σ-term is defined by:

- INT_CON: $M[\![c]\!]_\eta^\Sigma = c^M$ for $c \in \Sigma$ and $c^M \in I$
- INT_VAR: $M[\![x]\!]_\eta^\Sigma = \eta(x)$ for $x \in \Psi$, where $\eta : \Psi \longrightarrow X$ is a variable assignment and Ψ is a set of variables
- INT_APP: $M[\![(t_1 t_2)]\!]_\eta^\Sigma = (M[\![t_1]\!]_\eta^\Sigma)(M[\![t_2]\!]_\eta^\Sigma)$ for $t_1, t_2 \in T_\Sigma$

- INT_ABS: $M[\![\lambda x::u.t]\!]^{\Sigma}_{\eta} = f$ for $x \in \Psi$, $u \in T_{\Omega}$ and $t \in T_{\Sigma}$ where f is the (by extensionality) uniquely defined function $f : a \in TM[\![u]\!]^{\Omega} \mapsto M[\![t]\!]^{\Sigma}_{\eta(a/\mathtt{x})}$

Additional requirements for η (and ν), which arise from the restriction to type correct terms, are omitted here and may be found in [Reg94].

Some constants have to be in every model and their interpretations are:

- SEM_IMP: $\Longrightarrow^{M}(b_1)(b_2) = \begin{cases} \mathbb{F} & \text{if } b_1 = \mathbb{T} \text{ and } b_2 = \mathbb{F} \\ \mathbb{T} & \text{else} \end{cases}$
- SEM_OR: $\vee^{M}(b_1)(b_2) = \begin{cases} \mathbb{T} & \text{if } b_1 = \mathbb{T} \text{ or } b_2 = \mathbb{T} \\ \mathbb{F} & \text{else} \end{cases}$
- SEM_EQ: $=^{M}(a)(b) = \begin{cases} \mathbb{T} & \text{if } a \text{ and } b \text{ are equal} \\ \mathbb{F} & \text{else} \end{cases}$
- SEM_EPS: ε^{M} $(p::(\alpha \Rightarrow bool))$ chooses an arbitrary, fixed $x = ch(TM[\![\alpha]\!]^{\Omega})$ with $M[\![p(x)]\!]^{\Sigma}_{\eta} = \mathbb{T}$. This operator is called Hilbert operator an may be used to specify nondeterministic operations, for example the choice function of an equivalence class on page 228.

With this interpretation, the semantics of terms are defined. Based on the syntactic notion of a theory, satisfaction can be defined for axioms and theories.

Definition 2.1.8 ***Theory***

The theory $Th = (\Sigma, Ax)$ is a pair where

- Σ is a signature and
- $Ax = \{ax_i::bool\}$ and $ax_i \in T_{\Sigma(\emptyset)}$

The axioms Ax describe properties of the intended model.

Theories are often called *specifications* when they are used to specify requirements or realizations of a system.

Definition 2.1.9 ***Satisfaction***

A model M satisfies a formula $\varphi \in T_{\Sigma}$ of type `bool` ($M \models \varphi$) if

- $M[\![\varphi]\!]^{\Sigma}_{\eta} = \mathbb{T}$ for all variable valuations η.

The notion can be extended to theories $Th = (\Sigma, Ax)$ by $M \models Th$ if

- $M \models ax$ for all $ax \in Ax$.

The axioms Ax are closed boolean terms, and therefore, they are independent from the assignment η.

In this definition of satisfaction the type environment Γ and the type assignment ν are not needed, since type correctness of terms (and variables) is assumed.

2.1.4 Conservative Extensions

Conservative extensions are a method to safely extend HOL theories, see [GM93]. They are called conservative, since models of the extended theory can be constructed in terms of the basic theory and it does not affect the basic models (persistent construction). Refinements between a theory its conservative extensions are consistency preserving, since conservative extensions explicitly construct a model, and they are modular since the extended model may be reduced to the concrete one (see [Reg94, Section2.6]). Since HOLCF is a conservative extension of HOL, and since HOL has a model, the semantics of HOLCF is well defined. Realizing this conservative embedding was a challenge to Franz Regensburger in his thesis [Reg94].

In [Reg94, Section 5.3] a special form of conservative extension to introduce free data types is presented. This introduction ensures that the constructed type resides in the class `pcpo`. This has been implemented in form of the `domain` construct (see [Ohe95] or on page 50). For the implementation of interactive systems it is important, that the types reside into the type class `pcpo` since on functions between types of the type class `pcpo` the fixed point construction denotes the semantics for recursive functions. In order to guarantee that types are in the class `pcpo`, the methods presented in the following chapters provide techniques which ensure this without requiring the user to provide a partial order or to prove chain completeness, which are the characteristic properties of the class `pcpo`.

In this section the HOL method for the conservative introduction of new types is repeated from [GM93] and the introduction of free data types is presented as in [Reg94, Ohe95].

The following example (from the implementation of HOLCF [Reg94, page 172]) shows the conservative introduction of the continuous function space in HOLCF:

Example 2.1.2 ***Continuous Functions***

Continuous functions become a conservative extension of HOL, constructed by building a subset of values from the full function space from HOL. The first restriction of the subset excludes all structures which are not chain complete. This is expressed by the use of the type class `pcpo`. The second restriction allows only continuous functions between types with *cpo* structures. This is expressed by the predicate

cont:: (α::pcpo$\Rightarrow\beta$::pcpo)$\Rightarrow$bool[7]. The set of all continuous functions is denoted by the infix type constructor $\rightarrow$. To ensure the well-definedness of this definition the conservative extension method uses an abstraction function and a representation function.

```
Cfun1 = Cont +
types → 2    (* declares the arity of the type constructor *)
consts
    Cfun    :: (α ⇒ β) set          (* subset *)
    fapp    :: (α → β)⇒(α ⇒ β)        (* rep *)
    fabs    :: (α ⇒ β)⇒(α → β)        (* abs *)
rules
    Cfun_def              Cfun ≡ {f. cont(f)}
    Rep_Cfun              fapp fo ∈ Cfun
    Rep_Cfun_inverse      fabs (fapp fo) = fo
    Abs_Cfun_inverse      f ∈ Cfun ⟹ fapp(fabs f) = f
end
```

To show that this definition is conservative, it is necessary to show that the new type is not empty (otherwise the choice function would be undefined and cause inconsistencies). This is done by proving that there exists a function f with cont f. This function is the *witness* for the correctness of the type definition. The abstraction function fabs is written as the binder Λ and the representation function fapp is abbreviated by ‘. For applying the β-reduction of a continuous function it is necessary to prove that the function is continuous. This is expressed by the following theorem:

- beta_cfun cont c $\Longrightarrow$ (Λx.c x)‘u = c u

We sometimes call continuous functions *operations*.

In HOLCF there exist conservative extension methods for introducing new classes, arities, types and constants (see [Reg94, Section 2.6]). For classes it is necessary to give a witness that fulfils the class axioms, for arities a witness for the characteristic axioms is needed (see Section 2.1.2). The conservative introduction of a constant requires that the constant is defined by a term that denotes its meaning.

Some examples for the introduction of new constants in HOL are the formulation of Henkin's definitions [Hen63] for the quantors in [Reg94, page 86]:

- $\forall$_Def: $\forall \equiv \lambda$P.(P=(λx.True))
- $\exists$_Def: $\exists \equiv \lambda$P.P(εx.P)

In LCF [Pau87] *cpo*s and continuous functions form the basis of the logic for computable functions. In this thesis ADTs are based on data types over *cpo* structured domains.

[7] From [Reg94]: cont f$\equiv\forall$ Y.is_chain Y$\longrightarrow$range(λi.f(Y i))<<|f(lub(range Y))

2.1.5 Data Types

This section defines different notions of data types, and shows that for the specification of ADTs in HOLCF a type classes `eq` is useful.

Data types represent the values on which functions are working. To ensure the well-definedness of recursive functions definitions it is necessary to define them with a termination ordering. Another way is to define recursive functions with the help of a fixed point operator, denoting the least fixed point. The basis for this construct are the domains with chain complete partial orders and continuous functions. A central aspect in HOLCF is the existence of types with such order structures. In Isabelle's logic the fact that a type has a certain structure is expressed by declaring that the type is a member of a type class. For example the class `po` has a partial order for the definition of monotonicity and is a superclass of `pcpo`, the type class with pointed complete partial orders. `pcpo` provides a continuous function space and a fixed point operator `fix::(`$\alpha\rightarrow\alpha$`)`$\rightarrow\alpha$ for the denotational semantics of recursive functions and interactive systems.

Therefore, all data types used in specifications in the deductive software development process should reside in `pcpo`, the type class of pointed complete partial orders. Data types are specified in theories.

Definition 2.1.10 ***Data Type***

A data type $T = (\tau, Con)$, specified in a theory $Th = ((\Omega, C), Ax)$ is a type characterized by a set of constructor functions $Con = \{con_1, ..., con_n\}$ with

- $\tau \in T_{\Omega(\Xi)}$, and τ resides in the type class `pcpo`,
- $Con \subseteq C$,
- all con_i are continuous functions of type $\alpha_i \rightarrow \tau$ or constants of type τ. In the case of multiple arguments: the type $\alpha_i \rightarrow \tau = \overline{\alpha_{ij}} \rightarrow \tau$ which is an abbreviation for $\alpha_{i1} \rightarrow .. \rightarrow \alpha_{im_i} \rightarrow \tau$, and
- a typical induction rule IND $\in Ax$ for admissible predicates[8] depending on the type of the constructors:

  IND: $adm\ \ P \wedge P\ \bot \wedge (\forall \overline{x_{ij}.\tilde{s}(x_{ij})} \wedge \tilde{P}(\overline{x_{ij}}) \Longrightarrow P(con_i\text{`}\overline{x_{ij}}\text{`})) \Longrightarrow P(x::\tau)$ where the type dependent schemata $\tilde{P}(z::t) = P(z)$, if $t = \tau$ and else $true$, and $\tilde{s}(z) = z \neq \bot$, if the constructor is strict in the argument at position ij and else $true$.

The induction rule is an important part of a data type. It allows us to deduce admissible properties over all values of the type. In addition, it characterizes the constructor functions.

[8] From [Reg94]: `adm P ≡ ∀Y. is_chain Y ⟶ (∀i. P (Y i)) ⟶ P (lub (range Y))`

A data type without induction rule would only be a specification containing a type and some functions on that type.

There are two different kinds of data types: finite data types and infinite data types. Infinite data types may have infinite elements. The best known example is `stream` (see page 164 for a definition of streams). Mathematically this is expressed by allowing the data type constructor to be lazy, i.e. by not requiring it to be strict. For finite data types all constructor functions have to be `strict`:

- ConStrict: con‘⊥ = ⊥

This is equivalent to the requirement that all selectors of a data type be total:

- SelTotal: x≠⊥ ⟹ sel‘x ≠⊥

In this work we first focus on finite data types. After being able to implement finite ADTs we extend our methods to the implementation of infinite ADTs (see Chapter 6).

Note that the admissibility of the predicate in the induction rule is only needed for data types with infinite elements. Since the formulation of the induction rule is quite complicated (even without admissibility) there exists an abbreviation for it in HOLCF (see [Ohe95]). The rule may be specified in HOLCF with the `generated by` construct by:

`generated` τ `by` $\mathtt{con}_1$ | .. | $\mathtt{con}_n$

The definition of a data type is shown on the example of lists:

Example 2.1.3 ***Data Type List***

The data type list: $DLIST = (\alpha\ DList, \{dnil, dcons\})$ can be specified in HOLCF in the following theory:

```
DLIST = HOLCF +
types DList 1
ops curried
    dnil :: α DList
    dcons:: α → α DList → α DList
rules
DList_Ind   ⟦P ⊥;P dnil;
             ∀a d.⟦a≠⊥;d≠⊥;P d⟧ ⟹ P(dcons‘a‘d) ⟧ ⟹ P x
end
```

The data type constructor `DList` is specified and the axiom describing the induction rule which fixes the set of constructors and allows us to deduce properties about lists is given. The name `DList` is used, since in HOL `list` is already defined. With the `generated finite by` construct (the keyword `finite` is used if no admissibility is needed) the induction rule for `DList` may be specified by:

```
generated finite DList by dnil | dcons
```

A central idea of abstract data types is to give a small set of functions that characterize the properties of the data type. These functions can be used for the definition of all complex functions based on the data type. The basic functions can be divided into three groups [BW82]:

- constructor functions, whose result type is the constructed data type. They may be recursive in this type (for example `succ:: Nat`$\rightarrow$`Nat`).
- selector functions select components from the relevant sort (for example `fst`:: $<\alpha, \beta>\rightarrow\alpha$).
- discriminator functions are used to discriminate between variants of data types (for example `is_empty:: Set` $\rightarrow$ `tr`).

On finite data types we require in addition that

- there exists a continuous equivalence relation $\doteq_\tau$::$\tau\rightarrow\tau\rightarrow$`tr` to compare the elements of the type τ. This comparison should be
  - reflexive (on defined values), symmetric and transitive,
  - strict,
  - total,
  - $\doteq_\tau$ should be a congruence, i.e. for all constructor, selector and discriminator functions `f::`$\alpha\rightarrow\beta$ it should hold that $\lceil$`x` $\doteq_\alpha$`y`$\rceil$ implies if `f‘x`$\neq\perp$ and `f‘y`$\neq\perp$ then $\lceil$`f‘x`$\doteq_\beta$`f‘y`$\rceil$ where $\doteq_\alpha$ and $\doteq_\beta$ are the continuous equalities fitting to the type of `f`[9]

  We call $\doteq_\tau$ *continuous equality.*

A helpful function, which should be available for every data type constructor, is the map functional. It applies functions to every element of the type (the best known is `Map_List::` `(`$\alpha\rightarrow\beta$`)` $\rightarrow$ `List` α $\rightarrow$ `List` β). If τ=`tc`$(t_1, .., t_n)$, then the map functional Map_τ takes n functions f_i::$t_i\rightarrow s_i$ and has the type $(t_1\rightarrow s_1)\rightarrow..\rightarrow(t_n\rightarrow s_n)\rightarrow$`tc`$(t_1, .., t_n)\rightarrow$`tc`$(s_1, .., s_n)$.

[9] In Section 4.2.4 we introduce a predicate to express this property.

If the data type constructor is of arity 0 (for example `Nat`), then the map functional is the identity. The map functionals are needed to define the implementation of terms of type τ `list` where τ is the implemented type. The general map functional was also proposed for the `datatype` construct in HOL (see [Völ95]).

This leads to the following definition of ADTs:

Definition 2.1.11 ***Abstract Data Type***

A data type $T = (\tau, Con)$ specified in $Th = ((\Omega, C), Ax)$ is called an abstract data type, if there exist selector functions $Sel = \{sel_{ij}::\tau \to \alpha_{ij}\} \subseteq C$, discriminator functions $Dis = \{dis_i::\tau \to \mathtt{tr}\} \subseteq C$, a map functional $Map_\tau \in C$, and (on finite ADTs) a continuous equality $\doteq_\tau \in C$, so that the following rules are in Ax:

- DISCRIM:$\mathtt{x} \neq \bot \Longrightarrow \lceil dis_i\text{`}(con_i\text{`}(x))\rceil$ for all $1 \leq i \leq n$
- CONSEL:$\lceil dis_i\text{`}(x)\rceil \Longrightarrow con_i\text{`}(\overline{sel_{ij}\text{`}(x)\text{`}}) = x$ for all $1 \leq i, j \leq n$
- MAPCON:$\lceil dis_i\text{`}(x)\rceil \Longrightarrow Map_\tau\text{`}\overline{f\text{`}}x = con_i\text{`}(\overline{Map_\sigma\text{`}\tilde{f}\text{`}(sel_{ij}\text{`}x)})$ for all $1 \leq i, j \leq n$ where $Map_\sigma$ is the appropriate map functional of the component or the identity if the type of $sel_{ij}\text{`}(x)$ is basic and $\tilde{f}\text{`} \subset \overline{f\text{`}}$ are the arguments of the map functional, according to the type of the components.
- $\doteq_\tau$ has to be a strict and total equivalence relation. This is expressed by the following rules:
  - EQSTRICT1: $\bot \doteq_\tau \mathtt{x} = \bot$
  - EQSTRICT2: $\mathtt{x} \doteq_\tau \bot = \bot$
  - EQTOTAL: $\mathtt{x} \neq \bot \ \wedge\ \mathtt{y} \neq \bot \ \Longrightarrow\ \mathtt{x} \doteq_\tau \mathtt{y} \neq \bot$
  - EQREFL: $\mathtt{x} \neq \bot \ \Longrightarrow\ \lceil \mathtt{x} \doteq_\tau \mathtt{x}\rceil$
  - EQSYM: $\mathtt{x} \doteq_\tau \mathtt{y} \ =\ \mathtt{y} \doteq_\tau \mathtt{x}$
  - EQTRANS: $\lceil \mathtt{x} \doteq_\tau \mathtt{y}\rceil \ \wedge\ \lceil \mathtt{y} \doteq_\tau \mathtt{z}\rceil \ \Longrightarrow\ \lceil \mathtt{x} \doteq_\tau \mathtt{z}\rceil$
  - EQOBS$_i$:$\lceil \mathtt{x} \doteq_\alpha \mathtt{y}\rceil \ \Longrightarrow\ \mathtt{f}_i\text{`}\mathtt{x} \neq \bot \wedge \mathtt{f}_i\text{`}\mathtt{y} \neq \bot \longrightarrow \lceil \mathtt{f}_i\text{`}\mathtt{x} \doteq_\beta \mathtt{f}_i\text{`}\mathtt{y}\rceil$ for all $\mathtt{f}_i :: \alpha \to \beta \in Con \cup Sel \cup Dis$

  The existence of a continuous equality may be required by specifying the type to reside in the class `eq` (see page 234), and by the explicit specification of the observers. Our class `eq` (see Section 4.2.5) provides the predicate `is_Cobs` to express observability in HOLCF.

In the following we write $T = (\tau, Con, Sel, Dis, Map, \doteq)$ for a finite abstract data type, and $T = (\tau, Con, Sel, Dis, Map)$ for an arbitrary ADT.

This definition of ADTs has all features known from classical ADTs and (besides the powerful higher order logic of computable functions) it incorporates the notion of map functionals, which is very useful for higher order data types (see our methods in Sections

3.3.1, 3.3.2) and actually is integrated into Isabelle HOL logic [Völ95]. In addition our class `eq` (see Section 4.2.5) gives us the possibility to characterize congruences with predicates.

This definition is illustrated again with the example of lists:

Example 2.1.4 ***Abstract Data Type List***

The ADT list: $DLIST = (\alpha\ DList, \{dnil, dcons\}, \{dht, dtl\}, \{is\_dnil, is\_dcons\}, Map_{\alpha\ DList}, eq\_DList)$ can be specified in HOLCF in the following theory:

```
DLIST = EQ +
types DList 1
ops curried
          (* constructor functions *)
dnil      :: α DList
dcons     :: α → α DList → α DList
          (* discriminator functions *)
is_dnil   :: α DList → tr
is_dcons  :: α DList → tr
          (* selector functions *)
dhd       :: α DList → α
dtl       :: α DList → α DList
          (* Map on DLists *)
Map_DList :: (α → β) → α DList → β DList
          (* continuous equality for DList *)
eq_DList  :: α::eq DList → a DList → tr
rules
          (* Induction rule *)
DList_Ind [[P ⊥;P dnil;
              ∀a d.[[a≠⊥;d≠⊥;P d]] ⟹ P (dcons`a`d)]] ⟹ P x
          (* discriminator rules *)
is_dnil   ⌈is_dnil`dnil⌉
is_dcons  dcons`x`y≠⊥⟹⌈is_dcons`(dcons`x`y)⌉
          (* selector rules *)
dcons_app ⌈is_dcons`t⌉ ⟹ dcons`(dhd`t)`(dtl`t) = t
          (* Map rules *)
Map_dnil  ⌈is_dnil`t⌉ ⟹ Map_DList`f`t = dnil
Map_dcons ⌈is_dcons`t⌉ ⟹ Map_DList`f`t =
                                  dcons` (f`(dhd`t))`
                                 (Map_DList`f`(dtl`t))
          (* definition of the continuous equality *)
eq_DList1 ⌈eq_DList`dnil`dnil⌉
eq_DList2 dcons`x`y≠⊥⟹⌊eq_DList`dnil`(dcons`x`y)⌋
eq_DList3 dcons`x`y≠⊥⟹⌊eq_DList`(dcons`x`y)`dnil⌋
```

```
eq_DList4 eq_DList‘(dcons‘x‘s)‘(dcons‘y‘t) =
                  x≐y andalso eq_DList‘s‘t
end
```

The continuous equality for `DList` is based on the continuous equality of the base type. A proof that `eq_DList` is a continuous equality would justify the instantiation of `DList` into the class `eq`. In some requirements specification the existence of a continuous equality is required. This can be specified by requiring the type to be of the class `eq` and by specifying some observers for $\doteq$ (see Example 2.1.5). The definitions of `eq_DList` would allow us to deduce that it is a continuous equality. The observability can be expressed by the predicate `is_Cobs` from Section 4.2.4:

- `is_Cobs dcons`
- `is_Cobs is_dnil`
- `is_Cobs is_dcons`
- `is_Cobs dhd`
- `is_Cobs dtl`

The following general theorems for ADTs are obvious and will be used in some proofs in Section 3.2:

- DisCases: $\lceil dis_i‘x \rceil$ for some i
- MapId: $Map‘\overline{\Lambda x.x}‘z = z$

In programming languages, especially in functional languages data types are free. This means that constructors are distinct, and different arguments lead to different results of the constructor functions. This is captured by the following definition:

Definition 2.1.12 ***Free Data Type***

Let $T = (\tau, Con, Sel, Dis, Map, \doteq)$ be an ADT specified in $Th = (\Sigma, Ax)$. This ADT will be called a free data type, if the following rules are in Ax.

- Distinct:$\lfloor dis_j‘(con_i‘(x)) \rfloor$ for all $1 \leq i \neq j \leq n$
- Injective: $(con_i‘x = con_i‘y) \Longrightarrow (x = y)$ for all $1 \leq i \leq n$,

This is (without the observability) an initial characterization of the solution of the recursive domain equation for the data type. It is unique up to isomorphism.

This definition is illustrated again at the example of lists:

Example 2.1.5 ***Free Data Type List***

The ADT *DLIST* from Example 2.1.4 is specified as a free data type in the following theory:

```
DLIST = EQ +
types DList 1
arities DList (eq)eq      (* provides ≐ on DList *)
ops curried
          (* constructor functions *)
dnil      :: α DList
dcons     :: α → α DList → α DList
          (* discriminator functions *)
is_dnil   :: α DList → tr
is_dcons  :: α DList → tr
          (* selector functions *)
dhd       :: α DList → α
dtl       :: α DList → α DList
          (* Map on DLists *)
Map_DList :: (α → β) → α DList → β DList

rules     (* Induction rule *)
DList_Ind ⟦P ⊥;P dnil;
           ∀a d.⟦a≠⊥;d≠⊥;P d⟧ ⟹ P (dcons`a`d)⟧ ⟹ P x
          (* discriminator rules *)
axioms    (* further axioms *)
defvars f x y s t in
is_dnil1  ⌈is_dnil`dnil⌉
is_dnil2  ⌊is_dnil`(dcons`x`s)⌋
is_dcons1 ⌈is_dcons`(dcons`x`s)⌉
is_dcons2 ⌊is_dcons`dnil⌋
          (* selector rules *)
dcons_app ⌈is_dcons`t⌉ ⟹ dcons`(dhd`t)`(dtl`t) = t
          (* Map rules *)
Map_dnil  ⌈is_dnil`t⌉ ⟹ Map_DList`f`t = dnil
Map_dcons ⌈is_dcons`t⌉ ⟹ Map_DList`f`t =
                            dcons` (f`(dhd`t))`
                            (Map_DList`f`(dtl`t))
          (* injectivity of constructors *)
injective (dcons`x`s = dcons`y`t) ⟹ (x=y ∧ s=t)
end
```

`axioms defvars x in P x` is an Isabelle/HOLCF abbreviation for `x`$\neq\bot\Longrightarrow$`P x` in the following axioms.

From the monotonicity of the discriminators one can deduce that the constructors are distinct (here: `dnil` $\neq$ `dcons‘x‘t`).

Since such a description of free data types is very long it is helpful to have shorthands for their specification. In the specification language SPECTRUM [BFG+93a, BFG+93b] a data construct has been introduced that allows us to deduce these axioms. In HOL there exists a data construct as well.

In [Reg94] the introduction of free data types was defined and justified by the colimit construction. In [Ohe95] this introduction of free data types is implemented with the `domain` construct. With this, the free data type of lists may be specified by[10]:

```
DLIST = eq +
domain α DList = dnil | dcons (dhd::α) (dtl::α DList)
arities DList (eq)eq
rules
eq_obs   is_Cobs dcons ∧ is_Cobs is_dnil ∧
         is_Cobs is_dcons ∧ is_Cobs dhd ∧ is_Cobs dtl
end
```

Since the characterization of all observer functions may not be required for some data types, we define a class `EQ` in Section 4.3, in which all total and continuous functions are observers[11]. This will allow us to specify DLIST by:

```
DLIST = EQ +
domain α DList = dnil | dcons (dhd::α) (dtl::α DList)
arities DList (EQ)EQ
end
```

So we have seen that we could also need a subclass `EQ` of `eq` with some stronger properties. The class `EQ` corresponds to the class `EQ` of SPECTRUM. The class `eq` offers more flexibility in the implementation of abstract data types (we see this in Chapter 4). The differences between `eq` and `EQ` are discussed in Section 4.2.4, where these types classes are formally defined.

[10] The domain construct defines constructor, selector and discriminator operations, but in its current implementation not the map functional.

[11] Since `is_Cobs` works also with higher order functions another possibility would be to require the `when` functional to be an observer instead of the selector and discriminator functions. The `when` functional is an internal (higher order) functional from the `domain` construct which is used to define the selector and discriminator functions.

Data types specified by the `domain` construct can be directly translated into `datatype` declarations in a functional programming language as Haskell or ML. The precise syntax and features of the `domain` construct are contained in [Ohe95].

The data types used in most functional programming languages are free data types. Some experiments with non-free data types were made (see [Tur85, Tho90]), but did not become part of functional programming languages. However, in specifications non-free data types are used frequently, especially sets are a standard specification technique (Z [Dil94] is based on sets). Since non-free constructs (restrictions and quotients) are necessary for the implementation of ADTs and hence for the development of interactive systems a support for the implementation of non-free data types in HOLCF is needed. Another motivation is the lack of the type classes `eq` and `EQ` in HOLCF[12].

2.1.6 Executability and Pattern Matching

In the development towards functional programs, executability is a characteristic property of specifications which informally states that we may directly generate program code from the specifications or execute them with an appropriate interpreter. Usually code is generated, since this is easier and more useful (most of the time only syntactic translations are needed) than to build an interpreter for executable specifications.

The following definition of executability also includes some non-free data types, provided that all functions are executable. This leaves some freedom for the implementation of data types. However, on the non-free data types pattern matching is not supported.

Definition 2.1.13 ***Executable Data Type***

A abstract data type $T = (\tau, Con, Sel, Dis, Map, \doteq)$ is executable, if all functions in Con, Sel, Dis, Map, and $\doteq$ are executable.

A function is executable if

- it is a constructor function of a free data type (over executable data types[13]), or
- if it is conservatively introduced by a continuous term, built of executable subterms.

If the data type is in a subclass of `pcpo`, all characteristic operations for that class have to be instantiated by executable functions (Consider the instantiation of the continuous equality $\doteq$ in Section 4.2.5 as an example).

[12] In some HOLCF extensions there exists a class, which provides a so called weak equality with $\lceil x \doteq y \rceil \longrightarrow x = y$. However, this does not allow us to specify observer functions.

[13] This excludes the type $(\alpha \to \beta)$ `list`. The function space is not a data type since there is no induction rule for it. Specifying an induction rule for it, as in [Möl87], would not be conservative.

For example an executable definition of the map functional on `DList` would be:

```
DLIST = HOLCF +
domain α DList = dnil | dcons (dhd:: α) (dtl::α DList)
ops curried
Map_DList     :: (α → β) → α DList → b DList
defs
Map_DList_def Map_DList ≡ fix‘(ΛMap f l.
                          If is_dnil‘l
                          then dnil
                          else  dcons‘(f‘(dhd‘l))‘ (Map‘f‘(dtl‘l))
                          fi)
end
```

Note that executability does not imply termination. For example `g≡fix‘(Λf x.f‘x)` is executable, but does not terminate.

Using the fixed point constructor `fix` may lead to unreadable definitions and does not use the full power of functional languages, since there are more elegant notations with the same meaning. As in functional languages, pattern matching with non-overlapping and complete constructor patterns are allowed to increase usability (See [Har86, Rea89, Pau92, HR94] for a more formal definition of patterns).

Definition 2.1.14 ***Executability with Pattern Matching***

A specification $Th = (\Sigma, Ax)$ of an ADT T with pattern matching is executable, if

- the data type T is free and
- in the axioms Ax the patterns are complete (defined for all values except $\bot$) and disjoint.

This allows us to translate a function defined by pattern matching into an executable definition by a Λ-term.

The requirement that the patterns have to be complete comes from the difference between underspecification and $\bot$. From underspecification (no axioms for `f`) we cannot deduce `f`$=\bot$. However, from the developer's point of view it is acceptable, if for all undefined patterns dummy patterns defined by $\bot$ are inserted by the code generator. This is realized in the ML code generator of executable SPECTRUM specifications in [HR94].

Example 2.1.6 ***Executable Data Type List***

An executable definition with pattern matching for the data type `DList` is:

```
DLIST = EQ +
domain α DList = dnil | dcons (dhd::α) (dtl::α DList)
arities DList (eq) eq
ops curried
Map_DList     :: (α → β) → α DList → β DList
eq_DList      :: α::eq DList → α DList → tr

axioms
defvars f x y s t in
Map_DList_def1 Map_DList‘f‘nil = nil
Map_DList_def2 Map_DList‘f‘(dcons‘x‘s) = dcons‘x‘(Map_DList‘f‘s)
eq_DList1      ⌈eq_DList‘dnil‘dnil⌉
eq_DList2      dcons‘x‘s≠⊥⟹⌊eq_DList‘dnil‘(dcons‘x‘s)⌋
eq_DList3      dcons‘x‘s≠⊥⟹⌊eq_DList‘(dcons‘x‘s)‘dnil⌋
eq_DList4      eq_DList‘(dcons‘x‘s)‘(dcons‘y‘t) =
                  x≐y andalso eq_DList‘s‘t
inst_DList_eq  (op ≐) = eq_DList (* needs a witness *)
end
```

The instantiation into the class `eq` is essential. It requires to prove the class axioms of `eq` for `eq_DList`.

ML does not offer the possibility to instantiate the equality. It uses the canonical definition of the equality. In Haskell the equality may be instantiated by an arbitrary function, which may lead to incorrect programs. Our method gives us the possibility to characterize an observable equality for any data type (see Chapter 4), and to prove that the instantiation is correct.

In [BC93] a method which allows pattern matching over non-free data types is presented. It works with two different kinds of constructors: one for the construction the values and another for destructing them with pattern matching.

2.1.7 Predefined Type Constructors

Many data types that are used for specification and programming are based on primitive type constructors as sums, products, pairs etc. Therefore, we now define selector functions, discriminator functions, map functionals, and a continuous equality to construct arbitrary complex data types, which base on these predefined type constructors. For type constructors with an arity greater than one (pairs) the map functional has to take more than one function as arguments and apply them to each component.

For the introduction of constants with a type that involves type constructors, the `Map`τ functionals together with discriminator functions and selector functions are needed. For

type constructors introduced with the `domain` construct there exist such functions. The map functional may easily be defined for such types by pattern matching or by the `when` functional.

For the predefined data type constructors of HOLCF discriminator, and selector functions, and the map functionals can be defined as follows:

```
PREDEF = EQ +
(* all predefined data type constructors in HOLCF in pcpo are :*)
(*      ⊕ :: (pcpo,pcpo)pcpo    strict Sum *)
(*      ⊗ :: (pcpo,pcpo)pcpo    strict Product *)
(*      × :: (pcpo,pcpo)pcpo    cartesian Product *)

(* now Selectors and map−functionals are defined *)
consts
      selSs1  :: α ⊕ β → α
      selSs2  :: α ⊕ β → β
      disSs1  :: α ⊕ β → tr
      disSs2  :: α ⊕ β → tr
      mapSs   :: (α→γ) → (β→δ) → α⊕β → γ⊕δ
      eqSs    :: α::eq ⊕ β::eq → α⊕β → tr

      selSp1  :: α ⊗ β → α
      selSp2  :: α ⊗ β → β
      disSp1  :: α ⊗ β → tr
      disSp2  :: α ⊗ β → tr
      mapSp   :: (α→γ) → (β→δ) → α⊗β → γ⊗δ
      eqSp    :: α::eq ⊗ β::eq → α⊗β → tr

      selCp1  :: α × β → α
      selCp2  :: α × β → β
      disCp1  :: α × β → tr
      disCp2  :: α × β → tr
      mapCp   :: (α→γ) → (β→δ) → α×β → γ×δ
      eqCp    :: α::eq × β::eq → α×β → tr
defs

selSs1_def  selSs1 ≡ Λs. sswhen‘(Λx.x)‘⊥‘s
selSs2_def  selSs2 ≡ Λs. sswhen‘⊥‘(Λx.x)‘s
disSs1_def  disSs1 ≡ Λs. sswhen‘(Λx.TT)‘(Λx.FF)‘s
disSs2_def  disSs2 ≡ Λs. sswhen‘(Λx.FF)‘(Λx.TT)‘s
mapSs_def   mapSs ≡ Λf g. sswhen‘(sinl oo f)‘(sinr oo g)
eqCp_def    eqCp ≡ Λx y.selSs1‘x≐selSs1‘y andalso selSs2‘x≐selSs2‘y
```

```
selSp1_def  selSp1 ≡ sfst
selSp2_def  selSp2 ≡ ssnd
disSp1_def  disSp1 ≡ Λs. sswhen‘(Λx.TT)‘(Λx.FF)‘s
disSp2_def  disSp2 ≡ Λs. sswhen‘(Λx.FF)‘(Λx.TT)‘s
mapSp_def   mapSp ≡ Λf g x.(|f‘(selSp1‘x),g‘(selSp2‘x)|)
eqSp_def    eqSp ≡ Λx y.selSp1‘x≐selSp1‘y andalso selSp2‘x≐selSp2‘y

selCp1_def  selCp1 ≡ cfst
selCp2_def  selCp2 ≡ csnd
disCp1_def  disCp1 ≡ Λs. sswhen‘(Λx.TT)‘(Λx.FF)‘s
disCp2_def  disCp2 ≡ Λs. sswhen‘(Λx.FF)‘(Λx.TT)‘s
mapCp_def   mapCp ≡ Λf g x.<f‘(selCp1‘x),g‘(selCp2‘x)>
eqCp_def    eqSp ≡ Λx y.selCp1‘x≐selCp1‘y andalso selCp2‘x≐selCp2‘y

(* instantiations *)
arities
            ⊕ ::(eq,eq)eq
            ⊗ ::(eq,eq)eq
            × ::(eq,eq)eq
rules
inst_Ss_eq  (op ≐) = eqSs
inst_Sp_eq  (op ≐) = eqSp
inst_Cp_eq  (op ≐) = eqCp
end
```

As can be seen from these examples, the definitions of the selectors and constructors are schematic and can easily be defined by the `when` functional, which is provided by the `domain` construct.

2.2 Model Inclusion in HOLCF

This section defines a basis for the deductive software development using model inclusion in HOLCF. The refinement relation we use is model inclusion. Many of the development situations in Section 1.2.4 can be expressed by conservative extension and model inclusion. Another deductive development basis will use a special theory interpretation as refinement relation. Theory interpretations are a generalization of model inclusion and are presented in Section 2.3. Chapters 3 and 4 will compare these two bases for the deductive software development on the implementation of the restriction step and the quotient step.

The first notion of refinement in HOLCF is model inclusion, and it can be proved by theory inclusion. In this section we use the notion of HOLCF model from Definition 2.1.6 together

with the satisfaction relation $\models$ from Definition 2.1.9 as a loose semantics for theories as in [BW88a]. The next section contains the ideas of theory interpretation which will be worked out for the task of implementation of ADTs in HOLCF in the next chapters.

Model inclusion is based on a loose semantics:

Definition 2.2.1 ***Loose Semantics of Theories***

Let $Th = (\Sigma, Ax)$ be a theory, then the set of all models M' of all $Th' = (\Sigma', Ax')$ with $\Sigma \subseteq \Sigma'$ and $Ax \subseteq Ax'$ is the *loose semantics* of Th, if

- $M' \models Ax$

We write $\texttt{MOD}(Th)$ to denote the loose semantics of a theory.

This definition reflects the intuition that models with more features (functions and types) as required in the specification are in the semantics of the specification. With these loose semantics of HOLCF specifications we define a specific deductive software development basis.

Definition 2.2.2 ***Model Inclusion Basis***

The four tuple $(\mathcal{L}_{Th}, \texttt{MOD}(Th), \supseteq, \Longleftarrow)$ is called *model inclusion basis*, if

- $\mathcal{L}_{Th}$ is the set of all specifications,
- $\texttt{MOD}(\text{Th})$ are the loose semantics,
- $\supseteq$ is set inclusion on the models, and
- $\Longleftarrow$ is logical implication for HOLCF theories $Th^a, Th^c \in \mathcal{L}_{Th}$, defined by: $Th^a \Longleftarrow Th^c$ iff $Th^a \subseteq Th^{c\star}$ where * is the reflexive closure under the syntactic deduction relation $\blacktriangleright$ (see [Reg94, Section 2.5]). This logical implication is often called *theory inclusion*.

In the following we write $\texttt{C} \Longrightarrow \texttt{A}$ instead of $\texttt{A} \Longleftarrow \texttt{C}$ and $\texttt{S} \subseteq \texttt{T}$ instead of $\texttt{T} \supseteq \texttt{S}$.

The method to prove theory inclusion is to derive all axioms of Th^a from Th^c with a theorem prover. We sometimes write $\texttt{C} \vdash \texttt{A}$ for theory inclusion. We now prove that our model inclusion basis is a deductive software development basis by showing the properties of Definition 1.2.4.

- Executable specifications are consistent, since HOLCF is consistent and executable specifications are defined conservatively.
- Set inclusion $\subseteq$ is transitive,

- `C`$\Longrightarrow$`A` is correct, since the syntactic deduction relation $\blacktriangleright$ is correct [Reg94, page 81].

In addition we have:

- $\subseteq$ is consistency preserving, and
- $\subseteq$ is modular, if the theories are introduced by conservative extensions (see Section 2.1.4).

This simple notion of refinement has an obvious disadvantage: It does not relate models and theories with non-including[14] signatures . In many books on algebraic specifications like [EGL89, EM85, Wir90] the standard solution for this problem are homomorphisms.

Definition 2.2.3 ***Homomorphism***

Let $Th^a = (\Sigma^a, Ax^a), Th^c = (\Sigma^c, Ax^c)$ be theories. Then a homomorphism h is a mapping from $T_\Sigma a$ to $T_\Sigma c$ if

- HOMO: for all $f^a \in \Sigma^a$ and all $t_j \in T_\Sigma a$ holds: $h(f^a(\overline{t_j})) = h(f^a)(\overline{h(t_j)})$

This definition extends in the many sorted case to a family of mappings in the standard way.

A homomorphism is a mapping that respects structures, but the use of homomorphisms in software development has serious disadvantages. If we use homomorphisms to relate our theories we have to prove that the equations HOMO hold. For proving this we need the axioms of both theories Ax^a and Ax^c, since both theories are involved ($h :: T_\Sigma a \longrightarrow T_\Sigma c$). If the abstract axioms Ax^a contain a contradiction, Th^a is inconsistent and if Th^c is consistent we may prove the refinement (with HOMO) of Th^a into Th^c. Therefore, this refinement relation is not consistency preserving, since we refined an inconsistent specification by a consistent one. Therefore, homomorphisms cannot be used in this form for the deductive software development process (without requiring extra consistency proofs).

In equational logic the proof of HOMO is avoided by constructing the specification `A_by_C` as a homomorphic extension with the explicit abstraction function `abs` of the concrete specification and requiring that it refines the abstract specification behaviourally. However, the axiom {`instant`} (see on page 20) would not be conservative in HOLCF. Therefore, the method for the implementation of ADTs in HOLCF has to ensure consistency for this step[15].

Another drawback of homomorphism is that they are a concept of first order logic, and therefore, they are not defined for higher order terms. Theory interpretations can be regarded as the homomorphisms for higher order logics. We focus on them now.

[14] Neither $\Sigma^a \subseteq \Sigma^c$ nor $\Sigma^c \subseteq \Sigma^a$.

[15] There are further requirements from the implementation of interactive systems that make the use of equational logic inadequate. For example: fixed points, *cpo* structures and continuous functions.

2.3 Theory Interpretations

This section presents the concept of theory interpretation, a technique to define further refinement relations[16]. In [Far94b] there is a theory interpretation defined for a higher order logic. This section shortly presents theory interpretation, and shows why it cannot be applied to the implementation of ADTs for interactive systems. Therefore, we will define different theory interpretations for the implementation of ADTs in the following chapters. These methods will be compared to methods based on model inclusion in Chapters 3 and 4.

The motivation for using theory interpretation is: *"Theory Interpretation is a logical technique for relating one axiomatic theory to another with important applications in mathematics and computer science as well as in logic itself."* (Farmer in [Far94b]). In first order predicate logic theory interpretation is a standard method for problem reduction (see [End72, Sch70]). It guarantees the solution of an abstract problem, if the translated problem can be solved (satisfiability). The translation is like a homomorphism, fixed by a sort translation and a constant translation. In contrast to homomorphisms theory interpretations translate not only terms, but also formulas with quantifiers. If the abstract sort is translated into a subsort of a concrete sort, the quantifiers will be relativated (weakened) by a predicate that restricts the range of the concrete sort.

We will use theory interpretations for the translation of equivalence classes (in quotient types) into representing elements. This allows us to develop our quotient specifications into executable specifications in the sense of Definition 2.1.13. Theory interpretation is a generalization of model inclusion. Theory interpretations are a mathematical technique and they are used to reduce abstract problems to simpler ones, guaranteeing the satisfiability of the abstract problem. Usually theory interpretations are defined inductively over the structure of the terms.

Definition 2.3.1 ***General Theory Interpretation***

Let $Th^a = (\Sigma^a, Ax^a)$ and $Th^c = (\Sigma^c, Ax^c)$ be theories with signatures $\Sigma^a = (\Omega^a, C^a)$ and $\Sigma^c = (\Omega^c, C^c)$, then a translation (or mapping) $\Phi : T_\Sigma a \longrightarrow T_\Sigma c$ from Th^a to Th^c is called a theory interpretation, if it is defined inductively by a sort translation $\mu : T_\Omega a \longrightarrow T_\Omega c$ and a constant translation $\phi : C^a \longrightarrow T_\Sigma c$ in the following inductive form:

- GTI_Const Φ`(c)` = φ`(c)` for constants `c`$\in C^a$
- GTI_Var Φ`(x::`τ`)` = `x::`μ`(`τ`)` for Variables `x`
- GTI_App Φ`(f t)` = Φ_1`(`Φ`(f),`Φ`(t))` for a function `f` applied to an argument `t`

[16] Concrete theory interpretations for the quotient step and the restriction step of the implementation are defined in Chapters 3 and 4.

- GTI_ABS $\Phi(\lambda \mathtt{x}.\mathtt{t}) = \lambda \mathtt{x}.\Phi_2(\Phi(\mathtt{t}))$ for a λ abstraction.

$\Phi_1 : T_\Sigma c, T_\Sigma c \longrightarrow T_\Sigma c$ and $\Phi_2 : T_\Sigma c \longrightarrow T_\Sigma c$ are the rules, which describe how the translations of the terms are composed to the translation of the application and abstraction of terms. For a theory interpretation the following properties have to hold:

- CORRECTNESS: $Th^a \vdash \psi$ implies $Th^c \vdash \Phi(\psi)$ for all formulas $\psi \in T_\Sigma a_{(\emptyset)}$
- SATISFIABILITY: $M \models Ax^c$ and $M \models \Phi(Ax^a)$ implies that there exists $\widehat{M}$ with $\widehat{M} \models Ax^a$.

On page 60 there is an example for the definition of the rules Φ_1 and Φ_2. These rules are introduced to show two different kinds of theory interpretations. In the case that Th^a has a model M, we could define Φ to be the identity and we would have a trivial theory interpretation. However, since we do not know the consistency of Th^a in the development process, we define theory interpretations, which transform the abstract theories to simpler ones, which are closer to an implementation. One example for such a translation is the translation of theories with quotients into theories with representations.

In many cases $\widehat{M} = \widehat{\Phi}(M)$ can be provided by a schematic construction from Φ. CORRECTNESS states that Φ preserves correctness for all possible consequences of Th^a. This is important since if some properties are derived for the abstract theory (from Ax^a), these theorems will be required to hold in the representing theory (in the translated form). Theories extending Th^a can be seen as special properties of Th^a since they are based on Ax^a only. Correctness of those theories will be preserved, if Φ preserves correctness.

SATISFIABILITY guarantees the existence of an abstract model $\widehat{M}$ for Th^a. This will ensure consistency of Th^a if Th^c is consistent.

In the special case where Φ is the identity and $\Sigma^a \subseteq \Sigma^c$ the notion of theory interpretation reduces to model inclusion. The aspect of preserving the correctness is covered by theory inclusion and the satisfiability is provided by model inclusion, which is a consequence of theory inclusion ($\widehat{\Phi}$ is the identity). So theory interpretation is a generalization of model inclusion. However, the price for theory interpretation is, as we will see on the examples, a more complicated form of the resulting proof obligations to ensure the refinement.

Since the composition of mappings is a mapping and since theory interpretations are based on mappings they are obviously a transitive method. Satisfiability ensures that the refinement relations, defined by theory interpretations are consistency preserving. Φ translates theories, like a compiler, into more concrete ones and is, therefore, well suited for code generation. It will turn out (see Example 3.2.2) that theory interpretations are not modular and, therefore, it is a challenge to integrate them into the software development process.

The notion of theory interpretation for HOL was introduced by Farmer in [Far94b]. As we will see in an example of Farmer's theory interpretation it does not preserve continuity and

is, therefore, not suited for the implementation of interactive systems and we will define our own theory interpretation in the following chapters.

Farmer's theory interpretation consists of a sort translation μ, of a constant translation φ (which fit together in the sense of type checking), and of a restriction predicate U. The constant translation extends to higher order terms in a schematic way. If the translations fulfil some additional properties (which guarantee the CORRECTNESS and SATISFIABILITY), they will be called theory interpretation. These properties include the non-emptyness of the corresponding elements $\{x \mid U(x)\}$ and the invariance of the representing functions $\varphi(c::a \to b)$ with respect to the corresponding elements.

In Farmer's logic there are type terms and terms (as in HOL). Terms may be built of constants, variables, λ-abstraction and application of types. The extension of φ to these higher order terms depends on the types of the terms:

- FARMER_CONST: $\Phi(c) = \varphi(c)$ for constants c
- FARMER_VAR: $\Phi(x::s) = x::\bar{\mu}(s)$ for variables x
- FARMER_APP: $\Phi(f\ t) = \Phi(f)\ \Phi(t)$
- FARMER_ABS: $\Phi(\lambda x::s.t) = \lambda x::\bar{\mu}(s).$ `if` $\kappa_s(x)$ `then` $\Phi(t)$ `else` $\bot$

Where $\bar{\mu}$ is the extension from μ to type terms and $\kappa_s(x)$ is a type dependent predicate calculating the invariance of the functions argument.

We will not repeat the definition of $\bar{\mu}$ and κ_s from [Far94b], but we will demonstrate it by a small example, where κ_s is calculated for a functional sort.

Farmer's theory interpretation is a general theory interpretation in the sense of Definition 2.3.1. Φ_1 is the usually application of functions, and Φ_2 is a more complex translation, depending on the type of the involved variables.

Example 2.3.1 ***Farmer's Theory Interpretation***

In this example the abstract sort `B` is interpreted by a subset of `N`, which is characterized by the restriction predicate `isR::N→bool`. The theory B consists of:

- the sort `B`
- the constants
  - `T::B`
  - `F::B`
  - `not::B→B`
  - `Bid::B→B`
  - `Bcomp::(B→B)→(B→B)→B→B`

- Btwice::(B→B)→B→B

- the axioms for the functions
  - not(F) = T
  - not(T) = F
  - Bid = λx.x
  - Bcomp = λf g.λx.f(g(x))
  - Btwice = λf.Bcomp f f
  - Btwice not = Bid

The sorts are interpreted by:

- μ(B)=N

The elementary constants are interpreted by:

- φ(T)=1
- φ(F)=0
- φ(not)=λx.1−x
- φ(Bcomp)=Ncomp

Now we show the translation of the last axiom Btwice not = Bid using the definitions of φ, μ and Farmer's rules FARMER_CONST, FARMER_VAR, FARMER_APP and FARMER_ABS:

Φ(Btwice not = Bid)

= Φ(Btwice not) = Φ(Bid)

= Φ(Btwice) Φ(not) = Φ(Bid)

= Φ(λf::B→B.Bcomp f f) λx::N.1−x = Φ(λx::B.x)

= Φ(λf::B→B.Bcomp f f) λx.1−x
  = λx::N.if isR(x) then x else ⊥

= (λf::N→N.if $\kappa_{B\to B}$(f) then Φ(Bcomp f f) else ⊥) λx.1−x
  = λx.if isR(x) then x else ⊥

= (λf.if ∀x::N.if isR(x) then f(x)≠⊥ ⟶ isR(f(x)) else f(x)=⊥
    then Ncomp f f else ⊥) λx.1−x = λx.if isR(x) then x else ⊥

= if ∀x.if isR(x) then (1−x)≠⊥ ⟶ isR(1−x) else 1−x=⊥
  then Ncomp λx.1−x λx.1−x else ⊥ = λx.if isR(x) then x else ⊥

As can be seen from this example the premise calculation of functional arguments gives a test for all possible inputs. The resulting function is obviously a non-continuous function since it tests its argument f for all possible inputs[17].

Therefore, applying Farmer's theory interpretation to HOLCF would translate continuous higher order functions to non-continuous functions. We prove consistency of our specifications simply by developing them into executable specifications, for which consistency is trivial. Therefore, we need to implement continuous functions by continuous functions. That's why we define another theory interpretation and ensure correctness and satisfiability.

In contrast to the rules for Φ_1 and Φ_2 (FARMER_APP and FARMER_ABS) our rules for Φ_1 and Φ_2 in Section 3.2 use the more complex translation for the application, and the simple translation for the λ-abstraction. Our translations preserve continuity. An additional aspect is the treatment of type constructors: in $PF^\star$ (Farmers logic) there are no type constructors. We define our theory interpretation in a way that it also works for terms of constructed types. Furthermore, we have to cope with polymorphism.

Theory interpretation is (like conservative extension) a conservative approach, but the differences are: In conservative extension a theory is syntactically constructed (bottom-up), whereas theory interpretation constructs the model of the abstract theory semantically and the theory is translated to a concrete one (top-down). In the software development process this results in different programs[18]. We compare both methods in the following chapters.

Using theory interpretation in the deductive software development process requires to show that it preserves correctness and satisfiability. This will be guaranteed by showing that the chosen theory interpretation is satisfiable, if some invariance properties hold. These properties are additional proof obligations in the method: the user proves correctness and invariance, which guarantee the theory interpretation to be satisfiable.

[17] In addition the non-continuous test $y \neq \bot$ is used in $\kappa_{B \to B}$.

[18] A similar difference is between downward and U simulation in Chapter 6.

Chapter 3

Subdomains in HOLCF

The implementation of ADTs consists of two steps. The restriction step and the quotient step. The quotient step is treated in Chapter 4. We call the development from an abstract requirement specification with a subdomain into a concrete specification with a more general type *restriction step*. Restriction steps occur frequently in the development of interactive systems (see Section 1.2.4). A refinement relation for HOLCF should support the refinement of restriction steps.

The method for the implementation of interactive systems bases on the implementation of ADTs specified in HOLCF. To ensure that this method fits well into the deductive software development process we compare methods based on different refinement techniques for the restriction step in this chapter. These techniques are:

- theory interpretation (from Section 2.3) and
- model inclusion (from Section 2.2).

Therefore, this chapter is structured as follows: In Section 3.1 the motivations for subdomains are given and the role of invariance for subtypes is explained.

Since theory interpretation, as presented in Section 2.3, does not necessarily implement continuous functions by continuous functions we define a new theory interpretation in Section 3.2 that does this. The method for restrictions based on theory interpretation will not be used in the method for the implementation of ADTs in Chapter 5. Therefore, the reader, not interested in a theory interpretation that preserves continuous functions may skip this section with the technical correctness proofs of the method.

As mentioned on page 57, model inclusion cannot directly relate components with different interfaces. Therefore, the method presented in Section 3.3 constructs a new ADT[1] and

[1] Refining the requirement specification by a specification constructed on top of a concrete specification is called *constructor implementation* in [ST88].

requires to show that the new ADT refines the original ADT by model inclusion. The difficulty, solved in Section 3.3 is to show that the new type has *cpo* structure and that the functions operating on it are continuous. Section 3.4 compares both methods with the criteria from the development process. For that purpose a small example is analyzed. Section 3.5 describes the definition of a conservative subdomain construct in Isabelle's logic HOLCF and shows by an example how this construct is used.

3.1 Motivation

Subtypes describe a subset of values from a type by a restriction predicate. The term *subtyping* is used in types systems that allow us to use the subtypes instead of the more general types. For example if the natural numbers are a subtype of the integers, then subtyping allows us to use a function for integers also for natural numbers. A function defined for natural numbers is not applicable to integers values. Since type checking for systems with subtyping in general cannot give the most concrete type of a value (for example the type `nat` cannot be derived for the value of `2−1`) HOL and HOLCF have no subtyping.

In HOL and HOLCF "subtypes" have to be introduced as new types with conservative extensions (see Section 2.1.4). A restriction predicate allows us to define a subset of corresponding values that are isomorphic to the new type. Embedding functions are required for the conversions between the basic type and the new type. With this type checking can work around the problems of subtyping.

For the implementation of interactive systems we need domains that belong to the type class `pcpo` (see Section 1.4.4). If we formulate a new type as a subtype of a domain this new type is not necessarily a domain. We will see that an admissible restriction predicate suffices to introduce a subtype of a domain as a domain. We call a subtype of a domain that belongs to the class `pcpo` a *subdomain*.

In contrast to free extensions of data types the central idea of restrictions is that not all values from the concrete type are used to represent abstract values. There is a subset of concrete values that correspond to the abstract values, which may be defined by:

Definition 3.1.1 ***Corresponding Elements***

Let $\tau \in T_\Omega a$ be the required type and $\sigma \in T_\Omega c$ be the implementing type then the corresponding elements $\hat{\sigma}$ for a restriction predicate p are:

- $\hat{\sigma} \;=\; \{t{::}\sigma \mid p(t)\}$

The other values of σ are called *non-corresponding* elements.

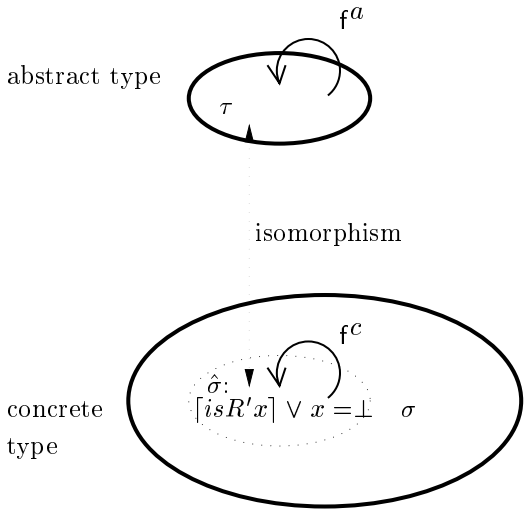

Figure 3.1: Preserving Function

An interesting aspect of subtypes is the definition of functions on the subtype. If we have functions for the conversions, then we call the function from the concrete type into the new subtype the abstraction function and the inverse function the representation function. The representation function is defined for all values in the new type, whereas the abstraction function is only partially defined for those values fulfilling the restriction predicate (see Example 2.1.2). With the these functions we can lift functions from the concrete type to functions on the new subtype. However, this will only be well defined if the concrete functions do not leave the subset of corresponding values. Those functions are called *preserving* functions. The restriction predicate is *invariant* under preserving functions.

To illustrate invariance we choose the most simple case that the function $\mathtt{f}^a \in \{\mathtt{c}_i^a\}$ is of the type $\tau \rightarrow \tau$ and the restriction predicate p is of type $\sigma \Rightarrow \mathtt{bool}$. The invariance of p for the corresponding function $\mathtt{f}^c \in \{\mathtt{c}_i^c\}$ is $\forall \mathtt{x}.p(\mathtt{x}) \Longrightarrow p(\mathtt{f`x})$. It is shown in Figure 3.1. A non-preserving function $\mathtt{f}^c$ would leave $\hat{\sigma}$ and therefore, the results of $\mathtt{f}^c$ could not be translated by the isomorphism. The correspondence between the required type τ and $\hat{\sigma}$ is established by an isomorphism. The main difference between theory interpretation and conservative extension is the form of this isomorphism. In theory interpretation the isomorphism is split into a mapping on theories and a reverse mapping on models, whereas in conservative extension two functions `abs` and `rep` are syntactically introduced for this isomorphism.

In the restriction step the corresponding operations $\mathtt{c}_i^c$ have to be preserving. The method for implementation of ADTs requires to prove that the functions are preserving. Therefore, preserving operations have to be chosen for the corresponding operations. Invariance will also hold, if the restriction predicate is stable with respect to the corresponding operations (Section 5.2 explains this more detailed).

Theory interpretation and model inclusion are illustrated by the following example on which a schematic translation between two components communicating over boolean values could be based.

Example 3.1.1 ***Restriction Step***

The requirement specification (Th^a) is:

```
BOOLEAN = DLIST +
types      B
arities     B   :: pcpo
ops curried
            T    :: B
            F    :: B
            Band :: B → B → B          (cinfixl 55)
            Bnot :: B → B
            c    :: B DList          (* a constant *)
rules
Band1      T Band x = x
Band2      F Band x = F
Bnot1      Bnot‘F = T
Bnot2      Bnot‘T = F
defs
c_def      c ≡ Map_DList‘Bnot‘(dcons‘T‘(dcons‘F‘dnil))
(* induction rule *)
generated finite B by T | F
end
```

Of course `tr` could have been used instead of `B` but `BOOLEAN` is used to present the requirements together in one specification.

The implementing specification (Th^c) is:

```
NAT = DLIST +
types N
arities N::pcpo
ops curried strict
            Zero :: N
            One  :: N
            succ :: N → N
            isB  :: N → tr
defs
one         One ≡ succ‘Zero
isB_def     isB ≡ Λx::N. is_Zero‘x orelse is_Zero‘(x−One)
```

```
(* induction rule *)
generated finite N by Zero | succ
end
```

To illustrate the CORRECTNESS property of our theory implementation we look at the formula, which follows from `BOOLEAN`:

`unique` $\exists!$ `(f::B→B).∀ x::B. f‘x = Bnot‘x`

The translation of this formula has to be valid in the concrete theory.

The other rules for the domain of `N` and for $-$ are omitted. They could easily be defined by the `domain` construct. The desired implementation is:

- `B` $\rightsquigarrow$ `N` (sort implementation)
- `T` $\rightsquigarrow$ `One`
- `F` $\rightsquigarrow$ `Zero`
- `Bnot` $\rightsquigarrow$ Λ `x::N . One − x`
- `Band` $\rightsquigarrow$ Λ `x::N y::N . If is_Zero‘x then Zero else y fi`
- The restriction operation is `isB`
- The corresponding values $\widehat{\texttt{N}}$ are $\{\bot$,`Zero`,`One`$\}$.

An implementation of `Band` by $+$ would violate the invariance of the implementation of `Band` since `isB‘(One+One)` is false. An implementation of `Band` by $+$ is possible if we build a quotient construction on `N`, identifying all values greater than zero. Quotient implementations are presented in Chapter 4.

In this chapter we implement an abstract ADT $T = (\tau, Con^a, Sel^a, Dis^a, Map_\tau)$[2] specified in a theory $Th^a = (\Sigma^a, Ax^a), \Sigma^a = (\Omega^a, C^a)$ by a concrete type $S = (\sigma, Con^c, Sel^c, Dis^c, Map_\sigma)$ specified in a theory $Th^c = (\Sigma^c, Ax^c), \Sigma^c = (\Omega^c, C^c)$. To cut down notations we write $\{\mathtt{c}_i^a\} \in T_\Sigma a$ for the set of all abstract operations[3] $Con^a \cup Sel^a \cup Dis^a \cup \{Map_\tau\}$ and $\{\mathtt{c}_i^c\}$ for the sets of *corresponding operations* ($\mathtt{c}_i^c \in T_\Sigma c$).

Since the quotient step is treated in Chapter 4 the implementation of T by S consists in this chapter of:

- A sort implementation $\tau \rightsquigarrow \sigma$, where $\tau \in T_\Omega a, \sigma \in T_\Omega c$.
- A constant implementation $\mathtt{c}_i^a \rightsquigarrow \mathtt{c}_i^c$ for all $\mathtt{c}_i^a \in \{\mathtt{c}_i^a\}$,

[2]The continuous equality does not need a special treatment for restrictions, and is therefore, omitted in this chapter.

[3]The `c` stands for constants. Note that in HOL a function is a higher order constant.

- A restriction predicate p of the form $p \equiv \lambda \mathtt{x}.(\mathtt{x} = \bot \vee \lceil \mathtt{isR}\text{`}\mathtt{x} \rceil)$, where `isR` is a continuous restriction predicate defined in Th^C.

The restriction predicate uses a continuous function `isR` of type $\sigma \to \mathtt{tr}$, which is `TT` for all values of the subdomain except $\bot$[4]. Thus `isR` characterizes a subdomain. Requiring the restriction predicate to have this form does not allow to express arbitrary subtypes (as for example in HOL [GM93, Mel89]), but it ensures that the subtype has a *cpo* structure. Since ADTs are specified with discriminator and selector functions there are many operations available to construct a restriction predicate. In Section 3.5 we weaken this restriction and require only the admissibility of the predicate characterizing the subdomain. However, for the comparison of theory interpretation and model inclusion we need `isR`.

Now two methods for the implementation of the restriction step are presented. Their comparison is in Section 3.4.

3.2 Theory Interpretation

This section presents the first method for the restriction step of the implementation. This method is based on theory interpretation, which in contrast to the theory interpretation presented in Section 2.3 interprets continuous functions by continuous functions and hence could be used for the development of interactive systems. However, the method presented in the next section, which is based on model inclusion is better suited[5] for the deductive software development process and it will be used in the method for the implementation of ADTs in Chapter 5. Therefore, the reader neither interested in the comparison of the two methods nor in a theory interpretation, which maps continuous functions to continuous functions may skip this section since it contains some technical details not needed for the implementation of interactive systems.

This section defines a special theory interpretation Φ for the restriction step of the implementation by defining a constant translations, a sort translation and by giving rules for the inductive translation of terms. To show that this special theory interpretation is well defined in the sense of Definition 2.3.1 on page 58 we have to ensure that the following properties hold for our theory interpretation.

- Correctness
- Satisfiability

The first requirement is a proof obligation for the user. From our definition of models it follows that it suffices to prove all axioms $\Phi(Ax^a)$ instead of all formulas ψ. We focus on the

[4] Since we do not want to exclude strict functions we do not require `isRep`$\bot$`=TT`.

[5] To compare the methods more precise both are presented.

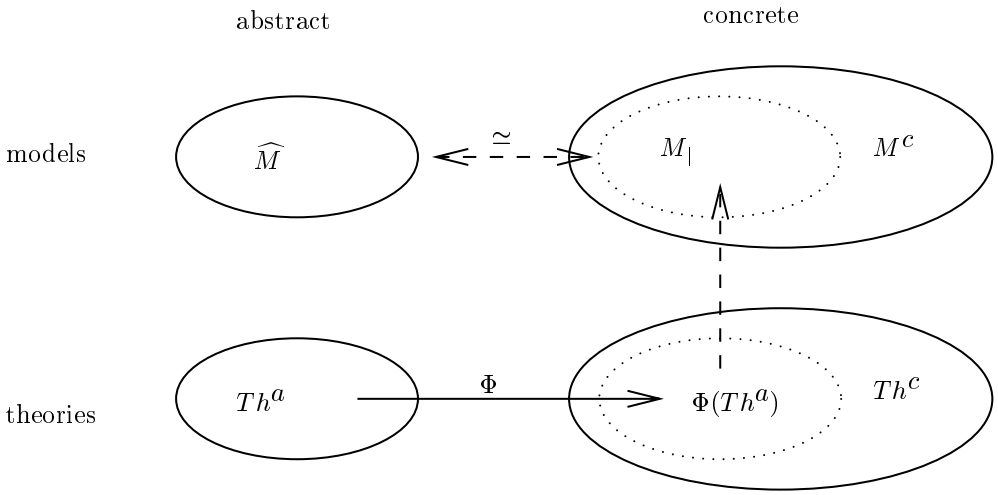

Figure 3.2: Satisfiability in HOLCF

special formula `unique` from Example 3.1.1 to demonstrate the normalization (see Section 3.2.5). To ensure the second requirement, satisfiability of theory interpretation, is our task in this section. It will turn out that this can be proved only under some assumptions that restrict the form of the specification. The method for the implementation, presented at the end of this section guarantees this restriction, since it embeds the implementation into the process of deductive software development.

Since our theory interpretation does not restrict functions (for reasons of continuity, see page 62) the model construction of $\widehat{M}$ is different from the construction used in Farmer's work. The model construction is depicted in Figure 3.2.

Theory Th^a is translated into Th^c, but only a part of Th^c is needed. This is modelled by a restriction of M^c to a restricted model $M_|$ that contains the semantics of $\Phi(Th^a)$. This reduct is equivalent to $\widehat{M}$. The proof of satisfiability proceeds as follows:

1. Definition 3.2.4: Φ
2. Definition 3.2.11: $M_|$
3. Theorem 3.2.2: $M_|[\![\Phi(Th^a)]\!] = M[\![\Phi(Th^a)]\!]$
4. Definition 3.2.16: $\widehat{M}$
5. Theorem 3.2.5: $M_| \models \Phi(Th^a)$ implies $\widehat{M} \models Th^a$

The new idea of our theory interpretation is that it checks the arguments of a function at the application and not at the λ-abstraction. This results in a simple translation rule Φ_2

and a more complex translation rule Φ_1 of the general definition of theory interpretation (see page 58). Since calculating the restriction predicate for the argument of a function is continuous (no $\forall$ quantification), continuous functions, based on continuous terms remain continuous. This trick can be understood as some kind of additional type checking in the concrete theory. Since the values of the abstract and the concrete type are represented by the same type the application guarantees that only "type correct" values are given to the functions.

However, things are not so easy, since the semantics is different. In Example 3.1.1 the identity operation on `B` is translated into the identity on `N`. This is correct for all applied occurrences of the identity, but using the semantics in a non-applied form causes troubles. Consider the formula:

- BOOL_ID: $\exists!$`f. f‘⊥=⊥` $\wedge$ `f‘T=T` $\wedge$ `f‘F=F`

It states that there exists only one identity function on `B`, and its translation should be true in `N`. Expanding the definition of $\exists!$`x.p(x)`$\equiv\exists$`x.p(x)`$\wedge$ $\forall$`y.p(y)` $\Longrightarrow$`x=y` shows that the semantics of `f` is directly used, since they are compared by = to the semantics of another function.

The first solution would be to translate the (application of) the equality of functions `f=g` by extensionality into $\forall$`x.f(x)=g(x)` before applying the theory interpretation Φ. However, this cannot be done, since some predicates (as $\exists!$ in the example) are defined polymorphically and therefore, the type dependent premise cannot be calculated for all instances correctly. This problem does only occur in predicates, since = is a non-continuous function and therefore, it must not be used in operations.

So the application of our theory interpretation would be restricted to theories that contain no polymorphic predicates applied to functional terms. This would be a handicap since one loses expressiveness, but the process of deductive software development provides a remedy: it requires, at the stage of the requirement specification, that all predicates describing the systems behaviour are fixed. Since predicates should be defined conservatively to ensure consistency (see page 23) there exists a defining term for the predicates. Replacing the polymorphic predicates applied to functions by their definition in a normalization step ensures that our theory interpretation can be applied (see Section 3.2.5).

Based on a sort translation μ and a constant translation φ the term translation Φ is defined.

3.2.1 Sort Translation μ

In order to define the implementation by theory interpretation formally it is necessary to have a translation from sort terms of the abstract requirement specification Th^a to sort terms from the concrete specification Th^c.

Definition 3.2.1 ***Sort Translation*** μ

Let $\tau \in T_\Omega a$ be an abstract sort and $\sigma \in T_\Omega c$ be a concrete sort then the sort translation $\mu : T_\Omega a_{(\Xi)} \longrightarrow T_\Omega c_{(\Xi)}$ is defined by:

- $\mu(\tau) = \sigma$ if τ is a type constant
- $\mu(\tau(\overline{t_i})) = \sigma(\overline{\mu(t_i)})$ if τ is a type constructor with variables unified by the type terms $\overline{t_i}$.
- $\mu(tc_n(\overline{t_i})) = tc_n(\overline{\mu(t_i)})$ if $tc_n \neq \tau$
- $\mu(\alpha) = \alpha$ if α is a variable in Ξ

We call σ the *corresponding sort* to τ.

In Example 3.1.1 the sort implementation `B` $\rightsquigarrow$ `N` can be modelled by the following sort translation μ by μ`(B)=N`. This translation can be applied to arbitrary sort terms of the abstract theory. For example μ`(`α `list→B)` $=$ α `list→N`.

To cut down notations, we use only translations, which translate one sort, since we would need a different restriction predicate and different premises for every implemented sort[6].

μ translates the abstract sort terms to concrete sort terms and leaves the other sorts unaffected. To ensure that this definition is well defined we need the requirement that the sort constructors $\{tc_j^a\} \subseteq \Omega^a$ of Th^a are contained in the sort constructors $\{tc_j^c\} \cup \{\tau\}$ of Th^c and τ. This is only a technical restriction to make the formalism more readable, since μ may be iterated.

3.2.2 Constant Translation φ

In order to define the implementation by theory interpretation formally it is necessary to have a translation from constants of the abstract requirement specification Th^a to corresponding terms from the concrete specification Th^c. Since constants of functional type are functions in HOLCF, not only constants of the specification, but also functions can be implemented with this definition of constant translation.

Definition 3.2.2 ***Constant Translation*** φ

Let $c_i^a \in \Sigma^a$ be abstract constants and $c_i^c \in T_{\Sigma^c}$ concrete terms then the constant translation $\varphi : \Sigma^a \longrightarrow T_{\Sigma^c}$ is defined by:

- φ_Corr : $\varphi(c_i^a :: u) = c_i^c :: \mu(u)$
- Φ_Else : $\varphi(c) = c$ for all $c \in \Sigma^a \wedge c \notin \{c_i^a\}$

[6]For the transitivity of our refinement relation we need the implementation of many sorts, but for practical applications it is not relevant, since the implementation may be repeated.

The c_i^c are called *corresponding constants* of the c_i^a. The constant translation has to respect the sort translation, i.e. the types of the corresponding functions have to equal to the results of the sort translation of the types of the abstract functions. This restriction ensures that the translated theory is type correct.

φ translates the implemented constants to corresponding terms and leaves the other constants unaffected. To ensure that this definition is well defined we need the requirement that the non-translated constants of Th^a are contained in the constants from Th^c. This can be formulated as: all constants from Th^a that are not in Th^c have to be translated by φ (Conservatively defined constants can be translated by translating their definitions).

Sort translation and constant translation are fixed by the user of the implementation method. In Example 3.1.1 the implementation would be:

- μ`(B)` = `N` (sort implementation)
- φ`(T)` = `One`
- φ`(F)` = `Zero`
- φ`(Bnot)` = Λ `x::N . One - x`
- φ`(Band)` = Λ `x::N y::N . If is_Zero‘x then Zero else y fi`
- Since `c` is defined conservatively it does not need to be translated by φ.

The constant translation respects the sort translation in the example, therefore the translated theory is type correct.

3.2.3 Term Translation Φ

On the basis of the sort translation μ and the constant translation φ, we can complete the definition of our theory interpretation by fixing the term translation[7]. For the translation of the application of terms we need a premise, which ensures the invariance. The premises are a generalization of the restriction predicate `isR` to the case of arbitrary sorts and sort constructors. In order to have a continuous restriction operation the premise has to be (equal to) `TT` for all corresponding values (except $\perp$) and `FF` for the other values.

Definition 3.2.3 ***Premises***

Let `isR::`$\sigma\rightarrow$`tr` be the continuous restriction predicate (of the implementation of the restriction step) on σ. Then the premise $\pi : T_\Omega a_{(\Xi)} \longrightarrow T_\Sigma c \Rightarrow$ `tr` is a translation, which produces for an abstract sort term $\rho \in T_\Omega a_{(\Xi)}$ a continuous representation predicate of type μ`(`ρ`)` $\rightarrow$ `tr`. The definition of the π-translation is:

[7] In the inductive Definition 2.3.1 on page 58 Φ_1 and Φ_2 are not fixed.

- π_VAR : $\pi(\alpha) = \lambda x.\texttt{TT}$ if α is a variable in Ξ
- π_FFUN : $\pi(s \Rightarrow t) = \lambda x.\texttt{TT}$
- π_TC_REP : $\pi(\tau(\overline{t_i})) = \lambda\ x\ .\ \texttt{cases}_m(\overline{dis_j\texttt{`}x \longrightarrow \texttt{and}_n(\overline{\pi_i(sel_{ji}\texttt{`}x)})})\ \texttt{and isR`}x$
- π_TC : $\pi(tc(\overline{t_i})) = \lambda\ x\ .\ \texttt{cases}_m(\overline{dis_j\texttt{`}x \longrightarrow \texttt{and}_n(\overline{\pi_i(sel_{ji}\texttt{`}x)})})$ if $tc \neq \tau$

  Where m is the number of different constructors and n_m are the numbers of selectors for each constructor and π_i are the restriction predicates of the sort of the selector's result type. If the selectors result type is the same as its argument (for example for recursive types) the fixed point construction would have to be applied[8].

Functions have no premises. They are checked when they are applied to arguments. Therefore, operations remain continuous. Obviously the resulting premises are continuous functions. Therefore, we may use Λ instead of λ, ` for the application of operations and $T_\Sigma c \rightarrow$ `tr` instead of $T_\Sigma c \Rightarrow$ `tr`. Premises are type dependent and can, therefore, not be formalized as a function in the logic HOLCF. The premise translation is part of the method and can be done automatically. No user input is necessary. The premises characterize all values that correspond to abstract values. For premises the following theorems can be derived for arbitrary type terms $s, t \in T_\Omega a$:

- π_CFUN : $\pi(s \rightarrow t) = \Lambda x.\texttt{TT}$
- π_CON : $\pi(\tau) = \Lambda x.\texttt{isR`}x$
- π_REST : $\pi(t) = \Lambda x.\texttt{TT}$ if the type τ does not occur in the type s.

The following example shows the calculation of premises for recursive types:

Example 3.2.1 ***Premise for Recursive Types***

The recursive type of finite lists is specified with the `domain` construct by:

```
domain   α DList   = dnil | dcons (fst::α) (rst::α DList)
```

Now the premises are calculated for Example 3.1.1:

```
π_(B DList)`dnil = TT
π_(B DList)`(dcons`x`s) = and` (π_B`x)` (π_(B DList)`s)
```

The premise for `B DList` is a recursive function. It may, for data types without pattern matching, be expressed by the following fixed point formulation:

[8]There is no principal difficulty in it, therefore, this case is only treated in Example 3.2.1.

```
π_(B DList) = fix‘ΛP l.If is_dnil‘l
                        then TT
                        else and‘ (isB‘(fst‘l))‘ (P‘(rst‘l))
                        fi
```

So all elements of a list are checked by the premise calculation.

Based on the premise π the term translation can be defined.

Definition 3.2.4 ***Term Translation*** Φ

Let μ, φ, π be sort, constant and premise translations. Then the term translation $\Phi : T_\Sigma a \longrightarrow T_\Sigma c$ is defined by:

- Φ_Con: $\Phi(c_i^a) = \varphi(c_i^a)$
- Φ_Var: Φ(x::u) = x::μ(u) for variables x
- Φ_Eq : Φ(a=b) = (Φa=Φb)
- Φ_Abs: $\Phi(\lambda$x::u.t) = λx::μ(u).Φt
- Φ_App: Φ(f t::u::pcpo) = Φ(f)(If πu‘(Φt) then Φt else $\perp$ fi)
- Φ_Rest: Φ(f t::u) = Φf Φt if u is not in pcpo.

The premises are tested when functions are applied to arguments. This ensures that the translation of applied Λ-terms does not leave the set of corresponding values. Therefore, it is not necessary to require that any Λ-term occurring in the specification is preserving.

This term translation is defined on arbitrary HOL terms. The following theorems can be proved by expanding the definitions of the HOLCF constants:

1. if f is continuous then $\Phi(f)$ is also.
2. $\Phi(\Lambda$x::u.t) = Λx::μ(u).Φ(t)
3. Φ(f‘ t::u) = If πu‘Φt then Φf‘Φt else Φf‘$\perp$ fi
4. $\Phi(t::u) = t::\mu(u)$ if τ does not occur in u.

As an example for the proof consider the proof of 2.

$$
\begin{array}{lcl}
\Phi(\Lambda \mathtt{x}::\mathtt{u}\ .\ \mathtt{t}::\mathtt{v}) & \stackrel{!}{=} & \Lambda\ \mathtt{x}::\mu(\mathtt{u})\ .\ \Phi(\mathtt{t}) \\
\Phi(\Lambda \mathtt{x}::\mathtt{u}\ .\ \mathtt{t}::\mathtt{v}) & = & \\
[\mathit{Syntax}] & = & \Phi(\mathtt{fabs}(\lambda \mathtt{x}.\mathtt{t})) \\
[\Phi\_\textsc{App}] & = & \Phi(\mathtt{fabs})(\mathtt{If}\ \pi(\mathtt{u}\Rightarrow\mathtt{v})(\Phi(\lambda\ \mathtt{x}.\mathtt{t}))\ \mathtt{then}\ \Phi(\lambda \mathtt{x}.\mathtt{t})\ \mathtt{else}\ \bot\ \mathtt{fi}) \\
[\pi\_\textsc{FFun}] & = & \Phi(\mathtt{fabs})(\Phi(\lambda \mathtt{x}.\mathtt{t})) \\
[\Phi\_\textsc{Con}] & = & \varphi(\mathtt{fabs})(\Phi(\lambda \mathtt{x}.\mathtt{t})) \\
[\phi\_\textsc{else}] & = & \mathtt{fabs}(\Phi(\lambda \mathtt{x}.\mathtt{t})) \\
[\Phi\_\textsc{Abs}] & = & \mathtt{fabs}(\lambda \mathtt{x}::\mu(\mathtt{u}).\Phi(\mathtt{t}))) \\
[\mathit{Syntax}] & = & \Lambda \mathtt{x}::\mu(\mathtt{u}).\Phi(\mathtt{t}) \qquad \square
\end{array}
$$

Since continuous functions are translated as general functions and, therefore, remain continuous we ignore the difference and use operations instead of functions whenever it is appropriate.

Two interesting facts can be derived in HOLCF since the quantifiers $\forall, \exists$ are defined in HOLCF as constants. For all types u in `pcpo`:

- $\Phi\_\forall$: $\quad \Phi(\forall \mathtt{x}::\mathtt{u}.\mathtt{p(x)}) \Longrightarrow (\forall \mathtt{x}::\mu\mathtt{u}.\ \lceil \pi \mathtt{u}\text{`}\mathtt{x}\rceil \Longrightarrow \Phi \mathtt{p(x)}\ )$
- $\Phi\_\exists$: $\quad \Phi(\exists \mathtt{x}::\mathtt{u}.\mathtt{p(x)}) \Longrightarrow (\exists\forall \mathtt{x}::\mu\mathtt{u}.\ \lceil \pi \mathtt{u}\text{`}\mathtt{x}\rceil \wedge \Phi \mathtt{p(x)}\ )$

As an example consider the proof of the first fact:

$$
\begin{array}{lcl}
\Phi(\forall \mathtt{x}::\mathtt{u}.\ \mathtt{p(x)}) & \stackrel{!}{\Longrightarrow} & \forall \mathtt{x}::\mu(\mathtt{u}).\ \lceil \pi \mathtt{u}\text{`}\mathtt{x}\rceil \Longrightarrow (\Phi \mathtt{p})\mathtt{x} \\
\Phi(\forall \mathtt{x}::\mathtt{u}.\ \mathtt{p(x)}) & = & \\
[\mathit{Syntax}] & = & \Phi(\forall \lambda \mathtt{x}.\mathtt{p(x)}) \\
[\forall\_\textsc{Def}] & = & \Phi((\lambda \mathtt{x}.\mathtt{p(x)}){=}(\lambda \mathtt{x}.\mathtt{True})) \\
[\Phi\_\textsc{Eq}] & = & \Phi(\lambda \mathtt{x}.\mathtt{p(x)}) = \Phi(\lambda \mathtt{x}.\mathtt{True}) \\
[\varphi\_\textsc{Else}] & = & \Phi(\lambda \mathtt{x}.\mathtt{p(x)}){=}\Phi(\lambda \mathtt{x}.\mathtt{True}) \\
[\Phi\_\textsc{Abs}] & = & \lambda \mathtt{x}.\Phi(\mathtt{p(x)}){=}\lambda \mathtt{x}.\Phi(\mathtt{True}) \\
[\Phi\_\textsc{App}] & = & \lambda \mathtt{x}.\mathtt{If}\ \pi \mathtt{u}\text{`}\Phi \mathtt{x}\ \mathtt{then}\ (\Phi \mathtt{p})\Phi \mathtt{x}\ \mathtt{else}\ (\Phi \mathtt{p})\bot\ \mathtt{fi}{=}\lambda \mathtt{x}.\mathtt{True} \\
[\Phi\_\textsc{Var}] & = & \lambda \mathtt{x}.\mathtt{If}\ \pi \mathtt{u}\text{`}\mathtt{x}\ \mathtt{then}\ (\Phi \mathtt{p})\mathtt{x}\ \mathtt{else}\ (\Phi \mathtt{p})\bot\ \mathtt{fi}{=}\lambda \mathtt{x}.\mathtt{True} \\
[\forall\_\textsc{Def}, \mathit{Syntax}] & = & \forall \mathtt{x}.\mathtt{If}\ \pi \mathtt{u}\text{`}\mathtt{x}\ \mathtt{then}\ (\Phi \mathtt{p})\mathtt{x}\ \mathtt{else}\ (\Phi \mathtt{p})\bot\ \mathtt{fi} \\
[\mathit{if}] & \Longrightarrow & \forall \mathtt{x}.\lceil \pi \mathtt{u}\text{`}\mathtt{x}\rceil \Longrightarrow (\Phi \mathtt{p})\mathtt{x} \qquad \square
\end{array}
$$

These theorems correspond to Farmer's definitions of relativations in his theory interpretations [Far94b].

One requirement for the correctness of theory interpretations in our approach is the invariance of the restriction predicate with respect to the corresponding operations.

3.2.4 Invariance and Preserving Functions

A function is called Π-*preserving*, if Π is invariant with respect to the function. Invariance is necessary to show that the reduct of the models suffices to define the semantics of translated terms. Informally invariance of the restriction predicate of the corresponding operations means that the results of the operations that are used instead of the abstract ones do not leave the set of corresponding elements (see Figure 3.1 on page 65).

In the case of term translations with premises π this means that the premise holds for the translations of all elements. So it is important that the restriction operation `isR` fits to the corresponding operations (see Section 5.2 for this aspect).

The aim of this section is to give syntactic verification conditions that ensure that the invariance holds for all possible terms. First a definition of invariance is needed:

Definition 3.2.5 ***Invariance and Preserving Function***

For a term translation Φ with φ, μ, π from Th^a to Th^c the invariance proof obligation for a functional term is defined by:

- INVARIANCE: $M[\![Inv(f::\mu u)]\!]^{\Sigma}_{\eta} = \mathbb{T}$ for all $f \in T_{\Sigma}c$ where
- INV_DEF: $Inv(f::\mu(s_1)\rightarrow \ldots \rightarrow \mu(s_{n+1})) = \forall \overline{x_j}.\ \bigwedge_j\ (\overline{\Pi_{s_j}(x_j)}) \Longrightarrow \Pi_{s_{n+1}}(f`(\overline{x_j}))$ where
- Π_DEF: $\Pi_s(x) = (\mathtt{x}{=}\bot \vee \lceil \pi \mathtt{s`x} \rceil)$.

A function f that fulfils $Inv(f)$ is called *preserving*.

It is obvious, that the application of two preserving terms gives an preserving term:

- INV_TERM: $Inv(t_1) \wedge Inv(t_2) \Longrightarrow Inv(t_1\ t_2)$

Π is introduced in addition to π in order to allow a continuous realization of the premise operation π. For reducing the invariance proof obligations to all representing constants we require that invariance holds for all variables. This is ensured by our method.

The invariance requirements for the user of the method can be generated automatically from the type of the abstract constants $\mathtt{c}_i^a$. The method for the implementation (see Section 3.2.8) requires the proof of these proof obligations. The user has to respect them when choosing the representations for the required operations and constants (see Section 5.2).

Definition 3.2.6 ***Invariance Requirement***

For a term translation Φ from Th^a to Th^c with premise π the invariance requirements are defined by:

- INV_FUN : $M[\![\forall \overline{\mathtt{x}_j}.\ \bigwedge_j\ (\overline{\Pi_{\mathtt{s}_j}(\mathtt{x}_j)}) \Longrightarrow \Pi_t(\mathtt{c}_i^c\text{`}\overline{x_j})]\!]_\eta^\Sigma = \mathbb{T}$
  for all $\mathtt{c}_i^a :: \mathtt{s}_1 \to ... \to \mathtt{s}_n \to t$ with $\varphi(\mathtt{c}_i^a) = \mathtt{c}_i^c$.

In the special case of simple constants this means:

- INV_CON: $M[\![\lceil \pi(\mathtt{t})\text{`}\mathtt{c}_i^c \rceil\ \vee \mathtt{c}_i^c = \bot]\!]_\eta^\Sigma = \mathbb{T}$ for all $\mathtt{c}_i^a :: t$ and

In Example 3.1.1 the invariance requirements are:

- for `T` : $M[\![\lceil \mathtt{isB`One} \rceil \vee \mathtt{One} = \bot]\!]_\eta^\Sigma = \mathbb{T}$
- for `F` : $M[\![\lceil \mathtt{isB`Zero} \rceil \vee \mathtt{Zero} = \bot]\!]_\eta^\Sigma = \mathbb{T}$
- for `not` : $M[\![\forall \mathtt{x}.(\lceil \mathtt{isB`x} \rceil \vee \mathtt{x} = \bot) \Longrightarrow (\lceil \mathtt{isB`(One-x)} \rceil \vee \mathtt{One-x} = \bot)]\!]_\eta^\Sigma = \mathbb{T}$
- for `and` : $M[\![\forall \mathtt{x} \forall \mathtt{y}.\Pi_{\mathtt{B}}(x) \wedge \Pi_{\mathtt{B}}(\mathtt{y}) \Longrightarrow \Pi_{\mathtt{B}}(\mathtt{If\ is\_Zero`x\ then\ Zero\ else\ y\ fi})]\!]_\eta^\Sigma = \mathbb{T}$
  where $\Pi_{\mathtt{B}}(x) = \lceil \mathtt{isB`x} \rceil\ \vee\ \mathtt{x} = \bot$

The invariance proof obligations may reduce from Π to π if the corresponding constants are defined.

Now it is shown that terms, resulting from the translation of abstract terms are preserving the invariance.

Theorem 3.2.1 *Inv_Φ_THM*

If Φ is a term translation from Th^a to Th^c with premises π, Π and invariance Inv then

- $M[\![Inv(\Phi t)]\!]_\eta^\Sigma = \mathbb{T}$ for $Inv(\eta(x))$ for all $x \in \Psi$

Proof

$\mathbf{t}=c$	$M[\![Inv(\Phi c)]\!]_\eta^\Sigma$	$=$	
	[INV_CON]	$=$	$\mathbb{T}$
$\mathbf{t}=\mathbf{x}$	$M[\![Inv(\Phi x)]\!]_\eta^\Sigma$	$=$	
	$[Hyp]$	$=$	$\mathbb{T}$

$$
\begin{array}{lllll}
\mathbf{t}{=}\mathbf{f}{::}(\mathtt{v}{\to}\mathtt{u})\text{`}\mathtt{z} & M[\![Inv(\Phi(\mathtt{f}\text{`}\mathtt{z}))]\!]^{\Sigma}_{\eta} & = & & \\
 & [\Phi\_\textsc{App}] & = & M[\![Inv(\Phi\mathtt{f}\text{`}\mathtt{If}\ \pi\mathtt{v}\text{`}\Phi\mathtt{z}\ \mathtt{then}\ \Phi\mathtt{z}\ \mathtt{else}\ \bot\ \mathtt{fi})]\!]^{\Sigma}_{\eta} \\
\mathit{case} & -\ \pi\mathtt{v}\text{`}\Phi\mathtt{z}{=}\bot & = & M[\![Inv(\Phi\mathtt{f}\text{`}\bot)]\!]^{\Sigma}_{\eta} \\
 & [\textsc{Inv\_Term}] & \Longleftarrow & M[\![Inv(\Phi f)\wedge Inv(\bot)]\!]^{\Sigma}_{\eta} \\
 & [IHyp] & = & \mathbb{T} \\
\mathit{case} & -\ \lfloor\pi\mathtt{v}\text{`}\Phi\mathtt{z}\rfloor & = & M[\![Inv(\Phi\mathtt{f}\text{`}\bot)]\!]^{\Sigma}_{\eta} \\
 & [\textsc{Inv\_Term}] & \Longleftarrow & M[\![Inv(\Phi f)\wedge Inv(\bot)]\!]^{\Sigma}_{\eta} \\
 & [IHyp] & = & \mathbb{T} \\
\mathit{case} & -\ \lceil\pi\mathtt{v}\text{`}\Phi\mathtt{z}\rceil & = & M[\![Inv(\Phi\mathtt{f}\text{`}\Phi\mathtt{z})]\!]^{\Sigma}_{\eta} \\
 & [\textsc{Inv\_Term}] & \Longleftarrow & M[\![Inv(\Phi f)\wedge Inv(\Phi z)]\!]^{\Sigma}_{\eta} \\
 & [IHyp] & = & \mathbb{T}
\end{array}
$$

$$
\begin{array}{lllll}
\mathbf{t}{=}\mathbf{\Lambda}x.f{::}\mathbf{v}{\to}\mathbf{w} & M[\![Inv(\Phi(\Lambda\mathtt{x}.\mathtt{f}))]\!]^{\Sigma}_{\eta} & = & & \\
 & [\Phi\_\textsc{Abs}] & = & M[\![Inv(\Lambda\mathtt{x}.\Phi\mathtt{f})]\!]^{\Sigma}_{\eta} \\
 & [\textsc{Inv\_Def}] & = & M[\![\Pi_{\mathtt{v}}(x){\Longrightarrow}\Pi_{\mathtt{w}}(\Phi f)]\!]^{\Sigma}_{\eta} \\
(\star) & \Pi_{\mathtt{v}}(x) & & \Longrightarrow Inv(\eta(x)) \\
 & [IHyp,(\star)] & = & \mathbb{T} \qquad\qquad \square
\end{array}
$$

So the invariance of the restriction predicate with respect to all corresponding terms can be guaranteed if the representing constants are preserving.

From INV_Φ_THM follows:

- DEF_Φ_TRUE: $M[\![\lceil\pi\mathtt{s}\text{`}\Phi(\mathtt{t}{::}\mathtt{s})\rceil\ \vee\ \Phi\mathtt{t}{=}\bot]\!]^{\Sigma}_{\eta} = \mathbb{T}$

This expresses the inability to write abstract terms for which the translation is defined, but which have no corresponding value. In Example 3.1.1 this means that we cannot write a term in `BOOLEAN` that corresponds to `2`.

The next step is to give the normalization, which ensures that the axioms of the abstract specifications are satisfiable.

3.2.5 Normalization

To show that we defined a theory interpretation in the sense of Definition 2.3.1 we have to prove satisfiability. The proof of satisfiability is only possible for a restricted class of axioms.

Normalization transforms (almost) all axioms to this form. The same problem arises when the semantics of functions is defined only for applied values, which simplifies many semantics (see [BFG+93a, BFG+93b] for an applied semantics definition of a parametrized specification language). In contrast to this purely algebraic question the implementation of ADTs is a part of the deductive software development process. This process ensures that all axioms are of the right form when an implementation step is performed.

As an example of the necessity of normalization consider the identity on the abstract sort. It should be uniquely defined. This can be expressed by:

- BOOL_ID: $\exists!$`f. f‘⊥=⊥ ∧ f‘T=T ∧ f‘F=F`

This holds in `BOOLEAN`, but translating this formula to `NAT`

- NAT_ID: $\exists!$`f. f‘⊥=⊥ ∧ f‘One=One ∧ f‘Zero=Zero`

gives an incorrect rule since `f` is not fixed on the other values. The reason is that $\exists!$ uses the semantics of function `f` and compares it directly to other functions. Expanding the definition of $\exists!$ ($\exists!$`x.p(x)` $\equiv$ $\exists$`x.p(x)` $\wedge$ $\forall$`y.p(y)` $\Longrightarrow$ `x=y`) yields

- BOOL_ID2: $\exists$`f.(f‘⊥=⊥∧f‘T=T∧f‘F=F)∧∀g.(g‘⊥=⊥∧g‘T=T∧g‘F=F)⟹f=g`

The translation of this formula

- NAT_ID2: $\exists$`f.(f‘⊥=⊥∧f‘One=One∧f‘Zero=Zero)∧`
  `∀g.(g‘⊥=⊥∧g‘One=One∧g‘Zero=Zero)⟹f=g`

would not hold in `NAT` since there may be a lot of other functions `g` that are not equal to `f`. The semantics SEM_EQ of = uses the functions directly without applying them. Normalizing with the extensionality of functions from HOL (`(f=g)` = $\forall$`x.f(x)=g(x)`) gives

- BOOL_ID3: $\exists$`f.(f‘⊥=⊥∧f‘T=T∧f‘F=F)∧`
  `∀g.(g‘⊥=⊥∧g‘T=T∧g‘F=F)⟹∀ x.f‘x=g‘x`

Since now the operations are compared by comparing their result on all input and since these applications are translated by Φ the result of the translation is the following formula:

- NAT_ID3: $\exists$`f.(f‘⊥=⊥∧f‘One=One∧f‘Zero=Zero)∧`
  `∀g.(g‘⊥=⊥∧g‘One=One∧g‘Zero=Zero)⟹`
  `∀x.(If isB‘x then f‘x else f‘⊥ fi = If isB‘x then g‘x else g‘⊥ fi)`

The requirement for this normalization is that all occurrences of = between operations on the abstract type can be eliminated. This requires to have normalizations for all polymorphic predicates that may be applied to abstract operations. In operations the identity cannot be used since this would violate continuity.

For all predicates that are introduced conservatively the normalization corresponds to an expansion of the predicate definitions. In the early phases of software engineering with specifications it may be useful to describe predicates abstractly (in a non-conservative way) by axioms. However, in the first phase of the deductive software development process it is necessary to refine the predicates into a precise requirement specification and to give an explicit (conservative) definition of those predicates to allow a compositional refinement.

Since the implementation of interactive systems belongs to the design and realization phase in software engineering, the requirement of conservative definitions for polymorphic predicates (on implemented operations) is no handicap for the method of implementing ADTs.

Normalization ensures that the specification is in the right syntactic form to successfully apply theory interpretation. The method for the implementations with restrictions in Section 3.2.8 checks these requirements automatically.

Definition 3.2.7 ***Normal***

A theory $T = (\Sigma, Ax)$ is called normal if

- no polymorphic predicates are applied to operations in Ax.

Especially no operations are compared by = (eg. $(\Lambda \mathtt{x}.\mathtt{x})=\bot$).

Definition 3.2.8 ***Normalization***

The process of replacing polymorphic predicates on continuous polymorphic functions by their definitions is called normalization. For normalization it is required that the polymorphic predicates are explicitly defined in the way of a conservative extension: $\mathtt{p} \equiv \lambda \mathtt{x}.\mathtt{q(x)}$. Normalization in addition uses the extensionality of operations

- NORM_EXT: $(\mathtt{f::s{\to}t{=}g} \equiv \forall \mathtt{x::s}.\ \mathtt{f`x} = \mathtt{g`x})$.

A specification which has only conservatively defined polymorphic predicates is called *normalizeable*.

Since HOLCF is conservative, for all predefined predicates in HOLCF such a definition exists. For example:

- NORM_DELTA: $\delta\mathtt{(f)} \equiv \mathtt{f} \neq \bot$

- NORM_NEQ: $\texttt{f} \neq \texttt{g} \equiv \neg(\texttt{f}{=}\texttt{g})$
- NORM_EX1: $\exists!\texttt{x.p(x)} \equiv \exists\texttt{x.p(x)} \wedge \forall\texttt{y.p(y)} \Longrightarrow \texttt{x}{=}\texttt{y}$

Normalization ensures that no operations are compared by =. Therefore, for normalized theories it suffices to define the semantics when operations are applied.

Definition 3.2.9 ***Applied Semantics***

According to Definition 2.1.6 of models the semantics of a functional term $\texttt{f}::\alpha{\Rightarrow}\beta$ is used when

1. f is compared by = to another functional term
2. f is an argument of a function $\lambda\texttt{x.z}$ then the semantics of $(\lambda\texttt{x.z})\texttt{f}$ is $M[\![z]\!]^{\Sigma}_{\eta(f/\texttt{x})}$. The semantics of f is stored in the valuation η and may be used in z.
3. f is under a λ-abstraction: $F = \lambda x.f$ then the semantics is used when the semantics of F is used.
4. f is applied to an argument t. Then $M[\![f(t)]\!]^{\Sigma}_{\eta} = (M[\![f]\!]^{\Sigma}_{\eta})(M[\![t]\!]^{\Sigma}_{\eta})$ the semantics of f is *really used* for a given argument.

Normalization eliminates case 1. by applying NORM_EXT, so it suffices to define the semantics of abstraction only for possible arguments:

- NORMALABS: $M[\![\Lambda x.f]\!]^{\Sigma}_{\eta} = f$ where $f{::}a \in \{M[\![\Phi t]\!]^{\Sigma}_{\eta}\}{\mapsto}M[\![f]\!]^{\Sigma}_{\eta(a/\texttt{x})}$

This is a partial definition. Since this semantics is used for translated theories all arguments have to be translations of abstract terms (corresponding elements). The difference to INT_ABS (see page 39) is only important when the semantics of function is "really used", which is only the case when functions are compared by =. Normalization ensures that this is not the case.

3.2.6 Reduct of Models

In Farmer's theory interpretation it is easy to give a model which allows us to show satisfiability, since the logic has subtypes, which may be characterized by arbitrary predicates. Therefore, the required model is simply characterized by the subtype defined by the restriction predicate. In HOLCF we do not have such subtypes in the models. Therefore, we build a reduct from the general models.

The key idea of the proof of satisfiability is that not all terms in $Th^{\mathcal{C}}$ are needed. The term translation Φ generates only a subset of values in $Th^{\mathcal{C}}$. It translates constants to

corresponding constants and the invariance of the corresponding functions ensures that some terms cannot be reached by the translation. In Example 3.1.1 one cannot write a term in `BOOLEAN` that corresponds to the natural number `2`. Therefore, we do not need the semantics of `2` in the proof of satisfiability and we restrict our models to models of corresponding terms. However, things become a bit more complicated, since for example the semantics of Λ`x::B.x` does not correspond to the semantics of Λ`x::N.x`. Therefore, the general definition of reduct is needed. It is based on the following definition.

Definition 3.2.10 ***Reduct of Type Models***

Let π be the premise of a term translation and let TM be a type model and $s \in T_\Omega$ be the restricted data type then the s-reduct of the type model $TM_{|s}$ is defined by:

- $TM_{|s} \;=\; (PU_{|s}, TC)$ where
- $PU_{|s} \;=\; \{X_{|s} \mid X \in PU\}$ where
- $X_{|s} \;=\; \begin{cases} \{x \in X \mid \lceil \pi(\mathtt{s}\text{`}\mathtt{x}) \rceil \vee \mathtt{x}{=}\bot\} & \text{if } TM[\![s]\!]^\Omega = X \\ X & \text{else} \end{cases}$

Since PU is closed under nonempty subsets, $TM_{|s}$ is well defined.

This type restriction depends on the type s. This is used to restrict the models selectively. This means that types that correspond to abstract types are only restricted when they are used for representation of abstract values. The type model restriction is so constructed that for the interpretation of type terms for reduced type models the following equivalence holds:

- TM$_{|}$_Φ: $\quad TM_{|s}[\![s]\!]^\Omega = \{M[\![\Phi(t::s)]\!]^\Sigma_\eta\}$

The proof is based on Inv_Φ_Thm. In Example 3.1.1 the type of `N` is only restricted when it is used to represent boolean values: $TM_{|\mathtt{B}}[\![\mathtt{N}]\!]^\Omega = \{\bot, \mathtt{Zero}, \mathtt{One}\}$ but $TM_{|\mathtt{N}}[\![\mathtt{N}]\!]^\Omega = \mathbb{N}_\bot$.

The reduct of models depends also on a given type:

Definition 3.2.11 ***Reduct of Models***

Let $TM_{|s}$ be a reduct of a type model. Then the s-reduct of the model $M_{|s}$ defined by:

- Red_Con: $\quad M_{|s}[\![c]\!]^\Sigma_\eta \;=\; M[\![c]\!]^\Sigma_\eta$ for $\mathtt{c}{\in}\Sigma$,
- Red_Var: $\quad M_{|s}[\![x]\!]^\Sigma_\eta \;=\; M[\![x]\!]^\Sigma_\eta$ for $\mathtt{x}{\in}\Psi$,
- Red_App: $\quad M_{|s}[\![f::s{\to}t\ z]\!]^\Sigma_\eta \;=\; (M_{|s\to t}[\![f]\!]^\Sigma_\eta)(M_{|s}[\![z]\!]^\Sigma_\eta)$ and
- Red_Abs: $\quad M_{|s\to t}[\![\Lambda x.z]\!]^\Sigma_\eta \;=\; f$ where f is the, by extensionality, exactly fixed function $f::a \in TM_{|s}[\![s]\!]^\Omega {\mapsto} M_{|t}[\![z]\!]^\Sigma_{\eta(a/\mathtt{x})}$.

The reduct is well defined for all translated terms, since they are preserving and therefore, the application of a function to an argument is of the restricted type.

With this definition the semantics of operations is reduced to the corresponding elements. The reduct restricts only corresponding functions. In Example 3.1.1 the identity on `N` that corresponds to the identity on `B` is reduced to an identity which is only defined for corresponding natural numbers, whereas the identity on natural numbers remains unaffected. With this reduct, the restriction is defined as an additional type among other types of the implementing theory.

The first theorem states that the semantics of translated terms are the same as the reducts of the semantics of translated terms. Exactly this is the aim of the definition of reduct. Then next step in Definition 3.2.16 is to construct a model for the abstract specification to show the satisfiability.

Theorem 3.2.2 Φ***-Reduct***

If Φ is a term translation from Th^a to Th^c and M a model of Th^c then

- $M_{|u}[\![\Phi(t::u)]\!]^\Sigma_\eta = M[\![\Phi(t)]\!]^\Sigma_\eta$

Proof

$\mathbf{t} = c$

$$\begin{array}{lll} M_{|u}[\![\Phi(c)]\!]^\Sigma_\eta & = & \\ [\text{RED\_CON}, \Phi\text{\_CON}] & = & M[\![\Phi(c)]\!]^\Sigma_\eta \end{array}$$

$\mathbf{t} = x$

$$\begin{array}{lll} M_{|u}[\![\Phi(x)]\!]^\Sigma_\eta & = & \\ [\text{RED\_VAR}, \Phi\text{\_VAR}] & = & M[\![\Phi(x)]\!]^\Sigma_\eta \end{array}$$

$\mathbf{t} = f::(s \to \mathbf{t})`z$

$$\begin{array}{lll} M_{|t}[\![\Phi(f`z)]\!]^\Sigma_\eta & = & \\ [\Phi\text{\_APP}] & = & M_{|t}[\![\Phi f` \texttt{If } \pi s` \Phi z \texttt{ then } \Phi z \texttt{ else } \bot \texttt{ fi}]\!]^\Sigma_\eta \\ case\ -\pi s` \Phi z = \bot & = & M_{|t}[\![\bot]\!]^\Sigma_\eta \\ [\text{RED\_CON}] & = & M[\![\bot]\!]^\Sigma_\eta \\ case\ -\lfloor \pi s` \Phi z \rfloor & = & M_{|t}[\![\Phi f` \bot]\!]^\Sigma_\eta \\ [\text{RED\_APP}] & = & (M_{|s \to t}[\![\Phi f]\!]^\Sigma_\eta) M_{|s}[\![\bot]\!]^\Sigma_\eta \\ [Hyp] & = & (M[\![\Phi f]\!]^\Sigma_\eta) M[\![\bot]\!]^\Sigma_\eta \\ case\ -\lceil \pi s` \Phi z \rceil & = & M_{|t}[\![\Phi f` \Phi z]\!]^\Sigma_\eta \\ [\text{RED\_APP}] & = & (M_{|s \to t}[\![\Phi f]\!]^\Sigma_\eta) M_{|\mathbf{s}}[\![\Phi \mathbf{z}]\!]^\Sigma_\eta \\ [Hyp] & = & (M[\![\Phi f]\!]^\Sigma_\eta) M[\![\Phi z]\!]^\Sigma_\eta \end{array}$$

$$
\begin{array}{lll}
\mathbf{t} = \mathbf{\Lambda} x::s.f & M_{|s\to t}[\![\Phi(\Lambda x.f)]\!]^{\Sigma}_{\eta} & = \\
& [\Phi\_\textsc{Abs}] & = M_{|s\to t}[\![\Lambda x.\Phi f]\!]^{\Sigma}_{\eta} \\
& [\textsc{Red\_Abs}] & = f::a \in TM_{|s}[\![s]\!]^{\Omega} \mapsto M_{|t}[\![\Phi f]\!]^{\Sigma}_{\eta(a/\mathtt{x})} \\
& [IHyp] & = f::a \in TM_{|s}[\![s]\!]^{\Omega} \mapsto M[\![\Phi f]\!]^{\Sigma}_{\eta(a/\mathtt{x})} \\
& [\mathrm{TM}_{|}\_\Phi] & = f::a \in \{M[\![\Phi(z::s)]\!]^{\Sigma}_{\eta}\} \mapsto M[\![\Phi f]\!]^{\Sigma}_{\eta(a/\mathtt{x})} \\
& [\textsc{NormalAbs}] & = M[\![\Phi(\Lambda x.f)]\!]^{\Sigma}_{\eta} \qquad \square
\end{array}
$$

This is a structural induction proof that respects all cases of terms. The interesting case is the case of the λ-abstraction. In this case the normalization (NormalAbs) plays an important role. It guarantees that the semantics of restricted functions is equal to semantics of unrestricted functions for all normalized translated terms.

3.2.7 Model Construction $\widehat{\Phi}$

For satisfiability it is necessary that for every theory interpretation Φ an "inverse" model construction exists. This model construction extends the models M of the concrete theory by introducing a new type (as in Section 2.1.4) and new constants conservatively. The main difference to the usual conservative extensions is that we give a scheme α as a methodical help for introducing the corresponding constants, which ensures that they have the type required from the abstract constants. In this section we use this scheme only on the level of models, but in Section 3.3 this scheme is used on a syntactic level to explicitly introduce operations. This scheme works also for type constructors and higher order functions and it can be computed automatically from the type of the abstract constants.

First the type model is defined.

Definition 3.2.12 ***Type Model Construction***

Let $\tau \in \Omega^a$ be the abstract sort and $\sigma \in T_{\Omega^C}$ the corresponding concrete sort term then the type model construction $\widehat{TM}$ for a concrete type model $TM = (PU, TC)$ is defined by:

- $\widehat{TM} = (PU, TC \cup \{\hat{\sigma}\})$ where
- $\hat{\sigma} = \{x \in TM[\![\sigma]\!]^{\Omega} \mid x{=}\bot \vee \lceil \pi(\tau)\text{`}x \rceil\}$

Since the definition of $\hat{\sigma}$ is equivalent to the Definition 3.2.10 of reduct of type model the following holds:

- TM_Equal: $TM_{|v}[\![v]\!]^{\Omega} = \widehat{TM}[\![v]\!]^{\Omega}$
- TM_Red: $\widehat{TM}[\![\mu v]\!]^{\Omega} = TM[\![\mu v]\!]^{\Omega}$

As in [Reg94, Section 2.6] for the new type abstraction and representation functions are defined to relate the new type with the other types.

Definition 3.2.13 ***Embedding Functions***

Let $\hat{\sigma}$ be a new type then the embedding functions are defined

- ABS_DEF: $abs::\sigma \Rightarrow \hat{\sigma}$ by
  $abs(y) = \begin{cases} y & \text{if } y{=}\bot \ \vee \ \lceil \pi(\tau)\,`\,y \rceil \\ \bot & \text{else} \end{cases}$
- REP_DEF: $rep::\hat{\sigma} \Rightarrow \sigma$ by $rep(y) = y$

Since the subset predicate for the types is selected carefully we can show, using the admissibility that $\hat{\sigma}$ is a pcpo and that *abs* and *rep* are continuous[9].

From the definitions it can be seen that the representations are preserving:

- Π_REP: $\Pi_u(rep`(x::u))$

where Π is the same abbreviation as in Π_DEF from Definition 3.2.5.

Based on this embedding operations the scheme for the introduction of constants can be defined. It works for arbitrary terms, including functions and other data type constructors. For the introduction of constants with a type that involves type constructors the map functionals together with constructor functions and selector functions are needed. They are defined for basic types in Section 2.1.5, and for new types the functions may be defined schematically with the `when` functional from the `domain` construct for the conservative introduction of data types.

Definition 3.2.14 ***Construction Scheme*** α

Let Φ be a term translation. Then the construction scheme $\alpha : T_\Omega a \longrightarrow T_\Sigma c \Rightarrow T_\Sigma a$ for a given target type and a corresponding constant is defined by:

- α_VAR: $\alpha_x(t) = t$ for $x \in \Xi$
- α_FUN: $\alpha_{a \Rightarrow b}(f) = \lambda x::a.\alpha_b(f(\tilde{\alpha}_a(x)))$
- α_TAU: $\alpha_{\tau(\overline{t_i})}(t) = \mathtt{abs}(Map_\sigma(\overline{\lambda y::\mu t_i.\alpha_{t_i}(y)})(t))$
- α_TC: $\alpha_{tc(\overline{t_i})}(t) = Map_{tc}(\overline{\lambda y::\mu t_i.\alpha_{t_i}(y)})(t)$ if $\mathtt{tc} \neq \tau$

Since `abs` is continuous the resulting function for every type index is continuous.

[9] The continuity of *abs* requires `isR` to be total. See page 104 for more details.

This scheme does the lifting from the corresponding terms $\mathtt{c}_i^c$ to abstract functions $\mathtt{c}_i^a$. It is a type dependent scheme and allows us to lift functions of arbitrary, higher order types. The lifting of concrete functions to abstract functions requires a representation of the arguments and an abstraction of the result. The abstraction is supported by α and for the representation we have a dual construction, which maps to representing values:

Definition 3.2.15 ***Dual Construction Scheme*** $\tilde{\alpha}$

Let Φ be a term translation. Then the dual construction scheme $\tilde{\alpha} : T_\Omega a \longrightarrow T_\Sigma a \Rightarrow T_\Sigma a$ for a given target type and a corresponding constant is defined by:

- $\tilde{\alpha}$_VAR: $\tilde{\alpha}_x(t) = t$ for $x \in \Xi$
- $\tilde{\alpha}$_FUN: $\tilde{\alpha}_{a \Rightarrow b}(f) = \lambda x{::}\mu a.\tilde{\alpha}(b, f(\alpha_a, x))$
- $\tilde{\alpha}$_TAU: $\tilde{\alpha}_{\tau(\overline{t_i})}(t) = \mathtt{rep}(Map_\tau(\overline{\lambda y{::}t_i.\tilde{\alpha}_{t_i}(y)})(t))$
- $\tilde{\alpha}$_TC: $\tilde{\alpha}_{tc(\overline{t_i})}(t) = Map_{tc}(\overline{\lambda y{::}t_i.\tilde{\alpha}_{t_i}(y)})(t)$ if $\mathtt{tc} \neq \tau$

Since `rep` is continuous the resulting function for every type index is also continuous.

This scheme does the representation from abstract to corresponding values. As on page 75 it may be proved that the continuous abstraction Λ is constructed by the scheme in a continuous way. In the following we, therefore, apply the construction directly to continuous functions.

Obviously, for any abstract type and any concrete constant the schematically generated function is continuous, since `abs`, `rep`, and the map functionals are continuous.

With the schemes α and $\tilde{\alpha}$ it is easy to construct the abstract model for any translation. The idea of the schemes is to build abstractions and representations for all terms. For terms constructed by type constructors the map functional is used to reach all subterms.

Definition 3.2.16 ***Model Construction*** $\widehat{\Phi}$

Let Φ be a translation and $M = (\Omega, C)$ a concrete model. Then the model construction $\widehat{\Phi}$ is defined by:

- $\widehat{\Phi}(M) = (\widehat{TM}, C \cup \{\mathtt{abs}, \mathtt{rep}\} \cup \{\mathtt{c}_i^a\})$

In the following we use $\widehat{M}$ instead of $\widehat{\Phi}(M)$.

The interpretation for `abs` and `rep` is *abs* and *rep* from Definition 3.2.13. The interpretation for the abstract constants with the schemes is based on *abs*, *rep* and on the interpretation of the corresponding constants.

- INT_abs: $\widehat{M}[\![\mathtt{abs}]\!] = abs$
- INT_rep: $\widehat{M}[\![\mathtt{rep}]\!] = rep$
- INT_$\mathtt{c}_i^a$: $\widehat{M}[\![\mathtt{c}_i^a :: u]\!] = \widehat{M}[\![\alpha_u(\mathtt{c}_i^c)]\!]$

The interpretation of terms in $\widehat{M}$ is as in Definition 2.1.7. With this the interpretations in Example 3.1.1 are:

- $\widehat{M}[\![\mathtt{not}]\!]^\Sigma_{\hat{\eta}} = \widehat{M}[\![\alpha_{\mathtt{B}\to\mathtt{B}}(\varphi(\mathtt{not}))]\!]^\Sigma_{\hat{\eta}} = \widehat{M}[\![\Lambda\mathtt{x.abs`(One{-}rep`x)}]\!]^\Sigma_{\hat{\eta}}$
- $\widehat{M}[\![\mathtt{Map\_DList`not`(dcons`T`dnil)}]\!]^\Sigma_{\hat{\eta}} =$
  $\widehat{M}[\![\alpha_{(\mathtt{B\ DList})}\Phi(\mathtt{Map\_DList`Bnot`(dcons`T`dnil)})]\!]^\Sigma_{\hat{\eta}} =$
  $\widehat{M}[\![\mathtt{Map\_DList`(\Lambda x.abs`x)`(Map\_DList`(\Lambda x.One{-}x)`(dcons`One`dnil))}]\!]^\Sigma_{\hat{\eta}}$

Note that the construction schemes construct only the semantics and does not introduce the constants syntactically, as in the conservative extension in Section 3.3.

For the proof of satisfiability two general theorems over α and $\tilde{\alpha}$ are necessary:

Theorem 3.2.3 α_THM

Let Φ be a translation with φ, μ, π and invariance Inv and Π as in Definition 3.2.6 and α the construction scheme of Definition 3.2.14 then

- $Inv(t::\mu u)$ implies $\widehat{M}[\![\alpha_u(t)]\!]^\Sigma_{\hat{\eta}} = M_{|u}[\![t]\!]^\Sigma_\eta$

The induction proof goes over the type structure of u since α and π depend on it.

Proof

$$
\begin{array}{lll}
u = \tau(\overline{t_i}) & \widehat{M}[\![\alpha_{\tau(\overline{t_i})}(t)]\!]^\Sigma_{\hat{\eta}} & = \\
& Hyp & : \ Inv(t) \\
& [\text{INV\_DEF}] & = \ \Pi_u(t) \\
& [\Pi\text{\_DEF}] & = \ \lceil \pi(\tau(\overline{t_i}))`t \rceil \vee t{=}\bot \\
& [\pi\text{\_TC\_REP}] & = \ \lceil (\lambda x.\mathtt{cases}_m(\overline{dis_j`x \longrightarrow \mathtt{and}_n(\overline{\pi_i(sel_{ji}`x)})})\mathtt{andisR`}x)`t \rceil \vee t{=}\bot \\
& \mathtt{beta\_cfun} & = \ \lceil \mathtt{cases}_m(\overline{dis_j`t \longrightarrow \mathtt{and}_n(\overline{\pi_i(sel_{ji}`t)})})\ \mathtt{and\ isR`}t \rceil \vee t{=}\bot
\end{array}
$$

For that j with $\lceil \mathtt{dis}_j`(t) \rceil$:

$$
\begin{array}{lll}
[Mod.Pon.] & \Longrightarrow & \lceil \mathtt{and}_n(\overline{\pi_i`(sel\_ji`t)})\ \mathtt{and}\ \mathtt{isR}`t\rceil \vee t{=}\bot \\
[\mathtt{and}-Def] & = & \bigwedge_i \overline{\lceil \pi_i`(sel\_ji`t)\rceil} \wedge \lceil \mathtt{isR}`t\rceil \vee t{=}\bot \\
[\Pi\_\textsc{Def}] & \Longrightarrow & \bigwedge_i \overline{\Pi_i(sel_{ji}`\mathtt{t})} \wedge \lceil \mathtt{isR}`t\rceil \vee t{=}\bot \\
(\star)\ [\textsc{Inv\_Def}] & \Longrightarrow & \bigwedge_i \overline{Inv(sel_{ji}`\mathtt{t})} \\
models & : & \widehat{M}[\![\alpha_{\tau(\overline{t_i})}(t)]\!]^{\Sigma}_{\hat{\eta}} \\
[\alpha\_\textsc{Tau}] & = & \widehat{M}[\![\mathtt{abs}`Map_\sigma`(\overline{\Lambda x.\alpha_{t_i}(x)})`t]\!]^{\Sigma}_{\hat{\eta}}
\end{array}
$$

For that j with $\lceil dis_j`(t)\rceil$:

$$
\begin{array}{lll}
[\textsc{MapCon}] & = & \widehat{M}[\![\mathtt{abs}`con_j`(\overline{Map_{x_i}`\overline{\Lambda x.\alpha_{y_i}`x}`(sel_{ji}`t)})]\!]^{\Sigma}_{\hat{\eta}} \\
[\textsc{Int\_App}] & = & \widehat{M}[\![\mathtt{abs}`con_j]\!]^{\Sigma}_{\hat{\eta}}(\overline{\widehat{M}[\![Map_{x_i}]\!]^{\Sigma}_{\hat{\eta}}\overline{\widehat{M}[\![\Lambda x.\alpha_{y_i}`x]\!]^{\Sigma}_{\hat{\eta}}}\widehat{M}[\![sel_{ji}`t]\!]^{\Sigma}_{\hat{\eta}}}) \\
[IHyp] & = & \widehat{M}[\![\mathtt{abs}`con_j]\!]^{\Sigma}_{\hat{\eta}}(\overline{\widehat{M}[\![Map_{x_i}]\!]^{\Sigma}_{\hat{\eta}}\overline{M_{|y_i}[\![\Lambda x.x]\!]^{\Sigma}_{\eta}}\widehat{M}[\![sel_{ji}`t]\!]^{\Sigma}_{\hat{\eta}}}) \\
[\textsc{MapId}] & = & \widehat{M}[\![\mathtt{abs}`con_j]\!]^{\Sigma}_{\hat{\eta}}(\overline{M_{|y_i}[\![sel_{ji}`t]\!]^{\Sigma}_{\eta}}) \\
[IHyp,(\star)] & = & \widehat{M}[\![\mathtt{abs}]\!]^{\Sigma}_{\hat{\eta}}(M_{|u}[\![con_j`(\overline{sel_{ji}`t})]\!]^{\Sigma}_{\eta}) \\
[\textsc{ConSel}] & = & \widehat{M}[\![\mathtt{abs}]\!]^{\Sigma}_{\hat{\eta}}(M_{|u}[\![t]\!]^{\Sigma}_{\eta}) \\
[\textsc{Abs\_Def},Hyp] & = & M_{|u}[\![t]\!]^{\Sigma}_{\eta}
\end{array}
$$

$\mathbf{u = tc(\overline{t_i})}$ *analog to* $\tau(\overline{t_i})$

$$
\begin{array}{llll}
\mathbf{u = a{\to}b} & \widehat{M}[\![\alpha_{a\to b}(t)]\!]^{\Sigma}_{\hat{\eta}} & = & \\
& [\alpha\_\textsc{Fun}] & = & \widehat{M}[\![\Lambda \mathtt{x}.\alpha_b`(\mathtt{t}`(\tilde{\alpha}_a`x))]\!]^{\Sigma}_{\hat{\eta}} \\
& [\textsc{Int\_Abs}] & = & f{::}c \in \widehat{TM}[\![a]\!]^{\Omega}{\mapsto}\widehat{M}[\![\alpha_b`(\mathtt{t}`(\tilde{\alpha}_a`x))]\!]^{\Sigma}_{\hat{\eta}(c/\mathtt{x})} \\
& [\textsc{TM\_Equal}] & = & f{::}c \in TM_{|a}[\![a]\!]^{\Omega}{\mapsto}\widehat{M}[\![\alpha_b`(\mathtt{t}`(\tilde{\alpha}_a`x))]\!]^{\Sigma}_{\hat{\eta}(c/\mathtt{x})} \\
& [\tilde{\alpha}\_\textsc{Thm}] & = & f{::}c \in TM_{|a}[\![a]\!]^{\Omega}{\mapsto}\widehat{M}[\![\alpha_b`(t`x)]\!]^{\Sigma}_{\hat{\eta}(c/\mathtt{x})} \\
& & & \quad and\ \ Inv(x) \\
(\star) & [\textsc{Inv\_Term},Hyp] & \Longrightarrow & \quad Inv(t`x) \\
& [IHyp,(\star)] & = & f{::}c \in TM_{|a}[\![a]\!]^{\Omega}{\mapsto}M_{|b}[\![t`x]\!]^{\Sigma}_{\eta(c/\mathtt{x})} \\
& [\textsc{Int\_Abs}] & = & M_{|u}[\![t]\!]^{\Sigma}_{\eta} \qquad \square
\end{array}
$$

In the proof there are the schemes π and α that have an argument, written as an index. The index is chosen depending on the parameter's result type. In the example $\pi_i(sel_{ji}`x)$

the index i of π is chosen such that it fits to the selector's result. If sel_{ji} is of type $\overline{\tau_i}{\to}\alpha$ then i is that value with $\mu(i) = \alpha$. In the same way the Map_{X_i} depend on the type.

In the first case $\mathbf{u} = \tau(\overline{\mathbf{t_i}})$ the assumption that there exists a j with $\lceil dis_j`(t)\rceil$ is valid, since τ is an ADT. The application of INV_DEF uses the fact that the selectors are total (SELTOTAL) on the representing values. This is no restriction, since ADTs are not lazy. It is obvious that the application of map functional makes only sense for finite ones, since it would not terminate for infinite (lazy) data types. So this theory interpretation works only for finite data types.

The next theorem complements the previous:

Theorem 3.2.4 $\tilde{\alpha}$_THM

Let Φ be a translation with premises π and Π and $\tilde{\alpha}$ the construction scheme of Definition 3.2.15 then

- $\widehat{M}[\![\tilde{\alpha}_u(t::u)]\!]^{\Sigma}_{\hat{\eta}} = \widehat{M}[\![t]\!]^{\Sigma}_{\hat{\eta}}$ and $Inv(t)$

The induction proof goes over the type structure of u since $\tilde{\alpha}$ and π depend on it.

Proof

$\mathbf{u} = \tau(\overline{\mathbf{t_i}})$

$$\begin{aligned} \widehat{M}[\![\tilde{\alpha}_{\tau(\overline{t_i})}(t)]\!]^{\Sigma}_{\hat{\eta}} &= \\ [\tilde{\alpha}\text{\_TAU}] &= \widehat{M}[\![\mathtt{rep}`Map_{\tau}`(\overline{\Lambda x.\tilde{\alpha}_{t_i}`x})`t]\!]^{\Sigma}_{\hat{\eta}} \end{aligned}$$

For that j with $\lceil \mathtt{dis}_j`(\mathtt{t})\rceil$:

$$\begin{aligned} [\text{MAPCON}] &= \widehat{M}[\![\mathtt{rep}`con_j`(\overline{Map_{X_i}`(\overline{\Lambda x.\tilde{\alpha}_{y_i}`x})`(sel_{ji}`t)})]\!]^{\Sigma}_{\hat{\eta}} \\ [IHyp] &= \widehat{M}[\![\mathtt{rep}`con_j`(\overline{Map_{X_i}`(\overline{\Lambda x.x})`(sel_{ji}`t)})]\!]^{\Sigma}_{\hat{\eta}} \\ [\text{MAPID}] &= \widehat{M}[\![\mathtt{rep}`con_j`(\overline{sel_{ji}`t})]\!]^{\Sigma}_{\hat{\eta}} \\ [\text{CONSEL}] &= \widehat{M}[\![\mathtt{rep}`t]\!]^{\Sigma}_{\hat{\eta}} \\ [\Pi\text{\_REP}] &= \widehat{M}[\![\mathtt{rep}`t]\!]^{\Sigma}_{\hat{\eta}} \ \textit{and}\ \Pi_u(\mathtt{rep}`t) \\ [\text{INV\_DEF}] &= \widehat{M}[\![\mathtt{rep}`t]\!]^{\Sigma}_{\hat{\eta}} \ \textit{and}\ Inv(\mathtt{rep}`t) \\ [\text{REP\_DEF}] &= \widehat{M}[\![t]\!]^{\Sigma}_{\hat{\eta}} \ \textit{and}\ Inv(t) \end{aligned}$$

For that j with $\lceil dis_j`(t)\rceil$:

$\mathbf{u} = \mathbf{tc}(\overline{\mathbf{t_i}})$ *analog to* $\tau(\overline{t_i})$

For that j with $\lceil dis_j`(t)\rceil$:

$$
\begin{array}{lll}
\mathbf{u = a{\to}b} & \widehat{M}[\![\tilde{\alpha}_{a\to b}(t)]\!]^{\Sigma}_{\hat{\eta}} = & \\
& [\tilde{\alpha}\_\textsc{Fun}] = & \widehat{M}[\![\Lambda \mathtt{x}.\tilde{\alpha}_b\text{`}(t\text{`}(\alpha_a\text{`}x))]\!]^{\Sigma}_{\hat{\eta}} \\
& [\textsc{Int\_Abs}] = & f{::}c \in \widehat{TM}[\![\mu a]\!]^{\Omega} \mapsto \widehat{M}[\![\tilde{\alpha}_b\text{`}(t\text{`}\alpha_a\text{`}x))]\!]^{\Sigma}_{\hat{\eta}(c/\mathtt{x})} \\
& [\textsc{TM\_Red}] = & f{::}c \in TM[\![\mu a]\!]^{\Omega} \mapsto \widehat{M}[\![\tilde{\alpha}_b\text{`}(t\text{`}\alpha_a\text{`}x))]\!]^{\Sigma}_{\hat{\eta}(c/\mathtt{x})} \\
& [\textsc{NormalAbs}] = & f{::}c \in \{M[\![\Phi z]\!]^{\Sigma}_{\eta}\} \mapsto \widehat{M}[\![\tilde{\alpha}_b(t\text{`}\alpha_a(x))]\!]^{\Sigma}_{\hat{\eta}(c/\mathtt{x})} \\
(\star) & [\textsc{Inv\_}\Phi\textsc{\_Thm}] \quad \Longrightarrow & Inv(c) \\
& [\alpha\_\textsc{Thm},(\star)] = & f{::}c \in \widehat{TM}[\![\mu a]\!]^{\Omega} \mapsto \widehat{M}[\![\tilde{\alpha}_b\text{`}(t\text{`}x)]\!]^{\Sigma}_{\hat{\eta}(c/\mathtt{x})} \\
& [IHyp] = & f{::}c \in \widehat{TM}[\![\mu a]\!]^{\Omega} \mapsto \widehat{M}[\![t\text{`}x]\!]^{\Sigma}_{\hat{\eta}(c/\mathtt{x})} \ \ and \ \ Inv(t\text{`}c) \\
& [\textsc{Inv\_Def},(\star)] = & f{::}c \in \widehat{TM}[\![\mu a]\!]^{\Omega} \mapsto \widehat{M}[\![t\text{`}x]\!]^{\Sigma}_{\hat{\eta}(c/\mathtt{x})} \ \ and \ \ Inv(t) \\
& [\textsc{Int\_Abs}] = & \widehat{M}[\![\Lambda \mathtt{x}.t\text{`}x]\!]^{\Sigma}_{\hat{\eta}} \ \ and \ \ Inv(t) \\
& [Ext] = & \widehat{M}[\![t]\!]^{\Sigma}_{\hat{\eta}} \ \ and \ \ Inv(t) \qquad \square
\end{array}
$$

With these theorems we can prove the main result of this section: the satisfiability of Φ:

Theorem 3.2.5 SATISFIABILITY_THM

If Φ is a term translation with φ, μ, π from Th^a to Th^c, Inv is the invariance and $M \models Th^c$ then for all axioms $a{::}bool \in Ax^a$:

- $M[\![\Phi a]\!]^{\Sigma}_{\eta} = \widehat{M}[\![a]\!]^{\Sigma}_{\hat{\eta}}$ if for all $x \in \Psi : \eta(x) = \hat{\eta}(x)$ and $Inv(\eta(x))$

Since Theorem 3.2.2 it suffices to prove: $M_{|u}[\![\Phi(a{::}u)]\!]^{\Sigma}_{\eta} = \widehat{M}[\![a]\!]^{\Sigma}_{\hat{\eta}}$

Proof

$$
\begin{array}{lll}
\mathbf{t} = \mathtt{c}^a_i & M_{|u}[\![\Phi \mathtt{c}^a_i]\!]^{\Sigma}_{\eta} \stackrel{!}{=} & = \\
& [\Phi\textsc{\_Con},\varphi\textsc{\_Corr}] = & M_{|u}[\![\mathtt{c}^c_i]\!]^{\Sigma}_{\eta} \\
& [\textsc{Inv\_Con},\alpha\textsc{\_Thm}] = & \widehat{M}[\![\alpha_u(\mathtt{c}^c_i)]\!]^{\Sigma}_{\hat{\eta}} \\
& [\textsc{Int\_}\mathtt{c}^a_i] = & \widehat{M}[\![\mathtt{c}^a_i]\!]^{\Sigma}_{\hat{\eta}}
\end{array}
$$

$$
\begin{array}{lll}
\mathbf{t} = x & M_{|u}[\![\Phi x]\!]^{\Sigma}_{\eta} & = \\
& [\Phi\_\textsc{Var}] & = M_{|u}[\![x]\!]^{\Sigma}_{\eta} \\
& [\textsc{Red\_Var}] & = M[\![x]\!]^{\Sigma}_{\eta} \\
& [\textsc{Int\_Var}] & = \eta(x) \\
& [Hyp] & = \hat{\eta}(x) \\
& [\textsc{Int\_Var}] & = \widehat{M}[\![x]\!]^{\Sigma}_{\hat{\eta}}
\end{array}
$$

$$
\begin{array}{lll}
\mathbf{t} = (\Lambda \mathbf{x}.\mathbf{z})::\mathbf{v}\rightarrow\mathbf{w} & M_{|v\rightarrow w}[\![\Phi(\Lambda x.z)]\!]^{\Sigma}_{\eta} & = \\
& [\Phi\_\textsc{Abs}] & = M_{|v\rightarrow w}[\![\Lambda x.\Phi z]\!]^{\Sigma}_{\eta} \\
& [\textsc{Red\_Abs}] & = f::c \in TM_{|v}[\![v]\!]^{\Omega} \mapsto M_{|w}[\![\Phi z]\!]^{\Sigma}_{\eta(\mathtt{c/x})} \\
& [IHyp] & = f::c \in TM_{|v}[\![v]\!]^{\Omega} \mapsto \widehat{M}[\![z]\!]^{\Sigma}_{\hat{\eta}(\mathtt{c/x})} \\
& [\textsc{TM\_Equal}] & = f::c \in \widehat{TM}[\![v]\!]^{\Omega} \mapsto \widehat{M}[\![z]\!]^{\Sigma}_{\hat{\eta}(\mathtt{c/x})} \\
& [\textsc{Int\_Abs}] & = \widehat{M}[\![\Lambda x.z]\!]^{\Sigma}_{\hat{\eta}}
\end{array}
$$

$$
\begin{array}{lll}
\mathbf{t} = f::(\mathbf{v}\rightarrow\mathbf{u})\mathbf{z} & M_{|u}[\![\Phi(f\text{`}z)]\!]^{\Sigma}_{\eta} & = \\
& [\Phi\_\textsc{App}] & = M_{|u}[\![\Phi f\text{`}\mathtt{If}\ \pi u\text{`}\Phi z\ \mathtt{then}\ \Phi z\ \mathtt{else}\ \bot\ \mathtt{fi}]\!]^{\Sigma}_{\eta}
\end{array}
$$

$$
\begin{array}{lll}
cases & -\pi u\text{`}\Phi z{=}\bot \qquad \Phi z{=}\bot & \\
& [if] & = M_{|u}[\![\Phi f \Phi z]\!]^{\Sigma}_{\eta} \\
& [\textsc{Red\_App}] & = (M_{|v\rightarrow u}[\![\Phi f]\!]^{\Sigma}_{\eta}) M_{|v}[\![\Phi z]\!]^{\Sigma}_{\eta}
\end{array}
$$

$$
\begin{array}{lll}
(\star) & [\textsc{Inv\_}\Phi\textsc{\_Thm}, Hyp] & \Longrightarrow Inv(\Phi f)\ \ and\ \ Inv(\Phi z) \\
& [IHyp, (\star)] & = (\widehat{M}[\![f]\!]^{\Sigma}_{\hat{\eta}}) \widehat{M}[\![z]\!]^{\Sigma}_{\hat{\eta}} \\
-\lfloor \pi\mathtt{u}\text{`}\Phi\mathtt{z} \rfloor & [\textsc{Def\_}\Phi\textsc{\_True}] & is\ impossible\ case \\
-\lceil \pi\mathtt{u}\text{`}\Phi\mathtt{z} \rceil & [if] & = M_{|u}[\![\Phi\mathtt{f}\Phi\mathtt{z}]\!]^{\Sigma}_{\eta} \\
& [\textsc{Red\_App}] & = (M_{|v\rightarrow u}[\![\Phi\mathtt{f}]\!]^{\Sigma}_{\eta}) M_{|v}[\![\Phi z]\!]^{\Sigma}_{\eta} \\
(\star\star) & [\textsc{Inv\_}\Phi\textsc{\_Thm}] & \Longrightarrow Inv(\Phi\mathtt{f})\ \ and\ \ Inv(\Phi z) \\
& [IHyp, (\star\star)] & = (\widehat{M}[\![f]\!]^{\Sigma}_{\hat{\eta}}) \widehat{M}[\![z]\!]^{\Sigma}_{\hat{\eta}} \qquad \square
\end{array}
$$

Now the results are combined and we gain a method for the restriction step.

3.2.8 Method for the Restriction Step

To find a method for the restriction step we collect all assumptions required by the theorems and put them together to a method. The general scheme is defined for an abstract theory $Th^a = (\Sigma^a, Ax^a)$ and a concrete theory $Th^c = (\Sigma^c, Ax^c)$:

1. Check if the theory Th^a is normalizeable and normalize it.
2. Define a term translation Φ from Th^a to Th^c (including φ, μ, π).
3. Check the translation syntactically (type check, etc.).
4. Generate proof obligations.
5. Prove proof obligations.
6. Generate code.
7. Optimize code.

In the case of implementing an ADT by theory interpretation the concrete steps are:

1. The first step is to check if normalization is possible. It requires that all polymorphic predicates that are used in the specification are defined conservatively. This can be done automatically. As mentioned on page 70 the software development process ensures that normalization is possible.
2. In the second step the user of the method has to fix:
   - the abstract sort $\tau \in T_\Omega a$,
   - the corresponding sort $\sigma \in T_\Omega c$,
   - the restriction predicate $\mathtt{isR}::\sigma \to \mathtt{tr}$,
   - the abstract constants $\mathtt{c}_i^a \in \Sigma^a$, and
   - the corresponding constants $\mathtt{c}_i^c \in T_\Sigma c$ for all $\mathtt{c}_i^a$,
3. The translation is checked:
   - Type correctness: for all $\mathtt{c}_i^a :: u$ the type of the corresponding constants has to be $\mathtt{c}_i^c :: \mu(u)$
   - The sorts have to be data types: $\tau :: \mathtt{pcpo}$ and $\sigma :: \mathtt{pcpo}$.
   - All constants of $Th^a \setminus Th^c$ (except the conservatively defined ones) have to be implemented by φ
   - All sort constructors $\{tc_j^a\}$ of Th^a are contained in the sort constructors $\{tc_j^c\} \cup \{\tau\}$ of Th^c

These tests can be performed automatically.

4. The proof obligations are:

   - The correctness preserving of Φ (see Definition 2.1.5): for all abstract axioms $ax \in Ax^a$ prove the translation $Th^c \vdash \Phi ax$.
   - The invariance (which is needed for satisfiability):
     $Th^c \vdash \forall \overline{\mathtt{x}_j}.\ \bigwedge_j (\overline{\Pi_{\mathtt{s}_j}(\mathtt{x}_j)}) \Longrightarrow \Pi_t(\mathtt{c}_i^c \text{`} \overline{x_j})$
     for all $\mathtt{c}_i^a :: \mathtt{s}_1 \to ... \to \mathtt{s}_n \to t$

5. The proof of the proof obligations has to be done by the developer. It can be completely proved within HOLCF.

6. It is possible to generate code, if the corresponding constants are executable. If so, the code is the translation $\Phi(Ax^a)$ of the requirement specification.

7. It is possible to eliminate some redundant checks. This can be done for nested terms, or for functions that are ensured to get corresponding values.

So the user has to define the implementation Φ and prove its correctness. The rest is done automatically.

3.2.9 Example: BOOLEAN by NAT

The method applied to Example 3.1.1 is:

1. The theory BOOLEAN on page 66 is normal.

2. The definition of Φ in the example is:

   - μ(B) = N (τ = B and σ = N)
   - φ(T) = One (c_1^a = T, c_1^c = One)
   - φ(F) = Zero (c_2^a = F, c_2^c = Zero)
   - φ(Bnot) = Λ x::N . One − x (c_3^a = Bnot, c_3^c = Λx.One−x)
   - φ(Band) = Λ x::N y::N . If is_Zero‘x then Zero else y fi
     (c_4^a = Band, c_4^c = Λx y.If is_Zero‘x then Zero else y fi)
   - The restriction predicate isR is isB

3. The translation Φ is type correct. Consider the type of c_4^c: N→N→N = μ(B→B→B) is the translated type of c_4^a.

4. The proof obligations for the invariance are:

   - NAT ⊢⌈isB‘One⌉∨One=⊥
   - NAT ⊢⌈isB‘Zero⌉∨Zero=⊥
   - NAT ⊢∀x.(⌈isB‘x⌉∨x=⊥)⟹(⌈isB‘(One−x)⌉∨(One−x)=⊥)
   - NAT ⊢∀x∀y.(⌈isB‘x⌉∨x=⊥)∧⌈isB‘y⌉∨y=⊥)⟹ (⌈isB‘(If is_Zero‘x then Zero else y fi)⌉∨
     (If is_Zero‘x then Zero else y fi)=⊥)

   The proof obligations for the correctness are:

   - for Bnot1:NAT ⊢(Λx.One−x)‘If isB‘Zero then Zero else ⊥ fi) = One
   - for Bnot2:NAT ⊢(Λx.One−x)‘If isB‘One then One else ⊥ fi) = Zero
   - for Band1:Since T Band x = x is an abbreviation for ∀x.Band‘T‘x=x the proof obligation is:
     NAT ⊢∀x.⌈isB‘x⌉⟹(Λx y.If is_Zero‘x then Zero else y fi)‘
     If isB‘One then One else ⊥ fi‘If isB‘x then x else ⊥ fi=x
   - for Band2:
     NAT ⊢∀x.⌈isB‘x⌉⟹(Λx y.If is_Zero‘x then Zero else y fi)‘
     If isB‘Zero then Zero else ⊥ fi‘If isB‘x then x else ⊥ fi=Zero
   - The simplified proof obligation for the induction rule is:

     ⟦P ⊥; P Zero; P One ⟧ ⟹ ∀x.(isB‘x ∨ x=⊥) ⟶ P x
   - c_def: is a definition, that is based on the constants that are implemented. It could have been written in a separate specification, using BOOLEAN. Therefore, no proof obligations are necessary for it.

5. If constants are defined, proof obligations can be simplified. The invariance obligations become:

   - NAT ⊢⌈isB‘One⌉
   - NAT ⊢⌈isB‘Zero⌉
   - NAT ⊢∀x.⌈isB‘x⌉⟹⌈isB‘(One−x)⌉
   - NAT ⊢∀x∀y.⌈isB‘x⌉∧⌈isB‘y⌉⟹
     ⌈isB‘(If is_Zero‘x then Zero else y fi)⌉

   If the corresponding constants are preserving, the correctness proof obligations may be reduced to:

   - for Bnot1:NAT ⊢One−Zero=One
   - for Bnot2:NAT ⊢One−One=Zero

- for Band1:NAT $\vdash\emptyset$
- for Band2:NAT $\vdash\emptyset$
- for B_Induct:NAT $\vdash[\![$P ⊥;P Zero;P One$]\!]\Longrightarrow\forall$x.(isB‘x∨x=⊥)⟶P x

The proof obligations for Band1 ad Band2 disappear since the invariance ⌈isB‘x⌉ $\Longrightarrow$ ⌈isB‘(If is_Zero‘x then Zero else x fi)⌉ holds.

6. Code generation eliminates all abstract constants by translation. Therefore, only c remains and the generated code for BOOLEAN is:

```
BOOLEAN_BY_NAT = NAT +
consts
        c :: N DList
rules
c_def   c ≡ Map_DList‘(Λx.If isB‘x then One−x else ⊥ fi)‘
          If (fix‘ΛP l.cases is_nil‘l ⟶ true,
                              is_cons‘l ⟶ and‘(isB‘(fst‘l))‘
                                              (P‘(rst‘l)))‘
               (dcons‘One‘(dcons‘Zero,dnil))
          then (dcons‘One‘(dcons‘Zero,dnil)
          else ⊥ fi
end
```

The executability is clear, since N is a free data type. The fixed point operation can be implemented (for every example) by a recursive function of the functions. In this case it is even possible to eliminate the test by optimizations.

7. the optimizations give the following efficient code:

```
BOOLEAN_BY_NAT = NAT +
consts
          c :: N DList
rules
c_def     c ≡ Map_DList‘(Λx.One−x)‘(dcons‘One‘(dcons‘Zero,dnil))
end
```

If we focus on the normalized formula:

uniqueN $\exists$f.$\forall$x.f‘x = Bnot‘x ∧ $\forall$g.$\forall$x.g‘x = Bnot‘x⟶$\forall$y.f‘y=g‘y

we see that it is translated and simplified into:

$\exists$f. $\forall$x.$\lceil$isB‘x$\rceil$ $\Longrightarrow$ f‘x=(Λx.One−x)‘x
$\wedge$ $\forall$g.$\forall$x.$\lceil$isB‘x$\rceil$ $\Longrightarrow$ g‘x=(Λx.One−x)‘x
$\Longrightarrow$ $\forall$y.$\lceil$isB‘y$\rceil$ $\Longrightarrow$f‘y=g‘y

this is obviously true and hence it is an example for the CORRECTNESS of our method.

3.2.10 Refinement for Restrictions

This section defines a refinement relation for the restriction step of the implementation based on the theory interpretation of the previous sections. With this refinement relation a basis for the deductive software development process is defined. On an example we will see that this basis may introduce inconsistencies, when it is applied to arbitrary modules (parts) of the system. Therefore, it is not modular in the sense of Definition 1.4.2.

The following definition of the theory interpretation basis includes the definition of a refinement relation for restrictions:

Definition 3.2.17 ***Theory Interpretation Basis***

Let $(\mathcal{L}_{Th}, \mathtt{MOD}, \supseteq, \Longleftarrow)$ be a model inclusion basis, and let Φ be a theory interpretation for restrictions with invariance Inv, then the quadruple $(\mathcal{L}_\Phi, \mathtt{MOD}, \supseteq_\Phi, \Longleftarrow_\Phi)$ is called *theory interpretation basis*, if

- $\mathcal{L}_\Phi \supseteq \mathcal{L}_{Th}$ is the set of all normalizeable specifications,
- $\supseteq_\Phi$ is for defined by:
  $M \supseteq_\Phi N := M \supseteq N$ or $M \supseteq \widehat{\Phi}(N)$ where $\widehat{\Phi}(N)$ is the model construction for N (see Definition 3.2.16). This refinement relation allows us to do refinement by model inclusions and theory interpretations.
- $\Longleftarrow_\Phi$ is for $M, N \in$ MOD defined by:
  $S \Longleftarrow_\Phi T := S \Longleftarrow T$ if a refinement by model inclusion has to be proved and $\Phi(S) \wedge Inv \Longleftarrow T$ if a theory interpretation Φ with invariance Inv has to be proved.

The theory interpretation basis is obviously a deductive software development basis and due to the satisfiability of the theory interpretation it is also consistency preserving. For modularity we look at the following example. It show us, that applying a theory interpretation only to a module can introduce a type error.

Example 3.2.2 ***Theory Interpretation with Modules***

We specify a function that takes a natural number and doubles it. The specification `DOUBLE(Dnat)` is modular, since it includes the natural numbers as module.

```
DOUBLE = Dnat + (* Isabelle syntax for DOUBLE(Dnat) *)
consts
       double    :: dnat → dnat → dnat
defs
       double_def   double ≡ Λn. n add n
end
```

The specification bases on the following specification module:

```
Dnat = EQ +
domain dnat = dzero | dsucc(dnat)
consts
       add    :: dnat → dnat → dnat
rules
       add1    dzero add n=n
       add2    dsucc‘n add m=dsucc‘(n add m)
end
```

See Section A.3.1 for a complete specification of the natural numbers. Now we use a theory interpretation Φ with sort translation μ`(dnat)=Fin` (see Section A.3.7). Φ translates natural numbers into finite natural numbers. Applying this theory interpretation to `Dnat` changes the type of the function `add` from `dnat` $\rightarrow$ `dnat` $\rightarrow$ `dnat` to `Fin` $\rightarrow$ `Fin` $\rightarrow$ `Fin`. Since `double` is still of type `dnat` $\rightarrow$ `dnat` $\rightarrow$ `dnat` the definition `double_def` in the specification `DOUBLE(`Φ`(Dnat))` is not type correct.

Applying theory interpretations to the whole system avoids this problem. In the example it translates the type of the function `double` from `dnat` to `Fin`.

Since theory interpretations may only be applied to normalizeable theories it might be the case that a component is normalizeable and the whole system is not. Therefore, theory interpretations are in general not modular, in the sense of Definition 1.4.2. In [Far94a] it is proved that theory interpretations are modular for conservative specifications. The result cannot be applied in general, since specifications are in general not conservative. In Section 6.3.3.4 we prove that some restricted form of theory interpretations, which are used for the quotient step, are compositional, and therefore, well suited for the implementation of interactive systems.

The origin of the problems with different implementations is that there is no abstraction any more. Especially an abstraction function would help to avoid these problems since it abstracts from the concrete type. Therefore, many refinement techniques use abstraction functions to relate different data types. Conservative extension is a method that is based on abstraction and representation functions.

3.3 Model Inclusion

The last section presented a method for the restriction step based on theory interpretation. In order to compare theory interpretation with model inclusion, this section gives a method for the implementation of the restriction step, based on model inclusion. This method introduces a subtype by a conservative extension. This allows us to prove the refinement for the restriction step of the implementation by theory inclusion. The method provides a subtype with *cpo* structure (called *subdomain*) and continuous functions, needed for the implementation of interactive systems. Tool support for this method is available in form of the `subdom` type constructor (see Section 3.5 and Appendix A.1), which we have developed in HOLCF. The method is compared with the method of the previous section and will be part of the method for the implementation of ADTs in Chapter 5.

The method extends a concrete specification by a subtype, which is shown to reside generally in the class `pcpo` and provides syntactic schemes for the conservative introduction of functions of the new type. If the corresponding concrete functions on the concrete type are preserving, the defined functions will be continuous.

For the logic HOL there exists a method [Mel89], which allows us to introduce subtypes. The method realizes this by encoding an induction rule into the predicate describing the values of the new type. In general we cannot apply this technique, since we introduce types with *cpo* structures and require the admissibility of our restriction predicate. Therefore, we have to provide rules, which allow us to deduce the induction rule for the abstract type from the induction rule of the concrete type.

We use the same notations as in Section 3.2. The implementation consists of a sort implementation μ from a type τ to a corresponding sort σ, a constant implementation φ, and a restriction operation `isR`, which is (equal to) `TT` for all values of the subtype[10].

The section is structured as follows: First, a new subtype is introduced by conservative extension and it is proved that it belongs to the class `pcpo` (Section 3.3.1). Then, invariance will be required (Section 3.3.2) because it is used to show continuity of the operations on the new ADT, which are schematically introduced in Section 3.3.3. The induction principle is proved in Section 3.3.4 and code generation is described in Section 3.3.5. In the last section the method presented in 3.3.6 is applied to a small example. Since the following pages contain a lot of proofs, which are schematic and could be used for the introduction of arbitrary subdomains, we present in Section 3.5 a generic implementation, which supports the definition of subdomains.

[10] For the implementation of this method we generalize the restriction predicate to arbitrary admissible predicates (see Section 3.5).

3.3.1 Introducing a Subdomain

This section conservatively introduces a new subtype with a *cpo* structure using abstraction and representation functions as in Example 2.1.2. The subtype will implement the abstract type, and is a subset of a concrete type. A subtype of a type of class `pcpo` is called *subdomain*, if the subtype also belongs to the class `pcpo`. We introduce subdomains conservatively. The first step is to introduce a subtype. The next step is to show that the new domain has a *cpo* structure. This is possible since the restriction predicate for the implementation of ADTs is continuous[11]. Since the new type has to have a *cpo* structure it has to include an undefined element $\bot$. The subset contains $\bot$ and all values from the concrete type, for which the restriction operation `isR` is `TT`.

The abstract type τ is introduced, isomorphic to a subtype of the concrete type σ (of class `pcpo`). Since the extension is syntactic, we use Isabelle syntax:

```
T0 = Th^c +
types
   τ n                        (* the arity of the type is n *)
consts
   τVal      :: σ set         (* subset *)
   τrep      :: τ ⇒ σ         (* representation *)
   τabs      :: σ ⇒ τ         (* abstraction *)
defs
   τVal_def  τVal ≡ {s. ⌈isR`s⌉ ∨ s=⊥}
rules
   τrep_Val  τrep t ∈ τVal
   τabs_rep  τabs (τrep t) = t
   τrep_abs  s∈τVal ⟹ τrep(τabs s) = s
end
```

The introduction of τ with the HOLCF method requires to show that it is not empty. This is obvious since τ`abs` $\bot$ is in τ by the definition of τ`Val_def`. The more difficult part is to show that τ has a *cpo* structure. A conservative introduction of a *cpo* structure proceeds in the following steps:

1. Introduce $\sqsubseteq$ and show that it is a partial order. The characteristic axioms for the class `po` are (see page 36):
   - reflexivity,
   - antisymmetry and
   - transitivity

[11] In our realization of `subdom` we will only require an arbitrary admissible predicate.

These axioms have to be proved for $\sqsubseteq$ before instantiating τ into the class `po`.

2. Show that τ`abs` $\bot$ is the least element in τ.

3. Show that τ is a `pcpo` with respect to $\sqsubseteq$ by proving the characteristic axioms:

   - `minimal` $\bot \sqsubseteq$ `x`
   - `cpo` `is_chain C` $\Longrightarrow$ `range C` $<<|$ τ`abs (`$\bigsqcup$`i. `τ`rep (C i))` where $<<|$ denotes the least upper bound of a chain. In HOLCF it is defined by: `is_lub S<<|x` $\equiv$ `S<|x` $\wedge$ `(`$\forall$`u. S<|u` $\longrightarrow$ `x`$\sqsubseteq$`u)` and $<|$ is an upper bound, defined by: `is_ub S<|x` $\equiv$ $\forall$`y.y`$\in$`S` $\longrightarrow$ `y`$\sqsubseteq$`x`

   After these proofs the *cpo* structure may be instantiated by fixing the least element $\bot$.

The partial order $\sqsubseteq$ for τ is defined in the following Isabelle theory:

```
T1 = T0 +          (* add an order *)
consts
   τ_⊑        ::   τ ⇒ τ ⇒ bool
   τ_⊥        ::   τ
defs
   τ_⊥_def         τ_⊥ ≡ τabs ⊥
   τ_⊑_def         τ_⊑ ≡ λ a b.τrep a ⊑ τrep b
end
```

The proofs of the partial order axioms are easy, since the new order $\sqsubseteq$ is based on the partial order on σ. The next theory instantiates τ in the class of partial order by instantiating the polymorphic partial order $\sqsubseteq$ on τ into τ`_`$\sqsubseteq$.

```
T2 = T1 +
arities
   τ :: po
rules
   inst_τ_po       (⊑::τ ⇒ τ ⇒ bool) = τ_⊑
end
```

The following lemmata for `T2` were proved:

```
   minimal_τ       τabs ⊥ ⊑ x

   ⊑_τ             p ⊑ q = τrep p ⊑ τrep q
```

monofun_τrep	monofun τrep
is_chain_τrep	is_chain C $\Longrightarrow$ is_chain(λj.τrep(C j))
lub_τ	is_chain C $\Longrightarrow$ range C <<| τabs ($\bigsqcup$i.τrep(C i))
cpo_τ	is_chain C $\Longrightarrow$ $\exists$ a::τ.range C <<| a
τ_eq	τrep a = τrep b $\Longrightarrow$ a=b
mfun_τabs	[|x$\in\tau$Val; y$\in\tau$Val; x $\sqsubseteq$ y|]$\Longrightarrow\tau$abs x $\sqsubseteq$ τabs y

The theorems minimal_τ and cpo_τ are the witnesses for the fact that τ has a *cpo* structure and hence τ may be instantiated into the class pcpo in the next theory.

```
T3 = T2 +
arities
   τ :: pcpo
rules
   inst_τ_pcpo    (⊥::τ) = τ_⊥
end
```

The proofs of the lemmata mostly reduce the order on τ to the order on σ. The only interesting proof is the proof of lub_τ. It is carried out in Isabelle by:

```
> val prems=goal T2.thy "is_chain C⟹range C<<|τabs(⨆i.τrep(C i))";
> by (cut_facts_tac prems 1);
```

The first proof state is:

```
 1. is_chain C ⟹ range C <<| τabs (⨆i. τrep (C i))
```

Eliminating the application of the least upper bound by:

```
> by (rtac is_lubI 1);
> by (rtac conjI 1);
```

This gives the proof state:

```
 1. is_chain C ⟹ range C <| τabs (⨆i. τrep (C i))
 2. is_chain C ⟹ ∀u. range C <| u ⟶ τabs (⨆i. τrep (C i)) ⊑ u
```

Eliminating the application of the upper bound by:

```
> by (rtac ub_rangeI 1);
> by (rtac allI 1);
```

This gives the proof state:

```
 1. ⋀i. is_chain C ⟹ C i ⊑ τabs (⨆i. τrep (C i))
 2. is_chain C ⟹ ∀u. range C <| u ⟶ τabs (⨆i. τrep (C i)) ⊑ u
```

The weaker order on τ is reduced to the weaker order on σ by:

```
> by (rtac (less_τ RS ssubst) 1);
```

This gives the proof state (without the second subgoal, which has not changed):

```
 1. ⋀i. is_chain C ⟹ τrep(C i) ⊑ τrep(τabs (⨆i. τrep (C i)))
```

Now the definition of the isomorphism (`τrep_abs`) has to be applied. It requires the argument to be in the set of corresponding values.

```
> by (rtac (τrep_abs RS ssubst) 1);
```

The first subgoal expands into:

```
 1. ⋀i. is_chain C ⟹ (⨆i. τrep (C i)) ∈ τVal
 2. ⋀i. is_chain C ⟹ τrep (C i) ⊑ (⨆i. τrep (C i))
```

The second subgoal is a property of the least upper bound and can be eliminated by:

```
> by (rtac is_ub_thelub 2);
> by (etac is_chain_τrep 2);
```

This is the proof state where a property of the least upper bound is needed. To show this property admissibility of the predicate is needed. It is used in the following rule, which is a forward composition:

```
> adm_def2 RS iffD1 RS spec RS mp RS mp;
⟦ adm P5; is_chain x2; ∀i. P5 (x2 i) ⟧ ⟹ P5 (lub (range x2))
```

This composed theorem is applied by:

```
> br (adm_def2 RS iffD1 RS spec RS mp RS mp) 1;
```

It creates the following three subgoals:

```
 1. ⋀i. is_chain C ⟹ adm (λu. u ∈ τVal)
 2. ⋀i. is_chain C ⟹ is_chain (λi. τrep (C i))
 3. ⋀i. is_chain C ⟹ ∀i. τrep (C i) ∈ τVal
```

The second and the third subgoals are reduced by:

```
> by (rtac allI 3);
> by (rtac τrep_Val 3);
> by (etac is_chain_τrep 2);
```

Now the definition of τ`Val` is expanded by rewriting the subgoals with:

```
> by (rewrite_goals_tac [τVal_def,mem_Collect_eq RS eq_reflection]);
```

This gives the proof state for the admissibility of the restriction predicate:

```
 1. ⋀i. is_chain C ⟹ adm (λu. ⌈isR`u⌉ ∨ u = ⊥)
```

It can be proved using the fact that `isR` is a continuous function:

```
> by (rtac adm_disj 1);
 1. adm (λx. ⌈isR`x⌉)
 2. adm (λx. x = ⊥)
> by (rtac adm_eq 1);
 1. cont (fapp isR)
 2. cont (λx. TT)
 3. adm (λx. x = ⊥)
> by (cont_tacR 1);
 1. adm (λx. x = ⊥)
> by (rtac adm_eq 1);
 1. cont (λx. x)
 2. cont (λx. ⊥)
> by (cont_tacR 1);
```

Now the proof of the first subgoal is finished. It remains to show that

```
 1. is_chain C ⟹ ∀u. range C <| u ⟶ τabs (⨆i. τrep (C i)) ⊑ u
```

The proof uses the same ideas as the first subgoal, but instead of the property `is_ub_thelub` it uses `is_lub_thelub`. Therefore, it is not shown here.

3.3.2 Invariance and Preserving Functions

p-preservingness is a property of the corresponding functions, which ensures that the functions do not leave the set of corresponding elements, i.e. provided that a function gets a value of the subtype, its result will be again of the subtype. A function f is called p *preserving* with respect to a predicate p if the predicate p is invariant under the functions, i.e. $\forall x.p(x) \Longrightarrow p(f(x))$ (see Section 3.2.4 for a precise definition of invariance). If we have the possibility to choose the predicate p we may choose it to be equal to true and, therefore, preservingness is trivial[12].

In the general case, we cannot choose the restriction predicate, for example, if we want to represent the values true and false by the natural numbers one and zero, the restriction predicate is fixed to the test whether the natural number is one or zero. Therefore, the invariance is a proof obligation in the method for the implementation of ADTs. It ensures continuity of the operations introduced in the next section. This might look laborious since it has to be proved in every implementation of interactive systems with a restriction step. In fact, there is a method without this explicit proof obligation. This method would encode the test of the restriction operation into a continuous abstraction function by:

$\mathtt{c}\tau\mathtt{abs\ x} = \mathtt{If\ isR`x\ then}\ \tau\mathtt{abs\ x\ else}\ \bot\ \mathtt{fi}$

Since this function can be shown to be continuous (if `isR` is total), all introduced operations could be based on this abstraction function and would be preserving and continuous by definition. However, this would lead to inefficient code (`isR` would be evaluated in every abstraction), which could only be optimized by the help of invariance. Furthermore, the refinement proof of totality of the abstract operations (for example the selector functions) would require to show that the corresponding functions are preserving. Therefore, for a given restriction predicate, invariance is a necessary proof obligation and proving it once helps to structure our correctness proofs.

In addition, invariance may help to find the restriction predicate and the corresponding values (see Section 5.2 for the use of the general method). Therefore, it is advantageous to treat invariance as an explicit proof obligation.

Intuitively the invariance, as described in Figure 3.1 (on page 65) states that the corresponding functions must not leave the subset of corresponding values. Formally this can be defined, as in Definition 3.2.5, by:

- $Th^{c} \vdash \forall \overline{\mathtt{x}_j}.\ \bigwedge_j (\overline{\Pi_{\mathtt{s}_j}(\mathtt{x}_j)}) \Longrightarrow \Pi_{\mathtt{t}}(\mathtt{c}_i^{c}\text{`}\overline{x_j})$
  for all $\mathtt{c}_i^{a} :: \mathtt{s}_1 \to ... \to \mathtt{s}_n \to \mathtt{t}$

The definition of the predicate Π is a type dependent generalization of the restriction predicate p to arbitrary types. For a special case we are able to derive the following theorem over the theory `T3`:

[12] See Section 5.2 for an example of a more appropriate choice of p.

`inv2cont` $\forall$`x.x`$\in\tau$`Val` $\longrightarrow$ `f'x`$\in\tau$`Val` $\Longrightarrow$ `cont` (λ`s.`τ`abs (f'(`τ`rep s))`

This theorem allows us to deduce continuity of the operations, which are introduced on the new type. The functions τ`abs` and τ`rep` are used in the schemes for the definition of the operations. For continuity it remains to show that the corresponding functions are preserving functions, i.e. that the restriction predicate is invariant.

3.3.3 Introducing Executable Operations

To construct an ADT that refines the abstract ADT $T = (\tau, \{c_i^a\})$ by model inclusion we need to implement not only the type τ but also the operations c_i^a. In the previous section we have implemented the type and now we implement the operations. As defined on page 51 operations are executable if they can be defined by continuous terms. Therefore, continuity is important here. We provide a method that specifies the abstract operations in terms of the corresponding concrete operations. Provided that the concrete operations are executable, the abstract operations are also. Invariance is a requirement for the continuity of our new operations c_i^a. Invariance requires the corresponding functions to be preserving.

This section defines syntactic construction schemes for the introduction of operations based on the corresponding constants c_i^c which have the same type as the abstract ones. The conversion is done by τ`abs` and τ`rep`. The schemes may also be applied, if higher order functions and higher order types with type constructors are used in c_i^a. Continuity is derived from the preservingness of the c_i^c by the above theorem `inv2cont`. The new operations will refine the abstract ADT, if the corresponding concrete operations are chosen correctly. This refinement proof is a proof obligation for the method of the implementation.

The schemes are of the same shape as the schemes used in the previous section (see page 85) to construct the semantics of the abstract theory, but they are used here on the syntactic level. They take the corresponding concrete term and the desired type as input, written as an index[13] and produce the desired executable operations.

Definition 3.3.1 ***Syntactic Construction Scheme*** α

Let Φ be a term translation with φ, μ and π. Then the construction scheme α : $T_\Omega a \longrightarrow T_\Sigma c \Rightarrow T_\Sigma a$ for a given target type and a corresponding constant is defined by:

- α_VAR: $\alpha_X(t) = t$ for type variables $x \in \Xi$
- α_FUN: $\alpha_{a \Rightarrow b}(f) = \lambda x{::}a.\alpha_b(f(\tilde{\alpha}_a(x)))$
- α_TAU: $\alpha_{\tau(\overline{t_i})}(t) = \tau\texttt{abs}(Map_\sigma(\overline{\lambda y{::}\mu t_i.\alpha_{t_i}(y)})(t))$
- α_TC: $\alpha_{tc(\overline{t_i})}(t) = Map_{tc}(\overline{\lambda y{::}\mu t_i.\alpha_{t_i}(y)})(t)$ if tc$\neq\tau$

[13] Of course these types have to match in the sense that the concrete type is the μ translated type of abstract type.

This scheme performs the lifting from the corresponding terms $\mathtt{c}_i^c$ to abstract functions $\mathtt{c}_i^a$. It is a type dependent scheme and allows us to lift functions of arbitrary, higher order types. The lifting of concrete functions to abstract functions requires a representation of the arguments and an abstraction of the result. The abstraction is supported by α and for the representation we have a dual construction, which maps to representing values:

Definition 3.3.2 ***Dual Syntactic Construction Scheme*** $\tilde{\alpha}$

Let Φ be a term translation with φ, μ and π. Then the dual construction scheme $\tilde{\alpha} : T_\Omega a \longrightarrow T_\Sigma a \Rightarrow T_\Sigma a$ for a given target type and a corresponding constant is defined by:

- $\tilde{\alpha}$_VAR: $\tilde{\alpha}_x(t) = t$ for type variables $x \in \Xi$
- $\tilde{\alpha}$_FUN: $\tilde{\alpha}_{a \Rightarrow b}(f) = \lambda x :: \mu a.\tilde{\alpha}_b(f(\alpha_a(x)))$
- $\tilde{\alpha}$_TAU: $\tilde{\alpha}_{\tau(\overline{t_i})}(t) = \tau\mathtt{rep}(Map_\tau(\overline{\lambda y :: t_i.\tilde{\alpha}_{t_i}(y)})(t))$
- $\tilde{\alpha}$_TC: $\tilde{\alpha}_{tc(\overline{t_i})}(t) = Map_{tc}(\overline{\lambda y :: t_i.\tilde{\alpha}_{t_i}(y)})(t)$ if tc$\neq\tau$

This scheme performs the representation from abstract to concrete (and corresponding) values. It may be demonstrated on the construction of the function `not` in Example 3.1.1. Starting with the translation of the abstract function ensures that the types match since this is a requirement for the translation:

$$
\begin{aligned}
\mathtt{not}::\mathtt{B}\to\mathtt{B} &\equiv \alpha_{\mathtt{B}\to\mathtt{B}}(\varphi(\mathtt{not})) \\
&\equiv \alpha_{\mathtt{B}\to\mathtt{B}}(\Lambda \mathtt{x}::\mathtt{N}.\mathtt{One}-\mathtt{x}) \\
&\equiv \lambda \mathtt{b}::\mathtt{B}.\alpha_{\mathtt{B}}((\Lambda \mathtt{x}::\mathtt{N}.\mathtt{One}-\mathtt{x})(\tilde{\alpha}_{\mathtt{B}}(\mathtt{b}))) \\
&\equiv \lambda \mathtt{b}.\mathtt{Babs}`((\lambda \mathtt{x}::\mathtt{N}.\mathtt{One}-\mathtt{x})(\mathtt{Brep}`\mathtt{b})) \\
&\equiv \lambda \mathtt{b}.\mathtt{Babs}`(\mathtt{One}-\mathtt{Brep}`\mathtt{b})
\end{aligned}
$$

The continuity of this function is ensured by `inv2cont` since for all $\mathtt{x} \in \{\bot, \mathtt{Zero}, \mathtt{One}\}$ the following holds:$(\lambda \mathtt{y}.\mathtt{One}-\mathtt{y})\ \mathtt{x} \in \{\bot, \mathtt{Zero}, \mathtt{One}\}$.

3.3.4 Deriving an Induction Rule

Since the abstract data type contains an induction rule (see Definition 2.1.10), and since we refine the abstract data type by model inclusion we have to prove this induction rule. We prove the abstract induction rule based on a concrete induction rule for the corresponding values. In other words we have to show that all corresponding values are generated from the constants corresponding to the abstract constructor functions.

The following rule, which is derivable from theory T3, helps us to derive the abstract induction rule:

all_sd $\forall$s.(s=$\bot$ $\vee$ $\lceil$isR`s$\rceil$) $\longrightarrow$ P (τabs s) $\Longrightarrow$ P (x::τ)

Thus, proving an arbitrary property over a variable of type τ can be reduced with all_sd to the proof of the property of all corresponding values ($\forall$s.(s=$\bot$ $\vee$ $\lceil$isR`s$\rceil$) $\longrightarrow$ P (τabs s)). The remaining proof can be done on the concrete type[14].

A different approach is chosen in the HOL logic [Gor85, GM93]. There exists a method for introducing subtypes and deriving the desired induction rule. In [Mel89] the type of trees is introduced by the following restriction predicate

- p n = $\forall$P.($\forall$tl.Every P tl$\Longrightarrow$P(node_REP tl))$\Longrightarrow$P n where Every P tl states that P holds for every element in the list of nodes and node_REP gives the encoding of trees.

As in this example, the desired induction rule is generally encoded into the restriction predicate in this method. In HOLCF such a restriction predicate could also be formulated, but since it is not admissible it cannot be used to show that the subtype is a pcpo. So we cannot use this HOL method to introduce subtypes, which belong to the class pcpo with derivable induction rules.

3.3.5 Code Generation

Since the schemes of Definitions 3.3.1 and 3.3.2 are conservative introductions of continuous functions they are executable in the sense of Definition 2.1.13. In programming languages with free data types, as ML, the datatype construct may be used to generate code for free data types.

Code generation has to ensure that only corresponding values may be generated. Since the c_i^C are preserving the only dangerous function is the free constructor of the data type τabs. By hiding this constructor we achieve correctness, but we lose the ability to define functions with pattern matching. In [BC93] there is an idea to work around this problem, which would correspond to exporting the continuous constructor function cτabs (see page 104) for the construction of abstract values and the τabs for pattern matching.

Code generation for the free data type is simple:

```
datatype tau = tauabs of sigma;     (* data type *)

fun taurep (tauabs x) = x;          (* representation function *)
```

[14] We will see an application of this rule for streams in Example 6.4.1 .

If the corresponding constant c_i^c for an abstract constant $c_i^a::t$ is executable, then the functions generated by the construction schemes are executable. Code generation for constants is schematically defined by:

- `val` $c_i^a::t = \alpha_t(c_i^c)$;

To ensure that only restricted values are constructed the free data type constructor `tauabs` has to be hidden in the signature.

3.3.6 Method for the Restriction Step

This method for implementing the restrictions is based on the method of conservative extensions in HOL and HOLCF. The general scheme is:

1. Define a translation Th^a to Th^c.
2. Check the translation syntactically (type check, etc.).
3. Apply conservative extension and generate proof obligations.
4. Prove proof obligations.
5. Generate code.

A more detailed description of these steps is:

1. First, the user has to fix the implementation as in Section 3.2.8:
   - the abstract sort τ,
   - the corresponding sort term σ (with $\mu(\tau)=\sigma$),
   - the restriction predicate $\mathtt{isR}::\sigma\rightarrow\mathtt{tr}$,
   - the abstract constants $\mathtt{c}_i^a$, and
   - the corresponding constants $\mathtt{c}_i^c$ for all $\mathtt{c}_i^a$ ($\varphi(\mathtt{c}_i^a) = \mathtt{c}_i^c$).
2. The translation is checked as in Section 3.2.8:
   - type correctness: for all $\mathtt{c}_i^a::u$ the corresponding type has to be $\mathtt{c}_i^c::\mu(u)$
   - the sorts τ and σ have to be data types: $\tau::\mathtt{pcpo}$ and $\sigma::\mathtt{pcpo}$
   - all constants of $Th^a \setminus Th^c$ (except the conservatively defined ones) have to be implemented by φ

- all sort constructors $\{tc_j^a\}$ of Th^a are contained in the sort constructors $\{tc_j^c\} \cup \{\tau\}$ of Th^c

This test can be checked automatically.

3. In this method the proof obligations arise from theory inclusion: for all abstract axioms $ax \in Ax^a$: $\widehat{Th} \vdash ax$, where $\widehat{Th}$ is the conservative extension from Th^c by a new `pcpo` type and by operations, defined by the schemes, as described in Sections 3.3.1 and 3.3.3. Invariance is a proof obligation, which helps to structure the proofs.

4. The proof of the proof obligations has to be done by the developer. It can be completely proved within HOLCF.

5. It is possible to generate code, if the corresponding constants are executable. If so, the code is generated out of $\widehat{Th}$.

So the user has to define the implementation Φ and has to prove invariance and theory inclusion of the generated conservative extension $\widehat{Th}$. The rest is done automatically.

3.3.7 Example: `BOOLEAN` by `NAT`

The method of Section 3.3.6 is illustrated again with the Example 3.1.1, where `B` is implemented by `N`.

1. The definition of the implementation in the example is:

   - μ`(B)` = `N` (τ = `B` and σ = `N`)
   - φ`(T)` = `One` (c_1^a = `T`, c_1^c = `One`)
   - φ`(F)` = `Zero` (c_2^a = `F`, c_2^c = `Zero`)
   - φ`(Bnot)` = Λ `x::N . One - x` (c_3^a = `not`, c_3^c = Λ`x.One-x`)
   - φ`(Band)` = Λ `x::N y::N . If is_Zero‘x then Zero else y fi` (c_4^a = `and`, c_4^c = Λ`x y.If is_Zero‘x then Zero else y fi`)
   - The restriction predicate `isR` is `isB`

2. The translation Φ is type correct. Consider the type of c_4^c: `N→N→N` = μ`(B→B→B)` is the translated type of c_4^a.

3. The proof obligations are all axioms (except the conservative definition of `c`) of `BOOLEAN`, based on the extended theory $\widehat{\texttt{NAT}}$.

   - $\widehat{\texttt{NAT}} \vdash$`Band1` `T Band x = x`
   - $\widehat{\texttt{NAT}} \vdash$`Band2` `F Band x = F`

- $\widehat{\texttt{NAT}} \vdash$Bnot1 Bnot`F = T
- $\widehat{\texttt{NAT}} \vdash$Bnot2 Bnot`T = F

The first extension of NAT is the introduction of a new type by:

```
NAT0 = NAT +
types B 0

consts
   BVal        :: N set              (* subset *)
   Brep        :: B ⇒ N              (* representation *)
   Babs        :: N ⇒ B              (* abstraction *)
defs
        BVal_def  BVal ≡ {s. ⌈isB`s⌉ ∨ s=⊥}
rules
        Brep_Val  Brep t ∈ BVal
        Babs_rep  Babs (Brep t) = t
        Brep_abs  s∈BVal ⟹ Brep(Babs s) = s
end
```

No explicit proofs are necessary, since all proofs were done schematically in Section 3.3.1. The next extension is the partial order:

```
NAT1 = NAT0 +    (* add an order *)
consts
   B_⊑       ::   B ⇒ B ⇒ bool
   B_⊥       ::   B
defs
   B_⊥_def        B_⊥ ≡ Babs ⊥
   B_⊑_def        B_⊑ ≡ λ a b.Brep a ⊑ Brep b
end
```

The next extension is the instantiation of the partial order:

```
NAT2 = NAT1 +
arities
   B :: po
rules
   inst_B_po      (⊑::B ⇒ B ⇒ bool) = B_⊑
end
```

The next extension is the instantiation of the *cpo*.

```
NAT3 = NAT2 +
arities
   B :: pcpo
rules
   inst_B_pcpo    (⊥::B) = B_⊥
end
```

Based on the new type the operations are introduced by the schemes. The result is:

```
NAT^ = NAT3 +
ops curried
   T   :: B
   F   :: B
   Band :: B → B → B      (cinfixl 55)
   Bnot :: B → B
   c   :: B DList         (* a constant *)
defs
T_def     T ≡ Babs‘One
F_def     F ≡ Babs‘Zero
Bnot_def   Bnot ≡ Λx.Babs‘(1−Brep‘x)
Band_def   Band ≡ Λx y.Babs‘(If is_Zero‘(Brep‘x)
                              then Zero else Brep‘y fi)
c_def     c ≡ Map_DList‘Bnot‘(dcons‘T‘(dcons‘F‘dnil))

generated finite B by T | F   (* derivable *)
end
```

The necessary proof obligations of the invariance are the same as in Section 3.2.9.

4. The proofs NAT $\vdash Inv$ and $\widehat{\texttt{NAT}} \vdash$ B can be carried out by functional refinement.

5. Code generation gives (in ML):

```
datatype B = Babs of N;
fun Brep(Babs x) = x;
val F = Babs One;
val T = Babs Zero;
fun Bnot x = Babs(One-Brep x);
fun Band x y = Babs(If is_Zero(Brep x) then Zero else Brep y);
```

The signature of the module must hide Babs.

	Theory Interpretation	Model Inclusion
transitive	$\surd$	$\surd$
modular	no	$\surd$
executable	$\surd$	$\surd$
code generation	compilation	new data type construction
pattern matching	no	no
applicable	normal theory	all theories
restrictions	finite data types	
proof obligations	$\Phi(Th^a), Inv$	Th^a,Inv

Figure 3.3: Methods for the Restriction Step

3.4 Summary and Comparison of Restrictions

This section compares the two methods for the implementation of the restriction step of the implementation of ADTs.

Both methods support the implementation of `pcpo` types by `pcpo` types without additional proof obligations for the instance of `pcpo`. Invariance is a proof obligation in both methods, but since Φ extends the size of the axioms we prefer the proof of Th^a. The admissibility, which is required to instantiate the conservative extension is ensured by the definition of the form of the restriction predicate (with the continuous function `isR`), and therefore it is not mentioned in Figure 3.3.

Pattern matching is not supported by both methods, although there are methods [BC93] that support pattern matching over non-free data types, by having a constructor and a destructor and by separating methodically between them. This would correspond to exporting the constructor τ`abs` only for pattern matching (usually only on the left hand side of function definitions) and `c`τ`abs` for the safe construction of arbitrary terms (on the right hand side). Since this would require a lot of uninteresting work in adapting the semantics and tools to the concepts presented in [BC93] it is not done in this work.

Theory interpretation is restricted to finite data types, since it uses the totality of the selector functions in the proof of Theorem 3.2.3. The conservative construction for the subdomain requires only an admissible predicate and, therefore, we can apply it to infinite data types as well.

To express the restriction step in a refinement relation we defined a (restricted) theory interpretation basis for the method based on theory interpretation. For the method based on model inclusion with conservative extensions we can use the general model inclusion basis of Definition 2.2.2. The refinement relation of the theory interpretation basis is not modular.

Therefore, the method in Chapter 5 uses conservative extension for the restriction step. However, for the implementation of quotients we need theory interpretations.

3.5 The subdom Constructor in HOLCF

Since the restriction step is an important part of the implementation we decided to support the development of restrictions by a (generic) type constructor. An example for a type constructor is `list`. It takes an arbitrary type α and builds the type α `list`. There are several operations on lists (see Example 2.1.4) available. The idea of our `subdom` constructor is similar, but it does not work for arbitrary types α, but only for those types with an admissible predicate. These types are collected in the type class `adm`.

The main goal of the realization of this type constructor was to show that it resides into the class `pcpo`, i.e. it has the

```
arity subdom::(adm)pcpo
```

All proofs of Sections 3.3.1 and 3.3.3 are carried out polymorphically, so that they are available on every type of the class `adm`.

Since the important construction schemata of Section 3.3.3 are type dependent we cannot define a function for them. However, since the most important case is the lifting of an operation of type $\sigma \to \sigma$ to the type $\tau \to \tau$ we defined a special function (`sd_lift`) for this lifting and proved some properties for it. The most important property is the following:

```
inv2cont_lift   inv f ⟹ cont lift f
```

The definition for invariance for this special case is also given.

All theories and theorems for the type constructor `subdom` and the class `adm` are in Appendix A.1. The realization respects some methods from [Reg94, Wen94] which ensure the technical correctness of the conservative extensions.

- The instantiation into type classes is done step by step. For example first `subdom` is defined in `SUBD0`. Then it is proved that the type `subdom` α is not empty. The next step (in theory `SUBD1`) is to define an order on `subdom` α. After having proved that it is a partial order, it is instantiated into the class `po` with the arity declaration `subdom::(adm)po` in theory `SUBD2`. The last step is to add the desired arity `subdom::(adm)pcpo` in theory `SUBD`.

- Axiomatic type classes are used, whenever it is possible. For the definition of the class `adm` we used axclasses and we proved the instantiations. We used the two step technique described in Section 2.1.2 for the introduction of characteristic constants. Since for HOLCF (by now) no version with axiomatic type classes is available, we used the stepwise instantiation into type classes as in [Reg94].

- Introduce continuous functions conservatively (step by step). A continuous function can be defined as composition of continuous functions. In our case τ`abs::`$\sigma\Rightarrow\tau$ is not continuous and hence we cannot use τ`abs` in the continuous function space$\rightarrow$. Therefore, we have to define a function in $\sigma\Rightarrow\tau$ first and after having proved that it is continuous we define a continuous function (for example, the introduction of the lifting operation in Appendix A.1.6).

The realization of the implementation with the `subdom` constructor has one small disadvantage: it does not allow us to formulate different restriction predicates on one type, since this would lead to inconsistent specifications. For example if we like to have the natural numbers `zero` and `one` as representations of boolean values and if we also require the natural numbers from `zero` to `255` to be the representations of characters, then we have this conflict.

In small case studies (where this case does not occur) we may ignore this, but it is easy to work around this problem, like the following example shows:

```
Istream0 = Stream + ADM +
domain α IS = Iabs(Irep::α stream)
defs
        is_Istream_def     adm_pred' ≡ λx.¬stream_finite (Irep`x)
```

This example shows the embedding of the type `stream` to the type `IS` with the domain construct. The restriction predicate for the class `adm` has to use the representation function `Irep`, defined by the embedding. After the proof of the admissibility of `adm_pred'` (in theorem `adm_is_Istream adm (adm_pred'::`α`::pcpo IS⇒bool)`) the embedded type `IS` is instantiated into the class `adm` and we can define the domain of infinite streams as subdomain of `IS`.

```
Istream = Istream0 + SUBD +
instance
        IS :: (pcpo)adm     (adm_is_Istream)
types   α Istream = α IS subdom (* defines infinite streams *)
end
```

The embedding slightly complicates the proofs, but it offers the advantage that the new subdomain of infinite streams may be reused in arbitrary developments.

The operations on infinite streams have to be preserving to ensure their continuity. This means for the concatenation that it is strict, otherwise we could build finite streams by concatenating one element to the undefined stream. The data type of infinite streams uses the following preserving functions on streams and lifts them into the new domain.

```
consts
Scons        :: α → α stream → α stream  (* strict cons *)
                (* lifting for streams *)
IS_lift      :: (α stream → β stream) → α IS → β IS
defs
Scons_def      Scons ≡ Λx.Λs.If is_&&‘s then x&&s else ⊥ fi
IS_lift_def    IS_lift ≡ Λf.Iabs oo f oo Irep
end
```

In addition to the predefined lifting operator `sd_lift` the lifting `IS_Lift` of the embedding is defined.

With these preserving corresponding functions, we define some operations on infinite streams by:

```
consts
        Ift     :: α Istream → α
        Irt     :: α Istream → α Istream
        Icons   :: α → α Istream → α Istream
        ith     :: dnat→α Istream → α
defs
        Ift_def    Ift ≡ Λx.ft‘(Irep‘(rep_sd x))
        Irt_def    Irt ≡ sd_lift(IS_lift‘rt)
        Icons_def  Icons ≡ Λx.sd_lift(IS_lift‘(Λs.Scons‘x‘s))
        ith_def    ith ≡ fix‘(Λith.Λn.Λs.If is_dzero‘n
                         then Ift‘s else
                         ith‘(dpred‘n)‘(Irt‘s) fi)
```

Chapter 4

Quotient Domains in HOLCF

Quotients are an important concept of the implementation of ADTs (see Section 1.3). In the development of interactive systems states can be regarded as quotients of streams and the types of the communication messages in the system can also contain quotients (For example if sets are transmitted). A *quotient domain* is a quotient with domain structure.

Quotients are used in the development, since they allow us to specify multiple representations for the same abstract value. The representations for every abstract value are grouped into an equivalence class by an equivalence relation. An implementation step, which involves multiple representations, is called *quotient step*.

Generating programs, working on the representations instead of the abstract values, is only correct if the (executable) functions have equivalent results for equivalent inputs. This congruence property of functions and types is called observability or behavioural correctness in the literature. Thus, for a method, that allows us to prove the correctness of the quotient step, we have to define the terms equivalence classes, quotients, congruences, and observability.

Section 4.1 contains the motivations for the different concepts. It demonstrates the quotient step with an example for the implementation of states by quotients of streams. In Section 4.2 we define higher order quotients, congruences and observability. These concepts base on partial equivalence relations (PERs). In the realization we tried to formulate as many concepts as possible in the HOL part of HOLCF in order to make them also available for the HOL logic. The result is a type constructor `quot` that takes a type with an arbitrary PER and makes a quotient type of it. The Isabelle realization of this type constructor is described in Appendix A.2.

In Section 4.3 we present an implementation method for quotients using the higher order quotient construction, and model inclusion as the refinement relation. Section 4.4 defines a simple form of theory interpretation for the elimination of quotients. This gives us the possibility to use functions working with representations instead of equivalence classes as

implementations of the abstract functions, provided that the representing functions are observer functions.

In Section 4.5 we compare both methods (model inclusion and theory interpretation) and propose a specification style that allows us to implement the quotient situations in the process of deductive software development[1] by model inclusion and conservative extensions. Section 4.2.5, defines a flexible type class `eq` which can be used to specify continuous equivalence relations for finite data types.

4.1 Motivation

With the restriction step we are able to implement ADTs, such that every abstract element is represented by a concrete one. For example, if we implement sets with a restriction step by ordered sequences, we have the situation that every set is represented by the ordered sequence of its elements. For this implementation we need a total order on the elements and the invariance of the corresponding functions requires to keep the representing elements sorted. For some abstract operations (union) a reordering of the elements is necessary. Therefore, it might be inefficient to have single representations and in the implementation of ADTs it is desired to allow *multiple representations* for the same abstract value. This supports efficient implementations.

In the implementations of ADTs different representations will not give different results, if all programs using the implemented ADT do not depend on the representations. In other words the programs cannot *observe* the difference in the representations. This leads to the general concept of behavioural implementations (see for example [Nip87]):

An ADT C implements an ADT A with respect to a set of programs P, if for all $p(C) \in P$ that use the ADT C holds: $p(C)$ implements $p(A)$.

To show that an implementation is behaviourally correct requires to reason over all programs in P, based on the ADTs. Showing the correctness of behavioural implementations in general depends on the set of observable programs P. This set, and also the correctness proof, base on the programming language in which $p \in P$ may be formulated. In [Nip87][Page 3] it was noted that "*descending to the level of programs for every correctness proof is undesirable. Ideally one would like a criterion on the level of data types which guarantees implementation on the level of programs.*"[2]

One possibility is to encode the programs into the logic of data types. Since we specify data types in HOLCF, we could also encode the notion of programs as in [Gan83] and

[1] In the development situations of Section 1.2.4 we need the quotient step for the elimination of states and in some schematic translations.

[2] In [Nip87] model-theoretic characterizations of behavioural implementation concepts for data types with nondeterministic operations are studied by using the semantics of programs as observers to characterize the behaviour of data types.

of behavioural implementation as in [Wan82] into HOLCF, but encoding both into the logic would fix the target programming language and would lead to a lot of additional formalization, which would make the deductive software development process much more complicated. However, omitting the treatment of observers P from the logic and the development system would be dangerous, since the behavioural correctness cannot be formally verified without observability concepts. Therefore, the crucial point for a method to prove behavioural correctness of implementations is to find a practicable embedding of the observers into the logic of data types.

For unique representations, as for example in the restriction step, we may construct an ADT `C` which refines `A`. Refinement is a behavioural implementation in this general sense, since we may regard the programs p as arbitrary terms over ADTs (as in [Sch85]). With the modularity of refinement we obtain that $p(\mathtt{C})$ refines (implements) $p(\mathtt{A})$. Therefore, model inclusion is a special case of behavioural implementation.

There are many approaches to behavioural implementation, but in HOL and HOLCF none of these has been yet realized. One reason might be that almost all known approaches use a simple equational logic. Instead of formalizing one known approach we combine the power of higher order logic and the *cpo* structured domains of HOLCF to a solution, which allows us to specify quotients over functions and streams, to eliminate quotients, and to express the congruence by specifying the observability for partial and higher order functions in the specification of the ADTs[3].

The following example motivates the different concepts that are defined in Section 4.2. The methods of the following sections use these concepts. The example defines a free ADT of states and shows how it is used to specify a buffer that receives messages and stores values until it is asked to give them back. The interesting aspect are the multiple representations in this example and how they are modelled. Every state has multiple histories as representations and there are two ways to model this. One abstract way is to define an equivalence class for every state. This means to define the type state as a quotient of histories. This allows us refine the specification of states by model inclusion. An executable way is to model multiple representations and to specify states in a way that permits multiple representations (with $\sim$ instead of $=$). The example shows the differences between pure conservative extension and theory interpretation.

Example 4.1.1 ***States for an one-element Buffer***

The requirement specification of the buffer consists of three modules: One specifies the free data type of states, another contains the types of the messages, and the main module contains the specification of the buffer component, using states and messages.

[3] In Section 4.6 we see that many approaches do not have these close integration of axiomatization and observability. Many approaches use observability as a separate semantic concept.

```
State = Dnat +    (* module for states *)
domain State = empty | state(dnat)
State :: eq
end
```

The module uses the domain of natural numbers.

```
Messages = EQ + (* module of messages *)
domain message  = req    | data (dnat)
domain answer   = stored | error | value(dnat)
arities message ::eq
        answer  ::eq
```

Using the `arities` declaration in this way requires that there exists a continuous equality on the types. We could also specify observer functions (for example with `is_Cobs data`, like on page 127), but we use this notation just as a shorthand to inform the reader that there is an equivalence relation on the types. In the concrete case study the equivalence relation is conservatively defined and it is continuous equality[4].

```
(* state-based specification of the buffer component *)
SBUF =  Stream + Messages + State +
ops curried
        Buffer  :: message stream → answer stream
        SBuffer :: State → message stream → answer stream
rules
Buffer_def Buffer ≡ SBuffer‘empty
SBuffer1   SBuffer‘empty‘(data‘n&&s)=stored&&SBuffer‘(state‘n)‘s
SBuffer2   SBuffer‘empty‘(req&&s)=error&&SBuffer‘empty‘s
SBuffer3   SBuffer‘(state‘n)‘(req&&s)=value‘n&&(SBuffer‘empty‘s)
SBuffer4   SBuffer‘(state‘n)‘(data‘m&&s)=error&&SBuffer‘(state‘n)‘s
end
```

This specification is executable, since it uses only free data types with pattern matching. However, for an implementation of the buffer specification in terms of a more concrete system using state-less stream processing functions it may be desired to eliminate the states from the specification[5]. In this example we implement the ADT `State` by a quotient of histories[6].

[4] Continuous equalities (see Definition 2.1.11) are continuous functions computing the result of a congruence (see Definition 4.2.4).

[5] In this simple example this might look strange, but consider the elimination of the data base in a distributed realization of a (monolithic) data base system as a more realistic example.

[6] With *history* we mean the inputs a component has received before the current message arrived.

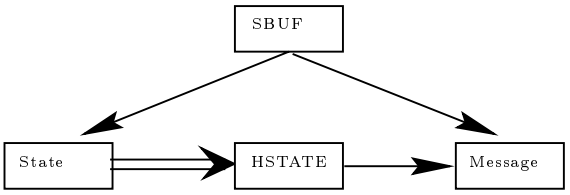

Figure 4.1: Quotient Implementation of `State`

Since our buffer stores only one message, it suffices to look at at the first element of the history streams. We suggest the following implementation:

- `State` $\rightsquigarrow$ `Stream`
- `empty` $\rightsquigarrow$ `req&&`$\bot$
- `state` $\rightsquigarrow$ Λ`n.data‘n&&`$\bot$

Beside `req&&`$\bot$ all other streams of messages, starting with a `req` are representations of the state `empty`. Therefore, we have multiple representations. We implement this step now by model inclusion. Therefore, we construct a model with the quotient constructor (see Section 4.2.2 for the definition). After the specification of an appropriate PER, which relates the equivalent histories, we can define the implementation in the following specification:

```
(* history–based specification of states *)
HSTATE = Dnat + Message + Stream +
types   State = message stream quot  (* quotient of streams *)
ops curried
        empty   :: State
        state   :: dnat → State
defs     (* partial equivalence classes use <[ ]> *)
        empty_def       empty  ≡ <[req&&⊥]>
        state_def       state  ≡ Λn.<[data‘n&&⊥]>
end
```

With this implementation of states we can derive all properties of the required specification of states, including the axioms of the `domain` construct. In other words the specification `HSTATE` is a refinement (by model inclusion) of the specification `STATE`. The situation is depicted in Figure 4.1. However, we lost the direct correspondence to a program, since quotients are not executable in the sense of our Definition 2.1.13 since quotients are not a free data type. The source of our problems is the specification of states. It uses injectivity for the constructors. In our example the injectivity rule of Definition 2.1.12 is:

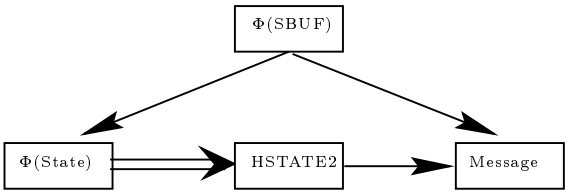

Figure 4.2: Implementation of `State` with Theory Interpretation

`injective` $\quad$ `state‘n=state‘m` $\Longrightarrow$ `n=m`

It is the basis for executability (see Section 2.1.13), but it excludes the executability of our implementation. The idea of our theory interpretation is to replace the equality $=$ in the axioms by a congruence $\sim$ and to require that the functions working on the type cannot observe differences between two equivalent elements.

Having replaced $=$ by $\sim$ in `STATE` and `HBUF` with the theory interpretation Φ (on `Message` Φ is the identity), we can implement it by the following specification:

```
HSTATE2 = Dnat + Message + Stream +
types   State = message stream
ops curried
        empty   :: State
        state   :: dnat → State
defs
        empty_def       empty  ≡ req&&⊥
        state_def       state  ≡ Λn.data‘n&&⊥
end
```

The situation is depicted in Figure 4.2. This implementation is executable and refines the modified specification of states. One interesting aspect is the continuous equality on states. It has to be refined by a continuous equality on streams which is observable by the function `state`, but not necessarily by the functions working on streams (for example the rest function). The continuous equality on histories is a non-trivial task, since the usual classes for equality specifications (see page 128) restrict equalities to flat types and our histories are streams and hence they are not flat.

To summarize the example we see that quotients are models for multiple representations and that quotients are not executable. Observability is used to eliminate quotients, a flexible class for equalities is required. So the main requirements for the treatment of the quotient step are:

- the specification of congruences, including

- the specification of observer functions,
- a quotient construction[7],
- a class `eq` which allows us to specify a continuous equality on non-flat domains.

In this chapter we implement an abstract ADT $T = (\tau, Con^a, Sel^a, Dis^a, Map_\tau, \doteq_\tau)$ specified in a theory $Th^a = ((\Omega^a, C^a), Ax^a)$ by a concrete ADT $S = (\sigma, Con^c, Sel^c, Dis^c, Map_\sigma, \doteq_\sigma)$ specified in a theory $Th^c = ((\Omega^c, C^c), Ax^c)$. To cut down notations we write $\{\mathtt{c}_i^a\}$ for the set of all abstract operations[8] $Con^a \cup Sel^a \cup Dis^a \cup \{Map_\tau\}$ and $\{\mathtt{c}_i^c\}$ for the sets of *corresponding operations* ($\mathtt{c}_i^c \in T_\Sigma c$).

Since the restriction step is treated in Chapter 3 the implementation from T by S in this chapter consists of:

- a sort implementation $\tau \rightsquigarrow \sigma$, where $\tau \in T_\Omega a, \sigma \in T_\Omega c$,
- a constant implementation $\mathtt{c}_i^a \rightsquigarrow \mathtt{c}_i^c$ for all $\mathtt{c}_i^a \in \{\mathtt{c}_i^a\}$, and
- a congruence $\sim$ on the concrete type σ[9]

The congruence groups the different representations for every abstract element together.

The next section defines PERs, quotients and congruences in HOLCF. The following sections of this chapter compare a method based on model inclusion to one based on theory interpretation.

4.2 PERs, Quotients, and Congruences

This section defines partial equivalence relations (PERs), quotients for higher order functions and streams. It also defines a predicate for the specification of higher order observer functions and congruences, which also may be applied to partial functions. The methods for the implementation of the quotient step in the following section use these concepts. The theories and the theorems proved for quotients are described in Appendix A.2.

[7] Even if the quotient construction is not executable it is required, since there we need the existence of a model for the satisfiability of our theory interpretation. For the quotient construct there are many other applications possible (see Chapters 7 and 8).

[8] Except $\doteq_\tau$, which is treated separately.

[9] For the implementation the existence of $\sim$ suffices, however to deduce more properties, we sometimes require the existence of a (flexible) continuous equality $\doteq$ on σ.

4.2.1 PERs

The higher order concept of PERs fits into the logic HOL, but with the gain in generality, we lose some nice properties. For example the composition of observer functions is not necessarily an observer function. Therefore, we can use these higher order quotients only for the specification. Integrating the HOLCF domains and finite data types into the concept of PERs results in the definition of the classes `percpo` and `eq`, which solve these problems.

Definition 4.2.1 ***Partial Equivalence Relation***

A relation $\sim$ on a type R is called *partial equivalence relation* (*PER*), if

- $\sim$ is symmetric: $\forall$`x,y`$\in R$.`x`$\sim$`y` implies `y`$\sim$`x`
- $\sim$ is transitive: $\forall$`x,y,z`$\in R$.`x`$\sim$`y` and `y`$\sim$`z` implies `x`$\sim$`z`

The domain D of a PER is the subset of R, on which $\sim$ is reflexive:

- $D := \{x \in R.x \sim x\}$

PERs are called partial equivalence relations, since they are, in contrast to equivalence relations, reflexive only on the domain D.

From these axioms we can derive (by symmetry and transitivity) that all values not in the domain `D` are not partially equivalent (see Appendix A.2.1).

- `x` $\sim$ `y` $\Longrightarrow$ `x` $\in$ `D`
- `x` $\notin$ `D` $\Longrightarrow \neg$ `x` $\sim$ `y`

In Isabelle, PERs are specified as (polymorphic) functions of type $\alpha \Rightarrow \alpha \Rightarrow$`bool` in the HOL part of HOLCF. All types for which such a function exists are collected in the type class `per` (see Appendix A.2.1). PERs are the basis for higher order quotients and observability.

Compared with equivalence relations, the main advantage of PERs is that for any types S, T with PER $\sim_S$ and $\sim_T$ (and domains $\mathtt{D}_S$ and $\mathtt{D}_T$) we may schematically define a PER on all functions of type $S \Rightarrow T$ by:

- `f` $\sim_{S \Rightarrow T}$ `g` := $\forall$`x y.x` $\in$ $\mathtt{D}_S$ $\wedge$ `y` $\in$ $\mathtt{D}_S$ $\wedge$ `x` $\sim_S$ `y` $\longrightarrow$ `f(x)` $\sim_T$ `g(y)` for all f, g of type $S \Rightarrow T$

It can be shown that the relation $\sim_{S \Rightarrow T}$ is symmetric and transitive and, therefore, PERs are closed under functional composition (see also [Rob89]). Intuitively this means that with PERs on `nat` and `bool` we have automatically PERs on all function types between them (for example `nat` $\Rightarrow$ `bool` $\Rightarrow$ `nat`). This nice property does not hold for equivalence relations, since in general equivalence relations on functions are not reflexive (since $x \sim y \not\Longrightarrow f(x) \sim f(y)$)[10].

[10] The identity is the only equivalence relation, which is closed under arbitrary functional composition.

4.2.2 Quotients

This section introduces higher order quotients, the following section defines a *cpo* structure on them. They are called higher order quotients, since they base on the higher order concept of PERs.

Definition 4.2.2 ***Higher Order Quotient***

Let $\sim$ be a partial equivalence relation on S. Then the *higher order quotient* is the set of all *partial equivalence classes*, defined by:

- QUOTIENT: $S_{/\sim} := \{< [x] >_\sim \mid x \in S\}$ where
- PARTIAL EQUIVALENCE CLASS: $< [x] >_\sim := \{y \in S \mid x \sim y\}$ for all $x \in S$

We sometimes call higher order quotients simply quotients.

Using quotients allows us to implement an abstract type (a set of elements) by a set of equivalence classes. With this, every abstract element is represented by a set of different concrete representations, which are grouped together by the partial equivalence relation.

In our realization in HOL (see Appendix A.2), we introduce a type constructor `quot`, which takes a type with an arbitrary PER and delivers the quotient type, consisting of a set of partial equivalence classes, characterized by some representants. The reader not interested in the technical details of the implementation may look at the theorems proved for higher order quotients in Appendix A.2.3 and believe that we introduce quotient domains and continue to read at Section 4.2.4.

We define a type constructor `quot`, which takes a type with a PER and builds its higher order quotient. The representation of these quotients are all sets of partial equivalence classes. We define a predicate `is_pec` of type α`::per⇒(`α `set) ⇒ bool`, which characterizes the set `s` of equivalent elements, for any representant `x`. It is defined by:

`is_pec_def` `is_pec x s` $\equiv$ $\forall$`y.y∈s=y∼x`

For any partial equivalence class this representant has to exist. Therefore, the specification of an equivalence class is:

`rpred s` $\equiv$ $\exists$`x.is_pec x s`

We named this predicate `rpred`, in analogy to the introduction of subdomains that follows a similar scheme (see Section 3.5). Defining the quotient type as the union of all equivalence classes, as in Definition 4.2.2 could give an empty type, if the PER is always false and, therefore, no representants exists. Since in HOL empty types are not allowed[11], we included the empty set as a representing set for the quotient. This is expressed in the definition of the corresponding values (`cor` and `Val_q`).

[11] The first proof obligation is always to show that an introduced type is not empty.

```
cor_def          cor f ≡ rpred f ∨ f={}
Val_q_def        Val_q ≡ {f. cor f}
```

The quotient type is constructed with a type constructor `quot`

```
types quot 1
arities quot :: (per)term
```

This denotes that `quot` is available on every type which belongs to `per`. The whole construction is on page 228. Another advantage is that the empty set is used as representant for the minimal element in the introduction of the *cpo* structure for quotients.

In order to define operations on quotient types, we need a function, which builds the equivalence class of an element, and an inverse function, which selects an arbitrary element from the equivalence class. These functions are also defined in the theory `QUOT0` (see Appendix A.2.3). The abstraction function <[.]> is of type α`::per` $\Rightarrow$ α `quot`. The representation function is called `any_in`. They are defined by:

```
peclass_def      <[x]> ≡ abs_q {y.y∼x}
any_in_def       any_in f ≡ @x.<[x]>=f
```

For these functions we could derive the following properties (see page 229 for a complete list of theorems):

```
qclass_eqI       x∼y⟹<[x]>=<[y]>
qclass_eqE       ⟦x∈D;<[x]>=<[y]>⟧⟹x∼y
qclass_eq        x∈D⟹<[x]>=<[y]>=x∼y
class_exhaust    ∃z::α::per.∀y.¬z∼y⟹∃s.(x::α quot)=<[s]>
all_class        ⟦∃z::α::per.∀y.¬z∼y;∀x::α.P<[x]>⟧ ⟹ P s
```

Theorem `qclass_eqI` states that the equivalence classes of equivalent elements are equal. This theorem ensures the satisfiability of our theory interpretation in Section 4.4. One disadvantage of our modelling is the form of the rules for induction and exhaustiveness on quotients. Since we included the empty set as a representant for quotients, we have to treat it separately, if the partial equivalence relation is reflexive, i.e if there is no element, which has no equivalence class. This is expressed by the premise $\exists$`z.`$\forall$`y.`$\neg$`z`$\sim$`y`. This ugly premise will disappear in Section 4.2.5, where we introduce the class `eq` and use $\perp$ as the only element, which is not in the domain `D` of the PERs.

All definitions of this section are formulated in the HOL part of HOLCF. Therefore, they could also be used if someone is working only with the HOL part of HOLCF. In the next section the *cpo* structure is introduced (for arbitrary PERs). To construct a quotient domain requires to instantiate the quotients into the type classes `po` and `pcpo` of HOLCF (see page 36).

4.2.3 Quotient Domains

This section defines a flat order on the higher order quotients[12] and shows that they have a *cpo* structure.

The introduction of the arities is done step by step (as in the introduction of subdomains in Section 3.5). First, we define a partial order in theory `QUOT1` (see page 230), then we show the characteristic axioms for the class `po`. Theory `QUOT2` defines the least element and shows that the quotient fulfills the characteristic axiom of the class `pcpo`.

The partial order on quotients is defined by:

`less_q_def` $\quad$ `less_q` $\equiv \lambda$`a`.λ`b`.`rep_q a`={} $\vee$ `a`=`b`

This is a flat order on the quotients with the least element `abs_q` {}. Of course the order is reflexive, transitive and antisymmetric. Therefore, it is a partial order. As was mentioned in Section 4.2.2, we define the least element to be equal to `abs_q` {}:

`UU_q_def` $\quad$ `UU_q` $\equiv$ `abs_q` {}

With this element the quotients are flat domains. See Appendix A.2.4 and Appendix A.2.5 for the theories and theorems of the order. The theorems for the quotient domain are mainly those of `QUOT0`. To derive more powerful theorems, we restrict our domains from the class `per` to the classes `percpo` and `eq` of domains with congruences (see Section 4.2.5).

All realized theories and theorems for the `quot` constructor are polymorphic. Therefore, they may be applied to every type (with a PER). This makes our construction quite flexible and there is no need for carrying out a lot of schematic proofs. This comes from the power of the polymorphism in the Isabelle system with the type classes.

4.2.4 Observability and Congruence

This section introduces observability for partial and higher order functions to define congruences. To specify observability is an important part in the specification of ADTs. It is the basis for the correctness of the behavioural implementation. This section defines a class `percpo` on which these concepts may be used in HOLCF and a predicate to express observability of higher order and partial functions.

Observer functions and congruences are, in equational logic two views of the same thing, since a congruence is substitutive with respect to all observer functions (of the signature of

[12] The quotients are called higher order, since they are build on higher order elements, even if the resulting structures are a flat domains.

a specification). In higher order logic we can define arbitrary functions by λ-abstraction. Therefore there is a difference between all functions of the signature and all possible functions and we use the definition of observer functions to have more fine-grained possibility to characterize congruences.

PERs are the basis for the higher order observability (i.e. PERs allow us to express the congruence property for higher order functions). We could define a predicate for HOL functions, which described the observability by `is_Cobs f` $\equiv \forall$ `x y.x∈D∧y∈D∧x∼y⟶f x∼f y`. Since we work with partial functions, we use HOLCF as logic and `pcpo` domains. Therefore, we need both concepts (PERs and `pcpo` domains) for an adequate specification of observability and congruences. The first step towards a formulation of observability is the introduction of a class `percpo` as a subclass of `per` and `pcpo`. This is in Isabelle defined by (see Appendix A.2.8)

```
axclass percpo < per,pcpo
```

After a definition of a PER on the continuous function space (almost the same as for the HOL functions) and after the proof of the witnesses for the fact that `percpo` is not empty and for the arity $\rightarrow$ `::` `(percpo,percpo)percpo` we are ready to define observability (for details consider Appendix A.2.8).

Definition 4.2.3 ***Observer Function***

Let $\sim_S$ and $\sim_T$ be PERs on the types S and T (and domains $\mathtt{D}_S$ and $\mathtt{D}_T$). Then a function f of type $S \rightarrow T$ is called *observer function*, if

- for all $x, y \in D_S$ with $x \sim_S y$ holds: if $f`x, f`y \in D_T$ then $f`x \sim_T f`y$

This definition treats partiality of functions with respect to the domain D. This concept could also be used in HOL.

In the following section we define the class `eq` with a continuous equality. One benefit of `eq` is that the domain of the PERs coincides with the definedness of elements in the type class `eq` (`(x::`α`::eq)∈D=x≠⊥`) This means that $\bot$ is the only element not in `D`. The axiomatization of observability for continuous functions in HOLCF defines a predicate `is_Cobs` by:

```
is_Cobs_def   is_Cobs f ≡
              ∀x y.x∈D∧y∈D∧x∼y⟶f`x∈D∧f`y∈D⟶f`x∼f`y
```

Since the functions may be partial we fail to prove the general composition theorem for observable functions (`is_Cobs f∧is_Cobs g⟶is_Cobs (Λx.f`(g`x))`). This is another motivation for the introduction of the class `eq`.

There are many other approaches to observability in the literature (some of them are described in Section 4.6). Many of them use observable types for the specification of observability and require all functions containing this type to be observer functions. For first order logic this is adequate, since the set of all functions consists simply of those in the signature, but in higher order logic we may build functions by lambda-abstraction or application of higher order terms. Therefore, we need the more fine-grained notion of observer functions. Another advantage is the integrated treatment of observability. We may specify observability simply by writing additional axioms for the data type (see for example page 50).

With the notion of observer function we can define congruences.

Definition 4.2.4 ***Congruence***

A (family of) PERs $\sim_{\tau_j}$ on an ADT $T = (\tau, \{\mathtt{c}_i^a\})$ is called (partial) congruence relation, if

- all functions $f \in \{\mathtt{c}_i^a\}$ of type $S \to T$ are observer functions with respect to the PERs $\sim_S, \sim_T \in \{\sim_{\tau_j}\}$

Usually we call the PER $\sim_\tau$ a congruence. In this case we ignore the other PERs of the family[13].

We specify the congruence property in HOLCF for an ADT $T = (\tau, \{\mathtt{c}_i^a\})$ by requiring that the predicate `is_Cobs` holds for all operations $f \in \{\mathtt{c}_i^a\}$.

4.2.5 The Class `eq`

This section defines a class `eq`, which allows us to specify the continuous equality operation (see Definition 2.1.11) and also to specify observer functions, needed for proving the behavioural correctness of the quotient step. To present the advantages of the class `eq` it is compared with the class `EQ` from SPECTRUM and in addition we define a class `EQ` in HOLCF, just to show the difference between `EQ` and `eq`. As we will see types of our class HOLCF are not normalizeable.

First, we look at the class `EQ` of the specification language SPECTRUM [BFG+93b]. It is defied by the following SPECTRUM specification:

[13] In the development process every type of the class `pcpo` should have a PER. We give methods for the definition of PERs in Section 4.3.2.

```
Predefined_Specification =
{
    class EQ;
        .==. : α::EQ ⟹ α × α → Bool;
        .==. strict total;
    axioms α::EQ ⟹ ∀ a,b:α in
        {weak_eq}  (a == b) = (a=b);
    endaxioms;
}
```

This requires the continuous function == to coincide with the equality on defined values. From the monotonicity of the function == it follows that all types in the class `EQ` have to be flat. For the elimination of states in state-based specifications by histories of streams (see Section 1.2.4 for the general development situation and page 118 for an example), we need a continuous equality on non-flat domains. In addition this class `EQ` does not allow different representations (without the use of theory interpretations). However, in a development with finite data types without multiple representations such an equality class is useful. Therefore, we introduce two equality classes: The flexible class `eq` and the rigorous class `EQ`.

We start with the flexible type class `eq`. It could be defined as subclass of `percpo` with an additional operation $\doteq$ for the continuous equality and some axioms describing it. Instantiating a type into this class `eq` correctly would require to instantiate it first into the class `per`. We prefer a more comfortable instantiation, especially a schematic definition of the PER by $x \sim\sim y \equiv \lceil x \doteq\doteq y \rceil$ would be nice since the developer would not have to define first the PER, show that it is symmetric and transitive, then to instantiate the PER and continue with the definition of the continuous equality. Instead we define our class `eq` with the following axioms for $\doteq$ (see Appendix A.2.9 for the whole theory):

```
ax_eq_refl_def   x≠⊥⟹⌈x≐≐x⌉
ax_eq_sym2       ⌈x≐≐y⌉⟶⌈y≐≐x⌉
ax_eq_trans2     ⌈x≐≐y⌉∧⌈y≐≐z⌉⟶⌈x≐≐z⌉
ax_eq_strict1    ⊥ ≐≐ x = ⊥
ax_eq_strict2    x ≐≐ ⊥ = ⊥
ax_eq_total      ⟦x≠⊥;y≠⊥⟧⟹x≐≐y≠⊥
ax_eq_per        x∼∼y=⌈x≐≐y⌉   (* just to define a per *)
```

These axioms are the axioms for the continuous equality from the definition of ADTs (on page 46). In addition they include the axiom `ax_eq_per` for the schematic definition of the PER. We can simply deduce that $\sim\sim$ is symmetric and transitive. Speaking in terms of axiomatic type classes this means that we can prove the subclass relation `per`$<$`eq`[14].

[14] The syntax of this construct allows us to give the names of theorems containing the witnesses to verify the arity.

This subclass relation is expressed by the following statement:

```
instance eq<per          (eq_sym_per,eq_trans_per)
```

For this class we can derive a lot of theorems. The most important is:

```
UU_eq  (x::α::eq)∈D=x≠⊥
```

The theorems for quotients are also more elegant, since the premise $\exists$z::α::per.$\forall$y.$\neg$z$\sim$y can be removed from the induction rule and from the exhaustiveness rules for quotients since $\forall$y.$\neg\bot\sim$y holds. A further theorem describes the composition of observer functions on the class eq:

```
is_Cobs_comp_eq  ⟦is_Cobs(f::β::eq→γ::eq);is_Cobs(g::α::eq→β)⟧
                    ⟹  is_Cobs (f oo g)
```

This theorems ensures the behavioural correctness of programs that are composed of observer functions. All theorems for eq are listed in Appendix A.2.10.

The class EQ

As we saw in Section 2.1.5 sometimes there was a class EQ used, which is equivalent to the equality class EQ of SPECTRUM. Using this class seamed only helpful to simplify some specifications because on this class all total functions are observers. The class is specified as subclass of the class eq

```
axclass EQ < eq
        ax_EQ   "∀a b.a≠⊥∧b≠⊥⟶(⌈a≐≐b⌉=(a=b))"
```

Theorems and the full theory of the class EQ are listed in Appendix A.2.10. However, we prefer not to use this class, since it does not allow us to define flexible implementations. All specifications using this class are not normalizeable since they implicitly contain the characteristic axiom of the class ax_EQ, which is a polymorphic predicate using =. Therefore, the simple theory interpretation of Section 4.4.3 cannot be applied to allow multiple representations for types of this class. So using EQ instead of eq for an ADT fixes the implementation of this ADT.

The next section defines a method for the implementation of the quotient step. It uses the quot constructor to extend the concrete specification and it requires to prove the refinement of the abstract specification by theory inclusion. Since it uses quotients it is not directly executable and we need a method with theory interpretation to come to a specification which is directly executable. This method is presented in Section 4.4.

4.3 Model Inclusion

This section implements quotients with model inclusion and conservative extension. The presented method uses the `quot` constructor to define a quotient.

The conservative introduction of a new type in HOLCF requires with the axiom $rep(abs(q)) = q$ that the representation is unique (see Section 2.1.4). Therefore, the main goal of having different representations of the same abstract value cannot be achieved by conservative extension, but different representations of the same abstract element can be grouped together by the partial equivalence relation.

Quotients are not a free data type and therefore, cannot be translated directly into functional programs. The translation from the equivalence classes in the quotients to single representing elements requires a correctness proof. This can be verified at the level of programs (see for example some approaches in Section 4.6) or, independently from programming languages at the level of specifications (by theory interpretation).

The reason for presenting an inexecutable model for quotients is that it is needed in the next section to show the satisfiability of the theory interpretation, which eliminates the quotient step. Furthermore quotients are useful in the abstract specification of systems (for example if states are used).

The *cpo* structure of the domains is ensured by the construction of quotients in Section 4.2.2. The method defines continuous operations on the quotient domains.

4.3.1 Method for the Quotient Step

This section describes the method for the implementation of the quotient step in HOLCF. It is in the following sections, applied to Example 4.1.1 where an ADT of states is implemented by a quotient over streams.

The method extends the concrete ADT $S = (\sigma, \{\mathtt{c}_i^c\})$ with the `quot` constructor. The main proof obligation is a theory inclusion, which ensures model inclusion of the extended theory[15]. The concrete steps are:

1. Define a PER on σ.
2. Instantiate it into the class `per`. The proof obligations are the characteristic axioms of `per`: symmetry and transitivity.
3. Extend the concrete ADT S with the type $\tau = \sigma$ `quot`.
4. Introduce the operations $\mathtt{c}_i^a$ as liftings from the corresponding operations $\mathtt{c}_i^c$ into an extended ADT $\widehat{S}$

[15] In this section we are working with the model inclusion basis of Section 2.2.

5. Prove that the extended ADT $\widehat{S}$ with the quotient type and the new operations is a refinement of the abstract ADT T. This requires to prove all axioms $ax \in Th^a$:
   (a) the induction rule for the abstract type τ and
   (b) the axioms for the specifications of the data type.

The following sections describe these steps more detailed.

4.3.2 Definition of the PER

This section describes a method for the definition of the PER on the concrete type σ. The aim of the PER is to group the different representations of the every abstract value together into one equivalence class.

In many cases the required PER is induced by equations between terms in the specification of the abstract terms. For instance, from the example of the introduction (page 16) we have the axioms:

```
set4    add‘x‘(add‘x‘s) = add‘x‘s
set5    add‘x‘(add‘y‘s) = add‘y‘(add‘x‘s)
```

Implementing `add` by `cons` induces us the following axioms for our PER:

```
PER4    cons‘x‘(cons‘x‘s) ∼ cons‘x‘s
PER5    cons‘x‘(cons‘y‘s) ∼ cons‘y‘(cons‘x‘s)
```

If we have a complete set (all possible equations between the constructor terms, for example {1=2,2=1,2=4,4=2,1=4,4=1}) of equations, then the induced relation is obviously symmetric and transitive.

If the definition of the PER is not so simple (for example if inequations are used in the specification), the developer has to define the PER explicitly and has to derive the desired properties. A special case is the definition of PERs on streams. In our example on page 118, we used only the first elements of the streams, but in general all elements can be used. For example two streams are partially equivalent, if all elements in the stream are partially equivalent. This PER could be defined with the following fixed point construction for the definition of the PER:

```
SPer0 = Stream + Wfp + PER +
types
        α StPER         = α::per stream × α stream ⇒ bool
consts
```

```
        SperFkt :: α::percpo StPER ⇒ α StPER
        Sper    :: α::percpo StPER
defs
        SperFkt_def     SperFkt F ≡ λ(s,t).ft‘s∼ft‘t ∧ F((rt‘s),(rt‘t))
        Sper_def        Sper ≡ wfp SperFkt
end
```

Since a PER is a predicate, we cannot use the least fixed point operator for the definition of continuous functions from HOLCF. We define the PER on streams (`SPer`) to be the weakest fixed point of a functional (`SperFkt`) that recursively describes the desired properties of the PER. The weakest fixed point construct on predicates (`wfp`) is defined by:

```
types
  α pred = (α ⇒ bool)
consts
  Charfun       :: α set ⇒ α pred
  wfp           :: (α pred ⇒ α pred) ⇒ α pred
Charfun_def   Charfun  ≡ λA. λx. x∈A
wfp_def       wfp PF   ≡ Charfun(gfp(Collect o PF o Charfun))
```

It uses the greatest fixed point of sets (see [Pau94a] for more details). Definitions of recursive predicates as weakest fixed points are standard in the FOCUS method [BDDG93] and their tool support is described in [SM97, Ohe97]. Roughly speaking the method starts with the proof that the predicate is monotone and then it requires to deduce properties from the fixed point definition of the predicate.

The easy way[16] to define PERs (on types of the class `eq`) is to define the PER schematically. Consider the following specification of the domain of natural numbers:

```
Dnat0 = EQ +    (* introduce dnat with continuous equality *)
domain dnat = dzero | dsucc (dpred :: dnat)
defs
        dnat_eq_def     (op ≐≐) ≡ fix‘(Λeq.Λx y.
                                If is_dzero‘x
                                then is_dzero‘y
                                else If is_dzero‘y
                                        then FF
                                        else eq‘(dpred‘x)‘(dpred‘y)
                                     fi
                                fi)
        dnat_per_def    (op ∼∼) ≡ λx y::dnat.⌈x≐≐y⌉ (* schematic *)
end
```

[16] The identity is trivially a PER, but since ⊥=⊥, there exists no possibility to define a continuous equality for it.

This specification contains the definition of the domain of natural numbers with the domain construct and the definition of the continuous equality as recursive continuous function with the fixed point construction from HOLCF. The PER on `dnat` is defined schematically. After the instantiation into the class `eq` it is automatically available on `dnat`. This is possible, since we included the axioms `ax_eq_per  x∼∼y=⌈x≐≐y⌉` into the characteristic axioms of the class `eq`. Therefore, we could also prove the fact that `PER` is a subclass of `eq` (see Section 4.2.5 for the definition of the class `eq` and Appendix A.2.9 for the theorems)

In the Example 4.1.1 we have the following definition of the continuous equality on histories. This induces the (schematic) definition of the PER.

```
domain HV  =  Habs (Hrep:: message stream)
HV_eq_def     (op ≐≐) ≡ Λx y.ft‘(Hrep‘x)≐ft‘(Hrep‘y)
HV_per_def    (op ∼∼) ≡ λx y.⌈x≐≐y⌉
```

Since we can only define one PER for every type, and since we want to specify components, which may be reused in greater systems we used the same embedding as in the example on page 114 for the definition of histories and the continuous equality on it. The definition of the continuous equality on histories bases on the continuous equality of messages.

To instantiate `HV` into the class `per` is not explicitly necessary, since we instantiate `HV` with the following statements into the class `eq`, which is a subclass of `per`.

```
instance HV :: eq     ( HV_eq_refl,HV_eq_sym,HV_eq_trans,
                        HV_eq_strict1,HV_eq_strict2,HV_eq_total,
                        HV_per_def2 )
```

The theorems contain witnesses for the characteristic axioms of the class `eq`. They could easily be derived from the definition `HV_eq_def`.

4.3.3 Introducing a Quotient Domain

Since we defined the `quot` constructor polymorphically for the type class `per` we can use it now simply by $\tau = \sigma$ `quot`. In our example this leads to the following type declaration for `State`:

```
types State = HV quot
```

The next step is to introduce the operations, which correspond to the operations of `State`.

4.3.4 Introducing Operations

This section describes the introduction of the abstract operations $\{\mathtt{c}_i^a\}$ on the basis of the concrete operations $\{\mathtt{c}_i^c\}$. The situation is similar to the introduction of operations for subdomains. We have corresponding terms for the required abstract operations and we have the abstraction function <[.]>, which builds the equivalence class and the representation function any_in, which selects an arbitrary element from the equivalence class (see Appendix A.2.3 and Appendix A.2.10 for theorems on these functions). Therefore, we can apply the same schemes to define the general conversions between an arbitrary abstract constant and its corresponding term. The schemes are described on page 105 and are not repeated here.

Since for the introduction of quotients there is no restriction predicate, we have no invariance requirements. Since the quotient domain is flat, continuity of the introduced abstract operations is quite simple. The interesting aspect is the monotonicity. For the simple scheme we derived a lifting theorem for strict functions:

monofun_lift $\forall$x. $\neg$ x $\sim$ f`$\bot$ $\Longrightarrow$ monofun ($\lambda$x. <[ f`(any_in x)]>)

In the example we have the corresponding functions and the following definitions of the abstract functions:

```
    (* corresponding functions *)
ops curried
        empty_  :: HV
        state_  :: dnat → HV
defs
        empty__def      empty_ ≡ Habs`(req&&⊥)
        state__def      state_ ≡ Λn.Habs`(data`n&&⊥)
    (* abstract functions *)
ops curried
        empty   :: State
        state   :: dnat → State
        SBuffer :: State → message stream → answer stream
        Buffer  :: message stream → answer stream
defs
        empty_def       empty  ≡ <[empty_]>
        state_def       state  ≡ Λn.<[state_`n]>
        SBuffer_def     SBuffer≡ Λs.HBuffer`(any_in s)
        Buffer_def      Buffer ≡ SBuffer`empty
```

As was mentioned in the motivating example in Section 4.1, the quotient domain is not executable However, one extension of this work could be to extend the definition of executable specifications to quotients by giving a verified interpreter or a similar tool to ensure

the behavioural correctness of these specification. Therefore, it makes sense to think about the continuity of the introduced operations.

4.3.5 Proof Obligations

The proof obligations of the quotient step with the model inclusion basis are:

- to show that the PER is symmetric and transitive, and
- to derive all abstract axioms from the concrete axioms by theory inclusion.

The first proof obligation was treated in Section 4.3.2. The refinement of the abstract axioms requires to prove all axioms in Ax^a. These are the axioms of the specified functions and the axioms for the domain construct, especially the induction rule. Since the proof of the axioms of the function depends on their specification, it is hard to give general rules to support these proofs. The only general style is that functions are frequently specified by equations. Proving an equation between equivalence classes requires to show that the representants are equivalent. We have the following rules:

```
        (* equality and symmetry for equivalence classes *)
qclass_eqI      x∼y⟹<[x]>=<[y]>
qclass_eqE      ⟦x∈D;<[x]>=<[y]>⟧⟹x∼y
qclass_eq       x∈D⟹<[x]>=<[y]>=x∼y
```

For the introduction of the induction rule we have the following theorems for induction over the class `per`:

```
all_class       ⟦∃z::α::per.∀y.¬z∼y;∀x::α.P<[x]>⟧ ⟹P s
all_class2      ⟦P(abs_q{});∀x.P<[x]>⟧ ⟹P s
```

for the induction over types of the class `eq` we have the following stronger rule:

```
eq_all_class    ∀x::α::eq.P <[ x ]> ⟹ P (s::α quot)
```

In addition to these rules there are a many other rules, supporting the refinement proofs (see Appendix A.2.10).

4.4 Theory Interpretation

This section defines a theory interpretation Φ for the quotient step of the implementation. Although we have a corresponding type σ, our theory interpretation Φ translates terms over τ into terms over τ. The implementation of type τ by a type σ could be done with a free domain construct (consider the example in Section 5.3) and the known lifting techniques. In this section we concentrate on the theory interpretation, which makes this implementation possible.

To ensure that a theory interpretation, as defined in Definition 2.3.1, is made it is necessary to show for an abstract theory $Th^a = (\Sigma^a, Ax^a)$ and a concrete theory $Th^c = (\Sigma^c, Ax^c)$:

- CORRECTNESS: $Th^a \vdash \psi$ implies $Th^c \vdash \Phi(\psi)$ for all formulas $\psi \in T_{\Sigma^a(\emptyset)}$
- SATISFIABILITY: $M \models Ax^c$ and $M \models \Phi(Ax^a)$ implies that there exists $\widehat{M}$ with $\widehat{M} \models Ax^a$.

The first requirement is a proof obligation for the user. From our definition of models it follows that it suffices to prove all axioms $\Phi(Ax^a)$ instead of all formulas ψ. The second requirement is our task in this section to show that the interpretation is satisfiable. This is trivial, since we define a very simple form of theory interpretation.

Theory interpretation for quotients replaces $=$ by $\sim$ and requires $\sim$ to be a congruence. If $=$ is used in polymorphic predicates, we cannot replace it by $\sim$. Therefore, normalization is needed again (as in Section 3.2.5). However, we do not repeat the normalization in this chapter.

The theory interpretation Φ for the quotient step is simple, since it has no premises. Therefore, there is no need for invariance requirements and this section is structured as follows: first, Φ is formally defined on normalized theories and satisfiability is proved. Then, the method is presented, including proof obligations and code generation. The treatment of Example 4.1.1 shows the applicability of this method. The section concludes with a definition of a software development basis containing a refinement relation for the quotient step.

4.4.1 Simple Sort Translation μ

The theory interpretation Φ bases on a simple sort translation μ. The sort translation μ is the basis for the term translation. In the quotient step it is the identity:

- μ_ID: μ(x) = x for all $x \in T^a_\Omega$

The only non-trivial requirement is that the sort translation generates the the following arity:

- μ_ARITY: $\tau :: (\overline{\texttt{eq}})\texttt{eq}$ where $\tau \in \Omega^a$ is the abstract sort constructor, and
- the observability specification `is_Cobs` f for all functions $f \in \{\texttt{c}_i^c\}$

This makes the congruence $\sim$ available on τ and ensures some nice properties, which we require for the satisfiability of the method[17]. The necessary instantiation is a normal refinement step in the deductive software development process towards executable programs.

4.4.2 Simple Constant Translation φ

The only translated constant is $=::\tau\Rightarrow\tau\Rightarrow\texttt{bool}$. It is translated into $\sim::\tau\rightarrow\tau\rightarrow\texttt{bool}$.

Definition 4.4.1 ***Simple Constant Translation*** φ

The simple constant translation $\varphi : \Sigma^a \longrightarrow T_{\Sigma^C}$ is defined by:

- φ_EQ : $\varphi(\lambda\texttt{x y .x=y}) = \lambda\texttt{x y.x}\sim\texttt{y}$ if $=$ is used on the type τ.
- φ_CONST : $\varphi(c) = c$ for all other constants $c \in \Sigma^a$.

Since there is no real sort translation and no real constant translation there are no additional requirements for type correctness.

4.4.3 Simple Translation Φ

Since there are no restrictions in the quotient step all terms are translated canonically by Φ.

Definition 4.4.2 ***Simple Term Translation***

Let μ, φ be simple sort and constant translations. Then the simple term translation Φ for quotients is canonically defined by

- Φ_CON: $\Phi(c) = \varphi(c)$

[17] Here is an aspect for future work. It could be possible to find a more restrictive formalization of the class `percpo` or a more restrictive formalization of the predicate `is_Cobs`, for example a restriction to total functions could ensure the satisfiability for restricted PERs. We restricted the satisfiability to the class `eq`. However, for the specification the class `percpo` can be used. This will be our proposal for a method based only on model inclusion, which allows multiple representations in Section 4.5.

- Φ_VAR: $\Phi(x::u) = x::\mu(u)$ if x is a variable
- Φ_ABS: $\Phi(\lambda x::u.t) = \lambda x::\mu(u).\Phi t$
- Φ_APP: $\Phi(f\ t) = \Phi f\ \Phi t$

Both rules Φ_1 and Φ_2 of the general theory interpretation on page 58 are defined without complicated terms (in contrast to the theory interpretations on pages 60 and 74). Since there are no premises in the translation the invariance is not needed for the quotient step. The simple term translation Φ is the theory interpretation for the quotient step. Its correctness is ensured since the translated axioms are proof obligations in the refinement process. The injectivity axiom of Example 4.1.1 is translated into

`injective  state‘n∼state‘m ⟹ n∼m`

The next section shows the satisfiability of simple translations.

4.4.4 Satisfiability

The task of this section is to give a model $\widehat{M}$, which ensures:

- SATISFIABILITY: $M \models Th^C$ and $M \models \Phi(Ax^a)$ implies there exists $\widehat{M}$ with $\widehat{M} \models Ax^a$.

We need the satisfiability of our theory interpretation to ensure that there exists a model of the abstract theory, if we have a model for the concrete theory. In the implementation of ADTs this will ensure, that replacing the equality by a congruence is a correct development step.

The key idea of the proof of satisfiability is to construct a quotient model, which bases on the type τ. The first step is to construct a type model:

Definition 4.4.3 ***Quotient Type Model Construction***

Let $\tau \in T_\Omega a$ be the abstract sort (of class `eq`). Then the type model construction $\widehat{TM}$ for a concrete type model $TM = (PU, TC)$ is defined by:

- $\widehat{TM} = (PU, TC \cup \{\widehat{\tau}\})$ where
- $\widehat{\tau} = \{f \in TM[\![\texttt{quot}\ \tau]\!]^\Omega\}$

In Section 4.3.1 it was shown that this type has a *cpo* structure. Now the quotient model $\widehat{M}$ is constructed.

Definition 4.4.4 ***Quotient Model*** $\widehat{M}$

Let Φ be a simple translation and $M = (\Omega^a, C)$ be a model of $\Phi(Ax^a)$. Then the quotient model $\widehat{M}$ is defined by:

- $\widehat{M} = (\widehat{TM}, C \cup \{abs, rep\})$ where *abs* and *rep* are defined as `abs_q` and `rep_q` in the introduction of the `quot` constructor (see page 124).

With this semantic extensions of the concrete models, we have abstract models. The only remaining task is to show that they are models of the requirement specification. This is quite easy, since we know that they are models of the concrete specification and, hence, they satisfy the translated axioms of the form `t` $\sim$ `s`. From this we may deduce with the rule `qclass_eqI` `x` $\sim$ `y` $\Longrightarrow$ `<[ x ]>` = `<[ y ]>` (see Appendix A.2.6), that the abstract axioms hold. Therefore, our model is a model of the requirement specification. The congruence property ensures that the behaviour (the results) of arbitrary compositions of observer functions are equal[18].

4.4.5 Method for the Quotient Step

To find a method for the quotient step all required assumptions are collected and put into the method. The general scheme is:

1. Check if the theory Th^a is normalizeable.
2. Apply normalization and the simple theory interpretation Φ.
3. Generate proof obligations for further refinement ($\Phi(Ax^a)$).

Since Φ is fixed the developer only has to apply it. No additional consistency definition and checks are needed. The proof obligations are simple refinement obligations and are not part of the quotient step, although they have to be proved in the further development towards executable specifications.

4.4.6 Example: State by Histories

The method is applied to Example 4.1.1. The theory `State` is normalizeable, since no polymorphic predicates that are not conservative are used. The `domain` construct for states is expanded. The only translated rule is `injective`:

[18] The theorem `is_Cobs_comp_eq` `[[is_Cobs f;is_Cobs g]]==>is_Cobs (f oo g)` in Section 4.2.5 ensures this for types in the class `eq`.

```
ΦState = Dnat +
types state 0
ops curried
        is_empty:: State → tr
        is_State:: State → tr
        empty   :: State
        state   :: dnat → State
rules
          (* Induction rule *)
State_Ind ⟦P ⊥;P empty; P (state‘n)⟧ ⟹ P x
          (* discriminator rules *)
is_empty1 ⌈is_empty‘empty⌉
is_empty2 ⌊is_empty‘(state‘y)⌋
is_state1 ⌈is_state‘(state‘y)⌉
is_state2 ⌊is_state‘empty⌋
          (* translated injectivity rule *)
injective (state‘n ∼ state‘m) ⟹ n∼m
          (* added by the theory interpretation *)
arity state::eq
rules   (* specify the congruence *)
        obs1 is_Cobs state
        obs2 is_Cobs is_empty
        obs3 is_Cobs is_state
end
```

From now on the rest of the example is conservative extension and model inclusion. The examples in Section 5.3 and Section 7.4 show these steps.

Since ML does not differentiate between the equality $=$ and the continuous equality $\doteq$ the code generated for the quotient step should use a functional language like Gofer [HJW92, Jon93]. Gofer allows us to redefine the equality of the class EQ. In addition Gofer has lazy type constructors and, therefore, allows us to generate prototype code for the stream processing functions.

4.4.7 Simple Theory Interpretation Basis

This section defines a refinement relation for the restriction step of the implementation based on the theory interpretation of the previous sections. With this refinement relation a basis for the deductive software development process is defined. This section shows the deficiency of the method and proposes a pragmatic way to specify data types in order to allow multiple representations.

The following definition of the simple theory interpretation basis includes the definition of a refinement relation for model inclusion:

Definition 4.4.5 ***Simple Theory Interpretation Basis***

Let $(\mathcal{L}_{Th}, \texttt{MOD}, \supseteq, \Longleftarrow)$ be a model inclusion basis, and let Φ be a simple theory interpretation for quotients, then the quadruple $(\mathcal{L}_\Phi, \texttt{MOD}, \supseteq_\Phi, \Longleftarrow_\Phi)$ is called *simple theory interpretation basis*, if

- $\mathcal{L}_\Phi \supseteq \mathcal{L}_{Th}$ is the set of all normalizeable specifications,
- $\supseteq_\Phi$ is for $M, N \in \texttt{MOD}$ defined by:
  $M \supseteq_\Phi N := M \supseteq N$ or $M \supseteq \widehat{\Phi}(N)$ where $\widehat{\Phi}(N)$ is the quotient model for N (see Definition 4.4.4). This refinement relation allows us to do refinement by model inclusions and simple theory interpretations.
- $\Longleftarrow_\Phi$ is defined by:
  $S \Longleftarrow_\Phi T := S \Longleftarrow T$ if a refinement by model inclusion has to be proved and $\Phi(S) \Longleftarrow T$ if a theory interpretation has to be proved.

It is obviously a deductive software development basis and due to the satisfiability of the theory interpretation it is also consistency preserving. Transitivity is trivial, since $=$ may only be replaced once by $\sim$. To see that Φ is not modular consider, as in Section 3.2.10, a specification `S`, which uses a specification `A`. If `A` is normalizeable then Φ`A` is well-defined, but if `S` is not normalizeable (for example due to a non-conservative polymorphic predicate in `S`) Φ cannot be applied to `S` and, therefore, simple theory interpretations are not modular in general.

Again, as in Chapter 3 the embedding of the implementation of ADTs in the deductive software development process provides a solution, since it ensures that specifications are normalizeable, when data types are implemented. Compositionality of Φ is proved in Chapter 6.

Another more pragmatic solution is to use $\sim$ instead of $=$ in the specification of the system and to specify the observers on the abstract type. This allows us to base the complete development on conservative extension and gives the opportunity to characterize the observers for every data type within the specification. An additional advantage of this specification style is that we are not restricted to the class `eq` and, therefore, we may specify quotients of more types (for example stream does not reside in the class `eq`). Since this would be a severe restriction of the specification style, the method in the next chapter assumes normalizeability of the theories and uses the simple theory interpretation basis.

	Theory Interpretation	Model Inclusion
transitive	$\surd$	$\surd$
modular	normal theories	$\surd$
compositional	$\surd$	$\surd$
code generation	$\surd$	not directly
pattern matching	$\surd$	*no*
representations	different	unique
applicable	normal theories	all theories
proof obligations	$PER, \Phi(Th^a)$,`is_Cobs`	PER, Th^a

Figure 4.3: Methods for the Quotient Step

4.5 Summary and Comparison of Quotients

The comparison of the methods (Figure 4.3) is difficult, since they are used for different purposes. Conservative extension extends theories with unique representation where theory interpretation allows different representations. If different representations are modelled with conservative extension by equivalence classes the resulting specifications are not executable. The requirement of unique representations in conservative extension can be weakened by normalization, which allows executable implementations. The behavioural correctness of this step is formalized in the logic and is ensured by code generation. Conservative extensions cannot be used in the deductive software development process without normalization.

The main difference to theory interpretation for the restriction step is that theory interpretation for quotients is modular for normal theories. The reason is that it does not translate sorts and, therefore, Φ does not fix a concrete representation. Both methods are transitive. The only disadvantage of simple theory interpretation is that it is in general not modular, but in Chapter 6 it is proved that simple theory interpretations are compositional. Simple theory interpretations are an elegant way to introduce observability into specifications, since they ensure satisfiability without program dependent proof obligations.

The proof obligations are interesting aspects. Both methods require the proof of the PER. In contrast to the previous chapter there is no restriction and, therefore, no invariance is required. The proof of the congruence property with the observability is the price we pay in theory interpretations for the possibility of code generation. Both methods have only simple refinement proof obligations.

For understanding how different representations and pattern matching are possible consider Figure 4.4. It is impossible to find a conservative extension (in the sense of HOL, which uses abstraction and representation functions), which allows us to have different representations of the same abstract object (`x=y` $\Longrightarrow$ `crep x = crep y`). The conservative extension presented in Section 4.4 uses equivalence classes as representations. This requires only a

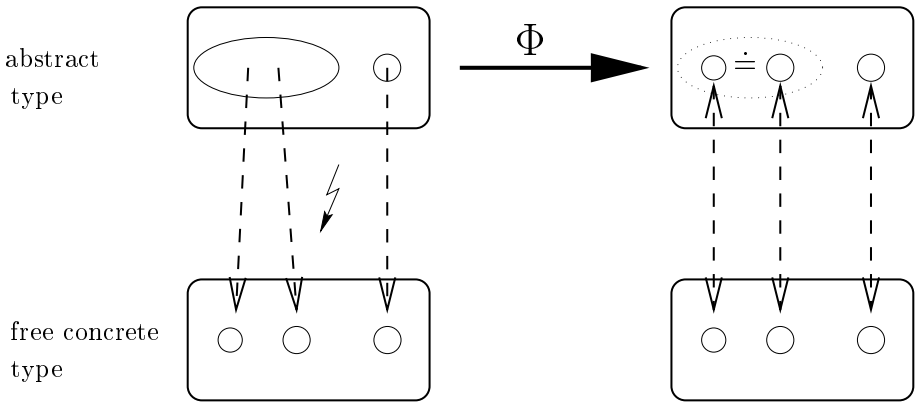

Figure 4.4: Theory interpretation for quotients

congruence $\sim$ on the corresponding type for a unique representation. The idea to solving this problem by theory interpretation is that it eliminates the quotient by weakening the equality to a congruence, which allows us to group together different elements by $\sim$. A simple theory interpretation weakens the axioms such that it is allowed to have different representations for the same abstract value. The resulting specification may be refined by free data types. Satisfiability ensures the correctness of this step.

The main reason for prefering the simple theory interpretation rather than conservative extension is that executability may be achieved in a completely formal development without fixing a certain target programming language by encoding it into the logic. The best solution is to use $\sim$ instead of $=$ in the requirement specifications, since it makes the use of theory interpretations superfluous and allows us to use the simple model inclusion basis in the deductive software development process.

4.6 Other Approaches for Behavioural Implementations

Other approaches to behavioural implementation in the literature can show that a certain programming language fulfils the specified observability. The main difference to our approach is that we express observability in the logic of data types and we do not need an extra treatment in the semantics. The module systems of functional programming languages (like ML), together with the predicate `is_Cobs` suffice to prove the behavioural correctness of implementations.

This section gives a short overview over existing approaches to behavioural implementation, especially under the aspect of the formalization of the quotient step with observers. More general overviews are in [Nip87][Section 5] or in [ONS96].

There are three classes of approaches, which support the verification of the quotient step. The first is to prove by induction over possible contexts that a program, based on a data type, cannot observe the difference between different representations of the data type. The second is to impose restrictions on the programming language in use. Members of the last approaches apply formalize the notions of programs or of implementations in the logic. This allows us to use the logical calculus to derive the correctness of the behavioural implementation.

4.6.1 Context Induction over Programs

Induction is a well known and powerful proof principle. In [Hen91] it was applied to reason over the structure of contexts ("terms with a hole"). This allows us to show behavioural properties by the so called **context induction**.

In [RH9x] a structured specification language is presented, which allows us to specify observability. The language supplies constructs for specifying quotients by identifying all elements that are behaviourally equal (could not be distinguished from observers). The paper presents a calculus for proving behavioural first order properties over structured specifications. This calculus is well suited for proofs over structured specifications, but for the proof of behavioural properties of basic specifications one of the infinitary many rules for context induction has to be applied.

In the examples the authors refer to [Hen91] for the proofs of basic behavioural properties. The proofs in this paper are nested inductions over all contexts. No general calculus is presented, but the contexts are formalized by algebraically specifying a simple programming language PROG with variables, assignment and sequential composition (almost as the approaches in Section 4.6.3).

Another way for proving behavioural properties, as the correctness of behavioural implementation, are the so called **behavioural theories** [MB9x]. These theories interpret the equality by a behavioural equality.

In [BMPW86] it was shown that the algebraic implementations for restrictions and quotients, called ENRICH-FORGET-IDENTIFY implementations, preserve the correctness of a certain class of programs. The programming language for these programs allows us to use variables, while loops, conditional statements and assignments.

The proof of behavioural implementations is reduced from all program contexts to the base case of empty contexts by showing that the implementations are *compatible* with the programs. Compatibility is captured by the notion of **homomorphism**. Of course the induction proof of this compatibility goes over the structure of the programs (contexts). So the main result of [BMPW86] is: algebraic implementations preserve correctness since they are homomorphisms with respect to the programs. Context induction is done once for the class of all those programming languages and it is not necessary to use context induction in the development process.

This result can also be seen the other way round: If the programming language is compatible with algebraic implementation, then it may be used as correct implementation language for the specifications. More approaches, restricting the programming languages are sketched in the following section.

4.6.2 Restricting the Programming Language

Schoetts idea in [Sch90, Sch85] is to define a strong *correspondence*[19] relation, which allows us to relate one abstract element to multiple representations and to require this relation to be, compatible like a homomorphism, with all functions of the type. Behavioural equivalence is defined over all terms with visible input and result. The distinction between visible and invisible sorts allows us to hide the implemented sort. It was shown that the correspondence relation ensures behavioural equivalences of visible terms.

In [Sch83] the requirements for behavioural correctness are carried over to the target programming language. It needs a good module system, which ensures that the hidden sort remains hidden and that it may only be accessed by the defined (and compatible) functions. This property is called HEP (homomorphic extension property).

Independently the notion of *simulations* between the semantics of data types was developed in [Nip86, Nip87]. It was shown that these relations can also be applied for nondeterministic data types.

The *logical relations* in [Mit86] are a generalization from relations to the second order λ-calculus. In [Meh95] they are extended to higher order logic.

4.6.3 Encoding the Programming Language into the Logic

The following approaches formalize the quotient step by encoding a programming language into the logic. This allows us to apply the used logical calculus to derive the correctness of behavioural implementations. All these logical approaches use theory interpretations to relate different theories in the logic.

In [Wan82] functions and procedures are added to the first order *dynamic logic* of Pratt [Pra76]. Having the programming language as part of the logic allows us to show the correctness of the implementation within the logic. Wand defines a theory interpretation in his logic, which allows us to replace $=$ by an equivalence relation, which is substitutive for all functions and procedures. This condition ensures the behavioural implementation. A satisfiability theorem for this theory interpretation is proved.

In [MVS85] it was argued that a similar *logical approach*, based on conservative extension and theory interpretation, is appropriate for program development since implementations

[19] With weak correspondence relation Schoett ensures behavioural inclusion, a concept, which allows us to express restrictions and partial implementations.

(theory interpretations) compose. No programming language is fixed and, probably therefore, the treatment of substitutivity is omitted.

In [Gan83] an imperative language with variables, assignments, while loops and procedures is used. Ganzinger assigns *denotational semantics* to it by defining an algebra for it. Within this algebra behavioural equivalence may be formulated easily. This behavioural equivalence is the basis for modular languages.

4.6.4 Comparison of the Approaches

Imposing the behavioural correctness of implementations as requirements on the target programming language (as in Section 4.6.2) means to delay the correctness proof until the target programming language is chosen. Furthermore, there is no calculus for verifying that a programming language fulfils these requirements. We prefer to be able to show the correctness of an implementation step independently from the chosen programming language and in the same development phase as the implementation. Waiting with the correctness proof until the programming language is chosen is not the goal of the implementation.

The only class of approaches, which provide a calculus for behavioural implementation, are those of Section 4.6.3. They encode (imperative) programming languages into the logic. However, encoding a programming language into the logic fixes the development to a specific language in a phase where this is not adequate, since we want to separate the implementation of data types from the choice of the target programming language. For example the implementation of sets by sequences should be provable without determinating whether C or Pascal is used as programming language

In the presented approaches observability is expressed by defining that the context consists of terms of type `in`$\rightarrow$`out`. To prove a property for all contexts requires, therefore, to reason over all possible terms. In the case of first order approaches the set of functions is fixed in the signature, but in approaches with higher order logic arbitrary functional terms may be formalized by applying higher order functions, or by λ-abstraction. Therefore, infinitely many contexts exist and we need a more fine grained notion of observability. Characterizing specific functions as observer functions by a higher order predicate `is_Cobs`$:(\alpha\rightarrow\beta) \Rightarrow$ `bool` would solve theses problems and would also integrate smoothly into the higher order logic[20]. Of course, the observability defined by types may be expressed by requiring that $\forall$`f::in`$\rightarrow$`out.is_Cobs f`.

A disadvantage of almost all approaches in the literature is that these approaches are not suited for our logic of the development of interactive systems, since except [Mit86] and [Meh95] they are all dealing with first order logic, and except [BMPW86] and [Meh95] they have no domains with continuous functions and fixed points. In [Meh95] the observability issue is not treated.

[20] Observer functions are defined as a subset of the signature in an equational setting in [vD88] to achieve modularization.

Comparing our method to those from the literature we can conclude that the general concept for proving the behavioural correctness by observability arguments is similar, but there are a lot of technical differences. Many of them come from the fact that we use higher order logic with domains. Therefore, we have a more difficult environment and we manage this with different techniques for partial observer functions, PERs and higher order quotients. One advantage of the higher order logic is that we can specify observability together with the ADT on an abstract (and program independent) level.

Working with functional languages , that have a module system like ML, has the advantage that if we export only functions that fulfil the predicate `is_Cobs`, then we will know that our programs are behaviourally correct. The correctness of this step bases, like the correctness of our code generation, on the direct correspondence between specification and program. For other programming languages the correctness proof of behavioural implementations can be carried out with one of the above approaches.

The following chapter defines a method for the implementation of ADTs including the restriction step and the quotient step. It uses the simple theory interpretation basis. Chapter 6 extends the method from the implementation of ADTs in two ways to the implementation of interactive systems.

Chapter 5

Implementation of ADTs in HOLCF

Why do we implement ADTs in HOLCF?
Why do we need HOLCF for ADTs?
Why do we use continuous abstraction and representation functions?
The answers to these questions (in reverse order) are: Continuous functions are an adequate model of computation. We generalize the implementation of ADTs to implementation of distributed and interactive systems. Because we want to compute the abstractions (for example from streams of bits to streams of bytes, or from histories to automata) and representations we need continuous functions abstraction and representation functions. Having continuous abstraction and representation functions on ADTs makes the generalization easy. Using continuous abstraction functions is also useful for simulations and prototyping. HOLCF provides *cpo* INDdomains and continuous functions and therefore we need it as foundation for our continuous abstraction and representation functions. The implementation of ADTs is an important part in software development and it was not available in HOLCF.

In the previous chapters we defined refinements in HOLCF, which fit into the deductive software development process of interactive systems. This chapter combines the methods of Section 3.3 and Section 4.4 to a method for the implementation of ADTs in HOLCF. Since we use the logic HOLCF we are able to extend our implementation of ADTs to the implementation of interactive systems in Chapter 6[1]. This results in a collection of concrete methods for the refinement of all situations in the development of interactive systems (see Section 1.2.4).

The general method for the implementation of ADTs has been studied since [Hoa72] under very different aspects (see [ONS96] for an overview). The implementation of ADTs consists of two important steps:

- the restriction step, which builds a subdomain and

[1] As mentioned in Section 1.2.4 we have two types of translations. Schematic translations, which are an application of our method, and individual translations, which are an extension of our method.

- the quotient step, which allows us to identify different elements.

Our method does not change these classical steps that proved to be useful in many years. We make these concepts more powerful by integrating them into the deductive development process of interactive systems. Since we use the logic HOLCF in this process, we use new techniques for the implementation of ADTs.

In the previous chapter we compared two different methods for the two steps of the implementation (theory interpretation and model inclusion). The results of this comparison of are: for the restriction step model inclusion with conservative extension is better, since it has simpler proof obligations, but for the quotient step simple theory interpretation is needed, since it is impossible to find a conservative extension, which implements quotients in an executable specification. A way to omit the simple theory interpretation is to specify the requirements in a form that admits implementations with multiple representations and, therefore, theory interpretations would be superfluous (see Section 4.5). However, since the requirement specifications are sometimes fixed we define our method for the most difficult case and we show possible simplifications for easier cases.

The method is visualized with a KorSo-development graph, which also shows the required proof obligations.

Our method has several parameters: the abstract sort, the concrete sort, the restriction predicate, the congruence, and the corresponding constants. Before the method (presented in Section 5.1) is applied to an example (see Section 5.3) the method is prepared for the developer in the deductive software development process by explaining how to use it (see Section 5.2). This includes hints for elegant choices of the parameters.

5.1 Method for the Implementation

This section describes a method for the implementation of ADTs in HOLCF. Since our methods are transitive, it suffices to focus on the implementation of one abstract sort by a concrete representation. This implementation may be repeated to implement more sorts[2]. The method consists of a restriction step followed by a quotient step (like the RESTRICT-IDENTIFY implementations, for example in [EKMP82]).

First, the intuition of the term *implementation* from the example in the introduction (on page 22) is formalized by the following definition. It defines the implementation of a requirement specification with an abstract ADT by a design specification containing the concrete ADT. The implementation (in general) consists of a combination of the restriction and the quotient step.

[2]The extension to mutually recursive sorts (as trees and branches) is a schematic extension, similar to those presented in Section 6.3.

Definition 5.1.1 ***Implementation of ADTs in HOLCF***

Let $A = (\tau, Con^a, Sel^a, Dis^a, Map^a, \doteq^a)$ be an abstract ADT specified in a theory $Th^a = (\Sigma^a, Ax^a), \Sigma^a = (\Omega^a, C^a)$ and let $C = (\sigma, Con^c, Sel^c, Dis^c, Map^c, \doteq^c)$ be a concrete ADT specified in $Th^c = (\Sigma^c, Ax^c), \Sigma^c = (\Omega^c, C^c)$. To cut down notations we write $\{\mathtt{c}_i^a\}$ for the set of all abstract operations[3] $Con^a \cup Sel^a \cup Dis^a \cup \{Map_\tau\}$ and $\{\mathtt{c}_i^c\}$ for the sets of *corresponding operations* ($\mathtt{c}_i^c \in T_\Sigma c$). Then the *implementation* of A by C consists of:

- a sort implementation μ, which implements the sort $\tau \in T_\Omega a$ by the corresponding sort term $\sigma \in T_\Omega c$.
- a constant implementation φ, which implements all (higher order) constants c_i^a by corresponding terms c_i^c.
- an admissible restriction predicate $p :: \sigma \Rightarrow \mathtt{bool}$, which describes the subset $\hat{\sigma}$ of the corresponding values by: $\hat{\sigma} := \{x \in \sigma \mid p(x) \vee x = \bot\}$
- a continuous equality $\doteq\_\tau :: \hat{\sigma} \to \hat{\sigma} \to \mathtt{tr}$. $\doteq\_\tau$ is the implementation of $\doteq^a$.

See Chapter 3 for the formal definition of μ and φ and for a detailed description of the method consider Sections 3.3.6 and 4.4.5. On infinite data types (without theory interpretation) we use congruences ($\sim$) instead of continuous equalities ($\doteq$).

The method performs first a restriction step RESTRICT and then a quotient step IDENTIFY (as almost all methods for the implementation of ADTs). It consists of the following steps.

1. Extend Th^c by an admissible predicate p and by preserving definitions of c_i^c. The result is in the specification `C_ext`[4].
2. Prove the admissibility of p and the invariance of p with respect to the c_i^c. These proof obligations are collected in the specification `A_inv`.
3. Introduce the subdomain $\hat{\sigma}$ and intermediate constants c_i^h for every corresponding constant c_i^c. This extends the specification `C_ext` into `C_sd`.
4. Prove that the equivalence relation $\doteq\_\tau$ is a continuous equality (by showing the substitutivity for all operations c_i^h). These proof obligations are collected in the specification `A_cong`[5].

[3] Except $\doteq_\tau$, which is treated separately and except Map^a, which may be defined conservatively (see page 52) and, therefore, it does not need to be implemented by the constant implementation.

[4] Theoretically it could be required that this step is included in `C`, but methodically it belongs to the implementation.

[5] Theory interpretation requires that $\hat{\sigma}$ has to be instantiated into the class `eq`. For that reason the class axioms of `eq` have to be proved for $\doteq\_\tau$. Theoretically the instantiation is not required to be part of the method (the arity would suffice), but methodically it belongs to the implementation since it is needed for the development towards executable specifications.

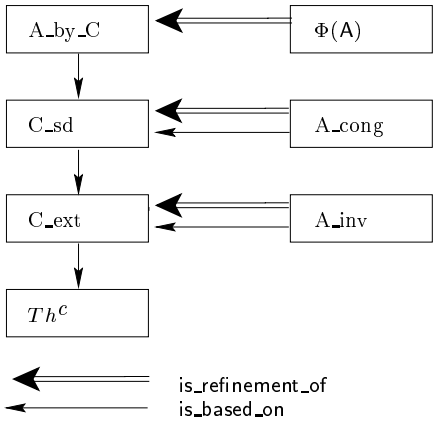

Figure 5.1: Development Graph for the Implementation of ADTs

5. Introduce the type τ as free extension from $\hat{\sigma}$ with the `domain` construct.

6. Add definitions for c_i^a, based on the abstraction and representation functions for τ (and on c_i^h with the general schemes described in Section 3.3.6). The resulting specification is called `A_by_C`.

7. Prove by theory inclusion $\Phi(\mathtt{A}) \hookrightarrow \mathtt{A\_by\_C}$, where Φ is the simple theory interpretation, defined by the sort implementation μ.

The method is depicted in the KORSO development graph in Figure 5.1.

As in Section 1.3 the notation for proof obligations is the combination of an `is_refinement_of` arrow with an `is_based_on` arrow. Note that both proof obligations can be hidden, since they are part of the main proof and the requirement of executability, but for methodical reasons it is better to have them explicitly given (see Section 5.2).

As in Section 1.3 every `is_refinement_of` edge of the graph gives a proof obligation. For that reason the proof obligations are:

1. Φ`(A)` $\hookrightarrow$ `A_by_C` (prove `A_by_C` $\vdash \Phi$`A` in HOLCF)
2. `A_cong` $\hookrightarrow$ `C_sd` (prove `C_sd` $\vdash$ `A_cong` in HOLCF)
3. `A_inv` $\hookrightarrow$ `C_ext` (prove `C_ext` $\vdash$ `A_inv` in HOLCF)

All proof obligations are theory inclusions and can be discharged with theorem provers as Isabelle.

The method is well suited for the deductive software development process, see Sections 3.2.10 and 4.4.7 for a discussion of the deductive software development bases. The only

disadvantage is that the simple theory interpretation requires the concrete type to reside in the class `eq`. This could give difficulties in the implementation of arbitrary quotients. However, as we saw in Example 4.1.1 our class `eq` is flexible enough to allow us to use quotients of streams. Therefore, we are able to give methods for all development situations in the deductive development of interactive systems in Chapter 6.

The next section explains how to use the method by describing how to fix the parameters of the implementation.

5.2 Using the Method

The description of the method in the previous section is defined precisely, but it does not describe how to fix the parameters in the implementation in an appropriate way. This section provides hints for the application of the implementation of ADTs in the deductive software development process. It explains how to fix the parameters of the implementation. They are:

- the sort implementation μ,
- the constant implementation φ,
- the restriction predicate p, and
- the continuous equality $\doteq\_\tau$.

The first step is to fix the desired sort implementation. The abstract sort τ is a sort in the specification of the abstract ADT. The concrete sort σ is a sort from the concrete ADTs. Arbitrary combinations are possible, but usually the concrete sort is executable or closer to a realization than the abstract one. In all our examples the sort implementation is fixed a priori. The selection of the other parameters is more interesting, since they depend on each other and of course on the sort implementation. The following classification of different realizations between the types helps to find the desired parameters.

5.2.1 Interface Situations

In order to apply the implementation of ADTs in the development we have to fix the parameters of the implementation method, by defining the observable equivalence relation, the restriction predicate and the corresponding constants. These parameters depend on the relation between the types of the interfaces of the components. There may be different situations. This section gives an overview over different interface situations, which are implemented by choosing the right parameters of the implementation method.

An ADT of an abstract component is implemented by an ADT of a concrete component. The situation depends on the types (in the ADTs) of the interfaces of the component: The type of the interface of the abstract component is called abstract type. The type of the concrete interface is called concrete type[6]. The following relations between the abstract type τ and the concrete type σ characterize the possible situations:

isomorphic types: $\tau \simeq \sigma$[7]. This situation may also be called "renaming". An example for this situation is the implementation of booleans by bits.

concrete subdomain: $\tau \simeq \hat{\sigma}$ where $\hat{\sigma} \preceq \sigma$[8]. This situation occurs, if not all values of the concrete type are needed to represent abstract values. An example for this situation is the implementation of sets by ordered sequences, where the ordered sequences are a subdomain of sequences.

abstract subdomain: $\hat{\tau} \simeq \sigma$ where $\hat{\tau} \preceq \tau$. This situation occurs, if not all values of the abstract type are used in a component. This allows the implementation of unbounded data types by bounded[9] ones, if the values beyond the bound are *actually not needed* (This is called "bounded implementation" in [Bre92] or partial implementation in [KA84]). An example for this situation is the implementation of natural numbers by 32 bit unsigned integers.

concrete quotient: $\tau \simeq \sigma_{|\sim}$ where $\sigma_{|\sim}$ is a quotient[10] of σ. This allows us to have "multiple representations" for the same abstract value. An example for this situation is the implementation of integers by a redundant type integer with two zeros $+0$ and -0 for the representation of the abstract value 0.

abstract quotient: $\sigma \simeq \tau_{|\sim}$. This allows us to implement different abstract elements by the same concrete one. This may be appropriate if not all information of the abstract type is *actually relevant*. This is called "indefinite representation" in [Bro93]. An example for this situation is the implementation of pairs consisting of values and a timing information by the type of values. This is only possible if the timing information is not needed for the implementation. Time sometimes may be useful in the specification, but not always in programs.

Situations that result from a combination of this situations for example a concrete quotient of a subdomain (as in the Section 5.3 of sets and sequences) are not analyzed in detail, since they can be modelled by two implementations, one for the subdomain and one for the quotient.

---

[6] Note that we focus here on the type being implemented and ignore other types of the interface of other types. These types are not affected in this implementation step.

[7] Isomorphic means that there exists functions $\mathtt{a}::\tau \Rightarrow \sigma$ and $\mathtt{r}::\sigma \Rightarrow \tau$ with $\mathtt{a(r(x))}=\mathtt{x}$ and $\mathtt{r(a(y))}=\mathtt{y}$.

[8] Subtype $\preceq$ means that there exists an admissible predicate Π with $\hat{\sigma}=\{\mathtt{x}::\sigma.\,\Pi\ \mathtt{x}\}$.

[9] A bounded type has a finite number of elements.

[10] See Definition 4.2.2 for the definition of a quotient with respect to a PER $\sim$.

The intuition of the terms "actually not needed" and "not actually relevant" in the situations of abstract subdomain and abstract quotient is that these situations may be functionally refined by specifications that do not need this irrelevant information. This will be explained in more detailed with Example 6.3.4 on page 190.

Having identified the interface situation, it will be easier to fix the other parameters of the method.

5.2.2 Parameters of the Method

In the development of interactive systems, we may arrive at different development situations (see Section 1.2.4). Many of these development situations require to implement an abstract component by a concrete one. In the previous section we described different interface situations by regarding the relations between the components interfaces. According to these situations we fix the sort implementation and the other implementation parameters. This section provides help to fix the parameters of the implementation method presented in Section 5.1.

Although our method performs the quotient step (with the PER) after the restriction step, we suggest to decide first whether multiple representations should be allowed or not since this influences the choice of the restriction predicate. If more representations are desired, the observers, which specify the congruence, have to be fixed. This may be done in a specification style, by requiring `is_Cobs f` for any observer `f` or, more constructive, by explicitly defining a continuous equality or a partial equivalence relation. The second way allows to characterize the continuous equality by theorems, whereas the first way is more abstract and requires these theorems to be proved later in the development. Some examples for different definitions of the PER are in Section 4.3.2. If multiple representations are not desired then the PER is not needed.

The other parameters that have to be fixed are the restriction predicate and the corresponding constants (the abstract constants are at least all constants in the abstract ADT, which use the implemented type and which are not conservatively defined). If we fix the corresponding constants first, then the invariance of p determines the restriction predicate p. The restriction predicate has to be *stable* (invariant), under the corresponding functions. Consider for example the case with one corresponding constant $c::\sigma\to\sigma$. Then the stability (invariance) of the restriction predicate is specified by:

$\forall$`x.p(x)` $\longrightarrow$ `p(c‘x)`

The FOCUS method for the recursive definition of predicates (see page 132) can be applied to the definition of the restriction predicate as well.

Besides the fact that it might be difficult to prove the admissibility of predicates, which such definitions, stable predicates can lead to undesired effects. Consider, for example, the

implementation of the two-valued booleans by natural numbers. We decide to represent `false` by `zero` and `true` by `one`, conjunction by multiplication, and disjunction by addition. Now we define the restriction predicate as greatest stable fixed point and perform our implementation with a trivial invariance proof obligation, since our predicate is stable (invariant). This leads (in our case) to undesired effects, since the result of `true or true` corresponds to the value two. In addition, we see from several different development situations (for example conservative introduction of a type with functions on page 41) that it is a natural choice to define the restriction predicate and to choose the corresponding predicates such that they are stable, i.e they preserve the invariance.

The constant implementation φ is fixed by giving corresponding constants c_i^c for all c_i^a. c_i^a are the constants in the abstract ADT `A`, which have the type τ in their types and are not conservatively introduced. On page 104 we described a method that allows us to use arbitrary corresponding functions by using a continuous function `isR` in the definition of a continuous abstraction function `c`τ`abs`.

`c`τ`abs x = If isR‘x then `τ`abs x else ⊥ fi`

This defines a partial abstraction function. With this definition we may introduce partial operations on τ. However, for total operations we have to ensure that `isR` is always true for the results of the corresponding functions. This is exactly the invariance requirement. Having fixed the restriction predicate the invariance helps to choose the corresponding constants. In the next section we see on an example how efficiency of the operations may help us to find the interface situation and how the invariance helps to define corresponding constants.

The essence of this section is: the choice of the PER and the restriction predicate are very important. For the special case of a stable predicate we have no invariance requirements in the method and for some PERs we get reflexivity, transitivity and symmetry for free. In general, for fixed restriction predicates the invariance proof obligations are a helpful tool for finding the right corresponding constants.

5.3 Example: Sets by Sequences

This section uses the method of Section 5.1 as explained in the previous section for the implementation of sets by sequences.

The first decision to implement sets by sequences is already fixed. The next parameter is the continuous equality. We decide that sequences with the same elements should represent the same set. This means that `≐_Set‘s‘t = same‘s‘t`.

The next decision is the restriction. Due to the efficiency of `has` we restrict the representations of sets to those sequences without duplicates by defining an operation, which tests

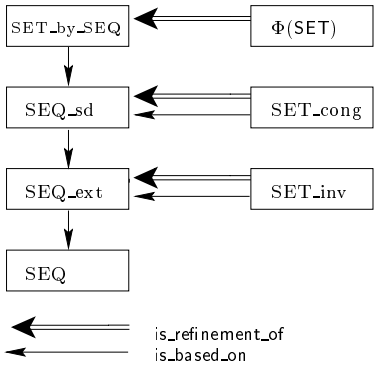

Figure 5.2: Development Graph of the Example

whether a sequence contains duplicates. Now the corresponding constants for `empty`, `add` and `has` are chosen. The intention is to take `eseq`, `cons` and `isin`, but `cons` does not preserve `no_dup` ($\lceil$`no_dup‘q`$\rceil \not\Longrightarrow \lceil$`no_dup‘(cons‘x‘q)`$\rceil$) and for that reason `add‘x‘q` has to be implemented by `If isin‘x‘q then q else cons‘x‘q fi` . This definition makes `no_dup` invariant for the corresponding constants.

Having fixed the parameters of the method the implementation can be carried out. The method, supplied with these parameters, generates concrete instances of the schemes in Figure 5.1. The user of the method has to fill the parameters into these schemes. For the example of implementing sets by sequences the generated specifications can be depicted in the development graph in Figure 5.2.

The contents (with the concrete definitions) of the specifications are:

```
SEQ = EQ +
domain a Seq = eseq | cons (first::α) (rest::α Seq)
ops curried strict total
    isin    :: α::eq → α Seq → tr
    contains:: α::eq Seq → α Seq → tr
    same    :: α::eq Seq → α Seq → tr
axioms
defvars x y :: α::eq
        s t :: α::eq Seq
in
    isin1      ⌊isin‘x‘eseq⌋
    isin2      isin‘x‘(cons‘y‘s) = (x≐y) OR (isin‘x‘s)
    contains1  ⌈contains‘eseq‘s⌉
    contains2  contains‘(cons‘x‘s)‘t = (isin‘x‘t) AND (contains‘s‘t)
    same       same‘s‘t = (contains‘s‘t) AND (contains‘t‘s)
end
```

The extension SEQ_ext of SEQ introduces corresponding constants for the constants in SET. The definitions are supplied by the user of the method.

```
SEQ_ext = Seq +
ops curried strict total
                (* corresponding constants *)
    empty_Seq   :: α::eq Seq
    add_Seq     :: α::eq → α Seq → α Seq
    has_Seq     :: α::eq → α Seq → tr
                (* continuous restriction predicate *)
    no_dup      :: α::eq Seq → tr
    p           :: α::eq Seq ⇒ bool
defs
empty_Seq_def   empty_Seq ≡ eseq
has_Seq_def     has_Seq ≡ isin
add_Seq_def     add_Seq ≡ Λx q.If isin‘x‘q then q else cons‘x‘q fi
no_dup1         ⌈no_dup‘eseq⌉
no_dup2         no_dup‘cons‘x‘q=(neg‘(isin‘x‘q)) AND (no_dup‘q)
p_def           p ≡ λx.⌈no_dup‘x⌉
end
```

The proof obligations for the subdomain construction are in:

```
SET_inv = SEQ_ext +
axioms
defvars x :: α::eq
        s :: α::eq Seq
in
admissible      adm p
inv_empty       ⌈no_dup‘empty_Seq⌉
inv_add         ⌈no_dup‘s⌉ ⟹ ⌈no_dup‘(add_Seq‘x‘s)⌉
end
```

The admissibility is trivial since the predicate is defined by a continuous restriction predicate, the operations are constructed such that they are invariant. This allows us to introduce the subdomain:

```
SEQ_sd = SEQ_ext +
instance Seq::(eq)adm (admissible)
types α Seq_sd = α Seq subdom
ops curried
               (* intermediate operations on the subdomain *)
```

```
       empty_     :: α::eq Seq_sd
       has_       :: α::eq → α Seq_sd → tr
       add_       :: α::eq → α Seq_sd → α Seq_sd
defs           (* introduces with general schemes for abs & rep *)
       empty_def empty_≡abs_sd empty_Seq
       has_def    has_≡Λx s.has_Seq`x`(rep_sd s)
       add_def    add_≡Λx s.abs_sd(add_Seq`x`(rep_sd x))
end
```

To derive the applied rules `has_`x`s=has_Seq`x`(rep_sd s)` we need to prove that `λx s.has_Seq`x`(rep_sd s)` is continuous. This can be proved with `inv2cont` (see page 104), since the function is preserving.

Now we define the continuous equality (in `SEQ_sd`) by:

```
defs
       eq_tau     x≐≐y ≡ same`(rep_sd x)`(rep_sd y)
       eq_per     (op ∼∼) ≡ λx y::α::eq.⌈x≐≐y⌉
```

The proof obligations for continuous equality are the characteristic axioms of the class `eq` and the observability for the corresponding intermediate functions:

```
SET_cong1 = SEQ_sd +
rules
        eq_refl_def  x≠⊥⟹⌈x≐≐x⌉
        eq_sym2      ⌈x≐≐y⌉⟶⌈y≐≐x⌉
        eq_trans2    ⌈x≐≐y⌉∧⌈y≐≐z⌉⟶⌈x≐≐z⌉
        eq_strict1   ⊥ ≐≐ x = ⊥
        eq_strict2   x ≐≐ ⊥ = ⊥
        eq_total     ⟦x≠⊥;y≠⊥⟧⟹x≐≐y≠⊥
        eq_per       x∼∼y=⌈x≐≐y⌉
end
SET_cong2 = SET_cong1 +
instance Seq_sd::(eq)eq  (eq_refl_def,eq_sym2,eq_trans2,
                          eq_strict1,eq_strict2,eq_total,eq_per)
rules
        obs1    is_Cobs has_
        obs2    is_Cobs add_
```

The use of the axiomatic type classes requires to split `SET_cong` into two parts, since for the specification of the observability we have to instantiate the PER before `is_Cobs` is available.

```
SET_by_SEQ = SEQ_sd +
domain α FSet = Fabs(Frep::α Seq_sd)
ops curried strict total
                (* abstract constants *)
        empty   :: α::eq FSet
        add     :: α::eq → α FSet → α FSet
        has     :: α::eq → α FSet → tr
defs    (* schematic definitions *)
        empty_def       empty ≡ Fabs‘empty_Seq
        has_def         has ≡ Λx s.has_‘x‘(Frep‘s)
        add_def         add ≡ Λx s.Fabs‘(add_‘x‘(Frep‘s))
end
```

ΦSET contains the main proof obligation of the implementation of sets by sequences. It results from the application from Φ to SET.

```
ΦSET = EQ +
types   FSet 1
arities FSet :: (eq)eq
ops curried strict total
        empty   :: α::eq FSet
        add     :: α::eq → α FSet → α FSet
        has     :: α::eq → α FSet → tr
generated finite FSet by empty | add
axioms
defvars x y s in
has1    ⌊has‘ x‘ empty⌋
has2    ⌈x≐y⌉⟶⌈has‘ x‘ (add‘ y‘ s)⌉
has3    ⌊x≐y⌋⟶has‘ x‘ (add‘ y‘ s) = has‘x‘s
quot1   add‘x ∼ add‘x oo add‘x
quot2   add‘x oo add‘y ∼ add‘y oo add‘x
end
```

Since the implementing specification SET_by_SEQ is executable it can be translated directly into the corresponding Gofer program. The main refinement proof, which proves all axioms of the specification ΦSET is similar to the proof in [Slo95].

In this chapter we presented an approach for the classical implementation of ADT with a restriction and a quotient step. Our approach uses HOLCF as specification logic and since we specify interactive systems in HOLCF, we are able to apply the methods for the implementation of ADT to the implementation of distributed systems in the next chapter.

Chapter 6

Implementation of Interactive Systems in FOCUS

This chapter contains concrete methods for the implementation of interactive systems in FOCUS . FOCUS is an approach to the specification and development of distributive and interactive systems. It contains general methods for the implementation of interactive systems. We specialize these methods by describing a lot of concrete methods for concrete situations in the development process. Furthermore we give tool support for the implementation of interactive systems in HOLCF by showing how to prove the resulting proof obligations with the Isabelle proof system.

A central idea of the implementation of interactive systems using the concept of streams [Kah74, BDD+92], is to define abstraction and representation functions on streams (see [Bro93]) and to require that they are inverse[1], as done in the conservative introduction of types (see Example 2.1.2 on page 41). Having continuous abstraction and representation functions allows us to model the computation between different abstraction levels. Having executable abstraction and representation functions allows us to simulate our interactive systems on a high abstraction level. This is especially useful for prototyping.

Our methods extend the implementation of ADTs to interactive systems in two different ways:

- schematic implementations, and
- individual implementations.

Furthermore there are direct applications of the implementation of ADTs to the implementation of state-based interactive systems.

[1] `abs(rep(a))=a` and `p(c)`$\Longrightarrow$`rep(abs(c))=c`

Schematic implementations schematically define `abs` and `rep` on streams, such that the implementation of ADTs is applied to every single message of channels of interactive systems. Abstraction and representation functions of schematic implementations are inverse by construction. If the ADT is implemented with continuous and executable functions schematic implementations will also be continuous and executable.

Individual implementations are more flexible. They allow us to define arbitrary abstraction and representation functions between the types. However, individual implementations require to proof the inverse theorems explicitly. We provide methods that support these proofs.

This chapter is structured as follows: Section 6.1 introduces basic concepts of the FOCUS approach to the specification of interactive systems. The following Section 6.2 contains direct applications of the implementation of ADTs to interactive systems, and especially examples for the introduction and elimination of states in the specification. Section 6.3 defines methods for the schematic implementation situations. We show compositionality of theory interpretation and derive an elegant compositionality result for downward simulation development. Section 6.4 presents concrete methods and examples for individual implementations.

6.1 FOCUS Notations

This section presents FOCUS, an approach for the specification of distributed and interactive systems[2] and introduces notations for the presentation of our methods in Sections 6.2 and 6.3.

There are many different ways to specify distributed and interactive systems [LT89, Lam94, BDD+92, BS, Mil89, LT88, CM88, UK95, Bac90, Gur92]. A good overview can be found in [BMS96a], where different methods are applied to one specification example, including a comparison [BMS96b]. In some approaches concrete methods and tool support are not considered to be important, since the approaches concentrate on the specification of systems. Other approaches are operational and do not need to be implemented [Gur92], but they do not give us the possibility to describe distributed systems in an abstract property oriented style. We decided to use FOCUS, because it includes denotational semantics for abstract requirement specifications and concepts for the refinement from abstract to concrete specifications. Since these concepts are very general, we define specific methods for different situations in the development of interactive systems (see Section 1.2.4)

In FOCUS [BDD+92, Bro93, SS95] interactive systems consist of components interacting over channels with other components and their environment. All channels (names and types) of a component are called its *interface*. FOCUS provides formal description techniques to describe interactive systems on different levels of abstraction. Furthermore,

[2] We will use these terms synonymously.

FOCUS supports the formal development of interactive systems and allows us to prove the correctness of a concrete system with respect to an abstract specification. In contrast to many other formal description techniques FOCUS allows us to functionally describe recursive components (or systems) with *feedback* channels. In FOCUS they get semantics by a fixed point operator which denotes the least fixed point of a recursive continuous function. HOLCF is an appropriate logic for FOCUS, since it supports continuous functions and fixed point construction by the use of domains with *cpo* structures.

A main aspect in FOCUS is compositionality. Compositionality ensures that if the components of a system are developed correctly, and the components are composed correctly, then the composed system will be a correct development of the original system. This allows us to develop the components independently and for that reason it is the basis for a correct development of large systems with many components.

FOCUS provides techniques for the specification and the development of interactive, distributed systems. It uses stream processing functions to specify systems and components. This section repeats definitions for stream processing functions and forms of composition and defines a notation for preserving stream processing functions.

6.1.1 Stream Processing Functions

Stream processing functions model interactive systems and components by describing their behaviour with streams of (input and output) values. First, we define streams:

Definition 6.1.1 ***Stream***

A `stream` is a (in general infinite) sequence of messages, specified in HOLCF by:

```
Stream = HOLCF +
domain α stream = && (ft::α) (lazy rt::α stream)
end
```

Streams are partially ordered by the prefix ordering and are members of the class `pcpo`, provided the messages are of the class `pcpo`.

Every stream corresponds to the history of a communication channel of the system. The domain `stream` is polymorphic. This means that for every message type τ (of class `pcpo`) τ `stream` denotes a channel for messages of type τ. This definition with the `domain` construct defines the selectors `ft` and `rt` and implicitly it also defines the discriminator `is_&&`. Therefore streams are ADTs.

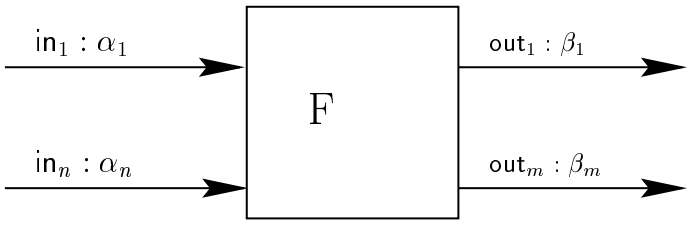

Figure 6.1: Stream Processing Function

With streams we define the notion of stream processing functions by:

Definition 6.1.2 ***Stream Processing Function***

Let α_1 `stream`, ..., α_n `stream` and β_1 `stream`, ..., β_m `stream` be streams, then every continuous function `f` of type α_1 `stream`, ..., α_n `stream` $\rightarrow$ β_1 `stream`, ..., β_m `stream` is a *stream processing function*. We abbreviate this type of stream processing functions by writing `f` :: $[\alpha_i^n \rightarrow \beta_j^m]$.

For the specification of stream processing functions `f` we use predicates: `F` ::$[\alpha_i^n \rightarrow \beta_j^m] \Rightarrow$`bool`. We write `F.f` to denote that a function `f` fulfils its specification `F`.

We can represent stream processing functions graphically by system diagrams. Furthermore these diagrams allow us to assign names to the communication channels. The system diagram in Figure 6.1 shows the signature and the channel names of the functions specified by `F`. Note that for the analysis of refinements we do not need the names of the channels `in`$_i$ and `out`$_j$, and therefore, will only write the types to the channels.

In the specification of `F` the operations available on the input and output types α_i and β_j may be used. ADTs provide these operations in a well structured way, together with induction rules to prove properties. Therefore, the operations on α_i and β_j should be specified with ADTs.

Without restricting the generality of our method, but to simplify the following specifications we focus on stream processing functions with one input and one output channel.

Definition 6.1.3 ***Preserving Stream Processing Function***

A stream processing function `f`::$[\alpha \rightarrow \alpha]$ specified in `F` (with `F.f`) is called Π-preserving with respect to a predicate Π::$\alpha \Rightarrow$`bool`, if

- $\forall$`x`.Π `x` $\Longrightarrow$ Π `(f‘x)`

With Π`(F)` we specify Π-preserving functions which fulfil `F`.

- Π(F).f = F.f $\wedge$ $\forall$x.Π x $\Longrightarrow$ Π (f‘x)

If Π is clear from the context we simply call the function f *preserving*.

The predicate Π is *invariant* for Π-preserving functions. We chose this application notation on specifications, since invariance in general is a predicate, depending on the type of the functions and the restriction predicate (see Definition 3.2.5). For example the invariance of $\Pi :: \alpha \Rightarrow$bool with respect to a function g of type $[\beta \rightarrow \alpha]$ is $\forall$x. Π (g‘x). Invariance is needed in the restriction step of the implementation (see Chapter 3) and will be the basis for improving the compositionality result (see Theorem 6.3.1).

6.1.2 Forms of Composition

A distributed system is composed by its components. Our forms of composition are sequential, parallel, and feedback composition [Kah74, BDD+92].

Definition 6.1.4 ***Sequential Composition***

The sequential composition of two stream processing functions f::$[\alpha \rightarrow \beta]$ and g::$[\beta \rightarrow \gamma]$ is defined by:

- (f;g)‘x = g‘(f‘x)

The notation is lifted to specifications by:

- F;G.h = $\exists$f g. F.f $\wedge$ G.g $\wedge$ h=f;g

Definition 6.1.5 ***Parallel Composition***

The parallel composition of two stream processing functions f::$[\alpha \rightarrow \beta]$ and g::$[\gamma \rightarrow \delta]$ is defined by:

- (f||g)‘(x,y) = (f‘x,g‘y)

The notation is lifted to specifications by:

- F||G.h = $\exists$f g. F.f $\wedge$ G.g $\wedge$ h=f||g

The feedback composition allows a component to read its own output. Semantically this is modelled by a fixed point construction.

Definition 6.1.6 ***Feedback Composition***

The feedback composition of a stream processing function $\mathtt{f::[(\alpha,\beta)\to\beta]}$ is defined by:

- $\mathtt{(\mu f)`x = fix`\Lambda y.f(x`y)}$

The notation is lifted to specifications by:

- $\mathtt{\mu F.h = \exists f.\ F.f \wedge h{=}\mu f}$

The feedback composition uses the fixed point operator of HOLCF. In order to have an operational meaning it is required that `fix` denotes the least fixed point. HOLCF allows us to use the weaker order $\sqsubseteq$ on INDdomains which belong to class `pcpo`. `fix` denotes the least fixed point with respect to $\sqsubseteq$. So feedback composition of distributed systems is the main reason for using *cpo* structured domains and continuous functions in the specification of interactive systems.

In the development of large systems it is important that the refinement relation allows us to develop the systems in separate parts. We introduced the notion of a deductive software development basis and defined modularity for the refinement relation $\models\!\!\approx\!\!>$ in Section 1.2.2. A modular refinement relation allows us to develop the system in arbitrary parts. As we have seen in Section 4.4.7 the theory interpretation in our method for the implementation is not modular. This is due to the very general definition of parts in the Definition 1.4.2 of modularity.

Now we define compositionality. A compositional refinement relation $\models\!\!\approx\!\!>$ supports a separate development of specific parts of the system. The definition uses the components of the system as parts. So compositionality allows us to structure the development of the system in the same way as the system is structured into components.

Definition 6.1.7 ***Compositionality***

Let ($\mathcal{L}$, M, $\models\!\!\approx\!\!>$, $\mid\!\!\sim$) be a deductive software development basis. Then $\models\!\!\approx\!\!>$ is *compositional* (with respect to ";","||", and "μ"), if for specifications $\mathtt{P}, \mathtt{P}_1, \mathtt{P}_2, \widehat{\mathtt{P}}, \widehat{\mathtt{P}}_1$, and $\widehat{\mathtt{P}}_2$ the following holds:

- $\mathtt{P}_i \models\!\!\approx\!\!> \widehat{\mathtt{P}}_i$ for $i = 1, 2$ implies $\mathtt{P}_1\,;\mathtt{P}_2 \models\!\!\approx\!\!> \widehat{\mathtt{P}}_1\,;\widehat{\mathtt{P}}_2$
- $\mathtt{P}_i \models\!\!\approx\!\!> \widehat{\mathtt{P}}_i$ for $i = 1, 2$ implies $\mathtt{P}_1||\mathtt{P}_2 \models\!\!\approx\!\!> \widehat{\mathtt{P}}_1||\widehat{\mathtt{P}}_2$
- $\mathtt{P} \models\!\!\approx\!\!> \widehat{\mathtt{P}}$ implies $\mu\mathtt{P} \models\!\!\approx\!\!> \mu\widehat{\mathtt{P}}$

Compositionality is a useful weakening of the strong requirement of modularity. Every modular refinement relation is compositional.

With these notations from FOCUS we will now define concrete methods for every situation in the development of interactive systems from Section 1.2.4.

6.2 Refinements for Interactive Systems

We have two refinement techniques for interactive systems. The well known functional refinement and the implementation of ADTs. By combining these techniques we define refinements for different development situations. Our methods for the implementation of interactive systems show how to use these refinements. Since our methods use the implementation of ADTs, we call our methods *implementation methods for interactive systems.*

In the development of interactive systems there are the following different situations (which have already been introduced in Section 1.2.4).

- behavioural development,
- ⋆ communication channel development,
- ⋆ restricted communication channel development,
- ⋆ interface simulation,
- structural development,
- state development,
- state elimination, and
- ⋆ dialog development.

In the next sections we define concrete refinements which allow us to prove the correctness of these development steps. These development situations are similar to those in [Bro93]. Some situations allow schematic and individual translations (marked with a ⋆). In this section we present methods for the other situations (marked with a •). Schematic implementations are treated in Section 6.3, individual translations are described in Section 6.4.

6.2.1 Behavioural Refinement

Behavioural refinement is a very general technique for the development of interactive systems. Many other methods are based on behavioural refinement.

The idea of behavioural refinement is to refine a component by another one which is more concrete and satisfies the specification of the more abstract one. The concrete function "behaves" like the abstract one.

Definition 6.2.1 ***Behavioural Refinement***

Let $(\mathcal{L}, M, \models\!\!\approx\!\!\gg, \vdash\!\!\rightsquigarrow)$ be a model inclusion basis. Then a stream processing function `c::[α→β]` specified by `C` (with `C.c`) is a behavioural refinement of a stream processing function `a::[α→β]` specified in `A` (with `A.a`), if

- `A` $\models\!\!\approx\!\!\gg$ `C` by deriving `A` $\vdash$`C` in HOLCF.

Since behavioural refinement is based on functional refinement (in the model inclusion basis), we will sometimes call it functional refinement.

The method for behavioural development only consists of one step:

1. Prove `A` $\vdash\!\!\rightsquigarrow$ `C` by deriving `C` $\vdash$`A` in HOLCF.

Since many other refinements are reduced to behavioural refinement, and since there are many examples for theses refinements, we do not give an example for behavioural refinement here.

6.2.2 Structural Refinement

Structural refinement allows us to define the structure of a black box specification by defining its architecture. The architecture of a system is a network of components or systems. In networks it is allowed to connect components with channels and to compose components with the composition operators: ";","||", and "μ".

To define structural refinement formally we would need a formal definition of networks. ANDL, the Agent Network Description Language [SS95] of FOCUS can be used to describe arbitrary networks. ANDL has a semantic translation into HOLCF, and a syntax to describe systems, such that we can use the resulting networks in our specifications. It is not our task to repeat the definition of ANDL here. It is enough to define a notation for arbitrary networks, specified in ANDL. We write `andl(A,B,...,Z)` to denote an arbitrary network of the components `A,B,...,Z`.

With this we can simply define structural refinement by:

Definition 6.2.2 ***Structural Refinement***

Let $(\mathcal{L}, M, \models\!\!\approx\!\!\gg, \vdash\!\!\rightsquigarrow)$ be a model inclusion basis. Then the network `andl(A,B,...,Z)::[α→β]` specified in `ANDL` (with `ANDL.andl(A,B,...,Z)`) is a *structural refinement* of a stream processing function `f::[α→β]`, specified in `F` (with `F.f`), if

- `F` $\models\!\!\approx\!\!\gg$ $\widehat{\texttt{F}}$, where $\widehat{\texttt{F}}$ extends `ANDL` by the equation `f=andl(A,B,...,Z)`.

Note that structural development is only possible, if the interfaces are identical. However, ANDL allows us to embed components into components with a larger interface by describing the embedding as a network.

The method for structural refinement consists of embedding and behavioural refinement.

1. Extend the concrete specification `F` by the equation defining the network into the specification $\widehat{\texttt{F}}$.
2. Prove $\texttt{F} \ \mid\!\!\rightsquigarrow \ \widehat{\texttt{F}}$ by deriving $\widehat{\texttt{F}} \vdash \texttt{F}$ in HOLCF.

Since the development step is a simple behavioural refinement, we do not give an example here.

Behavioural refinement and structural refinement can be treated with a simple model inclusion basis (see Definition 2.2.2). They are functional refinements. The following implementation techniques use the implementation of ADTs and require a deductive software development basis which allows us to express the implementation of ADT as a refinement (for example the simple theory interpretation basis of page 141).

6.2.3 State Refinement

This section introduces state-based specifications of interactive systems and shows how they can be developed with the method for the implementation of ADTs.

In the development of interactive systems, states are a comfortable way to describe systems. The concept is to characterize stream processing functions with an initial state and describe the behaviour for every state, together with the next state.

Now we define a simple notion of state-based specifications in FOCUS. See [Spi94, BS] for a more general treatment of states in FOCUS.

Definition 6.2.3 ***State-based Specification***

Let $\texttt{f}::[\alpha\rightarrow\beta]$ be a stream processing function. Let `T` be an ADT containing the specification of states of type τ. Then a state-based specification `F` of `f` with τ consists of

- an initial state $\texttt{init}::\tau$, and
- equations of the form $\texttt{f(s}_i\texttt{,x \&\& xs)=F}_i \texttt{ \&\& f(t}_i\texttt{, xs)}$, where
  - $\texttt{s}_i$ are different states in τ,
  - `xs` is a stream of messages,
  - $\texttt{F}_i$ are outputs of the function `f` for a single message $\texttt{x}::\alpha$, and

– $\mathtt{t}_i$ are the successor states in τ.

The semantics of such a specification is the set of all functions `f`, that start with `init` and continue as described with the state-based specification. We write $\mathtt{f}::\tau\rightarrow[\alpha\rightarrow\beta]$ to denote that `f` is specified with states of type τ and we write `F.f` to denote that `f` fulfils a state-based specification `F`.

An example for such a state-based specification is on page 171. This simple specification of state-based functions may easily be generalized to functions processing more than one message in one equation.

State refinement will be necessary, if an additional state is required in the specification, or in the more difficult case, if one state has to be eliminated from the set of all states.

In the development of interactive systems the situation state development is characterized by a change of the type τ in the state-based specification. And we define a refinement relation on state-based specifications.

Definition 6.2.4 ***State Refinement***

Let ($\mathcal{L}$, M, $\mid\!\approx\!\gg$, $\mid\!\rightarrow$) be a deductive software development basis, supporting the implementation of ADTs and let `A` and `C` be state based specifications of the stream processing function $\mathtt{c}::\sigma\rightarrow[\alpha\rightarrow\beta]$ specified by `C` (with `C.c`) and $\mathtt{a}::\tau\rightarrow[\alpha\rightarrow\beta]$ specified in `A` (with `A.a`). Then the stream processing function with `C.c` are state refinement of the functions `A.a`, if

- `A` $\mid\!\approx\!\gg$ `C`

In this definition we use the fact that we defined a deductive software basis that allows the implementation of ADTs (see Definition 4.4.5). Of course state-based specifications may also be refined by functional refinement (if neither the state space, nor the interface is changed).

For a state development we have two methods: one for the removement of a state, and another for the introduction of a new state. Note that these methods only change the states in the specification of distributed systems. The elimination of states from the specification of distributed systems is treaded in Section 6.2.4.

The method for the removement of states from an abstract specification `S` with state space τ and a state-based function `f` is:

1. Remove the state transitions from the requirement specification, by defining a specification `S1` that behaviourally refines the abstract specification.
2. Prove `S` $\mid\!\rightarrow$ `S1` for the state-less function.

3. Add the definition of a PER $\sim\sim$ on τ, which is true for all used states into the specification `S2`.

4. Prove `instance` τ`::per` $\hookrightarrow$ `S2` or `instance` τ`::eq` $\hookrightarrow$ `S2` (that $\sim\sim$ is symmetric and transitive).

5. Add the definition of the new state space τ`'` with `types` τ`'` $=$ τ `quot` to `S2`,

6. introduce the τ'_i states as equivalence classes of all used states τ_i from τ, and

7. define a new state based function `f'` on τ`'` as a lifting from the old state-based function `f`. The result is within the specification `S3`.

8. Then all axioms of `S1` with τ`'` instead of τ, τ'_i instead of τ_i, and `f'` instead of `f` automatically hold in `S3`.

`S3` is the desired specification with the reduced state-space. The development continues with the development of `S3`.

This method can also be applied to a bisimulation equivalence (see [Mil83]) between state-based specifications which base on completely different states. The bisimulation equivalence of the states will help to structure the proof in step 2. If we have a stream processing function `f` with a state-based specification $\sigma\rightarrow[\alpha\rightarrow\beta]$ and another state-based specification $\tau\rightarrow[\alpha\rightarrow\beta]$, then we will need abstraction and representation functions `abs`$::\sigma\rightarrow\tau$ and `rep`$:\tau\rightarrow\sigma$ with `abs‘(rep‘x)=x` and `rep‘(abs‘y)=y` for all reachable states `x`$::\tau$ and `y`$::\sigma$ to relate the equivalent states.

In the following example we show the simple case where τ is a subdomain of σ. Therefore the bisimulation proof will be very easy (since `abs` and `rep` are the identity on the states). To illustrate state-based specifications and state refinement we look again at Example 4.1.1 on page 118.

Example 6.2.1 ***Removing a State from a Buffer***

Our specification bases on the following ADT `T`, containing the specification of states:

```
T = Dnat +
domain TState = Tempty | strange | Tstate(dnat)
end
```

The state `strange` is used to model the effect when an empty buffer gets a request. The following specifications use the ADT `Message` from page 118:

```
TBUF =  Stream + Messages + T +
ops curried
        Buffer  :: message stream → answer stream
```

```
        TBuffer :: TState → message stream → answer stream
rules
Buffer_def Buffer ≡ TBuffer‘empty (* init = empty *)
TBuffer1   TBuffer‘Tempty‘(data‘n&&s)=stored&&TBuffer‘(Tstate‘n)‘s
TBuffer2   TBuffer‘Tempty‘(req&&s)=error&&TBuffer‘strange‘s
TBuffer3   TBuffer‘(Tstate‘n)‘(req&&s)=value‘n&&(TBuffer‘Tempty‘s)
TBuffer4   TBuffer‘(Tstate‘n)‘(data‘m&&s)=error&&TBuffer‘(Tstate‘n)‘s
TBuffer5   TBuffer‘strange‘(data‘m&&s)=stored&&TBuffer‘(Tstate‘m)‘s
TBuffer6   TBuffer‘strange‘(req&&s)=error&&TBuffer‘strange‘s
end
```

Because the state `strange` is redundant in our specification we want to eliminate it from our specification. The first step is to remove the translations from (and into) the state `strange`. Therefore we specify `TBuffer` without it and we have to show that this specification is a behavioural refinement of the previous one.

```
TBUF2 =  Stream + Messages + T +
ops curried
        Buffer  :: message stream → answer stream
        TBuffer :: TState → message stream → answer stream
Buffer_def Buffer ≡ TBuffer‘empty
TBuffer1   TBuffer‘Tempty‘(data‘n&&s)=stored&&TBuffer‘(Tstate‘n)‘s
TBuffer2   TBuffer‘Tempty‘(req&&s)=error&&TBuffer‘Tempty‘s
TBuffer3   TBuffer‘(Tstate‘n)‘(req&&s)=value‘n&&(TBuffer‘Tempty‘s)
TBuffer4   TBuffer‘(Tstate‘n)‘(data‘m&&s)=error&&TBuffer‘(Tstate‘n)‘s
```

This specification is under specified for the state `strange`. It behaviourally refines the function `Buffer` from `TBUF`.

To remove the superfluous state `strange` from the type `TState` formally, the next step is to change the type of `TBuffer` from `TState` to `State`. This is achieved by defining a quotient on `TState` with `empty`$\sim$`strange` since `empty` and `strange` are bisimulation equivalent, since they produce the same output.

The following specification refines `SBUF` from page 118 functionally and uses the lifting methods.

```
SBUF2 = TBUF2 + QUOT +
defs
       TState_per_def "(op ∼∼) ≡ λx y::TState.
                  ⌈is_Tempty‘x and is_Tempty‘y or
                   is_strange‘x and is_strange‘y or
                   is_TState‘x and is_TState‘y or
                   is_Tempty‘x and is_strange‘y or
```

```
                    is_strange‘x and is_Tempty‘y⌋"
rules   (* the following proof obligations are derivable *)
        TState_sym      (x::TState) ∼∼ y ⟶ y ∼∼ x
        TState_trans    (x::TState) ∼∼ y ∧ y ∼∼ z ⟶ x ∼∼ z
        (* now instantiate TState into the class per *)
instance TState::per (TState_sym,TState_trans)
        (* now quotients are available *)
types State = TState quot
consts
        empty   :: State
        state   :: dnat → State
        Buffer  :: message stream → answer stream
        SBuffer :: State → message stream → answer stream
defs    (* lift the operations *)
empty_def       empty ≡ <⌊Tempty⌋> (* = <⌊strange⌋> *)
state_def       state ≡ Λn.<⌊TState‘n⌋>
SBuffer_def     SBuffer ≡ Λs.TBuffer‘(any_in s)
end
```

This specification is a conservative extension of TBUF2. Therefore, it refines it trivially. We can continue our development with an implementation of SBUF, as described in Example 4.1.1 because the above specification SBUF2 is equivalent to the specification SBUF from the Example 4.1.1.

This example showed how quotients can be used in the development of state-based specifications by removing a state from the state space. The fact that the resulting specification is not executable is not important, since we are able to eliminate the states from our specification (see next section or the example on page 118).

If we want to add a state to a requirement specification we use the subdom construct to define the states of the requirement specification on the basis of our implementing specification, which has more states.

The method adds a state to the state space τ of state-based specification S. The new state space is named τ'

1. Define τ' with the domain construct like τ and add the new states to the domain construct[3] to the specification S1.

2. Add a definition of the predicate adm_pred' (into a specification S2) on the new type τ', such that it is true for all values σ_i that correspond to the values from the old state space. If possible use continuous functions for the definition of the predicate.

[3]For simplicity we assume that τ has been defined with the domain construct.

3. Prove the admissibility $\vdash\!\!\rightsquigarrow$ `adm adm_pred'::`τ`'⇒bool` $\vdash\!\!\rightsquigarrow$ `S2`. If continuous functions are used this step will be trivial.

4. Extend `S2` to `S3` by `types` τ = `subdom` τ`'` constructor and define a constant τ_i in the new subdomain by $\tau_i \equiv$`abs_sd(`σ_i`)` for every σ_i of the old type τ.

5. `S` $\vdash\!\!\rightsquigarrow$ `S3` holds automatically. Continue the development with `S1`.

If we use only continuous functions in the definition of the admissible predicate, we have no further proof obligations. Using the discriminators from the `domain` construct ensures this.

Consider the following example:

Example 6.2.2 ***Adding a State to the Buffer***

To model the effect that our buffer in a the state `strange` gets another request, we can add a more strange state (`ms`) to the specification of state in the previous example, and give the following more strange specification of the buffer:

```
T2 = Dnat +
domain TState2 = Tempty2 | strange2 | ms | Tstate2(dnat)
end

TBUF2 = Stream + Messages + T2 +
consts
        TBuffer2 :: TState2 → message stream → answer stream
rules
(* rules TBuffer1, ... ,TBuffer 5 like the rules in TBUF *)
TBuffer6   TBuffer2‘strange2‘(req&&s)=error&&TBuffer2‘ms‘s
TBuffer7   TBuffer2‘ms‘(data‘m&&s)=stored&&TBuffer2‘(Tstate2‘m)‘s
TBuffer8   TBuffer2‘ms‘(req&&s)=error&&TBuffer2‘ms‘s
```

This specification `T2` is now extended with the subdomain constructor to a specification `T3`, which functionally refines `T`.

```
T3 = T2 + SUBD +
defs   (* restriction predicate *)
St_adm adm_pred'≡ λs.⌈is_Tempty‘s or is_strange‘s or is_Tstate‘s⌉
rules
       (* this rule is easily derivable *)
proof_obligation adm (adm_pred'::TState2⇒bool)
       (* now instantiate State2 into adm *)
instance TState2::adm  (proof_obligation)
```

```
        (* now subdom is available on TState *)
        (* now introduce TState as subdomain *)
types    TState = TState2 subdom
        (* now introduce operations *)
consts
        Tempty :: TState
        Tstate :: dnat → TState
defs
        Tempty ≡ abs_sd(Tempty2)
        Tstate ≡ Λn.abs_sd(Tstate2‘n)
```

This specification functionally refines `T`. Now we construct a specification `TBUF3` identical to `TBUF`, except that it uses `T3` instead of `T`.

```
TBUF3 = Stream + Message + T3 +
(* rest is identical to TBUF *)
```

Since `T3` functionally refines `T`, and because all specifications in the example are conservative extensions, we conclude by modularity of functional refinement, that `TBUF3` is a refinement of `TBUF`. We could continue the development with implementing `TBUF3`.

These examples show how the implementation of ADTs is applied to the implementation of state-based interactive systems.

6.2.4 State Elimination

For the translation from state-based into purely functional specifications the elimination of states is an important step. The method for the elimination of states from the specification `S` of a state-based stream processing function `f::`$\tau\rightarrow[\alpha\rightarrow$`b`$]$ is:

1. Extend `S` with a PER on α `stream` into the specification `S2`.
2. Prove `instance` α `stream::per` $\vdash\!\!\rightsquigarrow$ `S2`.
3. Define the histories by `types` $\tau{=}\alpha$ `stream quot` .
4. Define the states as equivalence classes of histories.
5. Define a realization of the state-based stream processing function stream in terms of histories into the specification `S3`.
6. Prove `S` $\vdash\!\!\rightsquigarrow$ `S3`.

7. Eliminate the quotient using a theory interpretation (if desired)[4].

This method is presented in Section 4.4 together with the example from page 118.

Usually the elimination of states will first reduce the states by removing some elements of the state space. Then the states will be eliminated completely.

6.3 Schematic Implementations

In the previous section we defined some methods for the development of interactive systems. The methods support behavioural development, structural development, and the development of state-based systems. Another important group of development situations deals with the development of systems specified at different levels of abstraction. Many steps of communication protocols are good examples for specifications with different levels of abstraction, for example the realization of a string-based communication protocol by serial communication channels.

The following development situations from Section 1.2.4 are examples for different abstraction levels:

- communication channel development,
- restricted communication channel development,
- interface simulation, and
- dialog development.

FOCUS provides general techniques for the implementation of interactive systems at different levels of abstraction [Bro93, Bro92]. The key idea is, as in the method for conservative extension, to use abstraction and representation functions between the two levels. The main requirement is to prove that these functions exists and that they are inverse, at least for a subset of the concrete values. For an specification of an abstract system (or component), which uses the type τ, and a specification of a concrete realization of the system which uses the type σ the abstraction and representation functions have the following types[5]:

[4] Another way would be to specify the states with $\sim$ instead of $=$. This would allow us to omit the last step and in step 3 it would suffice to define `types` τ $=$ α `stream`.

[5] This scheme also fits to the implementation of ADTs (without theory interpretation).

```
ABS_REP =
consts
        abs:: σ → τ      (* abstraction function *)
        rep:: τ → σ      (* representation function *)
        cor:: σ ⇒ bool   (* restriction predicate *)
rules
        abs_rep abs‘(rep‘a)=a
        rep_abs cor c ⟹ rep‘(abs‘c)=c
        cor_rep cor (rep‘a)
end
```

To allow multiple representations we could use $\sim$ instead of $=$ in these equations. We use $=$ for readability and apply a simple theory interpretation if we want to allow multiple representations.

In FOCUS often only the first equation is required (for example in communication history refinement). However, having the second equation ensures us that the extension from the concrete theory to the abstract theory is conservative. The axiom `cor_rep` states that the representation of the abstract values are corresponding elements.

Our task in the implementation of interactive systems at different levels of specification is to define abstraction and representation functions together with a restriction predicate. Since we use FOCUS distributed systems with different levels of abstraction are specified with streams of messages. We classify the implementations into two groups according to the form of the functions `abs` and `rep`. Schematic implementations are schematic liftings from abstraction and representation functions, used in the implementation of ADTs, to `Sabs` and `Srep` on the streams.

Therefore, schematic implementations use schematic abstraction and representation functions. Assuming that `ABS_REP` contains the implementation of the ADT τ over σ, then the schematic liftings, which are again an instance of our general lifting methods in Section 3.3.3, have the following form:

```
SCHEMATIC = ABS_REP + Stream +
consts
    Sabs:: σ stream → stream τ     (* abstraction function *)
    Srep:: τ stream → stream σ     (* representation function *)
    Scor:: σ stream ⇒ bool         (* restriction predicate *)
defs
    Sabs_def Sabs ≡ fix‘(ΛSabs s.abs‘(ft‘s) && Sabs‘(rt‘s))
    Srep_def Srep ≡ fix‘(ΛSrep s.rep‘(ft‘s) && Srep‘(rt‘s))
    Scor_def Scor ≡ wfp (λc s.cor(ft‘s) ∧ c(rt‘s))
```

See Section 4.3.2 for a definition of the greatest fixed point for predicates. Provided that we have abstraction and representation functions for ADTs the existence of these

schematic functions is ensured, since they are conservatively defined by conservative extensions. The advantage of these schematic definitions is that we can derive the following theorems schematically.

```
Sabs_Srep  Sabs‘(Srep‘a)=a
Srep_Sabs  Scor c ⟹ Srep‘(Sabs‘c)=c
Scor_Srep  Scor (Srep)
```

Therefore, the schematic implementations are well suited for the implementation of interactive systems.

Individual implementations are arbitrary definitions of `Sabs` and `Srep`, in particular, it is allowed to relate one message of an abstract stream to many messages of the concrete stream. For example this allows us to implement a stream of bytes by a stream of bits with a sequential transmission of the bits (see page 193).

This section describes schematic implementations for different development situations. We have a composition result for downward simulation development in Section 6.3.3.2, which improves the composition of downward simulation from FOCUS. Individual implementations are described in Section 6.4.

The concrete methods for schematic implementations are of the following form:

1. Implement the ADT of the messages.
2. Lift the implementation to streams as described in `SCHEMATIC`

We now define concrete methods for implementations of interactive systems.

6.3.1 Communication Channel Development

Communication channel development is the most simple form of relating parts of systems, specified at different levels of abstraction. Communication channel development is the basis for other implementation methods.

Definition 6.3.1 ***Communication Channel Development***

An isomorphic pair of streams τ `stream` and σ `stream` is called *communication channel development*, if

- there exists inverse abstraction and representation functions with:

```
consts
    abs:: σ stream → τ stream    (* abstraction function *)
    rep:: τ stream → σ stream    (* representation function *)
rules
    abs_rep abs‘(rep‘a)=a
    rep_abs rep‘(abs‘c)=c
```

In the schematic case the types τ and σ have to be isomorphic. Consider the example of implementing a stream of characters by a stream of bytes.

Example 6.3.1 ***Streams of Characters***

This example uses a type `byte` of the specification `BYTE`. Then we can define the type of characters with the `domain` construct by:

```
CHAR = BYTE +
domain Char = b2c(c2b::byte)
end
```

This defines the ADT `Char` isomorphic to `byte`. Instantiating the general scheme of schematic implementations from page 177 generates us the communication channel development functions:

```
COM_CHN_SCH = Stream + CHAR +
consts
    abs:: byte stream → char stream    (* abstraction function *)
    rep:: char stream → byte stream    (* representation function *)
defs
    abs_def abs ≡ fix‘(Λsabs s.b2c‘(ft‘s) && sabs‘(rt‘s))
    rep_def rep ≡ fix‘(Λsrep s.c2b‘(ft‘s) && srep‘(rt‘s))
end
```

The proof that these functions are a communication development can be obtained from the general proof of schematic implementations, since `b2c‘(c2b‘c)=c` and `c2b‘(b2c‘b)=b` are valid in `CHAR`.

6.3.2 Restricted Communication Channel Development

Restricted communication channel development is a generalization from communication channel development.

Definition 6.3.2 ***Restricted Communication Channel Development***

A pair of streams τ `stream` and σ `stream` is called *restricted communication channel development*, if

- there exist abstraction and representation functions, and a restriction predicate with:

```
consts
    abs:: σ stream → τ stream    (* abstraction function *)
    rep:: τ stream → σ stream    (* representation function *)
    cor:: σ stream ⇒ bool        (* restriction predicate *)
rules
    abs_rep abs‘(rep‘a)=a
    rep_abs cor c ⟹ rep‘(abs‘c)=c
    cor_rep cor (rep‘a)
```

We write `COR` $\longrightarrow$ `A;R=I` to denote the axiom `rep_abs`, and `COR.R` for `cor_rep`

In the schematic case τ has to be a subdomain of σ. Consider the example of implementing a stream of bytes by a stream of natural numbers.

Example 6.3.2 ***Streams of Bytes***

This example defines the type `byte` in the specification `BYTE`. It uses the `subdom` constructor and extends the natural numbers.

```
BYTE = Dnat +
defs   (* restriction predicate *)
Dnat_adm_def adm_pred'≡ λn.⌈n≤256⌉
rules
       (* this rule is easily derivable *)
proof_obligation adm (adm_pred'::dnat⇒bool)
       (* now instantiate dnat into adm *)
instance dnat::adm  (proof_obligation)
       (* now subdom is available on dnat *)
       (* now introduce Byte as subdomain *)
types   byte = dnat subdom
consts (* continuous abstraction, representation function *)
       cabs :: dnat → byte
       crep :: byte → byte
defs
cabs_def  cabs ≡ Λn.If n≤256 then (abs_sd n) else ⊥ fi
cabs_def  crep ≡ Λn.rep_sd n  (* rep_sd is continuous *)
end
```

This defines the ADT `byte` as subdomain of `dnat` and a continuous abstraction function, since it is needed in the fixed point for the schematic liftings. Instantiating the general scheme of schematic implementations from page 177 generates us the restricted communication channel development functions:

```
RCOM_CHN_SCH = Stream + BYTES +
consts
    abs:: dnat stream → byte stream    (* abstraction function *)
    rep:: byte stream → dnat stream    (* representation function *)
    cor:: byte stream ⇒ bool           (* restriction predicate *)
defs
    abs_def abs ≡ fix‘(Λsabs s.cabs‘(ft‘s) && sabs‘(rt‘s))
    rep_def rep ≡ fix‘(Λsrep s.crep‘(ft‘s) && srep‘(rt‘s))
    cor_def cor ≡ wfp (λc n.⌈ft‘n≤256⌉ ∧ c(rt‘s))
end
```

The proof that these functions are a restricted communication development can be obtained directly from the general proof of schematic implementations.

The communication channel development situations are quite simple, since they are only dealing with channels. The following situations describe the implementations of situations in which also components occur.

6.3.3 Interface Simulation

This section presents a central refinement technique for the implementation of interactive systems. In [Bro93] these situations are called interaction refinements.

Definition 6.3.3 ***Interface Simulations***

Let $(\mathcal{L}, M, \models\!\!\approx\!\!\gg, \hookrightarrow)$ be a deductive software development basis, supporting the implementation of ADTs. Let τ `stream` and σ `stream` be a restricted communication channel development (with functions `abs` and `rep` with `A.abs` and `R.rep`). Let furthermore be $\mathtt{p}::[\tau\to\tau]$ a stream processing function specified in `P` (with `P.p`) and $\hat{\mathtt{p}}::[\sigma\to\sigma']$ a stream processing function specified in $\widehat{\mathtt{P}}$ (with $\widehat{\mathtt{P}}.\hat{\mathtt{p}}$), then the restricted communication channel development is called:

U simulation if $\mathtt{P} \models\!\!\approx\!\!\gg \mathtt{R};\widehat{\mathtt{P}};\mathtt{A}$

U^{-1} simulation if $\widehat{\mathtt{P}} \models\!\!\approx\!\!\gg \mathtt{A};\mathtt{P};\mathtt{R}$

Downward simulation if $\mathtt{P};\mathtt{R} \models\!\!\approx\!\!\gg \mathtt{R};\widehat{\mathtt{P}}$

Upward simulation if $\mathtt{A};\mathtt{P} \models\!\!\approx\!\!\gg \widehat{\mathtt{P}};\mathtt{A}$

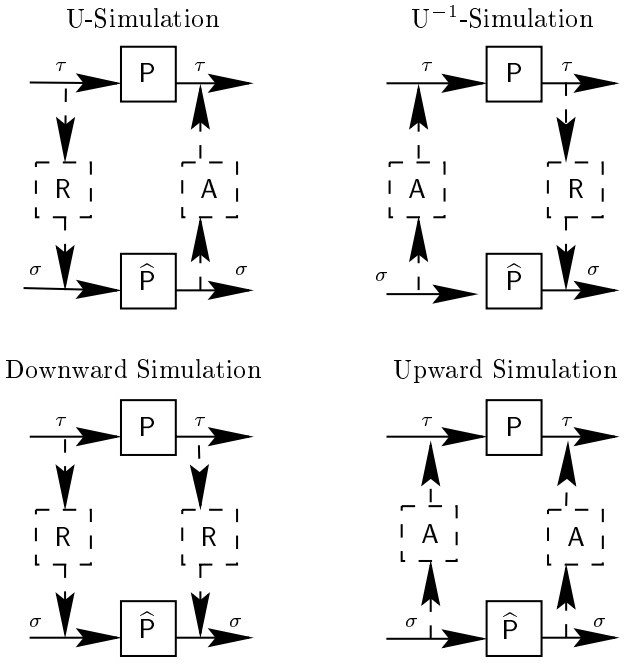

Figure 6.2: Simulations for Interface Interaction Refinement

Note that this definition includes the axioms `R;A=I`, `COR` $\longrightarrow$ `A;R=I`, and `COR.R` from the restricted communication channel development (see Definition 6.3.2 for the precise axioms).

This definition introduces different forms of simulations. They are depicted in Figure 6.2. To emphasize the methodical difference between abstraction and representation functions, and the components describing the behaviour of the system at the abstract and concrete level we used dashed lines for the components relating the different abstraction levels and solid lines for the other components.

We use U simulation and downward simulation in our methods for the implementation, since we do not find examples in the deductive software development for upward simulation and U^{-}1 simulation[6]. For that reason we will analyze their compositionality more detailed. U simulation constructs a component (with conservative extension based one the concrete component), which refines the abstract one by model inclusion. Therefore, U simulation is modular and trivially compositional.

The important difference between downward simulation and U simulation is that U simulation constructs a component with the same interface as the abstract one while downward simulation does not. For the process of software development this means that every component may be developed by a U simulation, independent of the components environment. Since downward simulation changes the interface it can only be applied, if both compo-

[6]Even in [BFG^{+}94] there is no such example.

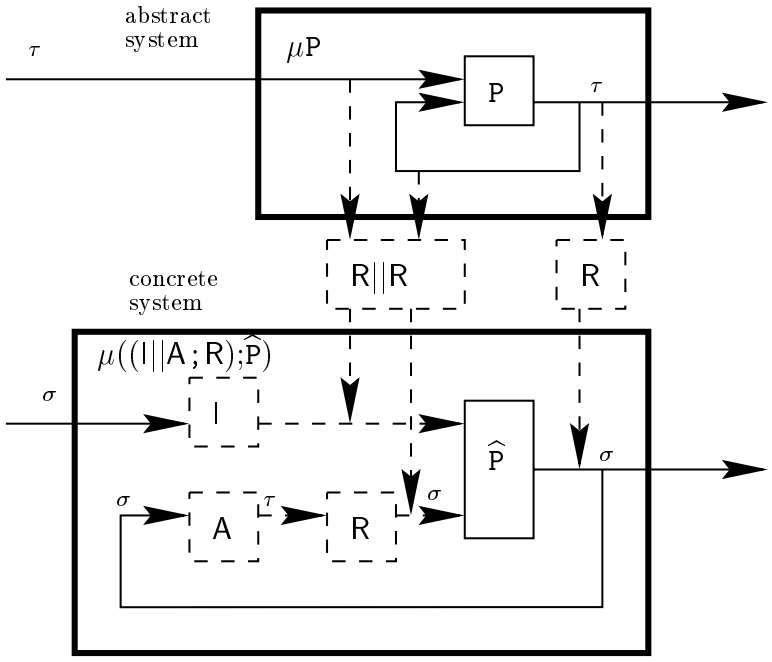

Figure 6.3: General Compositionality for the Feedback Operator

nents communicating over a channel of the abstract type are implemented in the same way (see the dialog development situation in Section 1.2.4). Downward simulation cannot be performed, if a component has a channel of the abstract type to the environment. This is clear, since only the system is developed, but not its environment. For the implementation of such components U simulations are needed.

Since downward simulation is no conservative extension (and no model inclusion) its compositionality requires an extra treatment. In [Bro92] the following compositionality results are proved for downward simulations.

- $\mathtt{P}_1;\mathtt{R} \mid\!\approx\!> \mathtt{R};\widehat{\mathtt{P}}_1$ and $\mathtt{P}_2;\mathtt{R} \mid\!\approx\!> \mathtt{R};\widehat{\mathtt{P}}_2$ imply $\mathtt{P}_1;\mathtt{P}_2;\mathtt{R} \mid\!\approx\!> \mathtt{R};\widehat{\mathtt{P}}_1;\widehat{\mathtt{P}}_2$,
- $\mathtt{P}_1;\mathtt{R} \mid\!\approx\!> \mathtt{R};\widehat{\mathtt{P}}_1$ and $\mathtt{P}_2;\mathtt{R} \mid\!\approx\!> \mathtt{R};\widehat{\mathtt{P}}_2$ imply $(\mathtt{P}_1||\mathtt{P}_2);(\mathtt{R}||\mathtt{R}) \mid\!\approx\!> (\mathtt{R}||\mathtt{R});(\widehat{\mathtt{P}}_1||\widehat{\mathtt{P}}_2)$, and
- $\mathtt{P};\mathtt{R} \mid\!\approx\!> (\mathtt{R}||\mathtt{R});\widehat{\mathtt{P}}$ implies $\mu\mathtt{P};\mathtt{R} \mid\!\approx\!> \mathtt{R};\mu((\mathtt{I}||\mathtt{A};\mathtt{R});\widehat{\mathtt{P}})$

The first two theorems state that downward simulation is compositional with respect to ";" and "||". The third rule is weaker, since it requires to use the complex system $\mathtt{R};\mu((\mathtt{I}||\mathtt{A};\mathtt{R});\widehat{\mathtt{P}})$ instead of $\mathtt{R};\mu\widehat{\mathtt{P}}$ to implement $\mu\mathtt{P};\mathtt{R}$. The graphical representation in Figure 6.3 shows the idea of that construction. The feedback operator ranges over the parallel composition of the identity (which only is in the formula because of type correctness) and the sequential composition of A and R which has no immediate motivation, but is required

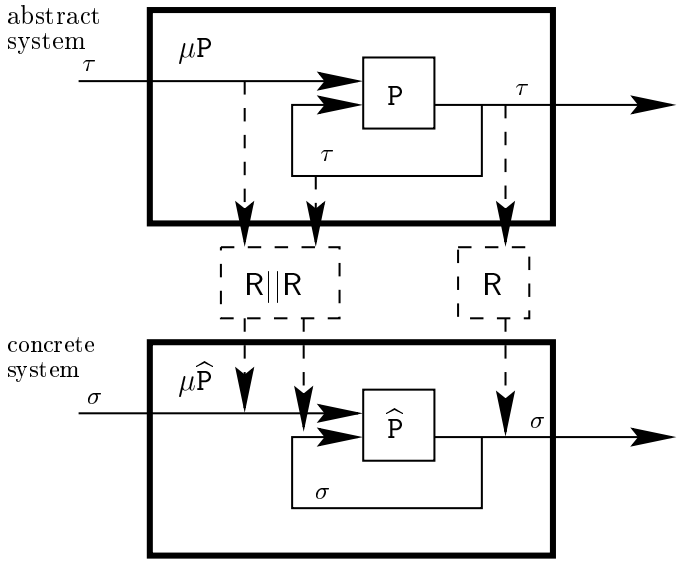

Figure 6.4: Improved Compositionality for the Feedback Operator

for proving compositionality. Informally speaking this is the way in which the concrete component ensures that its input values are representations of abstract values. However, since these input values are fed back from the component itself, it is easier to require the component to be preserving and to reason without the overhead of `A;R`.

This general, but complex composition rule may be simplified using the implementation of ADTs in HOLCF as described in Chapter 5. With this concrete method we will be able to improve the general compositionality of the feedback operator μ for downward simulation and prove the following rule (see Theorem 6.3.1):

- `P;R` $\models\!\!\approx\!\!\gg$ `(R||R);`$\widehat{\texttt{P}}$ implies μ`P;R` $\models\!\!\approx\!\!\gg$ `R;`$\mu\widehat{\texttt{P}}$.

This improved result is shown in Figure 6.4. The benefit of this solution is, that we have the optimal compositionality of the refinement relation. In the general case compositionality was enforced by including `A;R` into the feedback loop to have a compositional composition of components. So the development from `P` to $\widehat{\texttt{P}}$ will be modular, only if the system uses $\widehat{\texttt{P}}$ composed with `A;R`. Our solution (with invariance) will not need this composition with `A;R` in the feedback channel, but it needs the existence of a function `abs` with `A.abs` to ensure the correctness of this step. However, our proposed methods for the implementation of interactive systems define functions for `A` and `R` for every implementation. For the schematic definitions we get the rules directly from the corresponding rules for ADTs in HOLCF, and for the individual case we provide methods to prove these rules.

Now we analyze the interface situations (see Section 1.2.4) more detailed and then we give methods and examples for every interface situation. We focus on U simulation and Down-

ward simulation, since we do not find examples in the deductive software development for upward simulation and $U^{-}1$ simulation[7]. All examples assume `A` and `R` to be specifications of schematically lifted functions.

6.3.3.1 Implementation of Components with Isomorphic Types

The implementation of components with interfaces of isomorphic types is easy and the parameters of the method for the implementation of ADTs of the previous chapters are trivial: neither a restriction step nor a quotient step is needed. For this special case of the implementation of ADTs the `domain` construct may be used (see Section 2.1.5). It conservatively introduces continuous abstraction and representation functions which are by construction consistent. Therefore, it ensures that the following facts hold:

- `A;R = I`
- `R;A = I`

As an example we take the implementation of the boolean values by bits, allowing to implement components which base on booleans by components based on bits.

Example 6.3.3 ***Implementation of Boolean by Bits***

The abstract data type specification of `BOOL` is:

```
BOOL = HOLCF +
domain       B = T | F
ops curried strict total
    and   :: B → B → B   (cinfixl 55)
rules
    and1  T and x = x
    and2  F and x = F
end
```

The specification of the concrete data type BIT is:

```
BIT = HOLCF +
domain       Bit = L | H
ops curried strict total
    hw_and   :: Bit → Bit → Bit   (cinfixl 55)
rules
    hw1  H hw_and b = b
    hw2  L hw_and b = L
end
```

[7] Even in [BFG+94] there is no such example.

For both kinds of implementations the concrete specification is extended by the `domain` construct.

```
BOOL_by_BIT = BIT +
domain B = abs (rep :: Bit)
ops curried strict total
    T        :: B
    F        :: B
    and      :: B → B → B          (cinfixl 55)
defs
    T_def    T ≡ abs‘H
    F_def    F ≡ abs‘L
    and_def  a and b ≡ abs‘((rep‘a) hw_and (rep‘b))
end
```

The specification `BOOL_by_BIT` constructs the abstract functions (and constants) with the construction schemes of Definitions 3.3.1 and 3.3.2, which are generalizations of $\mathtt{R};\widehat{\mathtt{P}};\mathtt{A}$ for arbitrary types.

The differences between downward simulation and U simulation are proof obligations, compositionality proof, and code generation.

Since the `domain` construct ensures consistency of `A` and `R` the only remaining proof obligation for the correctness of the implementation of components for isomorphic types with U simulation is:

- $\mathtt{P} \mid\!\approx\!\!> \ \mathtt{R};\widehat{\mathtt{P}};\mathtt{A}$

In the example this would be `BOOL` $\mid\!\approx\!\!>$ `BOOL_by_BIT`. U simulation is a method for constructing an implementation which behaviourally refines the abstract specification. Therefore compositionality of U simulation follows from the compositionality of functional refinement.

Since the `domain` construct ensures consistency of `A` and `R` the only remaining proof obligation for the correctness of the implementation of components for isomorphic types with downward simulation is:

- $\mathtt{P};\mathtt{R} \mid\!\approx\!\!> \mathtt{R};\widehat{\mathtt{P}}$

In the example this would be `(rep‘x) hw_and (rep‘y) = rep‘(x and y)` $\mid\!\approx\!\!>$ {`BOOL + BIT + rep`}.

The interesting case in the compositionality of downward simulation is compositionality with respect to the feedback operator. Since `A;R=I` the optimal composition rule holds:

- `P;R` $\models\!\!\approx\!\!>$ `(R||R);`$\widehat{\texttt{P}}$ implies μ`P;R` $\models\!\!\approx\!\!>$ `R;`$\mu\widehat{\texttt{P}}$.

The proof is in [Bro92].

Code generation for U simulation generates the program out of the extended specification (in the example: `BOOL_by_BIT`). It uses the `datatype` construct of functional programming languages for the translation of the `domain` construct. The representation function is defined by pattern matching by:

```
fun rep (abs x) = x;
```

The translation of the other functions and constants from the extended specification is only syntactic, and that is why it is omitted here.

Code generation for downward simulation only uses the concrete specification (in the example: `BIT`). Therefore, it generates neither a new data type nor the simulation of the abstract functions and constants. Downward simulation requires the consistent abstraction and representation functions of the extended specification only for correctness and compositionality.

The only interesting remark about code generation for downward simulation is that it changes the interface of the implemented component. For (type) correctness of the system it is necessary that the components communicating with the implemented component are implemented in the same way.

For the development process this means: If two components, communicating over a channel, are implemented in the same way, it will be allowed to use downward simulation. For example if two components are communicating over a channel of type set, this channel may only be replaced by a channel of ordered sequences if both components use the sets implemented by ordered sequences. If one component uses for example hash tables, then the channel has to remain of type set and the only possible implementation is U simulation.

Only the system (and not the environment) is developed in the development process. Therefore, components communicating with the environment have to be developed by U simulation. See Section 6.3.4 for a way to develop components together with parts of their environment. Such developments are dialog developments and are treated in Section 6.3.4.

6.3.3.2 Implementation of Components with Concrete Subdomains

For the implementation of interactive components with an interface of concrete subdomains we cannot directly use the `domain` construct. We use the implementation of ADTs of the

previous chapters and only need a restriction step (See Section 5.2 for a detailed description how to fix the restriction predicate and the corresponding constants.). The restriction step of the implementation of ADTs in HOLCF conservatively introduces a new type, which is isomorphic to a concrete subdomain (see Section 3.3).

For U simulation the only remaining proof obligations are:

- $\mathtt{R};\widehat{\mathtt{P}};\mathtt{A} \mid\!\approx\!> \mathtt{P}$
- $\Pi(\widehat{\mathtt{P}})$ ($\widehat{\mathtt{P}}$ is Π-preserving (see Definition 6.1.3))

The invariance proof obligation $\Pi(\widehat{\mathtt{P}})$ does not appear in [Bro92, Bro93] but it is implicitly used in $\mathtt{P} \mid\!\approx\!> \mathtt{R};\widehat{\mathtt{P}};\mathtt{A}$[8]. The advantages of an explicit treatment of invariance are:

- a stronger compositionality result (Theorem 6.3.1),
- the continuity of the abstract functions, which are schematically introduced by conservative extension in the restriction step follows automatically (see Section 3.3.3 for details), and
- invariance is a methodical help for finding the corresponding functions and the restriction predicate (see Section 5.2).

For downward simulation the remaining proof obligations are:

- $\mathtt{P};\mathtt{R} \mid\!\approx\!> \mathtt{R};\widehat{\mathtt{P}}$
- $\Pi(\widehat{\mathtt{P}})$

Compositionality of downward simulation with respect to sequential and parallel composition is proved in [Bro92]. For compositionality with respect to the feedback we derive the following theorem:

Theorem 6.3.1 ***Compositionality of Downward Simulation***

Let A and R be a communication history refinement (with $\Pi \longrightarrow \mathtt{A};\mathtt{R}{=}\mathtt{I}$ and $\Pi.\mathtt{R}$). Let further P and $\widehat{\mathtt{P}}$ be the specifications of two stream processing functions with the invariance $\Pi(\widehat{\mathtt{P}})$; then the following compositionality for downward simulation holds:

- $\mathtt{P};\mathtt{R} \mid\!\approx\!> (\mathtt{R}||\mathtt{R});\widehat{\mathtt{P}}$ implies $\mu\mathtt{P};\mathtt{R} \mid\!\approx\!> \mathtt{R};\mu\widehat{\mathtt{P}}$.

[8] If P contains a total operation, for example a selector, then we have to show that the abstraction is defined. This can only be verified if the invariance holds.

The general composition rule is shown in Figure 6.3 on page 183, whereas the improved compositionality rule is depicted in Figure 6.4 on page 184.

Proof

The proof uses the general composition rule proved in [Bro92] and improves it. There it has been shown that:

- $\mathtt{P;R} \models\!\!\approx\!\!> (\mathtt{R}||\mathtt{R});\widehat{\mathtt{P}}$ implies $\mu\mathtt{P;R} \models\!\!\approx\!\!> \mathtt{R};\mu((\mathtt{I}||\mathtt{A;R});\widehat{\mathtt{P}})$

This may be simplified since $\Pi\,.\,\mathtt{R}$ ensures that the μ operator gets the values `x` of the subdomain (with Π `x`). The invariance $\Pi(\widehat{\mathtt{P}})$ ensures that it only produces output values of the concrete type (Π $(\widehat{\mathtt{P}}$ `x`)). Now the first rule of the conservative extension of the subdomain may be applied and reduces `A;R` to `I`. Elimination of the identity `I` gives the result.

This result holds also in the general case of individual implementations, since it only uses the requirements from the communication history refinement $\Pi \longrightarrow \mathtt{A;R{=}I}$ and $\Pi\,.\,\mathtt{R}$. However, in the general case these premises cannot be proved schematically.

The generated code for U simulation of this implementation also uses the `datatype` construct, but since not all values should be reachable the generated code hides the real constructor `abs` and only exports the implemented functions of the abstract data type (see Section 3.3.5 for details). Code generation for U simulation bases on the extended specification.

The generated code for downward simulation of this implementation uses only the concrete representations. As shown in Section 6.3.3.1 downward simulation may only be applied, if both components using the same channel are implemented in the same way.

6.3.3.3 Implementation of Components with Abstract Subdomains

The task of schematically implementing an interactive component with an interface of an abstract type by a component with an interface of a concrete type which is isomorphic to a subdomain of the abstract type is generally impossible. This is independent from the development of interactive components and their interfaces. The types are the only reason for that: for example consider the type of natural numbers `N` which cannot be implemented schematically by a bounded type. That is one motivation for the individual implementation, which allows such implementations.

If not all values of the abstract type are used, it is possible to implement it schematically. Another possibility are individual implementations, since they allow us to use many concrete messages for the representation of one abstract element (see Section 6.4). The implementation uses a partial embedding from the abstract specification into the concrete. The functions in the abstract specification have to be invariant, otherwise the refinement

would be impossible (since the embedding function is partial). Consider the following example which only focuses on the types of the components interfaces:

Example 6.3.4 ***Implementation of an Abstract Subdomain***

The abstract specification describes a modulo 256 counter[9].

```
MODCNT = N +
ops      cnt  :: N → N
rules
count    cnt n ≡ If n≐255 then 0 else (n+1) fi
end
```

The definition of the `cnt` function ensures that `cnt` is only specified on the "abstract subdomain" ($\{n \leq 255\}$) of `N`.

Let `BYTE` be a specification of bytes with a function `inc` to increment bytes by one, an order $\leq$, and a maximal value `FF`. Then the implementation is:

```
BYTECNT = BYTE +
domain   N = abs(rep::Byte)
ops      cnt  :: N → N
         255  :: N
rules
255_def  255=abs(FF)
partial  n≤255 ⟹ cnt n = abs‘(inc‘(rep‘n))
end
```

Since `inc‘FF` is undefined the embedding has to be partial. It is obvious that `cnt` of `BYTECNT` refines `cnt` of `MODCNT` behaviourally[10]

In the example `MODCNT` would have been called finite in [Bre92] and could be implemented without additional bounds restricting its implementations.

The situation of abstract subdomains is reduced by behavioural refinement into another implementation situation. Therefore, compositionality and code generation are not analyzed here.

[9] A similar example is treated in [Fuc95].

[10] The proof goes by induction over the subdomain.

6.3.3.4 Implementation of Components with Concrete Quotients

The implementation of components with concrete quotients bases on the implementation of ADTs in the previous chapter. The restriction step is not needed, but an observable equivalence relation has to be supplied. See Section 5.2 for details how to find this relation.

The situation of concrete quotients handles the case of multiple representations for one abstract element. There are values `x` and `y` with `x`$\neq$`y` and `abs(x)=abs(y)`. Since `rep(abs(x))=x` holds for abstraction and representation functions it follows that abstraction and representation functions for this situation cannot exist. Therefore, no U simulation can be defined for this step. Small modifications, however, would lead us to a specification which is implementable by a U simulation. These modifications are the result of a simple theory interpretation Φ (see Definition 4.4.1). It replaces the equality on the abstract type τ by an observable equivalence and requires the type τ to be in the class `eq` (see Section 4.2.5). This replacement has to be carried out in all specifications of the system, but since it has no additional proof obligation (see Section 4.4.4 for its correctness) this does not matter.

There is only one problem with this replacement: it cannot be applied to arbitrary specifications, but only to normalizeable ones (see Definition 3.2.8). Therefore, we have to prove compositionality for this implementation by simple theory interpretation.

Theorem 6.3.2 ***Compositionality of Simple Theory Interpretation***

> Let Φ be a simple theory interpretation as defined in Definition 4.4.1, then Φ is compositional.

Proof

We have to show that for specifications $\mathtt{P}, \mathtt{P}_1, \mathtt{P}_2, \widehat{\mathtt{P}}, \widehat{\mathtt{P}}_1$, and $\widehat{\mathtt{P}}_2$ the following holds:

1. $\Phi\mathtt{P}_1 \mid\!\approx\!\gg \widehat{\mathtt{P}}_1$ and $\Phi\mathtt{P}_2 \mid\!\approx\!\gg \widehat{\mathtt{P}}_2$ imply $\Phi(\mathtt{P}_1;\mathtt{P}_2) \mid\!\approx\!\gg \widehat{\mathtt{P}}_1;\widehat{\mathtt{P}}_2$,
2. $\Phi\mathtt{P}_1 \mid\!\approx\!\gg \widehat{\mathtt{P}}_1$ and $\Phi\mathtt{P}_2 \mid\!\approx\!\gg \widehat{\mathtt{P}}_2$ imply $\Phi(\mathtt{P}_1||\mathtt{P}_2) \mid\!\approx\!\gg \widehat{\mathtt{P}}_1||\widehat{\mathtt{P}}_2$, and
3. $\Phi\mathtt{P} \mid\!\approx\!\gg \widehat{\mathtt{P}}$ implies $\Phi\mu\mathtt{P} \mid\!\approx\!\gg \mu\widehat{\mathtt{P}}$.

Since $\Phi(\mathtt{P}_1;\mathtt{P}_2) = \Phi\mathtt{P}_1;\Phi\mathtt{P}_2$, if $(\mathtt{P}_1;\mathtt{P}_2)$ is normalizeable it remains to show that $(\mathtt{P}_1;\mathtt{P}_2)$ is normalizeable. This follows from the fact that $\mathtt{P}_1$ and $\mathtt{P}_2$ are normalizeable. `P` will be normalizeable, if `P` does not contain axiomatized polymorphic predicates. Since ";" does not introduce such predicates, normalizeability holds for $(\mathtt{P}_1;\mathtt{P}_2)$. The other rules (2. and 3.) hold because of the same argumentation.

Since this step only prepares an implementation by adding an observable equivalence, code generation only has to instantiate this equivalence into the class `eq` (see Section 4.4.6 for details). Another possibility for concrete quotients is the elimination of states. It allows us to weaken the requirement of executability, since it implements non-executable quotients by executable constructs (see example on page 118).

6.3.3.5 Implementation of Components with Abstract Quotients

For the implementation of components with interfaces of abstract quotients the method of behavioural refinement is used. As in Section 6.3.3.3 abstract quotients cannot be implemented in general, but only in a context which does not actually need all informations of the abstract type. This reduces the implementation of components with abstract quotients to behavioural refinement. Therefore, neither compositionality nor code generation have to be analyzed here.

An example for an abstract quotient is the implementation of a timed specification (pairs of values and time) by a specification without time information. The PER on the timed specification would be the projection on the values. Of course correctness would require to show that the implementation ensures the timing conditions of the requirement specification.

6.3.4 Dialog Development

Some refinements require knowledge about the environment of a component. If we know the environment of a component, for example because the component is a part of a specific system, then we may implement the component with additional restrictions, which are ensured by the specific system.

Dialog development is a situation, where (at least) two components, communicating over (at least) one channel are developed together. This can only be done, if the abstract system and the concrete system are specified by glass box specifications. Therefore dialog development is a special form of behavioural refinement between glass box specifications.

For example, if two components are only sending `zero` and `one` over a channel of type `nat`, then we can implement this dialog. The implementation involves an interface implementation of both components. The channel can now be schematically implemented by a boolean channel, since the implementation of the channel may use the knowledge from the environment (the dialog) that only `zero` and `one` are sent.

The situation is depicted in Figure 6.5. It shows how a U simulation of the system can lead to an downward simulation between some components (depicted with the dotted lines). A special case is that $\alpha = \hat{\alpha}$ and $\gamma = \hat{\gamma}$. In this case we do not need explicit abstraction and representation functions. One such example is in Section 7.4.

Dialog development is a methodical combination of other development steps, we can apply the same refinement relation as for structural development in Section 6.2.2.

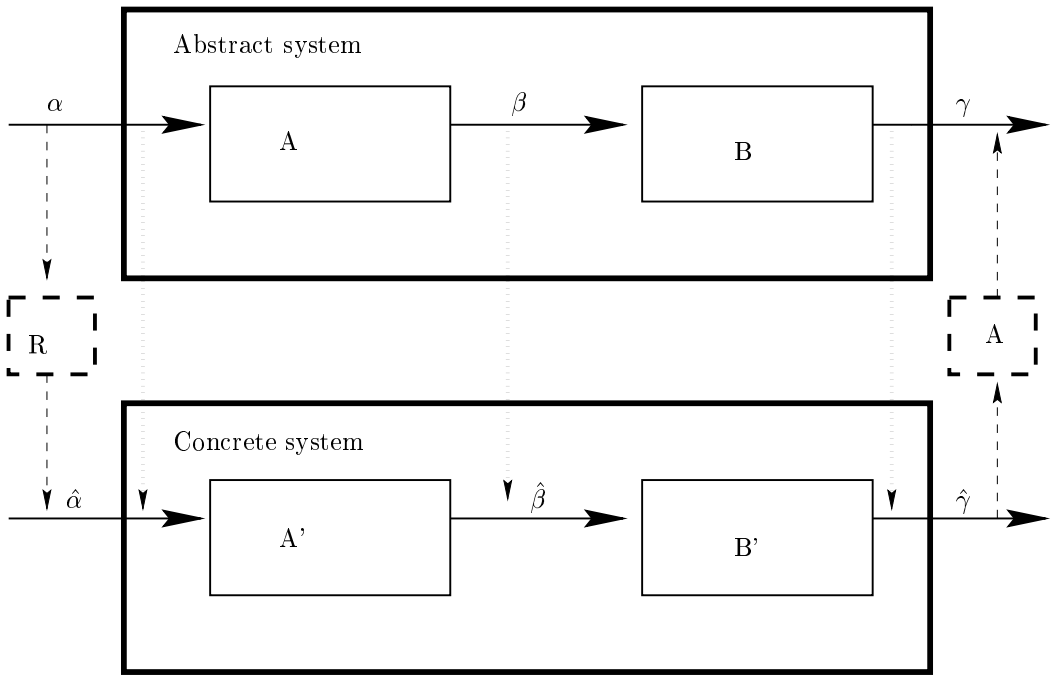

Figure 6.5: Dialog Development with U and downward Simulations

6.4 Individual Implementations

The previous section provided methods for the elimination of states, and for a lot of schematic implementations between different levels of abstraction. For every schematic development situation we can also give individual implementations. The only additional proof obligation is invertability with the axioms for `abs` and `rep` (see page 176).

This section presents an example for an individual implementation of a restricted communication channel development. We choose the implementation of a channel with byte by a channel with bits. The idea is that the bits are sequentially transmitted over the channel. In addition to the used specifications, we present the method for the proofs. This makes individual implementations easier to handle.

Example 6.4.1 ***Implementation of a Byte Stream***

This example shows the implementation of a byte stream by a bit stream. The used method is a restricted communication channel development. The restriction is a concrete subtype, realized with the `subdom` construct. The basic specification are the specification of bits and bytes are in Section 7.1.3 on page 202.

Since we do not use arbitrary bit streams as representations, but only those of a length divideable by eight[11]. We use the additional lifting to make the component reusable. Now we define the concrete subtype as subdomain of streams.

```
BITS0 = FStream + BIT +
domain BitS = absBitS(repBitS::bit stream)
consts
        is_evenS        :: bit stream ⇒ bool
defs
is_even_def     is_evenS ≡ wfp (λx.x=⊥∨rt⁸‘x≠⊥∧(E (rt⁸‘x)))
Bits_adm_def    adm_pred’≡ λn.is_evenS(repBitS‘n)
end
```

With `rt⁸‘x` we abbreviate the 8 times iteration of `rt` on the stream `x`. Before we can build the subdomain, we have to derive the admissibility of the predicate. By now this requires to use a theorem for the admissibility for `wfp` on flat streams (see [SM97, Ohe97] for the tool support for FOCUS). Because of this theorem we used `FStream`, the theory of flat streams, in our specification `BITS0`. The proof of admissibility requires to show that the functional, used in the definition of `is_evenS`, has the following properties:

- `monoP`
- `contP`
- `stream_monoP`

See [Ohe97] for a definition of these predicates. For our case study the only interesting aspect is that these proofs ensure, together with the flatness of `Bit`, the admissibility of the predicate defined as greatest fixed point with `wfp`. The admissibility of `adm_pred’` is proved as the theorem `adm_is_even`. Then we can instantiate `BitS` into the class `adm` and define the subdomain of even bits stream.

```
BITS = BITS0 + SUBD +
instance BitS::adm  (adm_is_even)
types   BitSd = BitS subdom
end
```

Now we are ready to define the abstraction and representation functions. We use, like the schemes on page 177 the fixed point constructor of HOLCF.

[11] We could also use quotients on bits streams to handle the case of non-representing bit streams.

The following specification is an executable specification of a translation between streams of bytes and streams of bits.

```
BYTES = Byte + BITS +
consts
   absBB   :: BitSd → Byte stream
   repBB   :: Byte stream → BitSd
defs
   absBB_def  absBB ≡ fix‘(Λabs.Λx.
                       mkBy‘(ft‘(repBitS‘(rep_sd x)))‘
                            (ft‘(rt‘(repBitS‘(rep_sd x))))‘
                            (ft‘(rt²‘(repBitS‘(rep_sd x))))‘
                            (ft‘(rt³‘(repBitS‘(rep_sd x))))‘
                            (ft‘(rt⁴‘(repBitS‘(rep_sd x))))‘
                            (ft‘(rt⁵‘(repBitS‘(rep_sd x))))‘
                            (ft‘(rt⁶‘(repBitS‘(rep_sd x))))‘
                            (ft‘(rt⁷‘(repBitS‘(rep_sd x))))‘
    &&  abs‘(abs_sd(absBitS‘(rt⁸‘(repBitS‘(rep_sd x))))))
   repBB_def  repBB ≡ (Λy.abs_sd(absBitS‘((
                            fix‘(Λrep.Λx.b1‘(ft‘x) &&
                            b2‘(ft‘x) && b3‘(ft‘x) &&
                            b4‘(ft‘x) && b5‘(ft‘x) &&
                            b6‘(ft‘x) && b7‘(ft‘x) &&
                            b8‘(ft‘x) && rep‘(rt⁸‘x))‘y)))
end
```

This looks more difficult than it is, since we have two different abstraction and representation functions: for the embedding, and for the subdomain. The abstraction makes a stream of bytes by taking the first eight elements in a stream. The next byte will be computed from the rest of the byte stream. The representation is dual. It takes a byte and concatenates all bits at the top of the representation from the rest.

The first theorems to prove, are the unfolding theorems for `absBB` and `repBB`. From these we can easily derive the pattern matching rules.

```
absBB_UU       absBB‘⊥ = ⊥
absBB_unfold2 ⟦a≠⊥;b≠⊥;is_evenS s⟧ ⟹
                 absBB‘(abs_sd(absBitS‘(a&&b&&s))) =
                 mkBy‘a‘b && absBB‘(abs_sd(absBitS‘s))
repBB_UU       repBB‘⊥=⊥
repBB_cons     repBB‘(b&&s)=abs_sd(absBitS‘(
                 b1‘b && b2‘b && repBitS‘(rep_sd(repBB‘s))))
```

With this lemmata we can derive the main theorems for the implementation of byte streams by bit streams.

```
abs_rep_BB    absBB‘(repBB‘x) = x
rep_abs_BB    repBB‘(absBB‘x) = x
```

The first theorem could be derived by induction on streams. However, induction on stream requires to show the admissibility of the predicate. In our case this was easy. A more elegant way is to use the lemma from streams to prove the equality of two streams:

```
stream.take_lemma   (⋀n.stream_take n‘x=stream_take n‘x’)⟹x=x’
```

This allows us to reduce the proof to an induction proof over the length of the stream. The second theorem `rep_abs_BB` is a theorem over all representing elements in the subdomain. Therefore, we used the rule from `SUBD0` (see Appendix A.1.3)

```
all_sd        ∀s.cor_sd s ⟶ P (abs_sd s) ⟹ P x
```

This allowed us to reduce the theorem to a theorem over stream, which we proved with help of the lemma `stream.take_lemma`

The result of the example is that individual translation can be carried out with the proof system, since tool support for all steps is available. The proofs are not complex, even if we used the embedding into `BitS`.

6.5 Summary for the Implementation of Interactive Systems

The methods for the implementation of ADTs in HOLCF can be applied to the implementation of interactive systems in FOCUS. The results are concrete methods which support the deductive development process of interactive systems.

These concrete methods provide several benefits for the development process of interactive systems, compared with the more abstract methods of FOCUS.

Proof Obligations: Consistency of `A` and `R` is ensured by construction, invariance is treated explicitly. For schematic implementations we get the axioms `A;R=I`, `COR ⟶ A;R=I`, and `COR.R` from the corresponding axioms in the implementation of ADTs without further proofs.

Compositionality: Invariance improves the compositionality result for downward simulation in the feedback case. The notation of simple theory interpretation weakens the specification (but ensures the existence of a model of the original specification) and allows multiple representations. Simple theory interpretations are compositional.

Classification: The methods for the implementation of ADTs characterizes different situations in the development of interactive systems. The classification includes a relation between the types of messages.

Methods: For every situation a concrete method is given. Downward and U simulation are integrated into the development process of interactive systems by characterizing its applicability.

State-bases Specifications: We have defined concrete methods for refining state-based specifications of stream processing functions. They allow us to remove states, to add states, and to show bisimulation equivalence between states. Furthermore it has been shown how states can be eliminated from the specifications.

Executability Since we have a method for the implementation of non-executable quotients (for states) we can implement ADT of states with the `quot` constructor. Simple theory interpretation, or a slight change in the specification style is the foundation of this step. Executability of the abstraction and representation functions are the basis for simulation and prototyping. It is shown how to generate functional programs from the implementations.

Tool Support The implementation of ADTs in HOLCF is realized with type constructors in the Isabelle system. Therefore the presented methods have tools support in the logic HOLCF.

Therefore our results are applicable methods for the implementation of interactive systems in HOLCF. We will apply them to the implementation of some critical aspects of a WWW server in the next chapter.

Chapter 7

Case Studies of a WWW Server

This chapter contains an extended case study with further applications of the implementation of ADTs in HOLCF. First, a library of ADTs in HOLCF is defined with the standard data types. The standard library show how our methods are applied and it is a basis for case studies. The definitions of the standard library use the `subdom` and `quot` constructors.

The second part presents case studies on the implementation of a WWW server. In Section 7.2 the structure of the case studies is given and the critical aspects are identified. The following sections describe the implementation of the critical aspects. Section 7.3 contains an implementation of the data base of a WWW server, and Section 7.4 focuses on a correct transmission of strings.

7.1 The Library of ADTs

This section contains the library of ADTs in HOLCF. We defined this library with the most frequently used ADTs, because the library is useful in many case studies and because it shows how our type constructors `subdom` and `quot` are applied. In contrast to free data types (introduced with the `domain` construct) the definition of subdomains and quotients involves a number of small steps. Having a standard library allows us to reuse these ADTs. A standard library is very useful for specifying large systems.

An adequate data modelling is an important basis for the implementation of interactive systems. For example the ARIANE 5 failure report [Lio96] identifies one cause in the chain of errors, which caused the disaster.

The internal SRI software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. This resulted in an Operand Error. The data conversion instructions (in Ada code) were

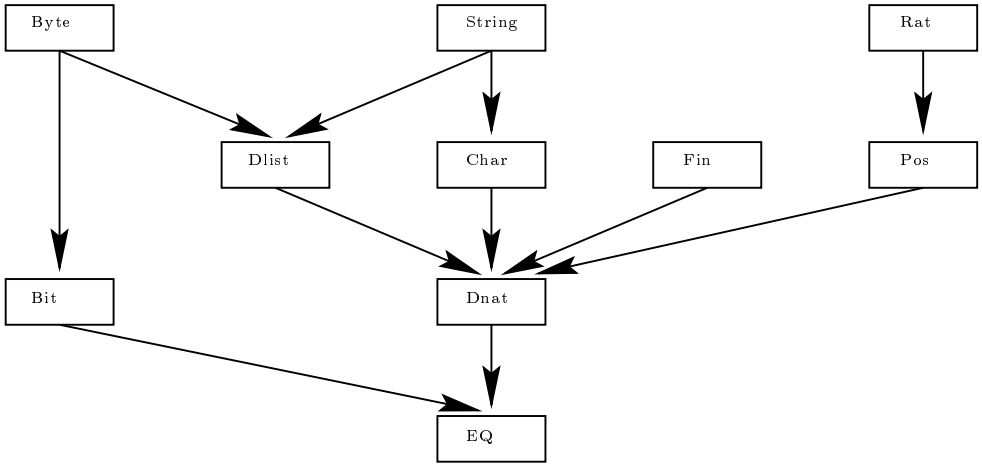

Figure 7.1: Structure of the Library

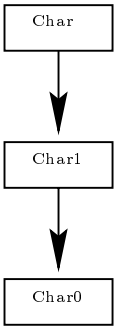

Figure 7.2: Structure of `Char`

not protected from causing an Operand Error, although other conversions of comparable variables in the same place in the code were protected.

Subdomains can model finite data types, as they are realized within computer systems. Thus, subdomains allow us to detect properties like overflow errors in calculations. We can avoid these errors with preserving functions and we can prove that such errors cannot occur. We define the type of 16-bit (unsigned[1]) integers as part of our library.

The library is a conservative extension of HOLCF. The structure of the library is depicted in the development graph in Figure 7.1. Conservative extensions require to extend HOLCF step by step. Therefore every box in the development consists of several small specifications, which directly base on each other. For example Figure 7.2 shows the structure of `Char`.

The library can be used as basis for further extensions. Two kinds of extensions are possible:

[1]Signed integers could also be introduced, but are omitted, since the methods are the same.

One is to model further data types and further functions, for example floating points. The other extension is to provide better proof support for the data types. In this work and in the library, we derived numerous theorems for the introduction and refinement of data types, however, for the efficient evaluation of expressions (for example 30*5=100+50), we would need even more tactics.

Recently in HOLCF a lifting constructor `lift` has been developed. It has the arity `lift::(term)pcpo`. This means that the constructor lifts arbitrary types from HOL to HOLCF domains. The resulting domains have flat structures. The advantage of this lifting is, that no admissible predicate is required to construct the domains, however for the construction of non-flat domains it cannot be applied. Since the development of interactive systems uses a lot of non-flat subdomains (see for example the domain of infinite streams on page 114) we need our `subdom` constructor in the development of distributed systems. Introducing the domain of natural numbers `dnat` with the `lift` constructor would allow us to lift the theorems from `nat` (in HOL) to `dnat` schematically[2] To show how our constructors are working we use them in this library.

There are many methodical aspects in the examples, concerning the use of the following HOLCF constructs.

- axiomatic type classes,
- conservative extensions,
- continuous functions,
- fixed point definitions,
- `domain` construct,
- `subdom` construct, and
- `quot` construct.

Since we presented the methods for the implementation in Chapter 5, and since it is not our goal to present all these methods in detail in this thesis, we just show how they are working on examples, and leave the collection of all HOLCF methods as a future work in Chapter 8.

As the library contains the basic ADTs, it uses the class `EQ` and the `domain` construct to define the (flat) data types. The flexible class `eq` is used in the case study to specify data types which should be implemented in terms of streams.

The following sections describe the specifications of the library. Appendix A.3 contains the full specifications with the derived properties.

[2]Reproving all arithmetic rules for + and ∗ on `dnat` took about one hour in the development of this library.

7.1.1 Natural Numbers

Natural numbers are needed in almost every case study. Since they can be easily defined with the `domain` construct they are not integrated into HOLCF. However, redefining the basic functions for every case study is not desired. Furthermore natural numbers are an important basis for the library of ADTs since many data types base on them.

The ADT of natural numbers is introduced with the `domain` construct by:

```
domain dnat = dzero | dsucc (dpred :: dnat)
```

The natural numbers are introduced by two steps, since they are instantiated into the class `EQ`. The first step is to define the general equality $\doteq\doteq$ on the type `dnat` by the definition of the equality on natural numbers. To derive the characteristic axioms of this equality we expand the fixed point from the definition and derive the axioms in the form of pattern matching. For example:⌈`dzero`$\doteq\doteq$`dzero`⌉. Having these rules, we can prove the characteristic axioms and instantiate the type `dnat` into the axiomatic type class `EQ` (see page 35 for axiomatic type classes, page 128 for the class `eq`, and Appendix A.3.1 for the complete theories and the theorems derived for natural numbers). Since we prefer to use $\doteq$ instead of $\doteq\doteq$ we derive the equality rules with $\doteq$ by simply expanding the definition of $\doteq$ on the class `eq` (see page 234).

With the example of natural numbers we show two different specification styles: The fixed point definitions of recursive functions and the definition with pattern matching without explicit use of the fixed point operator. The equality $\doteq\doteq$ and the function $\leq$ are defined with fixed points, the other functions like `add` and `mult` are defined with pattern matching. Both styles have their advantages. Defining a function by fixed points is a logical definition and we may use the defining equality $\equiv$ of Isabelle. Using $\equiv$ in the definition causes Isabelle to check, whether the definitions are conservative (for example no free variable may occur on the right side of a definition equation), however the fixed point definitions require to derive the pattern matching rules, since these rules are very helpful in theorem proving with the simplifier.

Defining a function with patterns (see Section 2.1.5) requires to ensure that the patterns do not overlap and, like in the definition, that no free variables occur. These restrictions are not checked by the Isabelle system (in its current version), however these checks are decidable and for example the code generator for SPECTRUM in [HR94, Hot95] checks these conditions. Sometimes, it is useful just to write axioms to specify functions abstractly, however using this specification style we have to be very careful that the specifications are not inconsistent. One example for such an abstract specification is the function `div`, which is only required to fulfil the axiom:

```
div1      n≠dzero ⟹ div`(n mult m)`n=m
```

We need this function only once, and therefore we do not give an algorithm for its definition, but we know that there is a function `div` which satisfies this axiom and therefore the specification is consistent.

Since natural numbers are frequently used in specifications we introduce some syntactical abbreviations for these numbers (for example `#2↑#0` for twenty). The syntax of these abbreviations is explained in [Pau94b].

For the definitions of the following specifications we also use the techniques presented on the specification of natural numbers. However, we focus more on other specification methods.

7.1.2 Lists

Lists are also an important specification element. They are defined polymorphically with the domain construct by:

```
domain α dlist = dnil | "##" (dhd::α) (dtl::α dlist) (cinfixr 65)
```

The annotation `(cinfixr 65)` denotes that `##` is a continuous infix operator with associates to the right with priority 65. All functions on `dlist` are defined with fixed point constructions. Witnesses for the instantiation of `dlist` into the type classes `eq` and `EQ` are proved, provided that the elements of the lists are of these classes.

7.1.3 Bytes

Bytes are included in the standard library for two reasons. One is that they are used in many case studies, especially in the implementations of communicating systems (see Example 6.4.1). The other reason is that they are used to demonstrate how larger domains can be introduced without the domain construct. Bytes base on bits, which are specified with the `domain` construct by:

```
domain Bit = L | H
```

Since `0` and `1` are natural numbers in HOL, we decided to use `L` and `H` for the representation of bits in HOLCF.

As was mentioned on page 50 the domain construct is a conservative definition of data types. This means that it derives all axioms of the data type (as described in Section 2.1.5). For example the distinctness of bits: `L` $\neq$ `H`. However, if we use the domain construct for larger data types it derives so many axioms that it takes very long to wait for the result. For example the following domain of bytes could not be introduced (within acceptable time) with the domain construct:

```
domain Byte=mkBy(b1::Bit)(b2::Bit)(b3::Bit)(b4::Bit)
               (b5::Bit)(b6::Bit)(b7::Bit)(b8::Bit)
```

The `subdom` constructor provides an elegant way to characterize such data types. With the `subdom` constructor we can model bytes as list of bits of length eight. This modelling can be easily extended to words of 32 or 64 bits.

Using the `subdom` constructor in a modular way requires to introduce embeddings (see Section 3.5). We use the following embedding for bytes:

```
Byte0 = Dlist + Bit + (* 8 bit bytes with embedding *)
types BitL = Bit dlist
domain B=Babs(Brep::BitL)
defs
        Badm_pred_def (adm_pred'::B⇒bool) ≡ λi.⌈dlen‘(Brep‘i)≐#8⌉
        (* defines equality and PER on B *)
        B_eq_def         (op ≐≐) ≡ Λx y.Brep‘x ≐ Brep‘y
        B_per_def        (op ∼∼) ≡ λx y::B.⌈x≐≐y⌉
end
```

Because the equality on `bit dlist` is lifted schematically to the type `B`, the proofs that this is an equality are trivial. With the equality rules for the `subdom` constructor (see Appendix A.2.10) we get the equality of bytes automatically. With the equality $\doteq$ on bytes we can easily deduce the distinctness, if needed. For example, if we need distinctness between two different bytes, we deduce it at the state in the proof where we need it. This is more efficient than to derive all axioms for the data type[3].

We use the additional type definition `BitL`, since the `domain` construct (in its current version) requires basic types on the selector positions.

7.1.4 Strings

Strings are an important part for modelling communication on a higher level of abstraction. Strings are modelled by lists of characters. Therefore we have to introduce characters first. In principle we could use the domain construct for the definition of characters, however as explained in the previous section we define characters as abstraction from a subdomain of natural numbers. We use the following embedding:

```
domain C=Cabs(Crep::dnat)
defs
        Cadm_pred_def adm_pred' ≡ λc.⌈Crep‘c ≤ #1 ↑ #2 ↑ #7⌉
```

[3]Another alternative currently under discussion is to liberate the `domain` construct from proving these schematically rules ever and ever again.

Since the restriction predicate is based on continuous functions its admissibility proof is trivial.

For the definitions of the functions on the subdomain we have two possibilities. One is to define them according to the methodical lifting schemata in Section 3.3.3 and to prove the invariance for their continuity. This style is supported with the lifting constructs for subdomains (see Section 3.5). The other form for the introduction of continuous functions is to define a continuous abstraction function. However, this bases on the fact that the admissible predicate, which characterizes the subdomain is monotone. We have to derive theorems like:

```
mono_Cadm_pred        ∀x y.x⊑y⟶adm_pred (Cabs‘x)⟶adm_pred (Cabs‘y)
```

Since the used domains are flat we can derive the continuity simply from monotonicity. Monotonicity is also easy to prove on flat domains.

Like for natural numbers we introduce syntactic abbreviations for characters, based on a continuous abstraction function `nat2chr`. This function allows us to define an encoding from natural numbers characters (We used ASCII encoding). Consider for example the definition `‡A≡nat2chr‘(#6 ↑ #5)`. Based on the specification of characters, strings can be easily defined as list of characters (see Appendix A.3.6).

7.1.5 Finite Numbers

Finite numbers are the first real[4] subdomain, which is needed in the development of interactive systems. An adequate modelling of finite numbers is necessary to detect overflow errors. We define positive numbers to be the natural numbers less or equal than $2^{16} - 1$. This allows us to use 16 bit words as representations. Finite numbers use the following embedding:

```
domain F=Fabs(Frep::dnat)
defs
Fadm_pred_def  adm_pred' ≡ λi.⌈Frep‘i≤(#2^#1↑#6) sub #1⌉
```

In the theory `Fin` (see page 250) we define a continuous abstraction function `nat2fin`. However, this function is not total, since the natural numbers greater than $2^{16} - 1$ are represented by $\perp$. The following definition of a non-preserving function defines an addition `Fadd` on finite numbers by:

```
Fadd_def        Fadd ≡ Λn m.nat2fin‘(fin2nat‘n add fin2nat‘m)
```

[4] It cannot be defined with the `domain` construct

This function is continuous, but not total. The reason for this undesired effect is that `add` is not preserving. However, with this function we can detect overflow errors, by proving for defined values `x` and `y` that `Fadd‘x‘y` is not defined. To avoid overflow errors we advocate the use of preserving functions (see Section 5.2.2 for a further discussion of invariance).

7.1.6 Rational Numbers

We choose fractional representations to introduce rational numbers. Positive rational numbers are pairs of natural numbers (numerator and denominator). The denominator must not be zero, and two fractions are equivalent, if they denote the same rational number (for example $\frac{1}{2} = \frac{2}{4}$). Since we restrict the denominator to non-zero numbers we have to build the subdomain of positive numbers. This is a usual subdomain and it is defined in the theory `Pos` (see Appendix A.3.8). The more interesting aspect is the quotient construction which identifies the fractions representing the same rational numbers. We obtain the following specification:

```
domain R = Rabs(num::dnat)(den::pos)
defs
R_eq_def        (op ≐≐) ≡ Λx y.num‘x mult pos2nat‘(den‘y) ≐ \
\                                 num‘y mult pos2nat‘(den‘x)
R_per_def       (op ∼∼) ≡ λx y::R.⌈x≐≐y⌉
```

In contrast to the previous embeddings this is not a canonical embedding. It uses a more complex definition of the equality between fractionals ($\frac{a}{b} \doteq \frac{c}{d} \equiv a * d \doteq c * b$). Proving the characteristic axioms for this equality was not as schematically as proving the characteristic axioms for canonical embeddings. Especially the transitivity rule for $\doteq\doteq$ requires to prove some arithmetical theorems for `dnat`. After the instantiation into the class `eq`, we can define the quotient construction and the fractional representations by:

```
types Rat = R quot
ops curried
        "−−" :: dnat → dnat → Rat (cinfixl 90)
defs
        fract_def       (op −−) ≡ Λx y.<⌈Rabs‘x‘(nat2pos‘y)⌉>
```

We also derived some rules to compute this equality (see Appendix A.3.9). Since the domains are flat, it suffices to prove monotonicity to ensure continuity . For monotonicity of the fractional representations, we used the theorem `monofun_class` (see Appendix A.2.10).

Future case studies will define additional ADTs, and functions, and will derive further useful theorems, which can be integrated into this library. However, the presentation of the

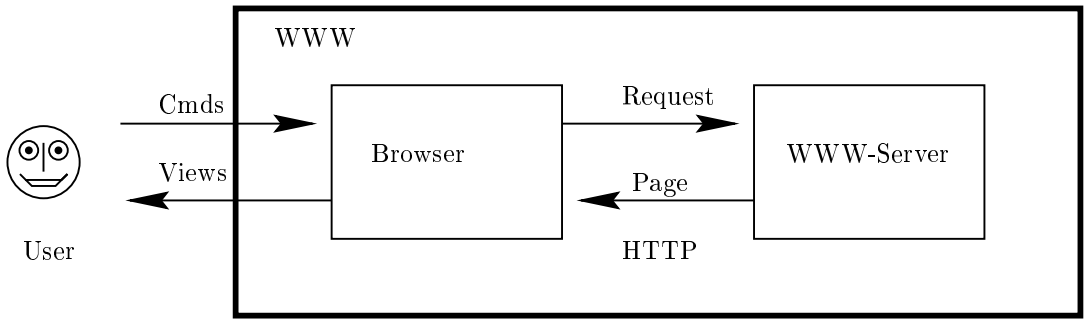

Figure 7.3: Structure of the System

standard library demonstrated some HOLCF methods, and it showed how to use the type constructors `subdom` and `quot` for the introduction of new data types. The specifications of the critical aspects of the WWW server in the following sections base on this library of ADTs for HOLCF.

7.2 Structure and Critical Aspects of the WWW

We decided to use implementations of a WWW server a case study, because the WWW is quite well known and because it is of increasing importance since the number of WWW servers doubles approximately every year [Pax94]. Since the WWW is well known we do not have to explain terms like *URL*, *HTTP* or *html-page* (see for example [Tan96]). A WWW server has many functions and requirements. It is not our goal to implement a complete WWW server, but we concentrate on some of the critical aspects of the requirement specifications.

According to the KorSys process model [SM96], for the verification of critical aspects in large systems, we choose some critical aspects of the system and formalize only the relevant parts. This formalization is the basis for the verification of the critical aspects of the system. The advantage is that we do not have to formalize the complete system, but only those parts which are relevant for the critical properties.

This section presents the structure of the WWW system and some critical aspects, which have to be ensured by the implementation of the system. The following sections show how the refinement relations in HOLCF and the implementation methods can be used to verify the critical aspects of the WWW server.

The structure of the WWW server is depicted in the structure diagram in Figure 7.3. It shows how a user can interact with the WWW. The user enters commands into the browser, which transmits requests to a WWW server. We do not model the internet with more than one users and more than one servers, since this it no relevant for our case study.

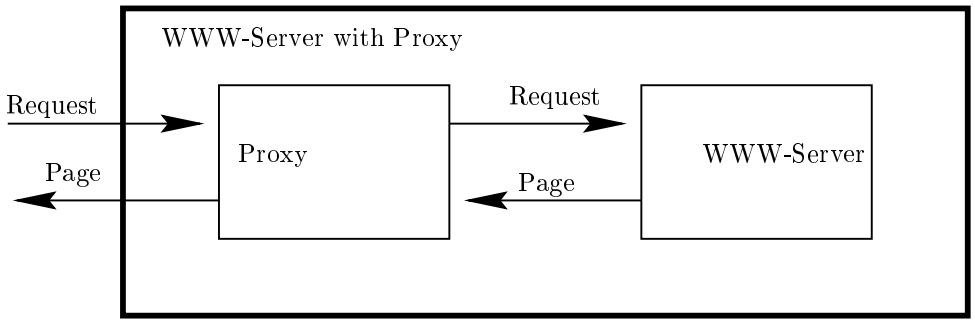

Figure 7.4: Cache in the WWW

The browser offers two features to the user: to surf to a certain address in the WWW, i.e. to ask for a html page with a certain URL and to create html pages and to put them into the net at a certain URL.

A further refinement in the WWW are proxy servers. Proxy servers are the caches in the WWW. Proxy servers can be introduced by structural development of the WWW server. A WWW server with a proxy server is a distributed system, depicted in Figure 7.4.

Proxy servers are very important in the WWW. However, since a WWW server with proxy has the same interface as a WWW server without proxy it is a simple structural development step. Thus, this step can be proved with behavioural refinement, for which many case studies have already been carried through [BFG+94]. Since a proxy is like a cache in a sequential system, the verification of the correctness of the proxy is like a sequential correctness proof of a cache. Therefore, we do not focus on this implementation step and concentrate on more difficult critical aspects of the WWW, which show how the implementation of ADTs can be used to verify critical aspects of a WWW system.

One critical aspect is that the WWW server does not lose stored pages. To verify this property we formalize the server. We use a state-based specification with a data base state. From this specification it can be easily derived that the pages are not lost in the WWW server. The implementation of this server has to ensure this property. Since the used refinement relation is modular, we have only to show that the implementation of the state-based specification is correct. This is done in Section 7.3.

Another critical aspect of the WWW is the transmission of messages. This aspect is closely related with the different abstraction levels in the specification of protocols. Since all URLs, requests, and html pages are encoded into strings, we focus on a correct transmission of strings. In Section 7.4 we implement the string transmission on the basis of a packet transmission, which is realized in internet protocols (see [Tan96]).

7.3 Database of a WWW Server

This section formalizes the functions (and the data types) of a WWW server, which are required to prove the critical aspect that the server does not lose stored informations. We give a state-based specification of the server in Figure 7.3 and the data types `Request` and `Page` of the interface. For the data state of this specification we specify a small database.

We implement the server with a stream processing functions which stores the received messages (like the buffer in Example 4.1.1). In the specification of the database we use $\sim$ instead of $=$ and formalize observer functions to characterize the congruence (See Section 4.2 for the definition of $\sim$ and Section 4.5 for the advantages of this specification style).

We start with a specification of the requests and assume that the specification `HTTP` contains the subdomains `Page` and `url` of strings.

```
REQUEST = HTTP +
domain Request = get(req_url::url)
               | put(put_url::url) (put_cont::Page)
arities Request::eq
end
```

Of course, we could have modelled `Request` also as a subdomain of streams, however for readability, we use the domain construct since it introduces all functions of the ADT in a compact way (see page 50). This specification is the basis for our small database. It contains only pages with the URLs as key. It is specified without the domain construct in order to allow an implementation based on streams without quotients.

```
DB = REQUEST +      (* Specification of the Data Base *)
types Db 0
arities Db::eq

(* the following operations are available on Db *)
ops curried strict
(* constructors *)
        emptyDb :: Db
        addDb   :: Db → url → Page → Db
(* selectors *)
        key     :: Db → url
        content :: Db → Page
        restDb  :: Db → Db
(* discriminators *)
        is_emptyDb :: Db → tr
        is_addDb:: Db → tr
```

```
(* further operations on databases *)
        isinDb  :: Db → url → tr
        find    :: Db → url → Page
        replace :: Db → url → Page → Db
        insert  :: Db → url → Page → Db
generated Db by emptyDb | addDb
axioms
defvars db u p db2 u2 p2 in
        (* discriminator rules *)
is_emptyDb1     ⌈is_emptyDb‘emptyDb⌉
is_emptyDb2     ⌊is_emptyDb‘(addDb‘db‘u‘p)⌉
is_addDb1       ⌈is_addDb‘(addDb‘db‘u‘p)⌉
is_addDb2       ⌊is_addDb‘emptyDb⌋
        (* selector rules *)
key1            key‘(addDb‘db‘u‘p)=u
content1        content‘(addDb‘db‘u‘p)=p
restDb1         restDb‘(addDb‘db‘u‘p)∼db
        (* weakened injectivity *)
injective       addDb‘db‘u‘p∼addDb‘db2‘u2‘p2⟹u=u2∧p=p2∧db∼db2
        (* observability to characterize ≐ *)
obs_is_emptyDb  is_Cobs is_emptyDb
obs_is_addDb    is_Cobs is_addDb
obs_addDb       is_Cobs addDb
obs_key         is_Cobs key
obs_content     is_Cobs content
obs_restDb      is_Cobs restDb
        (* rules for db functions *)
isinDb1         ⌊isinDb‘emptyDb‘u⌋
isinDb2         isinDb‘(addDb‘db‘u2‘p)‘u =
                     u≐u2 orelse isinDb‘db‘u
find1           find‘emptyDb‘u = errorPage
find2           find‘(addDb‘db‘u2‘p)‘u =
                     If u≐u2 then p else find‘db‘u fi
replace1        replace‘emptyDb‘u‘p ∼ emptyDb
replace2        replace‘(addDb‘db2‘u2‘p2)‘u‘p ∼
                     If u≐u2 then addDb‘db2‘u‘p
                     else addDb‘(replace‘db2‘u‘p)‘u2‘p2 fi
insert1         insert‘db‘u‘p ∼
                     If isinDb‘db‘u‘p
                     then replace‘db‘u‘p
                     else addDb‘db‘u‘p fi
end
```

The state-based specification of the WWW server is:

```
WWW_SERVER = DB + Stream +
ops curried strict
        server   :: Request stream → Page stream
        DBserver :: Db → Request stream → Page stream
rules

server_def      server ≡ DBserver‘emptyDb"
DBserver1       DBserver‘DB‘(get‘u)&&s = find‘DB‘u && DBserver‘DB‘s
DBserver2       DBserver‘DB‘(put‘u‘p)&&s = stored_Page &&
                                        DBserver‘(insert‘DB‘u‘p)‘s
end
```

This specification allows us to use the model inclusion basis for the functional refinement of the specification states and to give a refinement of states in terms of histories of request streams. This refinement does not use quotients and is therefore executable in the sense of our Definition 2.1.13 on page 51. Since the model refinement relation of the model inclusion basis is modular, we do not have to reprove our critical aspect for the executable implementation. It suffices to show that the DB specification is implemented correctly.

The critical property that the server does not lose pages, unless they are replaced by more recent ones is formalized by:

```
∀rs::Request stream. ∀i:nat. let r=ith(i,rs) in
      if ⌈is_get‘r⌉
      then ith(i,server‘rs)=last_page(i,req_url‘r,rs)
      else true
```

We do not go further into details of this formalization, since the implementation of the WWW server preserves all predicates, which hold for the requirement specification WWW_Server and we assume the predicate to be true for this case study[5]. If we refine the specification DB with a modular refinement relation, then this property is also true for the specification WWW_SERVER (see page 24 for a definition of modularity). The structure of the development is visualized in Figure 7.5.

We specified the database with the PER $\sim$, such that we can apply the behavioural refinement relation of HOLCF, which is modular, to the implementation of the database and therefore we do not need to reprove the critical aspect. The only remaining proof obligation is that the database is refined behaviourally by the database specification which bases on histories of streams.

[5]Of course this proof is a main part of the correct development of the system, but since a lot of those proofs have been carried through in FOCUS, we do not present here another case study. The interesting aspect of our implementation is that it preserves the properties since it is modular.

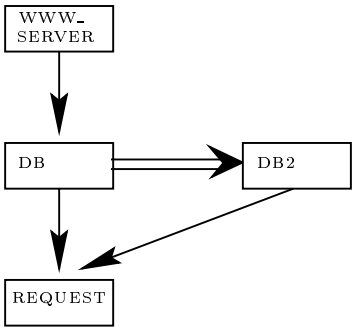

Figure 7.5: Development of the WWW Server

```
DB2 = REQUEST + Stream +
        (* embedding for histories *)
domain Db = Habs (Hrep:: Request stream)
ops curried strict
        (* collects all URLs in a stream *)
        collect_urls :: Request stream → url dlist
rules
        (* ignores get−requests *)
collect     collect_urls‘s =
               If is_&&‘s
               then If is_put‘(ft‘s)
                    then insert_dl‘(put_url‘(ft‘s))‘
                           (collect_urls‘(rt‘s))
                    else collect_urls‘(rt‘s) fi
               else dnil fi
defs
        (* equality on histories *)
Db_eq_def    x≐≐y=collect_urls‘(Hrep‘x)≐collect_urls‘(Hrep‘y)
Db_PER_def   x∼∼(y::Db)=⌈x≐≐y⌉
        (* instance Db into the class eq *)
instance Db::eq    (* witnesses for eq are needed here *)
        (* now define the database operations *)
ops curried strict
(* constructors *)
        emptyDb :: Db
        addDb   :: Db → url → Page → Db
(* selectors *)
        key     :: Db → url
        content :: Db → Page
```

```
        restDb  :: Db → Db
(* discriminators *)
        is_emptyDb :: Db → tr
        is_addDb:: Db → tr
(* further operations on databases *)
        isinDb  :: Db → url → tr
        find    :: Db → url → Page
        replace :: Db → url → Page → Db
        insert  :: Db → url → Page → Db
axioms
defvars db u p db2 u2 p2 in
        (* discriminator rules *)
is_empty_def    is_emptyDb ≡ Habs‘(get‘def_url)
is_addDb_def    is_addDb ≡ Λdb u p.Habs‘((put‘u‘p)&&Hrep‘db)
        (* selector rules *)
key_def         key ≡ Λdb.put_url‘(ft‘(Hrep‘db))
content_def     content ≡ Λdb.put_cont‘(ft‘(Hrep‘db))
restDb_def      restDb ≡ Λdb.Habs‘(rt‘(Hrep‘db))
        (* identical rules for other db functions *)
isinDb1         ⌊isinDb‘emptyDb‘u⌋
isinDb2         isinDb‘(addDb‘db‘u2‘p)‘u =
                     u≐u2 orelse isinDb‘db‘u
find1           find‘emptyDb‘u = errorPage
find2           find‘(addDb‘db‘u2‘p)‘u =
                     If u≐u2 then p else find‘db‘u fi
replace1        replace‘emptyDb‘u‘p ∼ emptyDb
replace2        replace‘(addDb‘u2‘p2‘db2)‘u‘p ∼
                     If u≐u2 then addDb‘u‘p‘db2
                     else addDb‘(replace‘db2‘u‘p)‘u2‘p2 fi
insert1         insert‘db‘u‘p ∼
                     If isinDb‘db‘u‘p
                     then replace‘db‘u‘p
                     else addDb‘db‘u‘p fi
end
```

The implementation of the database, based on histories of request streams is a generalization of the implementation of buffer of length one (see Section 4.1). The implementation of the database requires to handle histories of finite length. Histories are those messages which have already been processed by the server. Since the server starts with an empty history all histories are finite. We could also implement the database by "histories" of length one, if we take lists or databases as messages, however this is not desired, since these "histories" do not consist of the input values and therefore they would be rather strange histories.

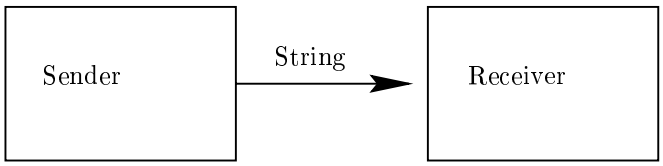

Figure 7.6: Communication with Strings

The proof that DB2 is a refinement of DB requires to derive all axioms for the requirement specification DB, including the observability axioms. Since we used $\sim$ instead of $=$ we can derive these axioms.

7.4 Transmission of Strings

The second critical aspect in our WWW server is the transmission of strings in our system. We represent html pages and URLs as strings and the HTTP protocol [Tan96] uses also special strings, which can be modelled by subdomains, for communication. Therefore it is important to ensure that strings are transmitted correctly.

In this section we use an individual translation between strings and packets, which is only correct for some strings. This restriction is ensured since we develop the system by a dialog development step. We refine the communication channel (with the general string communication) together with both components communicating over this channel into two components for which the restricted communication channel suffices.

In our system diagram on page 206 we have two communication channels between the browser and the WWW server. Since both channels can be implemented by special string processing channels, we concentrate on a string transmission from a sender to a receiver. This situation is depicted in Figure 7.6.

Since communication channels in general do not allow us to transmit strings of arbitrary length on the internet, we have to send strings in smaller units. It would be easy to identify a string-end character and to transmit every string by transmitting every character of the string separately, followed by the string-end character. This could be implemented like the implementation of bytes by bits on page 193 on a more hardware oriented level. However, since the internet has no direct channels between all browsers and servers, the messages on the internet have a header, which contains information about their target address. Sending every character separately would require to add such a message header to every single character. Since this would be very inefficient, messages have usually a maximum size between 1000 and 1500 bytes. Those messages are called packets in the internet.

The strings containing html pages (and URLs) are not restricted in their length. Especially they can be so long that they do not fit into one packet. This requires to split them into

several packets. These packets are sent one by one from the sender to the receiver and the receiver puts them together to the original string. Since in the internet packets do not necessarily arrive in the order in which they are sent, each packet header contains an finite (unsigned) integer number describing the position of the packet in the stream.

The first step is to formalize the notion of packets. We use pairs of finite numbers and strings with length less (or equal) than 1024.

```
STRING1024 = String2 +
(* embedding *)
domain S = Sabs (Srep::String)
defs
S_adm_pred_def  adm_pred' ≡ λs.⌈strlen‘(Srep‘s)≤#1↑#0↑#2↑#4⌉
        (* admissibility of S_adm_pred_def is proved
           by giving an explicit tactic *)
instance S::adm (S_adm) {| ((rewrite_goals_tac [S_adm_pred_def])
                            THEN (adm_cont_tacR 1)) |}
(* now define the subdomain *)
types S1024 = S subdom
end
```

The specification `String2` is an extension of `String` from the library (see page 249) by some specific functions used for packets (`head,tail::dnat→String→String`). With the type `S1024` of short string, we can define the type of packets by:

```
PACKET = STRING1024 + Fin +
domain Packet = mkP (header::Fin) (content::S1024)
```

The other aspects of the header are ignored here, since we have no addresses in our system. For packets we formalize two functions `split` and `combine`, which split a string into packets, and combine the packets to a string. They are specified by:

```
ops curried strict
       split   :: String → Packet dlist
       combine :: Packet dlist → String
       combine2:: Packet dlist → String (* auxiliary function *)
rules
split_def      split ≡ Λs. If strlen‘s≤#1↑#0↑#2↑#4
                          then mkP‘#0‘(Sabs‘s) ## dnil
                          else mkP‘(nat2int‘(div‘(strlen‘s)‘#1↑#0↑#2↑#4))‘
                                    (Sabs‘(head‘#1↑#0↑#2↑#4‘s))
                               ## split‘(tail‘#1↑#0↑#2↑#4‘s)
```

```
combine_def    combine ≡ Λl.combine2‘(sort‘l)
combine2_def   combine2 ≡ Λl.If is_dnil‘l then emptySt
                              else concat_dl‘(Srep‘(content‘(dhd‘l)))‘
                                             combine2‘(dtl‘s)
```

The following theorems hold in this theory:

```
combine_split      ⌈strlen‘s ≤ #1↑#0↑#2↑#4 mult (#2^#1↑#6 sub #1)⌉
                     ⟹ combine‘(split‘s)=s
split_combine      split‘(combine‘l)=l
```

Since `combine` is based on finite numbers and since every string in the packets has a length less or equal than 1024 we cannot prove the rule `combine_split` for arbitrary strings but for strings with length smaller than $1024 * (2^{16} - 1)$. This allows us to transmit strings up to a size of about 64 mega bytes. For many applications this might suffice, but since multimedia applications will become more important this limit could be exceeded sometimes. Exceeding this limit would lead to an integer overflow and we cannot transmit the string correctly. Therefore we have to restrict the size of our html pages to 64 MB or to use another encoding into packets.

We focus now on the task of string transmission with packets. For simplification we concentrate on the case that the messages arrive in the order in which they are sent (otherwise we would need an extra string-end packet and additional informations on the length of the string). This simplification allows us to use the packet numbers to indicate the end of a string. We do this by sending the packet with the highest number first, and then decrease the number in each packet by one. The packet with the number zero contains the last packet of the string. This trick works, since the messages in our channel arrive in the same order as they are sent.

The implementation of a channel with lists of packets as messages by a channel with packets as messages is an individual implementation (see Section 6.4). We define the abstraction and representation functions explicitly and then show that they are inverse (like in Example 6.4.1 on page 193).

```
consts
        absPP   :: packet stream → string stream
        absPPL  :: packet stream → dlist packet →
                    dlist packet stream
        repPP   :: string stream → packet stream
        repPPL  :: packet dlist → string stream → packet stream
axioms
defvars n p l s in
absPP_def     absPP‘s = absPPL‘dnil‘s
```

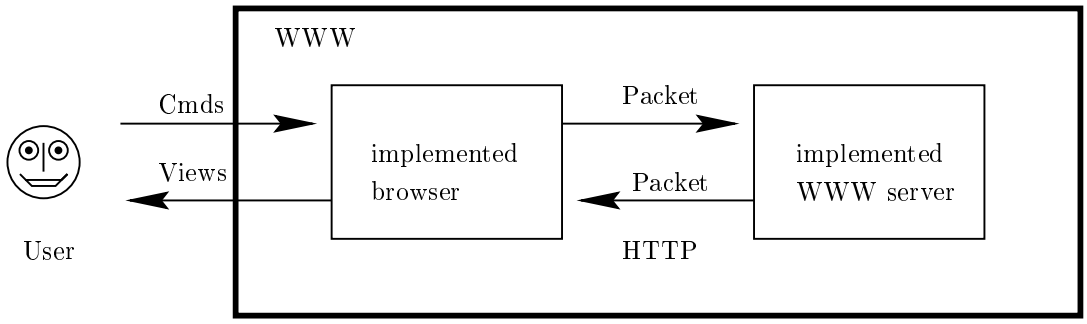

Figure 7.7: Communication with Packets

```
absPPL1       absPPL‘l‘(mkP‘#0‘p && s) =
                combine‘((mkP‘#0‘p)##l) && absPP‘s
absPPL2       absPPL‘l‘(mkP‘(Isucc‘n)‘p && s) =
                absPPL‘(mkP‘(Isucc‘n)‘p ## l)‘s

repPP_def     repPP‘x&&s = repPPL‘(split‘x)‘s
repPPL1       repPPL‘dnil‘s = repPP‘s
repPPL2       repPPL‘(p##l)‘s = p && repPPL‘l‘s
```

With the same proof techniques as in Example 6.4.1 we derive the following theorems:

```
∀s∈ps.⌈strlen‘s⌉≤#1↑#0↑#2↑#4 mult (#2^#1↑#6 sub #1)⌉
    ⟹  repPP‘(absPP‘ps)=ps
absPP‘(repPP‘lps)=lps
```

The restriction comes from the restriction of splitting strings into lists of packets with finite numbers. In Example 6.4.1 a subdomain is constructed for the representing elements. We omit this step here to have the restriction explicitly.

We cannot implement strings of arbitrary length with such a packet protocol. For our implementation of the channels between the WWW server and the browser this means that both components have to ensure that the strings containing the pages are not greater than 64 MB. This is a typical situation in the development of interactive systems: we can implement a channel in an efficient way, but only if we know that it is used correctly. Formally this is a dialog development (see Sections 1.2.4 and 6.3.4), i.e. we refine the system consisting of sender, receiver and the communication channel together. The refinement does not affect the external interfaces, but allows us to use downward simulations (see Definition 6.3.3) in the system, without changing the interface to the environment.

Out implementation of the WWW server is depicted in Figure 7.7. It does not change the user interface, but the browser ensures that the strings, which are split into packets are not greater than 64 MB. This involves especially that the functions working on the strings are preserving. We could define a concrete subdomain and proceed as described in Section 6.3.3.2. The resulting proof obligations would be simple behavioural refinements and are not presented here.

In our case only the browser is the critical part of the system, since the server does only store the arriving strings. If the browser does not produce large strings (pages), the server will not need to send them. One possible implementation would be to divide large pages and to store them at different URLs.

With the formalizations and implementations of the state correctness and with the correct string transmission we have demonstrated, even without presenting the formal proofs, how large interactive and distributed systems can be implemented with the methods from the implementation of ADTs in HOLCF.

Chapter 8

Future Work

This chapter describes possible extensions of our work, which are of theoretical and practical interest.

A theoretical challenge is to find out whether there exists a type class between `percpo` and `eq` which ensures the composition of observer functions and allows us to instantiate arbitrary streams. This could be the basis for a simple theory interpretation which does not require an equality $\doteq$ on the concrete type. This would allow us to use arbitrary streams instead of histories as representations of abstract types for theory interpretations. However, since our class `eq` is not restricted to flat domains we can instantiate streams to it and for the quotient construction a PER on the data type suffices. Since PERs are available on functions and streams, finding another class between `percpo` and `eq` is a theoretical challenge.

To implement a domain construct `Φdomain`, which specifies an ADT with constructor, selector and discriminator functions and uses $\sim$ instead of $=$ would provide us with a nice shorthand for specification of ADTs with observability constraints. Using this construct would make theory interpretation superfluous in the deductive software development process, and we could use the modular model inclusion basis as refinement relation. The implementation of the `Φdomain` construct is only a small modification of the `domain` construct, especially since the `Φdomain` construct is a specification of ADTs, it would not require to derive the axioms for the specification of the ADT.

A more practical extension of this work would be to extend our definition of executability to a definition which allows us to execute quotients. This could be done by defining an operational semantics for HOLCF. An executable quotient construction would lead to nice simplifications of the refinement relations, since it would make theory interpretation superfluous. This extension should also enclose the aspect of code generation, since a correct code generation is an important part in the deductive software development process. Since functional languages (like ML, Gofer, or Erlang) are very similar to our functional specifications, the correctness of a functional target language is easier to ensure. The

language Erlang [AVW91] is a functional language for distributed systems which could be used for code generation. Since the WWW is a growing interactive system we used it as case study. The most popular language in the WWW is Java [Fla95]. Java would also be a candidate for a target language of the code generator.

Since graphical description techniques are useful in the informal specification and development of interactive systems, they should also be integrated into the deductive software development process. FOCUS uses system diagrams to describe the structure of distributed systems. Integrating a graphical description technique into the deductive software development process requires to have formal semantics for it. The system diagrams have a semantics in HOLCF [SS95]. There are many other graphical description techniques for interactive systems (see Section 1.2). An important one are state-transition diagrams, since they can be used to graphically describe state-based systems.

To refine systems at the level of graphical description techniques, we do not only need a formal semantics, but also a refinement relation, which allows us to express refinements at the graphical level. For example inserting a new state into an automaton is a refinement step. Since our work provides refinements in HOLCF it is the basis for defining refinements of graphical description techniques, that have a formal semantics in HOLCF.

A very interesting aspect is to use the domains to give tool support for further theories in Isabelle and HOLCF. For example we could define the domains of strict and continuous functions or pulse driven functions as subdomains of the function space. The quotient constructor can be used to define quotients of streams, for example for the abstraction from timed streams to untimed streams.

Prototyping and simulation are extensions of high practical relevance of the implementation of ADT. They can be achieved by using executable abstraction and representations functions for a simulation between different levels of abstraction in the description of systems. For example we could define a simulation of a state-based component, which is realized by a distributed system, or a simulation of an abstract communication channel which is realized by a hardware communication channel and fulfils a certain protocol. The AUTOFOCUS tool [HSS96] could benefit from our basis for simulation and prototyping.

A further extension, important for the acceptance of formal methods in industry, is a detailed process model for the development of interactive systems. In the project KORSYS a coarse process model has been developed [SM96]. It requires to formalize only those parts of the system, which are relevant for proving the correctness of the critical aspects. The extension to a more detailed process model should also contain a detailed description of all methods available in HOLCF (including those not presented formally in this work).

The implementation of ADTs is an important basis for a formal software development framework. For example the thesis of W. Reif [Rei80] handles the implementation of ADTs in the dynamic logic. This work is the basis for the successful KIV system [Rei92]. The logic in which the implementation of ADT is solved determines the power of the deductive

software development framework. The dynamic logic of the KIV system is well suited for a development of sequential systems.

Since HOLCF is an adequate logic for the specification of interactive and distributed systems and since our work solves the implementation of ADTs in HOLCF, this thesis is the basis for a development framework for distributed systems.

Appendix A

HOLCF Extensions

The structure of the theory theories of ADTs in HOLCF is depicted in Figure A.1. Since the theory `EQ` contains all other theories for the implementation of ADT in HOLCF the theory ADT is simply defined by:

```
ADT = EQ
```

The subdomains are used in the theory `EQ`, since the instance `subdom::(eq)eq` is proved for the class `eq`.

The following sections describe the different theories.

A.1 The `subdom` constructor

This section contains the theories and the derived theorems for the `subdom` type constructor. The realization in HOLCF is described in Section 3.5, the method is described in Section 3.3.

A.1.1 ADM0

Two step introduction (see Section 2.1.2) of the type class `adm` with the characteristic constant `adm_pred`.

```
ADM0 = HOLCF +   (* adm is the class with admissible predicates *)

consts           (* general constant *)
        adm_pred'    :: α::term ⇒ bool
```

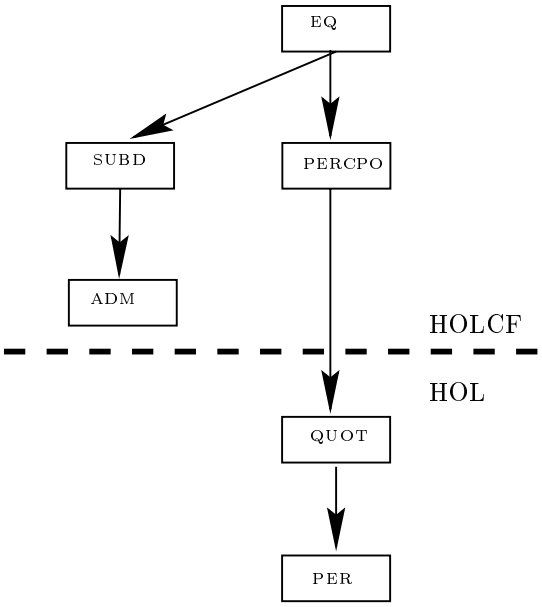

Figure A.1: Structure of ADTs in HOLCF

```
axclass adm<pcpo
        ax_adm_adm_pred  adm adm_pred'

consts            (* characteristic constant for adm *)
        adm_pred      :: α::adm ⇒ bool
defs    adm_pred_def      adm_pred ≡ adm_pred'
                  (* adm_pred for void *)
        adm_pred'_void    adm_pred'::(void⇒bool)≡λx.True
end
```

The type `void` is instantiated into `adm` to show that the class is not empty. Theorems for `ADM0`:

```
          (* characteristic axiom for the characteristic constant *)
adm_adm_pred      adm adm_pred
          (* admissibility for void *)
adm_void          adm (adm_pred'::void⇒bool)
```

A.1.2 ADM

Safe instantiation for `void` into `adm`.

```
ADM = ADM0 + (* instantiates void into adm *)
instance
        void::adm       {| (rtac adm_void 1) |}
end
```

Theorem for ADM:

```
inst_adm_void    adm_pred = (λx::void. True)
```

A.1.3 SUBD0

Introduces the subdom type constructor. The functions τabs, τrep and the restriction predicate p are called abs_sd, rep_sd and cor_sd in HOLCF. Val_sd contains the corresponding elements $\hat{\sigma}$. The introduction of the new type is described also on page 41. The following theories provide additional arities for subdom:

```
SUBD0 = ADM +
types   subdom 1
arities subdom :: (adm)term
consts
        cor_sd        :: α::adm ⇒ bool
        Val_sd        :: α::adm set
        rep_sd        :: α::adm subdom ⇒ α
        abs_sd        :: α::adm ⇒ α subdom
defs
        cor_sd_def       cor_sd f ≡ adm_pred f ∨ f=⊥
        Val_sd_def       Val_sd ≡ {f.cor_sd f}
rules
        rep_Val_sd       rep_sd t ∈ Val_sd
        abs_rep_sd       abs_sd(rep_sd t) = t
        rep_abs_sd       s ∈ Val_sd ⟹ rep_sd(abs_sd s) = s
end
```

Theorems for SUBD0:

```
        (* first show that subdom is not empty *)
cor_sd_UU        cor_sd ⊥
UU_in_Val_sd     ⊥∈Val_sd
not_empty_sd     ∃ x. x∈Val_sd
        (* prove the admissibility of the predicate cor_sd *)
adm_cor_sd       adm (λx.cor_sd x)
```

```
        (* now some general lemmas for subdom and predicates *)
cor_sd2Val_sd    cor_sd x⟹x∈Val_sd
adm_pred2Val_sd  adm_pred x⟹x∈Val_sd
cor_sd_rep_sd    cor_sd (rep_sd x)
cor_sdD          cor_sd x⟹adm_pred x ∨ x=⊥
        (* induction rule for subdomains *)
all_sd           ∀s.cor_sd s ⟶ P (abs_sd s) ⟹ P x
```

A.1.4 SUBD1

This theory defines a partial order on α `subdom`, based on the order of α.

```
SUBD1 = SUBD0 +          (* add an order *)
consts
        less_sd     :: α::adm subdom ⇒ α subdom ⇒ bool
defs
        less_sd_def    less_sd ≡ λa.λb.rep_sd a ⊑ rep_sd b
end
```

Theorems for `SUBD1`:

```
        (* show that less_sd is a partial order *)
refl_less_sd     less_sd a a
antisym_less_sd  ⟦less_sd f1 f2; less_sd f2 f1⟧ ⟹ f1 = f2
trans_less_sd    ⟦less_sd a b; less_sd b c⟧ ⟹ less_sd a c
```

A.1.5 SUBD2

This theory instantiates `subdom` into the class `po` and defines the least element for the *cpo* construction.

```
SUBD2 = SUBD1 +
arities  subdom :: (adm)po
rules
        inst_sd_po       ((op <<)::[α::adm subdom,α subdom]=>bool)=less_sd
consts
        UU_sd         :: α::adm subdom
defs
        UU_sd_def        UU_sd ≡ abs_sd ⊥
end
```

Theorems proved for SUBD2 (see page 101 for the proof of the axiom lub_sd):

```
        (* minimal *)
minimal_sd      UU_sd ⊑ x
        (* some theorems for the order *)
less_sd         p ⊑ q = rep_sd p ⊑ rep_sd q
sd_eq           rep_sd a = rep_sd b ⟹ a=b
abs_sd_less     ⟦x∈Val_sd; y∈Val_sd; x ⊑ y⟧⟹abs_sd x ⊑ abs_sd y
monofun_rep_sd  monofun rep_sd
is_chain_rep_sd is_chain C ⟹ is_chain(λj.rep_sd(C j))
        (* proof of cpo *)
lub_sd          is_chain C ⟹ range C <<| abs_sd (⨆i.rep_sd (C i))
cpo_sd          is_chain C ⟹ ∃ a::α::adm subdom.range C <<| a
```

A.1.6 SUBD

This theory instantiates subdom into the class pcpo and defines invariance and lifting for the simple case of the general construction schemes.

```
SUBD = SUBD2 +
arities subdom :: (adm) pcpo
rules
        inst_sd_pcpo    (⊥::α::adm subdom) = UU_sd
(* for some special functions on subdomains there is a lifting *)
consts
        inv     :: (α::adm → β::adm) ⇒ bool
        sd_lift0:: (α::adm → β::adm) ⇒ α subdom ⇒ β subdom
        sd_lift :: (α::adm → β::adm) ⇒ α subdom → β subdom
defs
        inv_def         inv f ≡ ∀x.cor_sd x⟶cor_sd(f`x)
        sd_lift0_def    sd_lift0 f ≡ λx.abs_sd (f`(rep_sd x))
        sd_lift_def     sd_lift f ≡ Λx.sd_lift0 f x
end
```

Theorems proved for SUBD:

```
inst_sd_pcpo2   ⊥=abs_sd ⊥
        (* strictness and totality for rep_sd and abs_sd *)
rep_sd_UU       rep_sd ⊥=⊥
abs_sd_UU       abs_sd ⊥=⊥
rep_sd_total    x≠⊥⟹rep_sd x≠⊥
```

```
rep_sd_total2    rep_sd x=⊥⟹x=⊥
abs_sd_total     ⟦s≠⊥;cor_sd s⟧ ⟹ abs_sd s≠⊥
        (* exhaustiveness (induction all_sd is in SUBD0) *)
exhaust_sd       x=⊥ ∨ (∃s.cor_sd s ∧ x=abs_sd s)
        (* a theorem on flatness *)
flat2flat_sd     flat(x::α::adm)⟹flat(y::α subdom)
        (* continuity proofs and invariance *)
cont_rep_sd      cont rep_sd
inv2contcont     ⟦!x.x∈Val_sd ⟶ F x∈Val_sd;cont F⟧
                 ⟹ cont(λs.abs_sd (F (rep_sd s)))
inv_def2         inv f=(∀x. cor_sd x ⟶ cor_sd (f`x))
invI             ∀x. cor_sd x ⟶ cor_sd (f`x) ⟹ inv f
invE             inv f ⟹ ∀x. cor_sd x ⟶ cor_sd (f`x)
         (* further theorems for invariance proofs *)
inv_If           ⟦inv f;inv g;cont B⟧⟹inv(Λx.If B x then f`x else g`x fi)
inv_oo           ⟦inv f;inv g⟧ ⟹ inv (f oo g)
inv2contcfun     inv f ⟹ cont(λs.abs_sd (f`(rep_sd s)))
         (* now theorems for the lifting *)
inv2cont_lift    inv f⟹cont (sd_lift0 f)
all_sd_lift      ⟦inv f;∀x.cor_sd x⟶P(abs_sd(f`x))⟧⟹P((sd_lift f)`x)
sd_lift_app      inv f⟹(sd_lift f)`x=abs_sd(f`(rep_sd x))
sd_lift_UU       sd_lift ⊥=⊥
sd_lift0_def2    sd_lift0 f = (λx. abs_sd (f`(rep_sd x)))
beta_lift0       inv f⟹(Λx.sd_lift0 f x)`x = sd_lift0 f x
```

Future case studies will reuse these theorems and provide further lemmata. Thus the collection of theorems available for the implementation is growing to the benefit of the deductive software development process.

A.2 The quot Constructor

This section describes the realization of the quotient constructor, together with the type class of partial equivalence classes. It contains theories and theorems of the realization of the concepts of Section 4.2.

A.2.1 PER0

This theory defines a class `per` for PERs and uses axiomatic type classes (see Section 2.1.2). It extends the HOL theory of sets.

```
PER0 = Set + (* axclass per with characteristic constant ∼ *)
consts
        "∼∼"    :: α ⇒ α ⇒ bool (infixl 55)
axclass
        per < term
                (* characteristic axioms *)
        ax_sym_per2     x ∼∼ y ⟶ y ∼∼ x
        ax_trans_per2   x ∼∼ y ∧ y ∼∼ z ⟶ x ∼∼ z
consts
                (* characteristic constant *)
        "∼"     :: α::per ⇒ α ⇒ bool (infixl 55)
                (* Domain *)
        D       :: α::per set
defs
        ax_per_def      (op ∼) ≡ (op ∼∼)
        Domain          D≡{x.x∼x}
                (* define ∼∼ on bool and fun *)
        bool_per        ((op ∼∼)::[bool,bool]⇒bool) ≡ (op =)
        fun_per         f ∼∼ g ≡ ∀x y.x∈D ∧ y∈D ∧ x∼y ⟶ f(x) ∼ g(y)
end
```

The characteristic constant $\sim$ is introduced with the two step technique described in Section 2.1.2. The following theorems on PERs hold:

```
        (* convert ⟶ to ⟹ *)
ax_sym_per      x ∼∼ y ⟹ y ∼∼ x
ax_trans_per    ⟦x ∼∼ y; y ∼∼ z ⟧ ⟹ x ∼∼ z
        (* first derive the characteristic properties of ∼ *)
sym_per         x ∼ y ⟹ y ∼ x
trans_per       ⟦x ∼ y; y ∼ z⟧ ⟹ x ∼ z
        (* further theorems on ∼ *)
sym_per2        x∼y=y∼x
sym2refl1       x∼y ⟹ x∼x
sym2refl2       x∼y ⟹ y∼y
not_per_sym     (¬ x ∼ y) = (¬ y ∼ x)
        (* theorems for the Domain D *)
DomainD         x ∈ D⟹ x ∼ x
DomainI         x ∼ x ⟹ x ∈ D
DomainEq        x ∈ D = x ∼ x
DomainI_left    x ∼ y ⟹ x ∈ D
DomainI_right   x ∼ y ⟹ y ∈ D
notDomainE1     x ∉ D ⟹ ¬ x ∼ y
notDomainE2     y ∉ D ⟹ ¬ x ∼ y
```

```
        (* witnesses for bool::per with general ∼∼ *)
bool_sym_per    (x::bool)∼∼y ⟶ y∼∼x
bool_trans_per  (x::bool)∼∼y ∧ y∼∼z ⟶ x∼∼z
        (* witnesses for "⇒" ::(per,per)per  with general ∼∼ *)
fun_sym_per     (x::α::per ⇒ β::per) ∼∼ y ⟶ y ∼∼ x
fun_trans_per   (f::α::per ⇒ β::per) ∼∼ g ∧ g∼∼h ⟶ f∼∼h
```

The proofs of the witnesses use the definition of the PER on the general constant $\sim\sim$. In this case `bool_per` and `fun_per`.

A.2.2 PER

This section instantiates `bool` and the function space into the class `per`. It uses the `instance` construct, which checks the witnesses, to declare the arities to the type checker.

```
PER = PER0 +  (* arities for per *)
instance bool :: per            (bool_sym_per,bool_trans_per)
instance fun  :: (per,per)per   (fun_sym_per,fun_trans_per)
end
```

The following theorems hold in `PER`:

```
        (* instantiation rules for bool, ⇒ *)
inst_bool_per   ((op  ∼)::[bool,bool]⇒bool) = (op =)
inst_fun_per    f∼g=(∀x y.x∈D∧y∈D∧x∼y⟶f x∼g y)
```

A.2.3 QUOT0

This section contains the introduction of the higher order quotients. It is explained on page 124.

```
QUOT0 = PER +  (* Quotient is type of partial equivalence classes *)
types   quot 1          (* represented by HOL sets *)
arities quot :: (per)term
consts
                (* checks the representant *)
        is_pec       :: α::per ⇒ (α set) ⇒ bool
                (* defines partial equivalence class *)
        rpred        :: (α::per set) ⇒ bool
                (* defines elements in the quotient *)
```

```
        cor          :: (α::per set) ⇒ bool
                (* represents the quotient *)
        Val_q        :: (α::per set) set
                (* abstraction and representation *)
        abs_q        :: (α::per set) ⇒ α quot
        rep_q        :: α::per quot ⇒ (α set)
defs
        is_pec_def       is_pec x s ≡ ∀y.y∈s=y∼x
        rpred_def        rpred f ≡ ∃x.is_pec x f
        cor_def          cor f ≡ rpred f ∨ f={}
        Val_q_def        Val_q ≡ {f. cor f}
rules
        rep_Val_q        rep_q t ∈ Val_q
        abs_rep_q        abs_q(rep_q t) = t
        rep_abs_q        s ∈ Val_q ⟹ rep_q(abs_q s) = s
consts          (* constants for equivalence classes *)
        peclass          :: α::per ⇒ α quot
        any_in           :: α::per quot ⇒ α
syntax          "@ecl"   :: α::per ⇒ α quot ("<[ _ ]>")
translations    <[x]> == peclass x
defs
        peclass_def      <[x]> ≡ abs_q {y.y∼x}
        any_in_def       any_in f ≡ @x.<[x]>=f
end
```

The `translations` (with the `syntax`) allow a more readable form of the theorems. The following theorems could be proved for quotients:

```
        (* first show that quot is not empty *)
UU_in_Val_q     {}∈Val_q
not_empty_q     ∃ x. x∈Val_q
        (* technical lemmata of the construction *)
quot_eq         rep_q a = rep_q b ⟹ a=b
        (* used predicates for equivalence relations *)
cor2Val_q       cor x⟹x∈Val_q
cor_rep_q       cor (rep_q x)
corD            cor x⟹rpred x ∨ x={}
rpredD          rpred f ⟹∃x.is_pec x f
is_pecD         is_pec x f⟹∀y.y∈f=y∼x
rep_rpred       rpred {y.y∼x}
rep_cor         cor{y.y∼x}
rep_Val_q       {y.y∼x}∈Val_q
rep_Val_empty   {}∈Val_q
```

```
empty2notrefl   {y. y ∼ z}={}⟹¬z∼z
        (* general induction and exhaustiveness *)
all_q           ∀s.cor s ⟶ P (abs_q s) ⟹ P x
exh_q           ∃s.cor s ∧ x=abs_q s
rep_abs_q_defined       ∃z.z∼s⟹abs_q{x.x∼s}≠abs_q{}
(* lemmas for the equivalence classes *)
        (* equality and symmetry for equivalence classes *)
qclass_eqI      x∼y⟹<[x]>=<[y]>
qclass_eqE      ⟦x∈D;<[x]>=<[y]>⟧⟹x∼y
qclass_eq       x∈D⟹<[x]>=<[y]>=x∼y
        (* exhaustiveness and 'induction' for classes *)
class_exhaust   ∃z::α::per.∀y.¬z∼y⟹∃s.(x::α quot)=<[s]>
class_exhaust2  x=abs_q {} ∨ (∃s.x=<[s]>)
all_class       ⟦∃z::α::per.∀y.¬z∼y;∀x::α.P<[x]>⟧ ⟹P s
all_class2      ⟦P(abs_q{});∀x.P<[x]>⟧ ⟹P s
        (* inequality *)
qclass_not_eqI  ⟦x∈D;¬x∼y⟧⟹<[x]>≠<[y]>
qclass_not_eqE  ⟦<[x]>≠<[y]>⟧⟹¬x∼y
qclass_not_eq   x∈D⟹<[x]>≠<[y]>=(¬x∼y)
        (* any_in *)
any_in_class    (s::α::per)∈D⟹any_in<[s]>∼s
```

The ugly premise $\exists z.\forall y.\neg z \sim y$ in the induction and exhaustiveness rules for equivalence classes comes from the fact that we have the empty set {} included in the values of the quotient.

A.2.4 QUOT1

This theory contains the introduction of the order for the higher quotient.

```
QUOT1 = QUOT0 + (* add an order for quotients *)
consts
        less_q     :: α::per quot ⇒ α quot ⇒ bool
defs
        less_q_def      less_q ≡ λa.λb.rep_q a={} ∨ a=b
end
```

The order is a flat order with the least abstraction of the empty set as least element. This can be seen from the theorems derived for QUOT1:

```
        (* show that less_q is a partial order *)
```

```
refl_less_q     less_q a a
antisym_less_q  ⟦less_q f1 f2; less_q f2 f1⟧ ⟹ f1 = f2
trans_less_q    ⟦less_q a b; less_q b c⟧ ⟹ less_q a c
         (* abs_q {} is least element *)
less_abs_empty  less_q (abs_q{}) x
```

A.2.5 QUOT2

This theory contains the instantiation into the class po of partially ordered domains and the definition of the least element.

```
QUOT2 = QUOT1 + HOLCF +        (* needs po of HOLCF *)
arities  quot :: (per) po
rules
inst_quot_po     ((op <<)::[α::per quot,α quot]⇒bool) = less_q
consts
        UU_q        :: α::per quot
defs
        UU_q_def       UU_q ≡ abs_q {}
end
```

The following theorem show that abs_q {} is the least element and that the order is chain complete.

```
        (* minimality of UU_q *)
minimal_q       UU_q⊑x
        (* further theorems for the order and chains *)
inst_quot_po2   (x::α::per quot)⊑y = (rep_q x = {} ∨ x = y)
quot_flat       x≠abs_q{}⟹x⊑y=(x=y)
quot_chainE     is_chain C ⟹ (∀i.C i=abs_q{})∨
                      (∃i.(C i)≠abs_q{}∧(∀j.i<j⟶C i=C j))
cpo_quot        is_chain C ⟹ ∃ a::α::per quot.range C <<| a
```

These theorems justify the instantiation of the quotients into the class pcpo in the next theory.

A.2.6 QUOT

The following theory instantiates the quotients into the class pcpo.

```
QUOT = QUOT2 +
        arities quot::(per)pcpo
rules
        inst_quot_pcpo   (⊥::α::per quot) = UU_q
end
```

The following theorems are derived for QUOT:

```
        (* theorem for ⊥ on quot *)
inst_quot_pcpo2 ⊥=abs_q{}
rep_q_UU        rep_q ⊥={}
abs_q_UU        abs_q {}=⊥
rep_q_total     x≠⊥⟹rep_q x≠{}
abs_q_total     ⟦s≠{};cor s⟧ ⟹ abs_q s≠⊥
        (* flatness *)
quot_flat2      (x::α::per quot)≠⊥⟹x⊑y=(x=y)
flat_quot       flat (x::α::per quot)
        (* some lemmas for the equivalence classes *)
rep_abs_q_defined        ∀s.∃z.z∼s⟶abs_q{x.x∼s}≠⊥
class_total      x∼x ⟹<[x]>≠⊥
```

A.2.7 PERCPO0

The type class percpo is introduced as subclass of pcpo and per. The conservative introduction proceeds in several step. The first step is to define a PER on the continuous function space $\rightarrow$ and on the type void.

```
PERCPO0 = QUOT + HOLCF +
defs
        (* define ∼∼ on void, → *)
void_per        (op ∼∼) ≡ λx y::void.False
cfun_per        (op ∼∼) ≡ λf g.∀x y.x∈D∧y∈D∧x∼y ⟶ f`x∼g`y
end
```

The following theorems are proved for PERCPO0:

```
        (* show void and cfun are in per *)
void_sym_per    (x::void) ∼∼ y ⟶ y ∼∼ x
void_trans_per  (x::void) ∼∼ y ∧ y ∼∼ z ⟶ x ∼∼ z
cfun_sym_per    (f::α::{pcpo,per}→β::{pcpo,per})∼∼g⟶g∼∼f
cfun_trans_per  (f::α::{pcpo,per}→β::{pcpo,per})∼∼g∧g∼∼h⟶f∼∼h
```

These theorems are the witness for the instances in the following theory.

A.2.8 PERCPO

This theory defines the axiomatic type class percpo as subclass of pcpo and per. Since there are no further axioms the instances for void and cfun can be proved without further witnesses. On the class percpo the predicate is_Cobs is defined.

```
PERCPO = PERCPO0 +
instance void :: per    (void_sym_per,void_trans_per)
instance "->" :: ({per,pcpo},{per,pcpo}) per
                        (cfun_sym_per,cfun_trans_per)
axclass percpo < per,pcpo
        (* no further axioms for percpo *)
instance void  :: percpo
instance "->"  :: (percpo,percpo)percpo

consts
        is_Cobs     :: (α::percpo → β::percpo) ⇒ bool
defs
        is_Cobs_def is_Cobs f≡∀x y.x∈D∧y∈D∧x∼y⟶f‘x∈D∧f‘y∈D⟶f‘x∼f‘y
end
```

The following theorems hold for PERCPO:

```
        (* derive instantiations *)
inst_void_per    (op ∼)=(λx y::void.False)
inst_cfun_per    (op ∼)=(λf g.∀x y.x∈D∧y∈D∧x∼y⟶f‘x∼g‘y)
        (* some rules for is_Cobs *)
is_Cobs_def2     is_Cobs f=(∀x y.x∈D∧y∈D∧x∼y⟶f‘x∈D∧f‘y∈D⟶f‘x∼f‘y)
is_CobsD         is_Cobs f⟹∀x y.x∈D∧y∈D∧x∼y⟶f‘x∈D∧f‘y∈D⟶f‘x∼f‘y
is_CobsI         ∀x y.x∈D∧y∈D∧x∼y⟶f‘x∈D∧f‘y∈D⟶f‘x∼f‘y⟹is_Cobs f
is_CobsE         ⟦is_Cobs f; xa∈D; x∈D; xa∼x; f‘xa∈D; f‘x ∈ D ⟧
                   ⟹  f‘xa ∼ f‘x
         (* application of observer functions *)
is_Cobs_app      is_Cobs (f::α::percpo→β::percpo→'c::percpo)
                   ⟹ is_Cobs (Λx.f‘x)
```

A.2.9 EQ0

This theory defines the flexible class eq with the characteristic constant eq. It is introduced ax axiomatic type class and described in Section 4.2.5.

```
EQ0 = PERCPO + SUBD +
ops curried
        "≐≐"    :: α → α → tr(cinfixl 55)
axclass eq < pcpo
        ax_eq_refl_def   x≠⊥⟹⌈x≐≐x⌉
        ax_eq_sym2       ⌈x≐≐y⌉⟶⌈y≐≐x⌉
        ax_eq_trans2     ⌈x≐≐y⌉∧⌈y≐≐z⌉⟶⌈x≐≐z⌉
        ax_eq_strict1    ⊥ ≐≐ x = ⊥
        ax_eq_strict2    x ≐≐ ⊥ = ⊥
        ax_eq_total      ⟦x≠⊥;y≠⊥⟧⟹x≐≐y≠⊥
        ax_eq_per        x∼∼y=⌈x≐≐y⌉   (* just to define a per *)
ops curried
        (* characteristic constant for eq *)
        "≐"     :: α::eq → α → tr      (cinfixl 55)
defs    ax_eq_def (op ≐) ≡ (op ≐≐)
        (* add an equality for subdom *)
        sd_eq_def       (op ≐≐) ≡ Λx y.rep_sd x ≐ rep_sd y
        sd_per_def      (op ∼∼) ≡ λx y::α::{adm,eq}.⌈x≐≐y⌉
end
```

The following theories are proved for EQ0:

```
        (* convert ⟶ to ⟹ *)
ax_eq_sym       ⌈x≐≐y⌉ ⟹ ⌈y≐≐x⌉
ax_eq_trans     ⟦ ⌈x≐≐y⌉; ⌈y≐≐z⌉ ⟧⟹⌈x≐≐z⌉
        (* derive the characteristic axioms *)
eq_refl         x≠⊥ ⟹ ⌈x≐x⌉
eq_sym          ⌈x≐y⌉ ⟹ ⌈y≐x⌉
eq_trans        ⟦ ⌈x≐y⌉; ⌈y≐z⌉ ⟧⟹⌈x≐z⌉
eq_per          x∼y=⌈x≐y⌉
eq_strict1      ⊥≐x=⊥
eq_strict2      x≐⊥=⊥
eq_total        ⟦x≠⊥;y≠⊥⟧⟹x≐y≠⊥
        (* derive the rules for per *)
eq_sym_per      (x::α::eq) ∼∼ y ⟶ y ∼∼ x
eq_trans_per    (x::α::eq)∼∼y∧y∼∼z⟶x∼∼z
eq_sym_per2     (x::α::eq)∼∼y=y∼∼x
```

```
eq_not_per_sym  (¬ (x::α::eq) ∼∼ y) = (¬ y ∼∼ x)
        (* derive some rules for continuous equality and per *)
eq2per          ⌈x≐y⌉⟹ x∼y
per2eq          x∼y ⟹ ⌈x≐y⌉
neq2nper        ⌊x≐y⌋⟹¬x∼y
eq_defined      ⟦x≠⊥;y≠⊥⟧⟹⌈x≐y⌉∨⌊x≐y⌋
nper2neq        ⟦x≠⊥;y≠⊥;¬x∼y⟧⟹⌊x≐y⌋
eq_nper2neq     ⟦x≠⊥;y≠⊥;¬x∼∼y⟧⟹⌊x≐y⌋
eq_sym_tr       (x≐y=z) = (y≐x=z)
eq_sym_eq       x≐y = y≐x
neq_sym         (x≐y≠TT) = (y≐x≠TT)
        (* characteristic axioms for subdom *)
rep_sd_eq_app   (Λx y.rep_sd x≐rep_sd y)`x`y=rep_sd x≐rep_sd y
sd_eq_refl      (c::α::{adm,eq} subdom)≠⊥⟹⌈c≐≐c⌉
sd_eq_sym       ⌈(x::α::{adm,eq} subdom)≐≐y⌉⟶⌈y≐≐x⌉
sd_eq_trans     ⌈(a::α::{adm,eq} subdom)≐≐b⌉∧⌈b≐≐c⌉⟶⌈a≐≐c⌉
sd_eq_strict1   (⊥::α::{adm,eq} subdom)≐≐b=⊥
sd_eq_strict2   b≐≐(⊥::α::{adm,eq} subdom)=⊥
sd_eq_total     ⟦a≠⊥;(b::α::{adm,eq} subdom)≠⊥⟧⟹a≐≐b ≠⊥
sd_per_def      a∼∼(b::α::{adm,eq} subdom)=⌈a ≐≐b⌉
```

A.2.10 EQ

This theory instantiates eq as subclass of per and percpo. This allows us to deduce nice properties for the quotients over the class eq. In addition the class EQ is defined.

```
EQ = EQ0 +

instance eq<per         (eq_sym_per,eq_trans_per)
instance eq<percpo

axclass EQ < eq
        ax_EQ   ∀a b.a≠⊥∧b≠⊥⟶(⌈a≐≐b⌉=(a=b))
end
```

The following theorems could be proved for EQ:

```
        (* convert ⟶ to ⟹ *)
weq2eq          ⟦ (xa::α::EQ)≠⊥; x≠⊥ ⟧ ⟹ ⌈xa≐≐x⌉ = (xa=x)
        (* different conversions between ≐ and = for EQ *)
```

```
weq             [[(x::α::EQ)≠⊥;y≠⊥]] ⟹ (x=y⟶⌈x≐y⌉) ∧ (x≠y ⟶ ⌊x≐y⌋)
eq2weq          [[ (x::α::EQ)≠⊥; y≠⊥; x=y ]] ⟹ ⌈x≐y⌉
neq2nweq        [[ (x::α::EQ)≠⊥; y≠⊥; x≠y ]] ⟹ ⌊x ≐ y⌋
eq_weq          [[(x::α::EQ)≠⊥;y≠⊥]] ⟹ (x=y=⌈x≐y⌉)
        (* theorems for ∼ on eq *)
not_UU_refl     (x::α::eq)≠⊥⟹x∼x
not_UU_D        (x::α::eq)≠⊥⟹x∈D
not_UUE         (x::α::eq)∼⊥⟹P
not_UU_I        x∈D⟹(x::α::eq)≠⊥
UU_eq           x∈D=((x::α::eq)≠⊥)
not_UU_per_eq1  ¬⊥∼(x::α::eq)
not_UU_per_eq2  ¬(x::α::eq)∼⊥
UU_not_inD_eq   (⊥::α::eq)∉D
eq_Domain       x≠⊥=((x::α::eq)∈D)
UU_eq_set_empty {y::α::eq. y ∼ ⊥} = {}
        (* quotients over eq *)
eq_class_total  (x::α::eq)≠⊥⟹ <⌈x⌉>≠⊥
eq_not_ex       ∃z::α::eq. ∀y. ¬ z ∼ y
any_in_class_sym        x≠⊥⟹(x::α::eq) ∼ any_in <⌈ x ⌉>
eq_class_strict <⌈(⊥::α::eq)⌉>=⊥
eq_any_strict   any_in (⊥::α::eq quot)=⊥
eq_any_total    x≠⊥⟹any_in x≠(⊥::α::eq)
eq_any_class    (x::α::eq quot)=<⌈any_in x⌉>
eq_exhaust      ∃s::α::eq.(x::α quot) = <⌈ s ⌉>
eq_all_class    ∀x::α::eq.P <⌈ x ⌉> ⟹ P (s::α quot)
        (* is_Cobs on eq *)
is_Cobs_eq      [[is_Cobs f;x≠⊥;y≠⊥;⌈x≐y⌉]]⟹f`x≠⊥∧f`y≠⊥⟶⌈f`x≐f`y⌉
is_Cobs_eq2     [[is_Cobs f;x≠⊥;y≠⊥;⌈x≐y⌉;f`x≠⊥;f`y≠⊥]]⟹⌈f`x≐f`y⌉
eq_less         [[c≠⊥;c⊑c']]⟹⌈c≐c'⌉
is_Cobs_comp_eq [[is_Cobs(f::β::eq→'c::eq);is_Cobs(g::α::eq→β)]]
                  ⟹  is_Cobs (f oo g)
total_EQ_is_Cobs      ∀x.(x::α::EQ)≠⊥⟶f`x≠⊥⟹is_Cobs(f::α→β::eq)
        (* further lemmas for the lifting *)
monofun_class   [[flat(z::α);f`⊥=⊥;∀b.¬(b::β)∼⊥]]
                  ⟹  monofun(λx::α::eq.<⌈(f`x)::β::percpo⌉>)
monofun_lift    [[∀x::β::percpo.¬x∼f`⊥]]
                  ⟹  monofun(λx::α::eq quot.<⌈f`(any_in x)⌉>)
```

A.3 The ADT Library for HOLCF

This section contains the theories and the derived theorems for the library of ADTs in HOLCF. The structure and the theories are explained in Section 7.1. The main theory is:

```
LIB = ADT + String + Byte + Fin + Rat
```

A.3.1 Dnat

The ADT of natural numbers is described in Section 7.1.1. The natural numbers are introduced in two steps, because they use axiomatic type classes. The first step is to define $\doteq\doteq$ and to deduce properties:

```
Dnat0 = EQ +   (* introduce dnat with equality ≐≐ *)
               (* domain with mixfix annotation for syntax, defines #0 *)
domain dnat = dzero ("#0") | dsucc (dpred :: dnat)
defs           (* equality ≐≐ *)
       dnat_eq_def     (op ≐≐) ≡ fix‘(Λeq. Λx y.
                                 If is_dzero‘x
                                 then is_dzero‘y
                                 else If is_dzero‘y
                                         then FF
                                         else eq‘(dpred‘x)‘(dpred‘y)
                                      fi
                                 fi)
       dnat_per_def    (op ∼∼) ≡ λx y::dnat.⌈x≐≐y⌉
end
```

Theorems for Dnat0:

```
       (* derive rules for the equality ≐≐ *)
dnat_eq_unfold    (op ≐≐)=(Λx y.If is_dzero‘x then is_dzero‘y
\                                 else If is_dzero‘y then FF
                                      else (dpred‘x)≐≐(dpred‘y) fi fi)
dnat_eq_app       x≐≐y=If is_dzero‘x then is_dzero‘y
                       else If is_dzero‘y then FF
                            else (dpred‘x)≐≐(dpred‘y) fi fi
dnat_eq_strict1   ⊥≐≐(x::dnat)=⊥
dnat_eq_strict2   (x::dnat)≐≐⊥=⊥
dnat1             ⌈#0≐≐#0⌉
```

```
dnat2              n≠⊥⟹⌊(dsucc‘n)≐≐#0⌋
dnat3              n≠⊥⟹⌊#0≐≐(dsucc‘n)⌋
dnat4              ⟦n≠⊥;m≠⊥⟧⟹(dsucc‘n)≐≐(dsucc‘m)=n≐≐m
        (* derive characteristic axioms for the class EQ *)
dnat_eq_total      ⟦(n::dnat)≠⊥;m≠⊥⟧⟹n≐≐m≠⊥
dnat_eq_eq         ⌈(n::dnat)≐≐m⌉⟹n=m
dnat_eq_refl       (n::dnat)≠⊥⟶⌈n≐≐n⌉
dnat_eq_sym        ⌈(x::dnat) ≐≐ y⌉ ⟶ ⌈y ≐≐ x⌉
dnat_eq_trans      ⌈(n::dnat)≐≐m⌉∧⌈m≐≐k⌉⟶⌈n≐≐k⌉
dnat_per_def2      (x::dnat) ∼∼ y = ⌈x≐≐y⌉
dnat_ax_EQ         ∀a b::dnat. a≠⊥∧b≠⊥ ⟶ ⌈a≐≐b⌉ = (a=b)
        (* Dnat is flat *)
flat_dnat          flat(x::dnat)
        (* Selector rule *)
dpred_dsucc        x≠⊥⟹dpred‘(dsucc‘x)=x
```

These theorems justify the instantiation into the class `EQ`. `TR` contains the strict operations `AND` and `OR` for the non-strict operations `andalso` and `orelse` of HOLCF.

```
Dnat = Dnat0 + TR + (* instantiate Dnat in eq *)

instance dnat :: EQ   ( dnat_eq_refl,dnat_eq_sym,dnat_eq_trans,
                        dnat_eq_strict1,dnat_eq_strict2,dnat_eq_total,
                        dnat_per_def2,
                        dnat_ax_EQ)
ops curried
        "≤" :: dnat → dnat → tr (cinfixl 62)
defs
        dnle_def (op ≤) ≡ fix‘(Λle n m.is_dzero‘n OR
                If is_dzero‘m OR is_dzero‘n then FF
                else le‘(dpred‘n)‘(dpred‘m) fi)
ops curried strict total
        sub     :: dnat → dnat → dnat  (cinfixr 70)
        add     :: dnat → dnat → dnat  (cinfixl 70)
        mult    :: dnat → dnat → dnat  (cinfixl 75)
        div     :: dnat → dnat → dnat
        "^"     :: dnat → dnat → dnat (cinfixl 79)
axioms
defvars n m in
        add1    #0 add n=n
        add2    dsucc‘n add m=dsucc‘(n add m)
        sub1    n sub #0 = n
        sub2    dsucc‘n sub dsucc‘m = n sub m
```

```
        mult1   #0 mult m =#0
        mult2   dsucc‘n mult m = m add n mult m
        pow0    n ^ #0 =dsucc‘#0
        pow1    n ^ (dsucc‘m) = n mult (n ^ m)
        div1    n≠#0 ⟹ div‘(n mult m)‘n=m
(* some nice writings for natural numbers *)
ops curried strict total
        "↑" :: "dnat → dnat → dnat" (cinfixl 80)
        on   :: "dnat" ("#1")
        two  :: "dnat" ("#2")
        three:: "dnat" ("#3")
        four :: "dnat" ("#4")
        five :: "dnat" ("#5")
        six  :: "dnat" ("#6")
        seven:: "dnat" ("#7")
        eight:: "dnat" ("#8")
        nine :: "dnat" ("#9")
        ten  :: "dnat"
axioms
defvars n m in
        exp_def          n ↑ m ≡ n mult ten add m
defs
        one     "#1 ≡ dsucc‘#0"
        two     "#2 ≡ dsucc‘#1"
        three   "#3 ≡ dsucc‘#2"
        four    "#4 ≡ dsucc‘#3"
        five    "#5 ≡ dsucc‘#4"
        six     "#6 ≡ dsucc‘#5"
        seven   "#7 ≡ dsucc‘#6"
        eight   "#8 ≡ dsucc‘#7"
        nine    "#9 ≡ dsucc‘#8"
        ten     "ten≡ dsucc‘#9"
end
```

The following theorems are derived for `Dnat`:

```
        (* characteristic axioms for dnat with ≐ *)
dnat_eq1            ⌈#0≐#0⌉
dnat_eq2            x≠⊥⟹⌊dsucc‘x≐#0⌋
dnat_eq3            x≠⊥⟹⌊#0≐dsucc‘x⌋
dnat_eq4            ⟦x≠⊥;y≠⊥⟧⟹dsucc‘x ≐ dsucc‘y = x ≐ y
        (* theorems for ≤ *)
dnle_unfold         (op ≤)=(Λn m.is_dzero‘n OR
```

```
                        If is_dzero‘m OR is_dzero‘n then FF
                        else (dpred‘n)≤(dpred‘m) fi)
dnle_app            n≤m=is_dzero‘n OR
                    If is_dzero‘m OR is_dzero‘n then FF
                    else dpred‘n ≤ dpred‘m fi
dnle_strict1        ⊥≤x=⊥
dnle_strict2        x≤⊥=⊥
dnle_total          ⟦n≠⊥;m≠⊥⟧⟹n≤m≠⊥
dnle1               ⌈#0≤#0⌉
dnle2               n≠⊥⟹⌈#0≤n⌉
dnle3               n≠⊥⟹⌊dsucc‘n≤#0⌋
dnle4               ⟦n≠⊥;m≠⊥⟧⟹dsucc‘n ≤ dsucc‘m = n ≤ m
dnle_refl           n≠⊥ ⟹ ⌈n ≤ n⌉
dnle_trans          ∀n h.(⌈n≤m⌉∧⌈m≤h⌉⟶⌈n≤h⌉)
dnle_trans2         ∀n h.(⌊n≤m⌋∧⌊m≤h⌋⟶⌊n≤h⌋)
          (* further theorems *)
zero_defined        #0≠⊥
one_defined         #1≠⊥
two_defined         #2≠⊥
three_defined       #3≠⊥
four_defined        #4≠⊥
five_defined        #5≠⊥
six_defined         #6≠⊥
seven_defined       #7≠⊥
eight_defined       #8≠⊥
nine_defined        #9≠⊥
         (* rules for add *)
add1b               n add #0 = n
add2b               n add dsucc‘m = dsucc‘(n add m)
add_com             n add m = m add n
add_ass             (m add n) add l= m add (n add l)
add_ass_left        x add (y add z)=y add (x add z)
add_cancel          a≠⊥⟶(a add n=a add m)=(n=m)
add_cancel2         a≠⊥⟶(n add a=m add a)=(n=m)
         (* rules for mult *)
mult1b              m≠⊥⟹m mult #0 = #0
mult2b              m mult dsucc‘n = m add m mult n
mult3               m mult #1 = m
mult3b              #1 mult m = m
mult_com            n mult m = m mult n
add_mult1           (m add n) mult k = m mult k add n mult k
add_mult2           k mult (m add n) = k mult m add k mult n
mult_ass            (m mult n) mult k = m mult (n mult k)
```

```
mult_ass_left      k mult (m mult n) = m mult (k mult n)
pos_mult_eq        [[a≠⊥;a≠#0]]⟹⌈a mult n≐a mult m⌉=⌈n≐m⌉
pos_mult_eq2       [[a≠⊥;a≠#0]]⟹a mult n=a mult m=(n=m)
```

These arithmetic rules for `dnat` are basically the same as the rules from the arithmetic theory of the Isabelle logic HOL.

A.3.2 Dlist

The ADT of lists is described in Section 7.1.2.

```
Dlist0 = Dnat +
domain α dlist = dnil | "##" (dhd::α) (dtl::α dlist) (cinfixr 65)
ops curried
lmap    :: (α → β) → α dlist → β dlist
dlen    :: α dlist → dnat
lmem    :: (α::eq)  → α dlist → tr                    (cinfixl 50)
defs

lmap_def    lmap ≡ fix‘(Λh f s. case s of dnil => dnil
                                        | x ## xs => f‘x ## h‘f‘xs)
dlen_def    dlen ≡ fix‘(Λlen s. case s of dnil => #0
                                        | x ## xs => dsucc‘(len‘xs))
lmem_def op lmem ≡ fix‘(Λ h e l. case l of dnil => FF
                                        | x ## xs => If e≐x then TT
                                                     else h‘e‘xs fi)
l_eq_def    (op ≐≐) ≡ fix‘(Λeq.Λx y.
                      If is_dnil‘x
                      then is_dnil‘y
                      else If is_dnil‘y
                           then FF
                           else dhd‘x≐dhd‘y andalso eq‘(dtl‘x)‘(dtl‘y)
                           fi
                      fi)
dlist_per_def   x∼∼(y::α::eq dlist) ≡ ⌈x≐≐y⌉
end
```

The following theorems are proved for `Dlist0`:

```
                (* unfold the fixed points *)
        (* lmap *)
```

```
lmap_def2        lmap = (Λf s.case s of
                           dnil => dnil |
                           x ## l => f‘x ## lmap‘f‘l)
lmap1            lmap‘f‘⊥ = ⊥
lmap2            lmap‘f‘dnil = dnil
lmap3            ⟦x≠⊥; xs≠⊥⟧ ⟹ lmap‘f‘(x##xs) = (f‘x)##(lmap‘f‘xs)
        (* dlen *)
dlen_def2        dlen = (Λs.case s of
                           dnil => #0 |
                           x ## l => dsucc‘(dlen‘l))
dlen_strict      dlen‘⊥ = ⊥
dlen1            dlen‘dnil = #0
dlen_eq          ⟦x≠⊥;y≠⊥⟧ ⟹ dlen‘(x##d)=dlen‘(y##d)
dlen2            x≠⊥ ⟹ dlen‘(x##xs) = dsucc‘(dlen‘xs)
dlen_total       x≠⊥ ⟹ dlen‘x ≠⊥
by (subgoal_tac x≠⊥⟶dlen‘x ≠ ⊥
         (* ≐≐ equality on lists *)
l_eq_unfold      (op ≐≐)=(Λx y.If is_dnil‘x then is_dnil‘y
                              else If is_dnil‘y then FF
                                   else dhd‘x ≐ dhd‘y andalso
                                        (dtl‘x)≐≐(dtl‘y) fi fi)
l_eq_app         x≐≐y=If is_dnil‘x then is_dnil‘y
                         else If is_dnil‘y then FF
                              else dhd‘x ≐ dhd‘y andalso
                                   (dtl‘x)≐≐(dtl‘y) fi fi
dlist_strict1    ⊥≐≐(y::α::eq dlist)=⊥
dlist_strict2    (x::α::eq dlist)≐≐⊥=⊥
dlist1           ⌈(dnil::α::eq dlist)≐≐dnil⌉
dlist2           ⟦(d::α::eq)≠⊥;l≠⊥⟧⟹⌊(d##l)≐≐dnil⌋
dlist3           ⟦(d::α::eq)≠⊥;l≠⊥⟧⟹⌊dnil≐≐(d##l)⌋
dlist4           ⟦(n::α::eq)≠⊥;m≠⊥;s≠⊥;t≠⊥⟧
                  ⟹  (n##s)≐≐(m##t)=(n≐m andalso s≐≐t)
        (* further properties of dlist *)
flat2dlist_flat  flat (x::α) ⟹ flat(y::α dlist)
        (* characteristic axioms for eq *)
dlist_total      ⟦(n::α::eq dlist)≠⊥;m≠⊥⟧⟹n≐≐m≠⊥
dlist_refl       (n::α::eq dlist)≠⊥⟹⌈n≐≐n⌉
l_eq_sym2        ⌈(n::α::eq dlist)≐≐m⌉=⌈m≐≐n⌉
l_eq_sym         ⌈(x::α::eq dlist)≐≐y⌉⟶⌈y≐≐x⌉
l_eq_trans       ⌈(n::α::eq dlist)≐≐m⌉∧⌈m≐≐k⌉⟶⌈n≐≐k⌉
dlist_per_def2   (x::α::eq dlist)∼∼y=⌈x≐≐y⌉
dlist_EQ_EQ      ∀a b::α::EQ dlist.a≠⊥∧b≠⊥⟶(⌈a≐≐b⌉=(a=b))
```

These theorems justify the following instantiations:

```
Dlist = Dlist0 + (* instantiate Dlist in eq and EQ *)
instance dlist::(eq)eq (dlist_refl,l_eq_sym,l_eq_trans,
                        dlist_strict1,dlist_strict2,dlist_total,
                        dlist_per_def2)
instance dlist::(EQ)EQ   (dlist_EQ_EQ)
ops curried strict total
        concat_dl :: α dlist → α dlist → α dlist
        insert_dl :: α::eq → α dlist → α dlist
axioms
defvars x l m in
concat1         concat_dl‘dnil‘l=l
concat2         concat_dl‘(x##l)‘m=x##concat_dl‘m‘l
insert_def      insert_dl‘x‘l=If isin‘x‘l then l else x##l fi
end
```

The following theorems are proved for `Dlist`:

```
        (* explicit instantiation rules *)
inst_dlist_eq       x≐y=If is_dnil‘x then is_dnil‘y
                        else If is_dnil‘y then FF
                             else dhd‘x ≐ dhd‘y andalso
                                  dtl‘x ≐ dtl‘y fi fi
dlist_eq_1          ⌈dnil≐dnil⌉
dlist_eq_2          ⟦ d≠⊥;l≠⊥ ⟧ ⟹ ⌊d ## l ≐ dnil⌋
dlist_eq_3          ⟦ d≠⊥;l≠⊥ ⟧ ⟹ ⌊dnil ≐ d ## l⌋
dlist_eq_4          n ## s ≐ m ## t = (n ≐ m) AND (s ≐ t)
```

A.3.3 Bit

The ADT of bits is described in Section 7.1.3. Bits are defined with the domain construct. The equality is defined together with the schematic PER.

```
Bit0 = EQ + TR +
domain Bit = L | H
defs
        Bit_eq_def      (op ≐≐) ≡ Λx y.is_L‘x AND is_L‘y
                                    OR is_H‘x AND is_H‘y
        Bit_per_def     (op ∼∼) ≡ λb c::Bit.⌈b≐≐c⌉
end
```

The following theorems justify the instance of Bit into the class eq:

```
        (* Bit is flat *)
flat_Bit            flat (x::Bit)
        (* derive characteristic axioms for the class EQ *)
Bit_eq_refl          c≠⊥ ⟹ ⌈c≐≐c⌉
Bit_eq_sym           ⌈(x::Bit)≐≐ y⌉ ⟶ ⌈y≐≐ ⌉
Bit_eq_trans         ⌈(a::Bit)≐≐ b⌉ ∧ ⌈b≐≐c⌉ ⟶ ⌈a≐≐c⌉
Bit_eq_strict1       (⊥::Bit) ≐≐ b=⊥
Bit_eq_strict2       b≐≐(⊥::Bit) = ⊥
Bit_eq_total         ⟦a≠⊥;(b::Bit)≠⊥⟧⟹a≐≐b ≠⊥
Bit_per_def          a∼∼(b::Bit)=⌈a≐≐b⌉
Bit_ax_EQ            ∀a b::Bit.a≠⊥ ∧ b≠⊥ ⟶ ⌈a≐≐b⌉ = (a=b)
```

Theory Bit instantiates the type Bit into the type class eq:

```
Bit = Bit0 +
instance Bit::EQ (Bit_eq_refl,Bit_eq_sym,Bit_eq_trans,
                      Bit_eq_strict1,Bit_eq_strict2,Bit_eq_total,
                      Bit_per_def,Bit_ax_EQ)
end
```

The following theorems are derived for Bit:

```
        (* explicit instantiation *)
inst_Bit_eq         x≐y=is_L‘x AND is_L‘y OR is_H‘x AND is_H‘y
        (* ≐ rules *)
Bit_eq1             ⌈L≐L⌉
Bit_eq2             ⌈H≐H⌉
Bit_eq3             ⌊L≐H⌋
Bit_eq4             ⌊H≐L⌋
```

A.3.4 Byte

The ADT of Bytes is described in Section 7.1.3. It consists of lists of bits of length eight.

```
Byte0 = Dlist + Bit+   (* 8 Bit Bytes with embedding *)
types BitL = Bit dlist
domain B=Babs(Brep::BitL)
defs
        Badm_pred_def (adm_pred'::B⇒bool) ≡ λi.⌈dlen‘(Brep‘i)≐#8⌉
```

```
        (* defines equality and PER on B *)
        B_eq_def        (op ≐≐) ≡ Λx y.Brep‘x ≐ Brep‘y
        B_per_def       (op ∼∼) ≡ λx y::B.⌈x≐≐y⌉
end
```

The following theorems are proved for `Byte0`:

```
        (* admissibility *)
adm_Badm_pre        adm (adm_pred'::B⇒bool)
        (* further theorems *)
flat_B              flat (x::B)
B_con_sel           Babs‘(Brep‘c)=c
        (* characteristic axioms for eq *)
B_eq_refl           (c::B)≠⊥⟹⌈c≐≐c⌉
B_eq_sym            ⌈(x::B) ≐≐ y⌉ ⟶ ⌈y ≐≐ x⌉
B_eq_trans          ⌈(a::B) ≐≐ b⌉ ∧ ⌈b ≐≐ c⌉ ⟶ ⌈a ≐≐ c⌉
B_eq_strict1        (⊥::B) ≐≐ b=⊥
B_eq_strict2        b ≐≐ (⊥::B) = ⊥
B_eq_total          ⟦a≠⊥;(b::B)≠⊥⟧⟹a≐≐b ≠ ⊥
B_per_def           a∼∼(b::B)=⌈a≐≐b⌉
```

These theorems justify the instance of the type `C` into the type classes `adm` and `eq`:

```
Byte = Byte0 +
instance B::adm (adm_Badm_pred)
instance B::eq  ( B_eq_refl,B_eq_sym,B_eq_trans,
                        B_eq_strict1,B_eq_strict2,B_eq_total,
                        B_per_def)
types Byte = B subdom
ops curried
        cBabs   :: Bit dlist → Byte    (* continuous abs *)
        mkBy    :: Bit→Bit→Bit→Bit→Bit→Bit→Bit→Bit→Byte
        b1      :: Byte → Bit
        b2      :: Byte → Bit
        b3      :: Byte → Bit
        b4      :: Byte → Bit
        b5      :: Byte → Bit
        b6      :: Byte → Bit
        b7      :: Byte → Bit
        b8      :: Byte → Bit
defs    (* defines continuous abstraction function *)
cBabs_def  cBabs ≡ Λl.If dlen‘l≐#8 then abs_sd(Babs‘l) else ⊥ fi
```

```
mkBy_def   mkBy ≡ Λa b c d e f g h.
                cBabs‘(a##b##c##d##e##f##g##h##dnil)
b1_def    b1≡ΛB.dhd‘(Brep‘(rep_sd B))
b2_def    b2≡ΛB.dhd‘(dtl‘(Brep‘(rep_sd B)))
b3_def    b3≡ΛB.dhd‘(dtl‘(dtl‘(Brep‘(rep_sd B))))
b4_def    b4≡ΛB.dhd‘(dtl‘(dtl‘(dtl‘(Brep‘(rep_sd B)))))
b5_def    b5≡ΛB.dhd‘(dtl‘(dtl‘(dtl‘(dtl‘
                  (Brep‘(rep_sd B))))))
b6_def    b6≡ΛB.dhd‘(dtl‘(dtl‘(dtl‘(dtl‘
                  (dtl‘(Brep‘(rep_sd B)))))))
b7_def    b7≡ΛB.dhd‘(dtl‘(dtl‘(dtl‘(dtl‘
                  (dtl‘(dtl‘(Brep‘(rep_sd B))))))))
b8_def    b8≡ΛB.dhd‘(dtl‘(dtl‘(dtl‘(dtl‘
                  (dtl‘(dtl‘(dtl‘(Brep‘(rep_sd B)))))))))
end
```

The following theorems are proved for `Byte0`:

```
        (* explicit instantiations *)
inst_B_adm          adm_pred =(λc.⌈dlen‘(Brep‘c)≐ #8⌉)
inst_B_eq           x≐y=Brep‘x ≐ Brep‘y
        (* prepare continuous functions *)
mono_Badm_pred      ∀x y.x⊑y⟶adm_pred(Babs‘x)⟶adm_pred(Babs‘y)
cont_cont_cBabs     cont(λx.if x≠⊥∧adm_pred(Babs‘x)
                            then abs_sd (Babs‘x) else ⊥)
cont_cont_cBabs4    cont(λx.If dlen‘x ≐ #8
                            then abs_sd (Babs‘x) else ⊥ fi)
cBabs_app           cBabs‘l=If dlen‘l≐#8
                            then abs_sd(Babs‘l) else ⊥ fi
mkBy_app            mkBy‘a‘b‘c‘d‘e‘f‘g‘h=
                        cBabs‘(a##b##c##d##e##f##g##h##dnil)
```

A.3.5 Char

The ADT of characters is described in Section 7.1.4.

```
Char0 = Dnat +          (* basic theory for ASCII−characters *)
(* embedding *)
domain C=Cabs(Crep::dnat)
defs
        Cadm_pred_def adm_pred' ≡ λc.⌈Crep‘c≤#1 ↑ #2 ↑ #7⌉
```

```
        (* defines equality and PER on C *)
        C_eq_def        (op ≐≐) ≡ Λx y.Crep‘x ≐ Crep‘y
        C_per_def       (op ∼∼) ≡ λx y::C.⌈x≐≐y⌉
end
```

The following theorems are proved for `Char0`:

```
        (* admissibility of the restriction *)
adm_Cadm_pred       adm (adm_pred'::C⇒bool)
        (* C is flat *)
flat_C              flat (x::C)
        (* further theorems *)
C_con_sel           Cabs‘(Crep‘c)=c C.rews) 1)
defined_127         #1 ↑ #2 ↑ #7 ≠⊥
zero_le_127         ⌈#0 ≤ #1 ↑ #2 ↑ #7⌉
         (* characteristic axioms for eq on C *)
C_eq_refl           (c::C)≠⊥⟹⌈c≐≐c⌉
C_eq_sym            ⌈(x::C) ≐≐ y⌉ ⟶ ⌈y ≐≐ x⌉
C_eq_trans          ⌈(a::C) ≐≐ b⌉ ∧ ⌈b ≐≐ c⌉ ⟶ ⌈a ≐≐ c⌉
C_eq_strict1        (⊥::C) ≐≐ b=⊥
C_eq_strict2        b ≐≐ (⊥::C) = ⊥
C_eq_total          ⟦a≠⊥;(b::C)≠⊥⟧⟹a ≐≐b ≠ ⊥
C_per_def           a∼∼(b::C)=⌈a ≐≐b⌉
```

These theorems justify the instance of the type `C` into the type classes `adm` and `eq`:

```
Char1 = Char0 +
instance C::adm (adm_Cadm_pred)
instance C::eq  ( C_eq_refl,C_eq_sym,C_eq_trans,
                        C_eq_strict1,C_eq_strict2,C_eq_total,
                        C_per_def)
types Char = C subdom
ops curried
        nat2chr :: dnat → Char
        chr2nat :: Char → dnat
defs
        (* defines continuous abstraction function *)
        nat2chr_def nat2chr ≡ Λn.If n≤#1↑#2↑#7
                                    then abs_sd(Cabs‘n) else ⊥ fi
        chr2nat_def chr2nat ≡ Λn.Crep‘(rep_sd n)
end
```

The following theorems are proved for `Char1`:

```
        (* derive explicit instantiation rules *)
inst_C_adm          adm_pred =(λc.⌈Crep‘c ≤#1↑#2↑#7⌉)
inst_C_eq           x≐y = Crep‘x≐Crep‘y
inst_Char_eq        x≐y = chr2nat‘x≐chr2nat‘y
        (* theorems for the continuity of nat2chr *)
mono_Cadm_pred      ∀x y.x⊑y⟶adm_pred (Cabs‘x)⟶adm_pred (Cabs‘y)
monofun_contAbs     monofun(λx.if x≠⊥∧adm_pred (Cabs‘x)
                                then abs_sd (Cabs‘x) else ⊥)
cont_contAbs3       cont (λx. if x≠⊥∧⌈x≤#1↑#2↑#7⌉
                              then abs_sd (Cabs‘x) else ⊥)
cont_contAbs4       cont (λx.If x ≤#1↑#2↑#7
                             then abs_sd(Cabs‘x) else ⊥ fi)
        (* rules for conversion nat <-> char *)
nat2chr_app         nat2chr‘n=(If n≤#1↑#2↑#7
                                then abs_sd(Cabs‘n) else ⊥ fi)
chr2nat_app         chr2nat‘n=Crep‘(rep_sd n)
nat2chr_strict      nat2chr‘⊥=⊥
chr2nat_strict      chr2nat‘⊥=⊥
nat2chr_total       ⌈x ≤ #1 ↑ #2 ↑ #7⌉⟹nat2chr‘x≠⊥
chr2nat_total       x≠⊥⟹chr2nat‘x≠⊥
nat2chr_chr2nat     nat2chr‘(chr2nat‘c)=c
chr2nat_nat2chr     ⌈n≤#1↑#2↑#7⌉⟹chr2nat‘(nat2chr‘n)=n
```

This ADT is the basis for the ASCII-encoding, which is given here only for some characters.

```
Char = Char1 + (* Char with ASCII-encoding *)
ops curried
        c_a :: "Char" ("‡a")
        c_b :: "Char" ("‡b")
        c_c :: "Char" ("‡c")
        c_A :: "Char" ("‡A")
        c_B :: "Char" ("‡B")
        c_C :: "Char" ("‡C")
        c0  :: "Char" ("‡0")
        c1  :: "Char" ("‡1")
        c2  :: "Char" ("‡2")
defs
        c0      "‡0≡nat2chr‘(#4 ↑ #8)"
        c1      "‡1≡nat2chr‘(#4 ↑ #9)"
        c2      "‡2≡nat2chr‘(#5 ↑ #0)"
        A       "‡A≡nat2chr‘(#6 ↑ #5)"
        B       "‡B≡nat2chr‘(#6 ↑ #6)"
        C       "‡C≡nat2chr‘(#6 ↑ #7)"
```

```
        a       "♮a≡nat2chr‘(#9 ↑ #7)"
        b       "♮b≡nat2chr‘(#9 ↑ #8)"
        c       "♮c≡nat2chr‘(#9 ↑ #9)"
end
```

Theorems of the following kind can be proved in one step:

```
        (* definedness *)
c_defined           ♮c≠⊥
        (* inequality *)
test                ⌊♮A ≐ ♮1⌋
```

These theorems are not derived for all characters, they can be inferred with the provided tactics, whenever they are needed.

A.3.6 String

The ADT of strings is described in Section 7.1.4.

```
String = Dlist + Char +
types String = Char dlist
ops curried strict total
        strlen :: String → dnat
        strcmp :: String → String → tr
        strcat :: String → String → String
defs
        strlen_def      strlen ≡ dlen
        strcmp_def      strcmp ≡ (op ≐)
axioms
defvars a s t in
        strcat1         strcat‘dnil‘s = s
        strcat2         strcat‘(a##s)‘t = a##(strcat‘s‘t)
end
```

Since these definitions are only equations between functions, we do not derive further theorems for them. With the simplifier we can substitute these definitions easily and prove theorems like

```
Test            ⌈strlen‘(♮0 ## ♮s ## ♮c ## ♮a ## ♮r ## dnil)≐#5⌉
```

A.3.7 Fin

The ADT of finite numbers is described in Section 7.1.5.

```
Fin0 = Dnat +  (* finite numbers (16 Bit) *)
(* embedding *)
domain F=Fabs(Frep::dnat)
defs
        Fadm_pred_def  adm_pred' ≡ λi.⌈Frep‘i≤(#2^#1↑#6) sub #1⌉
        (* defines equality and PER on F *)
        F_eq_def       (op ≐≐) ≡ Λx y.Frep‘x ≐ Frep‘y
        F_per_def      (op ∼∼) ≡ λx y::F.⌈x≐≐y⌉
end
```

The following theorems are proved for Fin0:

```
        (* characteristic axiom for adm *)
adm_Fadm_pred      adm (adm_pred'::F⇒bool)
        (* further theorems *)
flat_F             flat (x::F)
F_con_sel          Fabs‘(Frep‘c)=c
defined_2_16       #2^#1↑#6 sub #1≠⊥
zero_le_2_16       ⌈#0 ≤ #2^#1↑#6 sub #1⌉
        (* characteristic axioms for eq *)
F_eq_refl          (c::F)≠⊥⟹⌈c≐≐c⌉
F_eq_sym           ⌈(x::F) ≐≐ y⌉ ⟶ ⌈y ≐≐ x⌉
F_eq_trans         ⌈(a::F) ≐≐ b⌉ ∧ ⌈b ≐≐ c⌉ ⟶ ⌈a ≐≐ c⌉
F_eq_strict1       (⊥::F) ≐≐ b=⊥
F_eq_strict2       b ≐≐ (⊥::F) = ⊥
F_eq_total         ⟦a≠⊥;(b::F)≠⊥⟧⟹a ≐≐b ≠⊥
F_per_def          a∼∼(b::F)=⌈a≐≐b⌉
```

This justifies the instantiations into the classes adm and eq.

```
Fin1 = Fin0 +
instance F::adm (adm_Fadm_pred)
instance F::eq  ( F_eq_refl,F_eq_sym,F_eq_trans,
                        F_eq_strict1,F_eq_strict2,F_eq_total,
                        F_per_def)
types Fin = F subdom
ops curried
        nat2fin :: dnat → fin
```

```
        fin2nat :: fin  → dnat
defs
        (* defines continuous abstraction function *)
        nat2fin_def nat2fin ≡ Λn.If n≤#2^#1↑#6 sub #1
                                then abs_sd(Fabs‘n)else ⊥ fi
        fin2nat_def fin2nat ≡ Λn.Frep‘(rep_sd n)
end
```

The following theorems are proved for `Fin1`:

```
        (* explicit instantiations *)
inst_F_adm          adm_pred =(λc.⌈Frep‘c ≤#2^#1↑#6 sub #1⌉)
inst_F_adm          x≐y=Frep‘x≐Frep‘y
inst_Fin_eq         x≐y = fin2nat‘x≐fin2nat‘y
        (* prepare continuous operations *)
mono_Fadm_pred      ∀x y.x⊑y⟶adm_pred (Fabs‘x)⟶adm_pred (Fabs‘y)
monofun_contFAbs    monofun(λx.if x≠⊥∧adm_pred(Fabs‘x)
                               then abs_sd(Fabs‘x) else ⊥)
cont_contFAbs       cont(λx.if x≠⊥∧adm_pred (Fabs‘x)
                            then abs_sd(Fabs‘x) else ⊥)
cont_contFAbs4      cont(λx.If x ≤ #2^#1↑#6 sub #1
                            then abs_sd (Fabs‘x) else ⊥ fi)
        (* nat2fin and fin2nat *)
nat2fin_app         nat2fin‘n=(If n≤#2^#1↑#6 sub #1
                               then abs_sd (Fabs‘n) else ⊥ fi)
fin2nat_app         fin2nat‘n=Frep‘(rep_sd n)
nat2fin_strict      nat2fin‘⊥=⊥
fin2nat_strict      fin2nat‘⊥=⊥
nat2fin_total       ⌈x ≤ #2^#1↑#6 sub #1⌉⟹nat2fin‘x≠⊥
fin2nat_total       x≠⊥⟹fin2nat‘x≠⊥
nat2fin_fin2nat     nat2fin‘(fin2nat‘c)=c
fin2nat_nat2fin     ⌈n≤#2^#1↑#6 sub #1⌉⟹(fin2nat‘(nat2fin‘n)=n)
```

Now the operations on unsigned integers are defined:

```
Fin = Fin1 + (* defines non−preserving operations on Fin
               to show the overflow errors explicitly *)
ops curried
        Fzero   :: Fin
        Fsucc   :: Fin → Fin
        Fadd    :: Fin → Fin → Fin
defs
```

```
        Fzero_def       Fzero ≡ nat2fin‘#0
        Fsucc_def       Fsucc ≡ Λn.nat2fin‘(dsucc‘(fin2nat‘n))
        Fadd_def        Fadd ≡ Λn m.nat2fin‘(fin2nat‘n add fin2nat‘m)
end
```

We did not prove many theorems for finite numbers. The only interesting proof is that even, if `add` on `dnat` is total, this function `Fadd` on finite numbers is not. The reason is that it is not preserving.

```
        (* unfold the definition of Fadd *)
Fadd_app            Fadd‘n‘m=nat2fin‘(fin2nat‘n add fin2nat‘m)
        (* additional theorems on natural numbers *)
dsucc_le            x≠⊥⟹⌊dsucc‘x≤x⌋
le_not_eq           ⌊x≤dsucc‘y⌋⟹⌊x≐y⌋
not_n_add_n_le_n    ⌊n≐#0⌋ ⟹ ⌊n add n ≤ n⌋
sub_one_le          ⟦ ⌈#1≤n⌉; ⌈n≤m⌉ ⟧ ⟹ ⌈n sub #1 ≤ m sub #1⌉
        (* Fadd is not total *)
not_Fadd_total      ¬(∀x y.x≠⊥∧y≠⊥⟶Fadd‘x‘y≠⊥)
```

A.3.8 Pos

The ADT of positive natural numbers is described in Section 7.1.6.

```
Pos0 = Dnat +  (* positive numbers *)
(* embedding *)
domain P = Pabs(Prep::dnat)
defs
        Padm_pred_def adm_pred' ≡ λc.⌈neg‘(Prep‘c≐#0)⌉
        (* defines equality and PER on P *)
        P_eq_def        (op ≐≐) ≡ Λx y.Prep‘x ≐ Prep‘y
        P_per_def       (op ∼∼) ≡ λx y::P.⌈x≐≐y⌉
end
```

The following theorems are proved for `Pos0`:

```
        (* characteristic axiom for adm *)
adm_Padm_pred       adm (adm_pred'::P⇒bool
        (* further theorems *)
flat_P              flat (x::P)
P_con_sel           Pabs‘(Prep‘c)=c
        (* characteristic axioms for eq *
```

```
P_eq_refl          (c::P)≠⊥⟹⌈c≐≐c⌉
P_eq_sym           ⌈(x::P) ≐≐ y⌉ ⟶ ⌈y ≐≐ x⌉
P_eq_trans         ⌈(a::P) ≐≐ b⌉ ∧ ⌈b ≐≐ c⌉ ⟶ ⌈a ≐≐ c⌉
P_eq_strict1       (⊥::P) ≐≐ b=⊥
P_eq_strict2       b ≐≐ (⊥::P) = ⊥
P_eq_total         ⟦a≠⊥; (b::P)≠⊥ ⟧⟹a≐≐b ≠ ⊥
P_per_def          a∼∼(b::P)=⌈a≐≐b⌉
```

These axioms justify the instantiation into `eq` and `adm`.

```
Pos = Pos0 +
instance P::adm (adm_Padm_pred)
instance P::eq  ( P_eq_refl,P_eq_sym,P_eq_trans,
                      P_eq_strict1,P_eq_strict2,P_eq_total,
                      P_per_def)
types pos = P subdom
ops curried (* simplify proofs *)
        nat2pos :: dnat → pos
        pos2nat :: pos → dnat
defs
        nat2pos_def nat2pos≡Λn.If neg‘(n≐#0) then abs_sd(Pabs‘n) else ⊥ fi
        pos2nat_def pos2nat≡Λn.Prep‘(rep_sd n)
end
```

The following theorems are proved for `Pos`:

```
        (* explicit instantiations *)
inst_P_adm         adm_pred =(λc. ⌈neg‘(Prep‘c ≐ #0)⌉)
inst_P_eq          x≐y=Prep‘x≐Prep‘y
        (* prepare continuous functions *)
mono_Padm_pred     ∀x y.x⊑y⟶adm_pred(Pabs‘x)⟶adm_pred (Pabs‘y)
monofun_contPabs   monofun(λx.if x≠⊥∧adm_pred(Pabs‘x)
                           then abs_sd(Pabs‘x) else ⊥)
cont_contPabs      cont(λx.if x≠⊥∧adm_pred(Pabs‘x)
                        then abs_sd (Pabs‘x) else ⊥)
cont_contPabs4     cont (λx.If neg‘(x≐#0)
                         then abs_sd(Pabs‘x) else ⊥ fi)
        (* nat2pos and pos2nat *)
nat2pos_app        nat2pos‘n=(If neg‘(n≐#0)then abs_sd(Pabs‘n) else ⊥ fi)
nat2pos_app2       ⌊n≐#0⌋⟹nat2pos‘n=abs_sd (Pabs‘n)
nat2pos_strict     nat2pos‘⊥=⊥
nat2pos_total      ⟦n≠⊥;n≠#0⟧⟹nat2pos‘n≠⊥
```

```
nat2pos_zero        nat2pos‘#0=⊥
pos2nat_app         pos2nat‘n=Prep‘(rep_sd n)
pos2nat_total       x≠⊥⟹pos2nat‘x≠⊥
pos2nat_strict      pos2nat‘⊥=⊥
pos_not_zero        c≠⊥ ⟹ ⌊pos2nat‘c≐#0⌋
pos2nat_nat2pos     n≠#0⟹pos2nat‘(nat2pos‘n)=n
nat2pos_pos2nat     nat2pos‘(pos2nat‘p)=p
```

The positive numbers are the basis for the quotient construction, which leads to the rational numbers.

A.3.9 Rat

The ADT of fractionals for expressing rational numbers is described in Section 7.1.6.

```
Rat0 = Pos +  (* rationals with fractional representation *)
(* embedding *)
domain R = Rabs(num::dnat)(den::pos)

defs
R_eq_def         (op ≐≐) ≡ Λx y.num‘x mult pos2nat‘(den‘y) ≐ \
\                                  num‘y mult pos2nat‘(den‘x)
R_per_def        (op ∼∼) ≡ λx y::R.⌈x≐≐y⌉
end
```

The following theorems are proved for `Rat0`:

```
        (* general theorems on R *)
R_con_sel           Rabs‘(num‘r)‘(den‘r)=r
        (* derive the eq−axioms for R *)
R_eq_refl           (c::R)≠⊥⟹⌈c≐≐c⌉
R_eq_sym            ⌈(x::R) ≐≐ y⌉ ⟶ ⌈y ≐≐ x⌉
R_eq_trans          ⌈(a::R) ≐≐ b⌉ ∧ ⌈b ≐≐ c⌉ ⟶ ⌈a ≐≐ c⌉
R_eq_strict1        (⊥::R) ≐≐ b=⊥
R_eq_strict2        b ≐≐ (⊥::R) = ⊥
R_eq_total          ⟦a≠⊥;(b::R)≠⊥⟧⟹a ≐≐b ≠⊥
R_per_def           a∼∼(b::R)=⌈a ≐≐b⌉
```

These axioms justify the instantiation into the type classes `eq` and `adm`.

```
Rat = Rat0 +   (* type of positive rationals *)
instance R::eq  ( R_eq_refl,R_eq_sym,R_eq_trans,
                        R_eq_strict1,R_eq_strict2,R_eq_total,
                        R_per_def)
types Rat = R quot
ops curried
        "--" :: dnat → dnat → Rat (cinfixl 90)
defs
        fract_def       (op --) ≡ Λx y.<⌊Rabs‘x‘(nat2pos‘y)⌋>
end
```

The following theorems are proved for `Rat0`:

```
        (* theorems for fractal representations *)
fract_app            x--y = <⌊ Rabs‘x‘(nat2pos‘y) ⌋>
fract_rule           ⟦⌊n1≐#0⌋;⌊n2≐#0⌋;⌈z1 mult n2≐z2 mult n1⌉⟧
                       ⟹ z1--n1 = z2--n2
fract_zero_rule      ⌈#0≐n⌉⟹z--n=⊥
```

These rules allow us to deduce properties like `#2--#4 = #6--(#1 ↑ #2)` easily.

Bibliography

[AG90] C. Ashworth and M. Goodland. *SSADM: A Practical Approch*. McGraw-Hill, London, 1990.

[AI91] D. Andrews and D. Ince. *Practical Formal Methods with VDM*. Series in Software Engeneering. McGraw-Hill, 1991.

[AVW91] J. L. Armstrong, R. H. Virding, and M. C. Williams. *Erlang User's Guide and Reference Manual Version 3.2*. Ellemtel Utvecklings AB, Sweden, 1991.

[Bac78] R. J. R. Back. *On the Correctness of Refinement Steps in Program Development*. PhD thesis, University of Helsinki, 1978. Also available as report A-1978-5.

[Bac88] R. J. R. Back. A Calculus of Refinements for Program Derivations. *ACTA Informatica*, 25(6):593–624, 1988.

[Bac90] R. J. R. Back. Refinement calculus, part II: Parallel and reactive programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[BBB+85] F.L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, and B. Krieg-Brückner. *The Munich Project CIP, Vol 1: The Wide Spectrum Language CIP-L.*, volume 183 of *LNCS*. Springer, 1985.

[BC93] F. W. Burton and R. D. Cameron. Pattern Matching with Abstract Data Types. *Journal of Functional Programming*, 3(2):171–190, 1993.

[BDD+92] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T.F. Gritzner, and R. Weber. The design of distributed systems - an introduction to FOCUS. Technical Report TUM-I9203, Technische Universität München, Institut für Informatik, Januar 1992.

[BDDG93] Ralph Betschko, Sabine Dick, Klaus Didrich, and Wolfgang Grieskamp. Formal Development of an Efficient Implementation of a Lexical Scanner within the KorSo Methodology Framework. Forschungsberichte des Fachbereichs Informatik 93-30, Technische Universität Berlin, Franklinstraße 28/29 - D-10587 Berlin, October 1993.

[BFG+93a] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Specification Language SPECTRUM. An Informal Introduction. Version 1.0. Part I. Technical Report TUM-I9311, Technische Universität München. Institut für Informatik, May 1993.

[BFG+93b] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Specification Language SPECTRUM. An Informal Introduction. Version 1.0. Part II. Technical Report TUM-I9312, Technische Universität München. Institut für Informatik, May 1993.

[BFG+94] M. Broy, M. Fuchs, T. F. Gritzner, B. Schätz, K. Spies, and K. Stølen. Summary of Case Studies in FOCUS — a Design Method for Distributed Systems. SFB-Bericht 342/13/94 A, Technische Universität München, June 1994.

[BHN+94] M. Broy, U. Hinkel, T. Nipkow, Ch. Prehofer, and B. Schieder. Interpreter verification of a functional language, 1994.

[BJ95] Manfred Broy and Stefan Jähnichen, editors. *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, volume 1009 of *LNCS*. Springer, Heidelberg, Nov 1995.

[BMPW86] M. Broy, B. Möller, P. Pepper, and M. Wirsing. Algebraic Implementations Preserve pPogram Correctness. *Science of Computer Programming*, 7:35–53, 1986.

[BMS96a] M. Broy, S. Merz, and K. Spies, editors. *Formal Systems Specification – The RPC-Memory Specification Case Study*. Springer, LNCS 1169,, November 1996.

[BMS96b] M. Broy, S. Merz, and K. Spies. The RPC-Memory Specification Problem: A Synopsis. In M. Broy, S. Merz, and K. Spies, editors, *Formal Systems Specification – The RPC-Memory Specification Case Study*, pages 5–20. Springer, LNCS 1169,, November 1996.

[Boo93] G. Booch. *Object-Oriented Analysis and Design with Applications, Second Edition*. Benjamin/Cummings, 1993.

[Bre92] M. Breu. Bounded Implementation of Algebraic Specifications. *Lecture Notes in Computer Science*, 655:181–??, 1992.

[Bro82] M. Broy. Algebraic methods for program construction: The project CIP. In *SOFSEM 82*, 1982. Also in: P. Pepper (ed.): Program Transformation and Programming Environments. NATO ASI Series. Series F: 8. Berlin-Heidelberg-New York-Tokyo: Springer 1984, 199–222.

[Bro92] M. Broy. Compositional refinement of interactive systems. Technical Report SRC 89, DIGITAL Systems Research Center, 1992.

[Bro93] M. Broy. Interaction refinement the easy way. In M. Broy, editor, *Program Design Calculi. Springer NATO ASI Series, Series F: Computer and System Sciences, Vol. 118*, 1993.

[Bro96] M. Broy. Mathematical Methods in System and Software Engineering, 1996. Marktoberdorf Summer School.

[BS] Manfred Broy and Ketil Stølen. FOCUS on System Development – A Method for the Development of Interactive Systems. to appear.

[BW82] F.L. Bauer and H. Wössner. *Algorithmic Language and Program Development.* Springer, Berlin, 1982.

[BW88a] Broy and Wirsing. Ultra-Loose Algebraic Specification. *Bulletin of the European Association for Theoretical Computer Science*, 35, 1988.

[BW88b] M. Broy and M. Wirsing. Ultra-Loose Algebraic Specification. *Bericht MIP-8817, Universität Passau*, 1988.

[BW95] Manfred Broy and Martin Wirsing. Correct Software: From Experiments to Applications. In Broy and Jähnichen [BJ95].

[Che76] P. P. Chen. The Entity Relationship Model - Toward an Unified View of Data. *ACM Trans. on Database Sys.*, 1(1):9, March 1976.

[CM88] K. M. Chandy and J. Misra. *Parallel Programming Design.* Addison-Wesley, 1988.

[CZdR92] J. Coenen, J. Zwiers, and W.-P. de Roever. A Note on Compositional Refinement. Technical report, Eindhoven, 1992.

[Dav90] A. M. Davis. *Software Requirements: Analysis and Specification.* Prentice-Hall, Englewood Cliffs, NJ, 1990.

[DCC92] E. Downs, P. Clare, and I. Coe. *Structured systems analysis and design method (2nd ed).* Prentice-Hall, 1992.

[Dem78] Tom Demarco. *Structured Analysis and System Specification.* Prentice Hall, Englewood Cliffs, 1 edition, 1978.

[Den91] Ernst Denert. *Software-Engineering - methodische Projektabwicklung.* Springer, Berlin, 1991.

[Dil94] A. Diller. *Z: An Introduction to Formal Methods.* John Wiley & Sons, Chichester, UK, 2nd edition, 1994.

[EGL89] Hans-Dieter Ehrich, M. Gogolla, and U. Lipeck. *Algebraische Spezifikation Abstrakter Datentypen.* B.g. Teubner, Stuttgart, 1989.

[EKMP82] H. Ehrig, H.-J. Kreowski, B. Mahr, and P. Padawitz. Algebraic Implementation of Abstract Data Types. *Theoretical Computer Science*, 20:209–263, 1982.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer, 1985.

[EM90] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constrains.* Springer, 1990.

[End72] H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.

[Far94a] W. M. Farmer. A General Method for Safely Overwriting Theories in Mechanized Mathematics Systems. Technical report, The MITRE Corporation, 1994. ftp://math.harvard.edu/imps/doc/overwriting-theories.dvi.gz.

[Far94b] William M. Farmer. Theory Interpretation in Simple Type Theory. In J. Heering, K. Meinke, B. Möller, and T. Nipkow, editors, *Higher-Order Algebra, Logic and Term Rewriting*, volume 816 of *LNCS*, pages 96–123. Springer, 1994.

[Fla95] David Flanagan. *Java in a Nutshell: a desktop quick reference for Java programmers.* A Nutshell handbook. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1995.

[Fuc95] Maximilian Fuchs. Formal Design of a Model-N Counter. Technical Report TUM-I9512, Technische Universität München. Institut für Informatik, 1995.

[Gan83] Harald Ganzinger. Denotational Semantics for Languages with Modules. In Dines Björner, editor, *Formal Description of Programming Concepts II*, pages 3–23. North-Holland, Amsterdam, 1983.

[GM85] N. Gehani and A. McGettrick, editors. *Software Specification Techniques.* Addison-Wesley, Wokingham, 1985.

[GM93] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[Gor85] M.C.J. Gordon. HOL — A Machine Oriented Formulation of Higher-Order Logic. Technical Report 68, University of Cambridge, Computer Laboratory, 1985.

[GS77] C. Gane and E. Sarson. *Structured Systems Analysis*. Prentice-Hall, 1977.

[Gur92] Gurevich. Evolving Algebras. *BCTCS: British Colloquium for Theoretical Computer Science*, 8, 1992.

[Han80] G. Hansoul. *Systemes Relationelles et Algebres Multiformes*. PhD thesis, Universite de Liege, 1980.

[Har86] R. Harper. Introduction to Standard ML. Technical Report ECS-LFCS-86-14, University of Edinburgh,Department of Computer Science, November 1986.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex System. *Science of Computer Programming*, 8(3):231–274, 1987.

[Hen63] Leon Henkin. A theory of propositional types. *Fund. Math. 52*, 1963.

[Hen91] Rolf Hennicker. Context Induction: A Proof Principle for Behavioural Abstractions and Algebraic Implementations. *Formal Aspects of Computing*, 3(4):326–345, 1991.

[Het95] Rudolph Hettler. *Entity/Relationship-Datenmodellierung in axiomatischen Spezifikationssprachen*. Tectum-Verlag, 1995.

[HJW92] P. Hudak, S. Peyton Jones, and P. Wadler, editors. *Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2)*. ACM SIGPLAN Notices, May 1992.

[HKB93] Berthold Hoffmann and B. Krieg-Bruckner. *Program development by specification and transformation: the PROSPECTRA methodology, language family, and system*, volume 680 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1993.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

[Hoa72] C. A. R. Hoare. Proof of Correctness of Data Representation. *Acta Informatica*, 1:271–281, 1972.

[Hoa84] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1984.

[Hoa96] C.A.R. Hoare. Unified Theories of Programming, 1996. Marktoberdorf Summer School.

[Hot95] Alfred Hottarek. Übersetzung von SPECTRUM- in the large nach ML. Master's thesis, Technische Universität München, 1995. http://www4.informatik.tu-muenchen.de/papers/DA_Hottarek.ps.gz.

[HP87] D. J. Hatley and I. Pirbhai. *Strategies for Real-Time System Specification.* Dorset House, New York, N.Y., 1987.

[HR94] A. Hottarek and J. Rudolph. SPECTRUM nach ML Übersetzer. Fortgeschrittenenpraktikum, Technische Universität München, 1994.

[HSS96] Franz Huber, Bernhard Schätz, and Katharina Spies. AutoFocus - Ein Werkzeugkonzept zur Beschreibung verteilter Systeme . In Ulrich Herzog Holger Hermanns, editor, *Formale Beschreibungstechniken für verteilte Systeme*, pages 165–174. Universität Erlangen-Nürnberg, 1996. Erschienen in: Arbeitsbereichte des Insituts für mathematische Maschinen und Datenverarbeitung, Bd.29, Nr. 9 .

[Jac92] I. Jacobson. *Object-Oriented Software Engineering.* Addison-Wesley, Massachusetts, 1992.

[Jon93] M. P. Jones. *An Introduction to Gofer*, August 1993.

[KA84] Samuel Kamin and Mila Archer. Partial Implementations of Abstract Data Types: A Dissenting View on Errors. In Giles Kahn, David MacQueen, and Gordon Plotkin, editors, *Semantics of Data Types*, pages 317–336. Springer, 1984. Lecture Notes in Computer Science, Volume 173.

[Kah74] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, N.Y., 1974.

[KB94] Bernd Krieg-Brückner. Programmentwicklung durch Spezifikation und Translation. Technical Report 1/94, Universität Bremen, February 1994.

[Lam94] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lio96] J. L. Lions. ARIANE 5 Failure - Full Report, 1996. http://www.it.dtu.dk/ hra/ariane5rep.html.

[LT88] Kim G. Larsen and Bent Thomsen. A Modal Process Logic. In *Proceedings, Third Annual Symposium on Logic in Computer Science*, pages 203–210, Edinburgh, Scotland, 5–8 July 1988. IEEE Computer Society.

[LT89] N. A. Lynch and M. R. Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989. Also available as MIT Technical Memo MIT/LCS/TM-373.

[MB9x] R. Hennicker M. Bidoit. Behavioural Theories and The Proof of Behavioural Properties. *Theoretical Computer Science*, 199x. to appear.

[Meh95] M. Mehlich. *Implementation of Combinator Specifications.* PhD thesis, Ludwig-Maximilians-Universität München, 1995.

[Mel89] Tom Melham. Automating Recursive Type Definitions in Higher Order Logic. In Graham Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer-Verlag, 1989.

[Mil83] Robert Milner. *CCS - A Calculus for Communicating Systems*, volume 83 of *Lecture Notes in Computer Science.* Springer Verlag, 1983.

[Mil89] R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[Mit86] John C. Mitchell. Representation Independence and Data Abstraction. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 263–276. ACM, ACM, January 1986.

[Möl87] Möller. *Higher-Order Algebraic Specifications.* PhD thesis, TUM, 1987. Habilitationsschrift.

[MS96] Olaf Müller and Konrad Slind. Isabelle/HOL as a Platform for Partiality. In *Mechanization of Partial Functions (CADE-13 Workshop)*, 1996.

[MV94] C. C. Morgan and T. Vickers, editors. *On the Refinement Calculus.* Formal Approaches to Computing and Information Technology series (FACIT). Springer-Verlag, 1994.

[MVS85] T.S.E. Maibaum, Paulo A. Veloso, and M.R. Sadler. A theory of Abstract Data Types for Program Development: briging the Gap. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *Formal Methods and Software Engineering(TAPSOFT'85)*, volume 2, pages 214–230. Springer-Verlag, 1985.

[Naz95] D. Nazareth. *A Polymorphic Sort System for Axiomatic Specification Languages.* PhD thesis, Technische Universität München, 1995. Technical Report TUM-I9515.

[Nip86] Tobias Nipkow. Non-deterministic Data Types: Models and Implementations. *Acta Informatica*, 22(16):629–661, March 1986.

[Nip87] Tobias Nipkow. *Behavioural Implementation Concepts for Nondeterministic Data Types*. Ph.D. thesis, University of Manchester, Computer Science Department, May 1987.

[Nip91] T. Nipkow. Order-Sorted Polymorphism in Isabelle. In G. Huet, G. Plotkin, and C. Jones, editors, *Proc. 2nd Workshop on Logical Frameworks*, pages 307–321, 1991.

[NRS96] Dieter Nazareth, Franz Regensburger, and Peter Scholz. Mini-Statecharts: A Lean Version of Statecharts. TUM-I 9610, Technische Universität München, 1996.

[NS95] Tobias Nipkow and Konrad Slind. In P. Dybjer B. Nordström J. Smith, editor, *Proc. of Types for Proofs and Programs, LNCS 996*, pages 101–119, 1995.

[Ohe95] David von Oheimb. Datentypspezifikationen in Higher-Order LCF. Master's thesis, Technische Universität München, 1995. "http://www4.informatik.tu-muenchen.de/Papers/DA_Oheimb.html".

[Ohe97] David von Oheimb. Isabelle Case Study of a Buffer of Length One. Isabelle Theories, to be published, 1997.

[ONS96] Fernando Orejas, Marisa Navarro, and Ana Sánchez. Algebraic Implementation of Abstract Data Types: a Survey of Concepts and new Compositionality Results. *Mathematical Structures in Computer Science*, 6(1):33–67, February 1996.

[Pau87] L. C. Paulson. *Logic and Computation, Interactive Proof with Cambridge LCF*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1987.

[Pau92] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1992.

[Pau94a] Lawrence C. Paulson. A Fixedpoint Approach to Implementing (co)Inductive Definitions. In A. Bundy, editor, *Automated Deduction — CADE-12*, volume 814 of *Lecture Notes in Computer Science*, pages 148–161. Springer, 1994.

[Pau94b] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

[Pax94] V. Paxon. Groth Trends in Wide-Area TCP Connections. *IEEE Network Magazine*, 33:8–17, July 1994.

[PBDD95] Peter Pepper, Ralph Betschko, Sabine Dick, and Klaus Didrich. Realizing Sets by Hash Tables. In Broy and Jähnichen [BJ95].

[Pra76] Vaughan R. Pratt. Semantical Considerations on Floyd-Hoare Logic. In *17th Annual Symposium on Foundations of Computer Science*, pages 109–121. IEEE, October 1976.

[PW95] Peter Pepper and Martin Wirsing. A Method for the Development of Correct Software. In Broy and Jähnichen [BJ95].

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Solutions Manual: Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, 1991.

[Rea89] C. Reade. *Elements of Functional Programming.* Addison-Wesley, Wokingham, Berks., 1989.

[Reg94] Franz Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF.* PhD thesis, Technische Universität München, 1994.

[Reg95] Franz Regensburger. HOLCF: Higher Order Logic of Computable Functions. In E.T. Schubert, P.J. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 293–307. Springer, 1995.

[Rei80] Wolfgang Reif. *Korrektheit von Spezifikationen und generischen Moduln.* PhD thesis, Karlsruhe, 1980.

[Rei92] Wolfgang Reif. The KIV System: Systematic Construction of Verified Software. In *11th Conference on Automated Deduction*, volume 607 of *lncs*, pages 753–757, 1992.

[RH9x] M. Bidoit R. Hennicker, M. Wirsing. Proof Systems for Structured Specifications with Observability Operators. *Theoretical Computer Science*, 199x. to appear.

[Rob89] Edmund Robinson. How Complete is PER? In *Fourth Annual Symposium on Logic in Computer Science*, pages 106–111, 1989.

[Roy70] Winston W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *WESCON Technical Papers, v. 14*, pages A/1–1–A/1–9, Los Angeles, August 1970. WESCON. First Occurence of the Term Waterfall Model.

[Rut67] H. Rutishauser. *Description of Algol 60.* Handbook for Automatic Computation, Vol. 1a. Springer-Verlag, Berlin, 1967.

[Sch70] J. R. Schoenfield. *Mathematical logic.* Addison-Wesley, 1970.

[Sch83] O. Schoett. A Theory of Program Modules, their Specification and Implementation. Technical Report CSR-155-83, University of Edinburgh, 1983.

[Sch85] Oliver Schoett. Behavioural Correctness of Data Representations. Internal Report CSR-185-85, Department of Computer Science, University of Edinburgh, April 1985.

[Sch90] O. Schoett. Behavioural correctness of data representations. *Science of Computer Programming*, 14:43–57, 1990.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley Sons, 1994.

[Slo95] Oscar Slotosch. Implementing the Change of Data Structures with SPECTRUM in the Framework of KORSO Development Graphs. Technical Report TUM-I9511, Technische Universität München. Institut für Informatik, 1995.

[SM96] O. Slotosch and W. Mala. KORSYS-Prozeßmodell mit Kritikalen Aspekten. http://www4.informatik.tu-muenchen.de/proj/korsys/public/pmod.ps.gz, 1996.

[SM97] Robert Sandner and Olaf Müller. Theorem Prover Support for the Refinement of Stream Processing Functions. In *Proc. 3rd Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, Lecture Notes in Computer Science. Springer, 1997.

[Spi94] K. Spies. Funktionale Spezifikation eines Kommunikationsprotokolls. SFB-Bericht 342/08/94 A, Technische Universität München, May 1994.

[SS71] Dana Scott and Christopher Strachey. Toward a Mathematical Semantics for Computer Languages. pages 19–46. April 1971.

[SS95] Bernhard Schätz and Katharina Spies. Formale Syntax zur logischen Kernsprache der FOCUS-Entwicklungsmethodik. TUM-I 9529, Technische Universität München, 1995. http://www4.informatik.tu-muenchen.de/reports/TUM-I9529.html .

[ST88] D. Sannella and A. Tarlecki. Toward Formal Development of Programs from Algebraic Specifications: Implementation Revisited. *Acta Informatica*, 25:233–281, 1988.

[Sta94] R. Stabl. The CSDM System - an Environment for the Algebraic Specification Language SPECTRUM. In *Bericht zum GI-Workshop, Bad Honnef*, 1994.

[Tan96] Andrew S. Tanenbaum. *Computer Networks*. P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, third edition, 1996.

[Tho90] S. J. Thompson. Lawful Functions and Program Verification in Miranda. *Science of Computer Programming*, 13:181–218, 1989/90.

[Tur85] D.A. Turner. Miranda — a Non-Strict Functional Language with Polymorphic Types. In Jouannaud, editor, *Conference on Functional Programming Languages and Computer Architecture*, pages 1–16. Springer Verlag, 1985.

[UK95] R. T. Udink and J. N. Kok. ImpUNITY: UNITY with procedures and local variables. *Lecture Notes in Computer Science*, 947:452–472, 1995.

[vD88] N. W. P. van Diepen. Implementation of Modular Algebraic Specifications (extended abstract). In H. Ganzinger, editor, *Proceedings of the European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 64–78. Springer Verlag, 1988.

[Völ95] Norbert Völker. On the Representation of Datatypes in Isabelle/HOL. Technical Report 379, University of Cambridge Computer Laboratory, 1995. Proceedings of the First Isabelle Users Workshop.

[Wan82] M. Wand. Specifications, Models, and Implementations of Data Abstractions. *Theoretical Computer Science*, 20(1):3–32, 1982.

[WB94] Michal Walicki and Manfred Broy. Structured Specifications and Implementation of Nondeterministic Data Types. TUM-I 9442, Technische Universität München, 1994. http://www4.informatik.tu-muenchen.de/reports/TUM-I9442.ps.gz.

[Wen94] Markus Wenzel. Axiomatische Typklassen in Isabelle. Master's thesis, Institut für Informatik, TU München, 1994.

[Wir72] Niklaus Wirth. The Programming Language PASCAL. *Acta Automatica*, 1(4), December 1972.

[Wir90] M. Wirsing. Algebraic Specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science.*, chapter 13, pages 675–788. North-Holland, Amsterdam, 1990.

[WM85] Paul T. Ward and Stephen J. Mellor. *Structured Development for Real-Time Systems*, volume volume 2: Essential Modeling Techniques. Yourdon, Inc., Englewood Cliffs, New Jersey, 1985.

List of Figures

List of Examples

List of Definitions

Index

Mathematical Symbols:

Definitions and Rules:

Isabelle:

Keywords: