



Basic Research in Computer Science

BRICS RS-94-44

S. Agerholm: A HOL Basis for Reasoning about Functional Programs

A HOL Basis for Reasoning about Functional Programs

Sten Agerholm

BRICS Report Series

RS-94-44

ISSN 0909-0878

December 1994

**Copyright © 1994, BRICS, Department of Computer Science
University of Aarhus. All rights reserved.**

**Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.**

**See back inner page for a list of recent publications in the BRICS
Report Series. Copies may be obtained by contacting:**

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk**

**BRICS publications are in general accessible through WWW and
anonymous FTP:**

**<http://www.brics.dk/>
[ftp ftp.brics.dk](ftp://ftp.brics.dk) (cd pub/BRICS)**

A HOL Basis for Reasoning about Functional Programs

by
Sten Agerholm

BRICS¹
Department of Computer Science
University of Aarhus
Ny Munkegade
DK-8000 Aarhus C, Denmark

¹Basic Research in Computer Science, Centre of the Danish National Research Foundation.

Summary

Domain theory is the mathematical theory underlying denotational semantics. This thesis presents a formalization of domain theory in the Higher Order Logic (HOL) theorem proving system along with a mechanization of proof functions and other tools to support reasoning about the denotations of functional programs. By providing a fixed point operator for functions on certain domains which have a special undefined (bottom) element, this extension of HOL supports the definition of recursive functions which are not also primitive recursive. Thus, it provides an approach to the long-standing and important problem of defining non-primitive recursive functions in the HOL system.

Our philosophy is that there must be a direct correspondence between elements of complete partial orders (domains) and elements of HOL types, in order to allow the reuse of higher order logic and proof infrastructure already available in the HOL system. Hence, we are able to mix domain theoretic reasoning with reasoning in the set theoretic HOL world to advantage, exploiting HOL types and tools directly. Moreover, by mixing domain and set theoretic reasoning, we are able to eliminate almost all reasoning about the bottom element of complete partial orders that makes the LCF theorem prover, which supports a first order logic of domain theory, difficult and tedious to use. A thorough comparison with LCF is provided.

The advantages of combining the best of the domain and set theoretic worlds in the same system are demonstrated in a larger example, showing the correctness of a unification algorithm. A major part of the proof is conducted in the set theoretic setting of higher order logic, and only at a late stage of the proof domain theory is introduced to give a recursive definition of the algorithm, which is not primitive recursive. Furthermore, a total well-founded recursive unification function can be defined easily in pure HOL by proving that the unification algorithm (defined in domain theory) always terminates; this proof is conducted by a non-trivial well-founded induction. In such applications, where non-primitive recursive HOL functions are defined via domain theory and a proof of termination, domain theory constructs only appear temporarily.

Acknowledgments

This work was supported in part by the DART project funded by the Danish Research Council (October 1991–December 1993) and in part by BRICS funded by the Danish National Research Foundation (January 1994–June 1994). I am grateful to my supervisor Glynn Winskel for providing this financial support and for the freedom he has allowed me in my work. Glynn also read a draft of the thesis. I would like to thank the following people for discussions concerning this work: Flemming Andersen, Ralph-Johan Back, Richard Boulton, Esben Dalsgård, Mike Gordon, Tom Melham, Kim Dam Petersen, Laurent They and Glynn Winskel. Torben Amtoft has been a good room mate for more than two and a half years, and commented on a draft of chapter 2. Larry Paulson helped with questions about the LCF system and dug up the LCF proof of correctness of the unification algorithm. I am grateful to Tom Melham and Mike Gordon for allowing me to stay at Cambridge University in the Autumn 1992. Tom made the practical arrangements.

Contents

1	Introduction	1
1.1	Domain Theory	2
1.2	The HOL System	3
1.3	Logic of Computable Functions	4
1.4	Goals	5
1.5	Outline	6
2	Overview	9
2.1	A Formalization of Domain Theory	10
2.2	The HOL-CPO System	20
2.2.1	Notations for Cpos and Pointed Cpos	21
2.2.2	Notation for Cpo-typable Terms	22
2.2.3	Other Syntactic-based Proof Functions	25
2.2.4	Other Tools	26
2.3	Recursive Functions	28
2.3.1	The Factorial Function	28
2.3.2	Ackermann's Function	30
2.3.3	Equality of Two Recursive Functions	33
2.4	Recursive Domains	36
2.4.1	Finite Values	37
2.4.2	Lazy Sequences	38
2.4.3	Infinite Values	41
2.5	Reasoning about Infinite Values	43
2.5.1	Structural Induction	43
2.5.2	Fixed Point Induction	45
2.5.3	Co-induction	46
2.6	Conclusion	47
3	Basic Concepts of Domain Theory	49
3.1	Representation	50
3.2	Partial Order	51
3.3	Complete Partial Order	53
3.4	Continuous Functions	54
3.5	Dependent Lambda Abstraction	55
3.6	Constructions	55
3.6.1	Discrete	56
3.6.2	Product	57

3.6.3	Continuous Function Space	60
3.6.4	Lifting	63
3.6.5	Sum	65
3.7	Identity and Composition	67
3.8	Fixed Point Operator	68
3.9	Fixed Point Induction	69
3.10	Proof of Continuity of Composition	70
3.10.1	Composition Preserves Continuity	70
3.10.2	Composition is Continuous	73
3.11	Discussion	75
3.11.1	Sets as Types	76
3.11.2	Sets as Subtypes	77
3.11.3	Comments on the Formalization	80
4	Recursive Domains	81
4.1	Finite-valued Recursive Domains	83
4.1.1	Lists	83
4.1.2	Trees	87
4.2	Infinite Sequences	88
4.2.1	Lazy Sequences	89
4.2.2	Lazy Lists	94
4.3	Infinite Labeled Trees	97
4.3.1	A Type of Infinite Trees	98
4.3.2	A Pointed Cpo of Infinite Trees	100
4.3.3	Constructors for Infinite Trees	102
4.4	Infinite-valued Recursive Domains	103
4.4.1	Example: Lazy Lists	104
4.4.2	The Method in General	105
4.5	More General Domains	107
4.5.1	Broad Trees	108
5	The HOL-CPO System	111
5.1	Notations for Cpos and Pointed Cpos	112
5.1.1	Algorithm for Proving Cpo Facts	113
5.1.2	Proving Cpo Facts	113
5.2	Notation for Cpo-typable Terms	114
5.2.1	Algorithm for Parsing	115
5.2.2	Algorithm for Type Checking	116
5.2.3	Examples of Parsing	118
5.2.4	Type Checking	120
5.2.5	Switching the Interface On and Off	122
5.3	Proving Inclusiveness	122
5.4	Extending Notations	126
5.5	Derived Definition Tools	128
5.6	Other Tools	130
5.6.1	Universal Cpos	130
5.6.2	Reduction by Definition	131

5.6.3	Fixed Point Induction	132
5.6.4	Cases on Lifted Cpos	133
5.6.5	Calculating Bottom in the Function Space	133
5.6.6	Function Equality	133
6	Some Simple Examples	135
6.1	Booleans and Conditionals	135
6.1.1	A Sum Cpo of Truth Values	135
6.1.2	A Discrete Universal Cpo of HOL Booleans	139
6.2	Natural Numbers	141
6.2.1	Reduction Theorems	143
6.2.2	The Factorial Function	144
6.2.3	Proof of a Reduction Theorem	146
6.3	Using Fixed Point Induction	150
6.4	A Simple Language and Its Semantics	159
7	LCF Examples	165
7.1	The LCF System	166
7.1.1	The Logic $PP\lambda$	167
7.1.2	Extending Theories	168
7.1.3	Rewriting	169
7.2	Natural Numbers	169
7.2.1	Theorems about Addition	172
7.2.2	Theorems about Equality	173
7.3	A Recursive Function	174
7.4	A Mapping Functional for Lazy Sequences	178
7.5	Conclusion	182
8	Verifying the Unification Algorithm	185
8.1	Terms	186
8.1.1	Occurrence Relation	187
8.1.2	Variable Set	188
8.2	Substitutions	188
8.2.1	Application	189
8.2.2	Domain and Range	190
8.2.3	Agreement and Equality	191
8.2.4	Composition	192
8.2.5	Generality	193
8.2.6	Idempotent Substitutions	193
8.3	Unifiers	194
8.3.1	Most-general and Idempotent Unifiers	195
8.3.2	Best Unifiers and their Existence	196
8.4	The Unification Algorithm	198
8.4.1	The Type of Attempts	198
8.4.2	Domain Theory	199
8.4.3	Defining the Algorithm	202
8.5	A HOL Unification Function	204

8.6	Proof of Correctness	205
8.6.1	The Well-founded Ordering	205
8.6.2	The Induction Proof	207
8.7	Discussion	208
9	Conclusion	209
9.1	Extension of HOL	210
9.2	Embedding Semantics vs. Implementing Logic	210
9.3	Alternative Formalizations	212
9.4	Limited Treatment of Recursive Domains	212
9.5	Related Work	213
9.6	Evaluation of HOL-CPO	214
9.7	Future Work: ‘Real’ Domain Theory	215
A	Well-founded Sets	217

Chapter 1

Introduction

Writing computer programs is difficult. Often, a malfunction is detected in programs which have been tested extensively. Sometimes, this can be a source of major irritation, for instance, if a text editor suddenly fails. At other times, the consequences of failure can be extremely high—a threat to human life—for applications in areas like transportation and health care, not to mention nuclear power generation.

The use of formal methods in developing safety-critical software can increase the confidence, and probability, that the software will behave as desired in a certain application. Formal methods are based on rigorous techniques, rooted in mathematics. Mathematical proofs of correctness are often suggested as a technique for ensuring reliability of a software system; a proof of correctness must establish that a formal mathematical understanding of a programs behavior meets a formal specification of the intended behavior. As pointed out by Cohn [Co89], the idea that software (and hardware) is proven “correct” is appealing but very misleading. A proof of correctness with respect to a specification does not guarantee desired behavior or non-failure since the proof may very well be wrong, and the specification may not specify intended behavior; even further, the compiler and executing hardware may be wrong, etc. In general, proofs of correctness are difficult to conduct since they tend to be long and complicated, and full of tedious details. A machine can help to keep track of the details, and to automate proofs of the most trivial details as well. Hence, a machine proof of correctness can often further increase the probability, but not guarantee, that programs behave as desired.

There are various ways of giving formal meaning, or formal semantics, to programming languages. Operational semantics specifies how a programming language executes on an abstract machine and denotational semantics is concerned with giving mathematical models for what programs mean; a program is mapped directly to its meaning, called its denotation, by a so-called semantic function from programming languages syntax to semantic domains. The denotation of a program is usually a mathematical value, such as a natural number or a function. The advantage of denotational (over operational) semantics is that it is more abstract, ignoring unimportant execution behavior and using mathematical concepts. The mathematical theory underlying denotational semantics is provided by *domain theory*—the study of complete partial orders, continuous functions and least fixed points.

This thesis presents a domain theoretic basis, called *HOL-CPO*, for reasoning about “functional programs” in the HOL (Higher Order Logic) theorem proving system. Along with a formalization of basic concepts of domain theory in higher order logic, HOL-CPO

provides a collection of proof functions and other tools to support reasoning about “functional programs”. To be honest, we shall omit the extra level of complication due to having an explicit programming language syntax and a semantic function from the syntax to semantic domains. Reasoning is conducted directly in domain theory about mathematical functions, which may be viewed as functional programs. We will see examples of function definitions in functional programming languages like Standard ML [MTH90, Pa91] and Miranda [BW88] which are translated easily to domain theory, and vice versa.

1.1 Domain Theory

Denotational semantics was pioneered by Christopher Strachey in the early 60’s (see e.g. [Mo90, Sc86]). In his work, Strachey used the untyped λ -calculus as a way of writing denotations, though, at the time, the untyped λ -calculus did not have a formal model in which λ -terms represented mathematical functions. It was a fundamental breakthrough when Dana Scott discovered a model of the untyped λ -calculus in the late 60’s by constructing a non-trivial solution D_∞ of the recursive domain equation

$$D_\infty \cong [D_\infty \rightarrow D_\infty].$$

Scott’s work underpinned the area of denotational semantics with a rich mathematical theory, called domain theory (see e.g. [GS90, Gu92, Wi93]). In recent years, research in domain theory has been conducted in category theory, where new important results have contributed to a renewed interest in the theory of domains.

A central purpose of domain theory is to give an understanding of recursive definitions. A function f may be recursively defined by an equation of the following form:

$$f(x) = \dots f(\dots x \dots) \dots,$$

where f appears on both sides of the equality. A well-known example is a recursive specification of the *factorial* function which takes a natural number n as an argument and produces the factorial $n! = n * (n - 1) * \dots * 1$ as a result:

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{fact}(n - 1) & \text{if } n > 0. \end{cases}$$

In general, such an equation does not need to specify a unique function, or even any function. Another question is whether it is possible to give a canonical value for a solution of such an equation.

Domain theory is concerned with the existence and uniqueness of solutions of equations as canonical least fixed points. The fixed points are taken of non-recursive functionals determined by the recursive definitions. If a recursive definition determines a function $F : E \rightarrow E$ on a ‘domain’ E , which contains a least element \perp , read “bottom”, with respect to an ordering relation \sqsubseteq , then the term being defined is interpreted as the *least* fixed point of F . A canonical value for this fixed point is specified as the least upper bound of the ω -chain

$$\perp \sqsubseteq F(\perp) \sqsubseteq F^2(\perp) \sqsubseteq \dots \sqsubseteq F^n(\perp) \sqsubseteq \dots$$

of partial approximations of the term, obtained by recursively unfolding the recursive definition. Domains are ensured to always contain such least upper bound by requiring that they are *complete partial orders* (cpo). A cpo D may be viewed as a set with structure, induced by a partial ordering relation \sqsubseteq_D (often the subscript is omitted). In general, our domains are not required to possess bottom elements; they are so-called bottomless cpo, or predomains. However, we shall add bottom elements to take the least fixed points of functions, which are required to be *continuous* to allow this.

Apart from giving an understanding of recursive functions, domain theory gives an understanding of nontermination and partial functions, via the bottom element of domains. Further, it provides a range of techniques for reasoning about recursive definitions, e.g. fixed point induction and well-founded induction (see [Wi93]).

Another major issue in domain theory is the construction of solutions to recursive domains equations. Many programming languages allow the use of recursively defined types. Even if they do not it may be that their semantics is most straightforwardly defined through the use of recursively defined domains. A recursively defined domain is constructed as a solution of a recursive domain equation of the form:

$$D \cong F(D)$$

Here, F is an operator on domains defined using standard constructions on domains like product and function space. If D_1 and D_2 are domains then their product $D_1 \times D_2$, consisting of pairs of elements in D_1 and D_2 , respectively, is a domain ordered component-wise. The function space $[D_1 \rightarrow D_2]$, consisting of continuous functions from D_1 to D_2 , is a domain ordered pointwise. The three most prominent techniques for solving recursive domain equations are:

- the categorical method using embedding project pairs, based on Scott’s original inverse limit construction [SP82, Pa87, Pl83],
- via universal domains like $P\omega$ in which domains are encoded as retracts [Sc76, St77, Ba84], and
- information systems providing a representation of domains for which equations are solved by the fixed point method [Sc82, LW91, Wi93].

None are considered in this thesis since they do not fit in well with other goals (see below). Instead we apply a few ad hoc methods to introduce more restricted classes of recursive domains.

1.2 The HOL System

The HOL system [Go89a, GM93] is not a fully automated theorem prover but a mechanized proof-assistant for proving theorems in higher order logic. It is a general-purpose theorem prover since it is based on an expressive higher order logic (the HOL logic) and built on top of a functional programming environment ML (which stands for Meta Language). The HOL logic is a typed logic with terms, types, and theorems represented in the ML language. The logic is organized in theories each of which contains a set of types, constants, definitions, axioms and theorems. The purpose of the HOL system is to provide tools for constructing such theories.

Theories can be extended with new constants and types by giving definitions and axioms. Definitional extension is safe, meaning it preserves consistency of the HOL logic, because new constants and types are defined in terms of existing ones. Axiomatic extension is not safe and usually not accepted in the HOL community.

The representation of theorems in ML as an abstract type guarantees that theorems can only be created by formal proof. A proof is a derivation using a number of inference rules, pre-proved theorems and axioms. An inference rule is a function in ML which takes a number of theorems (premises) as arguments and produces a theorem as a result. All inference rules are derived from eight primitive rules, or other so-called derived inference rules. Conversions are special cases of inference rules which take no theorem arguments but instead a term argument.

Inference rules support forward proofs of theorems. However, a more natural goal-directed (or backwards) proof style is also supported—by the subgoal package. This allows proofs of theorems to be constructed by applying tactics interactively, in order to reduce goal terms to truth. A tactic is an ML function which typically implements the backwards use of one or more inference rules.

The HOL logic has a set theoretic semantics. All types denote sets and the function type denote total functions of set theory. The HOL system supports extensions well, through its expressive underlying logic and the meta language ML which can be used to program special-purpose proof functions and other tools. In particular, it supports reasoning about certain concrete recursive finite-valued datatypes and primitive recursive functions on these types well, due to the type definition package [Me89]. It is difficult to introduce more general recursive types and functions since one must first prove their existence in the logic. Finally, HOL has a large collection of built-in types, theorems and proof tools to support all kinds of reasoning. This is important since it means that one does not have to start from scratch when a new extension is considered.

1.3 Logic of Computable Functions

Scott's Logic of Computable Function was implemented in the LCF theorem prover, a system designed specifically for reasoning about the semantics of programming languages. The originator of the LCF system was Robin Milner, who implemented the first version of the system, a simple proof checker, at Stanford University (see the bibliography of [GMW79]). Out of the experiences with this system grew Edinburgh LCF, for which the general-purpose programming language ML was designed and developed [GMW79]. Later a version of the system called Cambridge LCF was developed by Paulson [Pa87] at Cambridge University. The LCF project is arguably one of the most significant in mechanical theorem proving. For instance, the tactical approach to proof was developed as part of this project and the successor of ML, called Standard ML [MTH90, Pa91], has become one of the most widely used functional programming languages. In fact, the HOL system is a direct descendant of the LCF system and shares an implementation in ML and many ideas on mechanical theorem proving with LCF.

Experiences with the LCF system, due to numerous applications (see the bibliography of [GMW79, Pa87]), are mixed. While, on the one hand, it seems to support reasoning about infinite-valued (lazy) types and non-strict functions (lazy evaluation) well, it seems to be less suited for reasoning about finite-valued (strict) types and strict functions (eager

evaluation), due to the presence of bottom elements in all types [Pa84a, Pa84b, Pa85]. The underlying logic of the LCF system is a first order logic of domain theory where well-formed terms have types which all denote cpos with bottom elements.

Another problem with LCF is that it is an implementation of a logic of domain theory; hence, there is no access to domain theory itself. For instance, there is no definition of the fixed point operator in LCF. The fixed point operator is provided via axioms and a primitive rule of inference, called fixed point induction. But using other techniques for recursion, or reasoning directly about fixed points, allows more theorems to be proved than with just fixed point induction. Further, the semantic notion of admissibility, or inclusiveness, of predicates for fixed point induction is only available via an incomplete syntactic check, performed by the rule of fixed point induction which is implemented in ML. Thus, if a predicate is not accepted by the syntactic check, then it cannot be used with fixed point induction in LCF, though it may be inclusive by the semantic definition in domain theory. The main properties of the LCF system are described in more detail in chapter 7.

1.4 Goals

It is often argued that machine-assisted program verification should be conducted in a semantics-based extension of a safe and general-purpose theorem prover such as the HOL system (see e.g. [Ag92]). The program verifiers for imperative programs described by Gordon in [Go89b] and by the present author in [Ag91, Ag92] satisfy this. Further, Andersen (et al.) [An92, APP93] employs this approach to develop a theorem prover for the UNITY theory of concurrent programs. Another example is the CCL (Classical Computational logic) extension of Isabelle [Pa90] by Coen [Co92] for reasoning about functional programs.

The work presented here is based on the same idea. By embedding domain theory in the HOL system, we hope to overcome some of the limitations of the LCF system, which are due to the fact that it is a direct implementation of a logic. In a way, the work may be viewed as an embedding of (the logic of) the LCF system within the HOL system, which is performed in such a way that the benefits of both the domain theoretic LCF “world” and the set theoretic HOL “world” are preserved. The last point is important: the benefits of the HOL system should be available in the extension of HOL. It should be possible to benefit from the ease of set theoretic reasoning, compared with domain theoretic reasoning which often involves bottom. It should be possible to mix the two different kinds of reasoning. Furthermore, the rich collection of types, theorems and tools provided with the HOL system should be directly accessible when reasoning about recursive definitions in domain theory. This thesis gives a thorough treatment of recursively defined functions I believe that a thorough treatment would not fit in directly with preserving the benefits of set theoretic reasoning in HOL (cf. the introduction to chapter 4 or chapter 9).

Domain theory is a rich mathematical theory which is useful to reason about various aspects of programming languages and individual programs. It should therefore be available to the HOL user. The main results of this thesis are:

- A formalization of basic domain theoretic concepts in HOL with syntactic-based proof functions and other tools to support reasoning about mathematical function

(the denotations of functional programs). This extension of HOL is called HOL-CPO.

- A good number of examples to demonstrate the use and usability of the HOL-CPO system. A larger example shows the proof of correctness of a unification algorithm. It is demonstrated that domain and set theoretic reasoning can be mixed to advantage, e.g. most reasoning in LCF involving the bottom element can be eliminated, and a richer class of recursive functions is supported than in pure HOL.
- A thorough comparison of LCF and HOL-CPO.
- A method for introducing derived definitions of recursively defined well-founded functions in HOL based on domain theory and well-founded induction.
- Ad hoc methods and ideas for defining recursive domains with finite and infinite values.

Parts of the work have also been published in [Ag93, Ag94a].

1.5 Outline

The contents of each chapter may be outlined as follows:

Chapter 2: Overview. The purpose of this chapter is to provide an overview of the formalization and main points and results of the work. The presentation is relatively non-technical and independent of HOL syntax. The chapter describes examples which are and which are not presented later in the thesis. In particular, it treats the well-founded recursive Ackermann function, co-induction for lazy sequences (lazy lists without the empty list) and an example which combines several techniques of domain theory for reasoning about recursive definitions.

Chapter 3: Basic Concepts of Domain Theory. A formalization of basic concepts of domain theory such as complete partial orders, continuous functions and least fixed points is described. A few constructions on domains are also presented, for instance, the discrete construction (associates the discrete ordering with a subset of a HOL type), the lifting construction (adds a bottom element to a cpo) and the function space construction.

Chapter 4: Recursive Domains. Some recursive domains with finite values may be introduced via recursive HOL types and the discrete construction, or as variants of such cpos. Cpo constructions for recursively defined cpos of lazy sequences and lazy lists, which contain infinite values, are introduced but the method is not easily generalized to more complicated domains. Ideas on more powerful techniques for defining some recursive domains are presented.

Chapter 5: The HOL-CPO System. Syntactic notations for writing cpos, continuous functions and inclusive predicates are presented. These are implemented by an interface and a number of syntactic-based proof functions. The notations can be extended interactively. Other tools such as a tactic for fixed point induction and derived definition tools for introducing cpos and elements of cpos are also presented.

Chapter 6: Some Simple Examples. Some first simple examples illustrate the use of HOL-CPO. The examples illustrate how to introduce new cpos and recursive continuous function easily, and how to extend the notations for writing cpos and elements of cpos easily. Proof by fixed point induction is also illustrated.

Chapter 7: LCF Examples. A few examples presented in chapter 10 of Paulson's book on Cambridge LCF are conducted in HOL-CPO in order to allow a comparison of the two systems. The examples illustrate reasoning about finite-valued domains like the natural numbers, arbitrary recursive functions and infinite-valued domains like lazy sequences.

Chapter 8: Verifying the Unification Algorithm. As an extended example we present a proof of correctness of the unification algorithm which was verified earlier by Manna and Waldinger [MW81] and by Paulson in LCF [Pa87]. The proof involves substantial theories of substitutions and unifiers and is based on the use of well-founded induction to prove termination of the algorithm.

Chapter 9: Conclusion. Conclusions are drawn and further work suggested.

The presentation is fairly technical and does require some knowledge of domain theory and the HOL system. Chapter 2 should be more accessible than the other chapters (except the conclusion).

Chapter 2

Overview

The purpose of this chapter is to provide a more accessible presentation of HOL-CPO, the extension of HOL with domain theory, than the following chapters provide; these are quite HOL technical and provide more details (i.e. discussions, explanations, definitions and theorems). It is hoped that a reader with some or little knowledge of domain theory and the HOL system can read and understand this chapter well enough to obtain a brief overview of the work.

The HOL-CPO system consists of various integrated parts: a formalization of basic concepts of domain theory, an interface, a number of proof functions for syntactic notations and various other theorems and functions to support domain theoretic reasoning in HOL. It has no built-in functions to support the definition of recursive domains, but a few example domains of both finite and infinite nature have been formalized; in particular, recursive domains of lazy (infinite) sequences and lazy lists have been formalized with the proof principles of structural induction [Pa84a] and co-induction [Pi94] to reason about infinite values, in addition to fixed point induction.

Through these extensions, HOL-CPO provides the concepts and techniques of fixed point theory to reason about infinite data values, nontermination and arbitrary recursive (continuous) functions. And, just as important, it allows set and domain theoretic reasoning to be mixed such that the benefits of both theories are available at the same time. Set theory is, for instance, well-suited to reason about primitive recursive functions and recursive datatypes with finite values.

In this chapter, we survey the formalization of domain theory and the associated tools. The range of new possibilities that HOL-CPO offers to the HOL user is demonstrated by considering a number of illustrative examples.

Notation

A lighter presentation is obtained by introducing some standard notation rather than HOL's ASCII notation (cf. [GM93], chapter 17):

- Types and constants in sans serif, e.g. `bool`.
- Use of subscript, e.g. `FixE` instead of `Fix E`.
- Mathematical notation: \forall instead of `!`, $x \neq y$ instead of `~(x=y)`, and so on.
- Built-in constants: \leq for `<=`, \in for `!N`, \subseteq for `SUBSET`, etc.

The constants `IN` and `SUBSET` are defined in the predicate sets library of HOL [Me92], a library which supports reasoning about sets written as predicates, e.g. $\{x, y\} : \alpha \rightarrow \text{bool}$.

2.1 A Formalization of Domain Theory

Domain theory is the study of complete partial orders (cpos) and continuous functions between cpos. A complete partial order is a partial order (po) which contains least upper bounds of all its chains. A continuous function is a certain kind of monotonic function which preserves such least upper bounds. The concepts of domain theory can be formalized by giving their semantic definitions in HOL. In this section we concentrate on how this is done; in particular, the section does not provide a thorough introduction on domain theory—such can be found in the textbooks by Winskel [Wi93], Gunter [Gu92] and Schmidt [Sc86].

A partial order (po) is a pair consisting of a set and a binary relation such that the relation is reflexive, transitive and antisymmetric on the set. We could formalize such pairs in various ways in HOL. The set component can be denoted by a HOL type α such that a partial order is represented as a HOL function $R : \alpha \rightarrow \alpha \rightarrow \text{bool}$ (corresponding to the binary relation) where `bool` is the type of boolean truth values. The properties of pos mentioned above should hold for all elements of the underlying type α , e.g. reflexivity would be stated as $\forall x. R x x$.

One serious disadvantage of this approach is that we will not be able to talk about the cpo of continuous functions. The HOL function type is denoted by the set of all functions between two sets and therefore may include non-continuous functions. This would be a serious drawback later since we shall use this cpo frequently. Instead, the set component should correspond to a subset of a HOL type. The most direct way to accomplish this is to represent a partial order as a pair (A, R) in HOL where the type of A is $\alpha \rightarrow \text{bool}$. As before, the type of R is $\alpha \rightarrow \alpha \rightarrow \text{bool}$ but here the conditions on R should hold for elements of A only (see below). Another equivalent approach would be to define the underlying set of a po to be precisely the subset of a type for which the relation is reflexive, i.e. the subset $\{x \mid R x x\}$.

In literature, a partial order is usually confused with the underlying set such that A is written for (A, R) . Much the same confusion can be provided in the formalization by introducing constants `rel` and `ins` where `rel` is used to obtain the relation component of a pair and `ins` is used to state whether a term is in the set component of a pair. Hence, in terms below the variable A (and later D and E) ranges over pairs of sets and relations, that is, it has HOL type

$$A : (\alpha \rightarrow \text{bool}) \times (\alpha \rightarrow \alpha \rightarrow \text{bool}).$$

The constant `rel` is simply defined to equal the projection function `SND` and the definition of `ins` is straightforward too:

$$\vdash \forall a.A. a \text{ ins } A = a \in (\text{FST } A)$$

using the projection function `FST` and the membership predicate \in on sets (so, $a \in P$ means $P(a)$ equals true). Hence, the constant `ins` simply extends \in to po pairs. We can introduce a similar extension of the subset inclusion predicate \subseteq as follows:

$$\vdash \forall B A. B \text{ subset } A = B \subseteq (\text{FST } A)$$

So the definition of **subset** says that a set is a subset of a partial order when it is a subset of the underlying set.

We shall introduce a few syntactic conveniences for use in this chapter only. Assuming variables a and b in a partial order A , the statement $a \text{ ins } A$ is written as $a \in A$ and the statement $\text{rel } A a b$ is written as $a \sqsubseteq_A b$. Besides we write $B \subseteq A$ instead of $B \text{ subset } A$ (assuming B is a set of appropriate type). Note that by introducing this syntactic sugar we in fact overload the symbols \in and \subseteq which are used for both sets and partial orders. The context will tell the reader which versions of the real HOL constants are meant (**ins** or **IN**, or **subset** or **SUBSET**). In addition, the following rules apply: variables like B and Z are always (and only) used for sets and variables like A , D and E are always (and only) used for partial orders. Finally, it is convenient to introduce the following abbreviation for universal quantifications: when we write a term like

$$\forall x, y, z \in A. P(x, y, z)$$

what we really mean is the relatively long-winded term

$$\forall x. x \in A \Rightarrow (\forall y. y \in A \Rightarrow (\forall z. z \in A \Rightarrow P(x, y, z))).$$

Further, comma and ‘long space’ are used to separate terms: $\forall f, x, y \in A. \dots$ This means for all f and for all x and y in A and so on. A similar syntactic sugar is used for λ -abstractions (but we return to that later).

The notion of partial order can now be introduced in HOL via a new constant **po** which is a predicate on pairs:

$$\vdash \forall A. \text{po } A = \text{refl } A \wedge \text{trans } A \wedge \text{antisym } A$$

The conditions on **pos** are defined by the theorems:

$$\begin{aligned} \vdash \forall A. \text{refl } A &= \forall x \in A. x \sqsubseteq_A x \\ \vdash \forall A. \text{trans } A &= \forall x, y, z \in A. x \sqsubseteq_A y \wedge y \sqsubseteq_A z \Rightarrow x \sqsubseteq_A z \\ \vdash \forall A. \text{antisym } A &= \forall x, y \in A. x \sqsubseteq_A y \wedge y \sqsubseteq_A x \Rightarrow x = y \end{aligned}$$

These definitions are the usual ones.

In domain theory, the ordering relation of a partial order is usually interpreted as an approximation ordering stating when one (partial) result of a computation approximates another. The ordering is usually read as “approximates” or “is less defined than” (or more precisely, “is at most as defined as”). This meaning is clearest for more complex partial orders of continuous functions or lazy lists, as we shall see later.

A partial order A may have a least defined element, i.e. an elements which approximates all other elements of the **po**:

$$\vdash \forall a A. a \text{ is_least } A = a \in A \wedge (\forall b \in A. a \sqsubseteq_A b)$$

From the antisymmetry condition on partial orders we can derive that least elements are unique when they exist:

$$\vdash \forall A. \text{po } A \Rightarrow (\forall a a'. a \text{ is_least } A \wedge a' \text{ is_least } A \Rightarrow a = a')$$

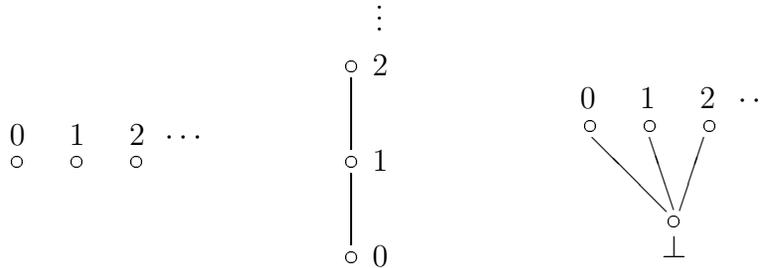


Figure 2.1: Partial orderings on natural numbers.

The least element is usually called bottom so it is convenient to introduce a term for this element (if it exists) using the choice operator:

$$\vdash \forall A. \text{bottom } A = (\textcircled{a}. a \text{ is_least } A)$$

The choice operator (see [GM93]) yields an arbitrary element of some type such that a predicate is satisfied (as in $\textcircled{x}.P(x)$). If this is not possible, i.e. if the predicate is everywhere false, then it yields any element of the type (all types are non-empty).

The bottom element of a po A is usually denoted by the symbol \perp_A so we shall write this instead of $\text{bottom } A$. Bottom is a kind of undefined element which stands for nontermination or “no value at all”.

Many familiar sets become partial orders when they are equipped with appropriate binary relations. As one example, consider the set of natural numbers corresponding to the HOL type `num` (numerals) which is a po with the less-than-or-equals ordering \leq :

$$\vdash \text{po}(\{n \mid 0 \leq n\}, \leq)$$

However, this is not the kind of relation on natural numbers we shall consider below since there are important properties that it does not enjoy. First of all, we cannot really interpret the less-than-or-equals ordering as an approximation ordering. Most readers would agree that all natural numbers are equally defined. The approximation ordering on natural numbers is therefore the equality relation which is called the discrete ordering. This means that any natural number is related to itself only, different numbers are incomparable by this relation.

The two different orderings on natural numbers are shown as graphs in figure 2.1. The discrete ordering corresponds to the first of the three graphs and the less-than-or-equals ordering corresponds to the second graph. Each node represents a natural number and each arc represents an ordering relationship. In the graphs, the lower elements are less defined than the upper elements. Arcs on elements themselves, corresponding to reflexivity, are not shown. Similarly, arcs which can be derived from transitivity are not shown. Therefore, the discrete ordering is shown as nodes only (no arcs). Also note that 0 is a ‘bottom’ or least ‘defined’ element with the less-than-or-equals ordering. It is less than all natural numbers above 1 due to transitivity. In the third graph of figure 2.1 a new element which is a ‘real’ bottom element has been associated with the natural numbers. This is done by extending the discrete ordering to a so-called flat ordering where the new element is below all numbers.

Let us write `Nat` for the discrete partial order $\text{discrete } \{n \mid 0 \leq n\}$ where the discrete construction is defined by:

$$\vdash \forall Z. \text{discrete } Z = Z, (\lambda d_1, d_2 \in Z. d_1 = d_2)$$

Introducing a construction for lifting a po by extending the underlying set with a new bottom element:

$$\begin{aligned} &\vdash \forall A. \\ &\quad \text{lift } A = \\ &\quad \{\mathbf{Bt}\} \cup \{\mathbf{Lft } d \mid d \in A\}, \\ &\quad (\lambda xy. x = \mathbf{Bt} \vee \exists dd'. x = \mathbf{Lft } d \wedge y = \mathbf{Lft } d \wedge d \sqsubseteq_A d') \end{aligned}$$

where \mathbf{Bt} and \mathbf{Lft} are the constructors of a new concrete datatype of syntax in HOL, we can write the flat (lifted-discrete) po of numbers as lift Nat . Hence, the symbol \perp of figure 2.1 refers to $\perp_{\text{lift Nat}}$ which is the bottom of this po, due to the way in which the ordering is defined, and equals the constant \mathbf{Bt} . It is easy to see that discrete and lift define constructions on partial orders:

$$\begin{aligned} &\vdash \forall Z. \text{po}(\text{discrete } Z) \\ &\vdash \forall A. \text{po } A \Rightarrow \text{po}(\text{lift } A) \end{aligned}$$

That is, a set associated with the discrete ordering is always a po and lifting preserves partial orders.

Another reason why the less-than-or-equals ordering is not a ‘good’ ordering in domain theory is that it is possible to choose a subset of natural numbers which is not bounded from above with this relation, e.g. the set of all numbers itself or the set of even numbers (see below). This should not be possible since complete partial orders are partial orders where certain sets (like this) are always bounded from above. This notion of boundedness is an important concept which we define next.

An upper bound of a subset B of some partial order A is an element $a \in A$ which is approximated by all elements $b \in B$. The following definition of the constant is_ub introduces this notion:

$$\vdash \forall aBA. a \text{ is_ub } (B, A) = a \in A \wedge \text{po } A \wedge B \subseteq A \wedge (\forall b \in B. b \sqsubseteq_A a)$$

If there are one or more upper bounds then there may be a least one. A least upper bound (lub) is an upper bound which approximates all other upper bounds:

$$\vdash \forall aBA. a \text{ is_lub } (B, A) = a \text{ is_least } (\{b \mid b \text{ is_ub } (B, A)\}, \sqsubseteq_A)$$

From the uniqueness of least elements we can derive that lubs are unique if they exist.

Using the choice operator, an expression for the least upper bound can be introduced. Defining a constant lub as follows:

$$\vdash \forall BA. \text{lub}(B, A) = (\text{Ch } a. a \text{ is_lub } (B, A))$$

we can prove that if a lub exists then lub yields an upper bound and in fact the unique least upper bound:

$$\vdash \forall aBA. a \text{ is_lub } (B, A) \Rightarrow (\text{lub}(B, A)) \text{ is_ub } (B, A) \wedge (\text{lub}(B, A) = a)$$

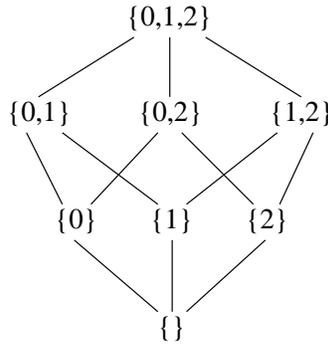


Figure 2.2: Power set of natural numbers 0, 1, and 2.

as is stated by the second conjunct in this theorem.

A finite subset of the natural numbers like $\{2, 4, 6, 8\}$ has many upper bounds with the less-than-or-equals ordering, e.g. 9, 10 and 20. It also has a least upper bound which is 8, certainly this is an upper bound and less than both 9 and 10. If this finite set is extended to the infinite set of all even numbers $\{2, 4, 6, \dots\}$ (which must be written as $\{n \mid \text{EVEN } n\}$ in HOL) then an upper bound does not exist. For any natural number n we can always find an even number which is larger than n .

On the other hand, the power set $Pow(\{0, 1, 2\})$ of natural number 0, 1 and 2 which is a partial order with the subset inclusion ordering \subseteq has a least upper bound for any subset of its elements, see figure 2.2. This lub is obtained by taking the union of all sets of the subset, e.g. the lub of $\{\{0\}, \{1, 2\}\}$ is the element $\{0, 1, 2\}$.

Least upper bounds are important in domain theory since they support an interpretation of infinite values like functions or lazy lists as the lub of chains of finite partial approximations. The definition of complete partial order guarantees that such lubs always exist. Besides, the definition of continuity guarantees that continuous functionals on cpos with bottom always have a least fixed point and that a canonical value exists for this fixed point. The fixed point operator yields this value which is the lub of a chain of partial approximations obtained by iterating the function a finite number of times over the bottom element. Furthermore, a recursive function is defined as a lub via the fixed point operator. The precise definitions follow next.

A chain is a non-decreasing sequence $X : \text{num} \rightarrow \alpha$ of elements of a partial order D . Here, non-decreasing means that the n -th element of the sequence $X(n)$ approximates the next element in the sequence, which is $X(n + 1)$:

$$\vdash \forall X D. \text{chain}(X, D) = (\forall n. Xn \in D) \wedge (\forall n. Xn \sqsubseteq_D X(n + 1))$$

Such chains are sometimes called ω -chain and the cpos we introduce below are called (bottomless) ω -cpo, or predomains.

Chains are used a lot below so it is convenient to introduce some syntactic sugar. First, we shall write $\text{chain}_D X$ for $\text{chain}(X, D)$. Besides, the term $\text{lub}_D X$ is written for $\text{lub}(\{Xn \mid 0 \leq n\}, D)$ and similarly, $a \text{ is_lub}_D X$ is written for $a \text{ is_lub}(\{Xn \mid 0 \leq n\}, D)$ assuming a is an element of D (or has the right type).

A partial order is called a *complete* partial order when it contains all lubs of chains. This central notion is introduced by the constant **cpo**, defined as follows in HOL:

$$\vdash \forall D. \text{cpo } D = \text{po } D \wedge (\forall X. \text{chain}_D X \Rightarrow \exists d. d \text{ is_lub}_D X)$$

Hence, in cpos the constant **lub** always yields a least upper bound of chains:

$$\vdash \forall D. \text{ cpo } D \Rightarrow \forall X. \text{ chain}_D X \Rightarrow (\text{lub}_D X) \text{ is_lub}_D X$$

A cpo with a least element is called a cpo with bottom, or a pointed cpo:

$$\vdash \forall E. \text{ pcpo } E = \text{ cpo } E \wedge (\exists e. e \text{ is_least } E)$$

In a pointed cpo E the term \perp_E is a bottom and therefore approximates all other elements w.r.t. to the underlying ordering of the cpo:

$$\vdash \forall E. \text{ pcpo } E \Rightarrow (\forall e \in E. \perp_E \sqsubseteq_E e)$$

In figure 2.1 above, the first and third graphs correspond to cpos whereas only the third one corresponds to a pointed cpo. The graph of figure 2.2 is a cpo and a pointed cpo as well. This follows fairly easily since all chains in the examples are finite, i.e. they are constant from a certain point. The lub of a finite chain is the largest element of the chain (this element is repeated forever in an infinite sequence representing a finite chain). A chain in a cpo with the discrete ordering as in the first graph of figure 2.1 is always a finite chain and in fact a constant chain since it contains exactly one element. Hence, the discrete construction always yields a cpo:

$$\vdash \forall Z. \text{ cpo}(\text{discrete } Z)$$

A chain in a cpo with a flat ordering as in the third graph of figure 2.1 may be bottom to start with before it becomes a constant chain (it can also be constantly bottom). It does not follow from this argument, however, that the lifting construction always preserves cpos since in general the cpo argument of **lift** needs not be discrete. However, **lift** can be proved to be a cpo constructor, in fact a constructor for pointed cpos, by a similar (more complicated) argument:

$$\vdash \forall D. \text{ cpo } D \Rightarrow \text{ pcpo}(\text{lift } D)$$

To conclude the examples observe that a chain in a finite po as in figure 2.2 is always constant from a certain point, and hence, this is also a cpo.

The conditions on continuous functions must guarantee that a fixed point can be obtained as the lub of a chain of partial approximations. First of all, a continuous function from a cpo D_1 to a cpo D_2 must be a monotonic function which maps elements of D_1 to elements of D_2 , and furthermore, it must be determined by this action on elements of its domain D_1 . Monotonicity is defined as follows:

$$\begin{aligned} \vdash \forall f D_1 D_2. \\ \text{ mono } f (D_1, D_2) = \\ \text{ cpo } D_1 \wedge \text{ cpo } D_2 \wedge \text{ map } f (D_1, D_2) \wedge \text{ determined } f D_1 \wedge \\ (\forall d, d' \in D_1. d \sqsubseteq_{D_1} d' \Rightarrow f(d) \sqsubseteq_{D_2} f(d')) \end{aligned}$$

where **map** and **determined** are defined by:

$$\begin{aligned} \vdash \forall f D_1 D_2. \text{ map } f (D_1, D_2) &= (\forall d \in D_1. f(d) \in D_2) \\ \vdash \forall f D_1 D_2. \text{ determined } f D_1 &= (\forall d \notin D_1. f(d) = \text{ARB}) \end{aligned}$$

The constant `ARB` is predefined in HOL and it chooses an arbitrary element of some type. Since HOL functions written using standard λ -notation “ $\lambda x. f[x]$ ” typically are not determined we introduce a special kind of dependent lambda abstraction for writing functions which are always determined as follows:

$$\vdash \forall D f. \text{lambda } D f = (\lambda x. (x \in D \rightarrow f(x) \mid \text{ARB}))$$

In particular, when `lambda` is applied to a λ -abstraction as in “`lambda D ($\lambda x. f[x]$)`” we will write the shorter term “ $\lambda x \in D. f[x]$ ” instead. As for universal quantifications we write “ $\lambda x, y \in D. f[x]$ ” for the longer term “ $\lambda x \in D. \lambda y \in D. f[x]$ ”.

The determinedness condition is necessary because we work with partial function, namely HOL functions between subsets of types, i.e. the subsets which are the underlying sets of cpos. Without this condition, a monotonic (or continuous) function might be seen as a representative for an equivalence class of HOL functions. The equivalence classes are induced by a function equality which works on subsets of types (monotonic functions are specified as maps between subsets of types rather than on types) unlike the HOL function equality which works on all elements of a type (extensional equality). Since it is very cumbersome to work with equivalence classes of functions we pick instead a certain fixed representative in each equivalence class by requiring functions are determined. (This problem is discussed further in chapter 3, see section 3.4 in particular.)

A continuous function is a monotonic function which preserves lubs of chains:

$$\begin{aligned} \vdash \forall f D_1 D_2. \\ \text{cont } f (D_1, D_2) = \\ \text{mono } f (D_1, D_2) \wedge (\forall X. \text{chain}_{D_1} X \Rightarrow f(\text{lub}_{D_1} X) = \text{lub}_{D_2} (\lambda n. f(Xn))) \end{aligned}$$

Note that if X is a chain in D_1 and f is a monotonic function from D_1 to D_2 then monotonicity ensures the term “ $\lambda n. f(Xn)$ ” is a chain in D_2 . Otherwise the definition of continuity would not make sense.

The notion of determinedness is necessary to ensure that continuous functions constitute a cpo with the pointwise ordering on functions. This is called the cpo of continuous functions or the continuous function space and defined by `cf`:

$$\begin{aligned} \vdash \forall D_1 D_2. \text{cfs}(D_1, D_2) = \{f \mid \text{cont } f (D_1, D_2)\} \\ \vdash \forall D_1 D_2. \text{cf}(D_1, D_2) = \text{cfs}(D_1, D_2), (\lambda f, g \in \text{cfs}(D_1, D_2). \forall d \in D_1. f(d) \sqsubseteq_{D_2} g(d)) \end{aligned}$$

which is a cpo construction, as claimed:

$$\vdash \forall D_1 D_2. \text{cpo } D_1 \wedge \text{cpo } D_2 \Rightarrow \text{cpo}(\text{cf}(D_1, D_2))$$

The determinedness condition ensures that we can prove the antisymmetry condition on partial orders (which involves HOL equality, see the discussion above). Note that saying a function f is continuous: `cont f (D1, D2)`, is the same thing as saying f is in the continuous function space: $f \in \text{cf}(D_1, D_2)$. Also note that this construction is a dependent subset of the type of HOL functions between the underlying HOL types. Since the HOL logic does not provide dependent types we must simulate these some way, e.g. using subsets of HOL type as here (this approach is also used in [JM93]). Hence, as explained in the beginning of this section the set component of a partial order cannot be represented by a HOL type if the continuous function space construction is needed.

Since the ordering relation on the continuous function space is defined pointwise we can prove that chains, lubs and bottoms are all calculated pointwise:

$$\begin{aligned}
&\vdash \forall X D_1 D_2. \text{chain}_{\text{cf}(D_1, D_2)} X \Rightarrow (\forall d \in D_1. \text{chain}_{D_2}(\lambda n. X \ n \ d)) \\
&\vdash \forall X D_1 D_2. \text{chain}_{\text{cf}(D_1, D_2)} X \Rightarrow (\text{lub}_{\text{cf}(D_1, D_2)} X = (\lambda d \in D_1. \text{lub}_{D_2}(\lambda n. X \ n \ d))) \\
&\vdash \forall D E. \text{cpo } D \wedge \text{pcpo } E \Rightarrow \perp_{\text{cf}(D_1, D_2)} = (\lambda d \in D_1. \perp_{D_2})
\end{aligned}$$

From the last theorem it follows that `cf` is also a constructor for pointed cpos (and not just cpos):

$$\vdash \forall D E. \text{cpo } D \wedge \text{pcpo } E \Rightarrow \text{pcpo}(\text{cf}(D, E))$$

Note that the domain cpo of the function space only need to be a cpo rather than a pointed cpo. If it is a pointed cpo it makes sense to talk about strictness of functions. We say that a function is strict if it maps bottom of the domain to bottom of the codomain.

Some functions are obviously continuous, some not. For instance, any determined function which is a map from a discrete cpo to any cpo is continuous:

$$\begin{aligned}
&\vdash \forall f D Z. \\
&\quad \text{cpo } D \wedge \text{map } f (\text{discrete } Z, D) \wedge \text{determined } f (\text{discrete } Z) \Rightarrow \text{cont } f (\text{discrete } Z, D)
\end{aligned}$$

In particular, if we instantiate the variable D in this theorem with a discrete universal cpo as for example the natural numbers `Nat` and similarly instantiate Z with the universal set of natural numbers `UNIV : num → bool`, where `UNIV` is the predicate which is always true, then all antecedents hold trivially, and we conclude:

$$\vdash \forall f. \text{cont } f (\text{Nat}, \text{Nat})$$

This can then be used to obtain that for instance HOL addition `$+` is a continuous operation on the cpo of natural numbers `$+ ∈ cf(Nat, Nat)`. A similar fact holds for any two universal sets, which may have different types, not just the natural numbers:

$$\vdash \forall f : \alpha \rightarrow \beta. \text{cont } f (\text{discrete UNIV}, \text{discrete UNIV})$$

This theorem is very useful to prove that HOL functions on ‘simple’ cpos are continuous. We can also derive the following result about flat (i.e. lifted discrete) cpos:

$$\vdash \forall f. \text{cont } f (\text{discrete UNIV}, \text{lift } (\text{discrete UNIV}))$$

which states that any function from a discrete universal to a flat universal cpo is continuous.

There are two continuous operations directly associated with the lifting construction on cpos. One is for lifting an element to a lifted cpo:

$$\begin{aligned}
&\vdash \forall D. \text{Lift}_D = (\lambda d \in D. \text{Lft } d) \\
&\vdash \forall D. \text{cpo } D \Rightarrow \text{cont Lift}_D (D, \text{lift } D)
\end{aligned}$$

and the other is for extending a continuous function between any cpo and a pointed cpo to a strict continuous function from the lifted cpo to the pointed cpo:

$$\begin{aligned}
&\vdash \forall D E. \text{Ext}_{(D, E)} = (\lambda f \in \text{cf}(D, E), x \in \text{lift } D. (x = \text{Bt} \rightarrow \perp_E \mid f(\text{unlift } x))) \\
&\vdash \forall D E. \text{cpo } D \wedge \text{pcpo } E \Rightarrow \text{cont Ext}_{(D, E)} (\text{cf}(D, E), \text{cf}(\text{lift } D, E))
\end{aligned}$$

where `unlift` is defined by:

$$\vdash \forall y. \text{unlift}(\text{Lft } y) = y$$

Note that both of these functions become parameterized by cpo variables in their domains; **Lift** takes arguments in D , and is parameterized by D , and **Ext** takes arguments in $\text{cf}(D, E)$ and **lift** D , and is parameterized with D and E . This is due to the use of the dependent **lambda** abstraction (rather than the ordinary HOL abstraction) which is necessary in order to ensure that the functions are determined. The determinedness condition makes all functions on domains with cpo variables parameterized by these cpo variables. Functions therefore become difficult to read and write, but a parser and a pretty-printer can hide the parameters in most cases (see section 2.2).

Finally, let us consider the main goal of this development, namely the definition of a canonical value for the least fixed point of a continuous function on a pointed cpo. First we define the power of a function applied to bottom by primitive recursion on the natural numbers (considered as a type in HOL, not as a cpo):

$$\begin{aligned} \vdash (\forall E. \text{pow } E \ 0 = (\lambda f \in \text{cf}(E, E). \perp_E)) \wedge \\ (\forall E n. \text{pow } E \ (n + 1) = (\lambda f \in \text{cf}(E, E). f(\text{pow } E \ n \ f))) \end{aligned}$$

It is straightforward to see that the values returned by **pow** constitute a chain for a continuous function by the monotonicity condition:

$$\vdash \forall E. \text{pcpo } E \Rightarrow \text{chain}_{\text{cf}(\text{cf}(E, E), E)}(\text{pow } E)$$

It therefore makes sense to take the least upper bound of **pow**:

$$\vdash \forall E. \text{Fix}_E = \text{lub}_{\text{cf}(\text{cf}(E, E), E)}(\text{pow } E)$$

which we call the least fixed point operator. Some readers might be more used to a characterization of the fixed point operator like

$$\vdash \forall E. \text{pcpo } E \Rightarrow (\forall f \in \text{cf}(E, E). \text{Fix}_E f = \text{lub}_E(\lambda n. \text{pow } E \ n \ f))$$

which can be derived from the previous definition. Note that it follows directly from the definition that **Fix** is a continuous function (the **lub** is in the continuous function space).

It still remains to be justified that the fixed point operator yields a fixed point of a continuous function on a pointed cpo and that this is the least fixed point. The fixed point property of **Fix** is stated as follows:

$$\vdash \forall E. \text{pcpo } E \Rightarrow (\forall f \in \text{cf}(E, E). f(\text{Fix}_E f) = \text{Fix}_E f)$$

and the fact that it is a least fixed point follows from the following theorem:

$$\vdash \forall E. \text{pcpo } E \Rightarrow (\forall f \in \text{cf}(E, E), d \in E. f(d) \sqsubseteq_E d \Rightarrow \text{Fix}_E(f) \sqsubseteq_E d)$$

which states that it is the least prefixed point; since it is a fixed point it is also a prefixed point. This theorem is sometimes called Park induction.

Another important induction principle of proof is called fixed point induction which gives a method to prove properties of least fixed points by looking at their finite partial approximations. It is stated as follows:

$$\begin{aligned} \vdash \forall EP, f \in \text{cf}(E, E). \\ \text{pcpo } E \wedge \text{inclusive}_E P \Rightarrow \\ P(\perp_E) \wedge (\forall x. P(x) \Rightarrow P(f(x))) \Rightarrow P(\text{Fix}_E f) \end{aligned}$$

where the constant **inclusive** is defined by:

$$\begin{aligned} &\vdash \forall DP. \\ &\text{inclusive}_D P = \\ &(\forall X. \text{chain}_D X \wedge (\forall n. P(Xn)) \Rightarrow P(\text{lub}_D X)) \end{aligned}$$

This notion of inclusiveness, sometimes called admissibility, ensures that chains of elements in some predicate have lubs in the predicate. From this and continuity the fixed point induction theorem follows easily.

Monotonicity ensures the existence of the lub of finite approximations of a fixed point. The continuity condition is used to prove that this lub is indeed the least fixed point. In fact, the existence of a least fixed point could be proved from monotonicity alone without appealing to continuity (see e.g. exercise 4.23 in Gunter's book [Gu92]). However, using continuity the proof is simpler because it avoids the use of ordinals.

The goal of this entire development has now been achieved: the solutions of recursive specifications of the form $x = F(x)$ can be found within pointed cpos as the least fixed point of the (continuous) functional F . In particular, x can be a function f . Hence, we have a way to give semantics to recursive function definitions. Fixed point induction provides a way to reason about recursive definitions using an inductive argument. But Park induction, the fixed point property or the definition of the fixed point operator can be used as well, just as any other techniques for recursion derivable in HOL.

Before we conclude this section we shall briefly consider two more constructions on cpos, in addition to the discrete, lifting and continuous function space constructions that we considered above. The product construction yields a cpo which consists of all HOL pairs of elements of two cpos. The underlying ordering relation is defined componentwise:

$$\begin{aligned} &\vdash \forall D_1 D_2. \\ &\text{prod}(D_1, D_2) = \\ &\{(d_1, d_2) \mid d_1 \in D_1 \wedge d_2 \in D_2\}, \\ &(\lambda(d_1, d_2)(d'_1, d'_2). d_1 \sqsubseteq_{D_1} d'_1 \wedge d_2 \sqsubseteq_{D_2} d'_2) \end{aligned}$$

The product of for instance three cpos is written as $\text{prod}(D_1, \text{prod}(D_2, D_3))$.

The sum construction, called **sum**, is similar but it is based on the HOL sum type instead of the type of pairs. We don't use this construction in this chapter so we shall not show its definition, which is a bit ugly due to the use of injections and their inverses (see chapter 3).

Both the product and the sum constructions yield cpos if their arguments are cpos:

$$\begin{aligned} &\vdash \forall D_1 D_2. \text{cpo } D_1 \wedge \text{cpo } D_2 \Rightarrow \text{cpo}(\text{prod}(D_1, D_2)) \\ &\vdash \forall D_1 D_2. \text{cpo } D_1 \wedge \text{cpo } D_2 \Rightarrow \text{cpo}(\text{sum}(D_1, D_2)) \end{aligned}$$

but only the product construction yields a pointed cpo, provided its arguments are pointed cpos:

$$\vdash \forall E_1 E_2. \text{pcpo } E_1 \wedge \text{pcpo } E_2 \Rightarrow \text{pcpo}(\text{prod}(E_1, E_2))$$

The bottom element is calculated componentwise:

$$\vdash \forall E_1 E_2. \text{pcpo } E_1 \wedge \text{pcpo } E_2 \Rightarrow \perp_{\text{prod}(E_1, E_2)} = (\perp_{E_1}, \perp_{E_2})$$

just as lubs and chains, in fact (the theorems are listed in section 3.6.2).

A continuous function from a product can be curried such that it takes its arguments one at a time. This construction called currying has been defined and proved to be a continuous function, and hence, in particular, to preserve continuity, i.e. to yield a continuous function if its function argument is continuous. Function application has also been proved to be a continuous function and to preserve continuity. In addition, the functions for projection, injection, tupling of functions and function sum have been defined and proved to be continuous functions, among others. They are not used in this chapter so we shall not bother the reader with the definitions here (see chapter 3).

In the formalization of domain theory presented above we have not considered the solution of recursive domain equations. This is the difficult part of domain theory. If one turns to representations of domains as retracts of $P\omega$ [Sc76, Ba84] or information systems [Sc82, LW91, Wi93] then solutions to recursive domain equations can be created by the fixed point operator. Alternatively, solutions to recursive domain equations can be obtained as inverse limits (ω -limits) in the category of domains with projections. In any case, the formalizations of these methods would not match well with the formalization presented here (see chapter 4). Instead we consider a few ad hoc methods in section 2.4.

2.2 The HOL-CPO System

The formalization presented above supports the reasoning about domain theoretic concepts in the HOL system. Thus, we can use built-in tools of HOL to prove that terms are cpos, continuous functions and inclusive predicates. Such theorems can then be used in recursive (fixed point) definitions or for fixed point induction. This way domain theory looks like a separate block built on top of HOL, which it is in a way. However, a few ML functions can (fairly) easily be implemented to support the view that domain theory is an integrated part of the system; the advantage is that the formalization would become easier to use in practice. Special-purpose proof functions and other features such as a simple interface have been implemented to support the use of the extension of HOL with domain theory. The resulting system is called HOL-CPO, hereafter.

The most important proof functions of HOL-CPO are syntactic-based functions which implement informal notations of cpos and cpo-typable terms (there is a notation for both cpos and pointed cpos). A term is called *cpo-typable* or just *typable* if it can be proved to be an element of some cpo. Note in particular that we can treat the problem of showing that a function is continuous as a special case of the problem of deducing which cpo a term is an element of (i.e. the cpo of continuous functions). The proof functions are fairly simple and therefore exclude many cpos and typable terms. However, the notations can be extended interactively by proving constants and other terms are cpos or elements of cpos and then declaring these theorems to the system. Furthermore, derived definition tools for introducing new constants to extend the notations have been based on the proof functions and the declaration tools.

A better interface to the formalization constitutes another useful feature of HOL-CPO. In the previous section, we noted on the inconvenience of the cpo parameters on function constructions, such as $\text{Ext}_{(D,E)}$ (and Fix_E). It is desirable to get rid of these parameters some way. This can be handled by implementing a special-purpose parser and pretty-printer and hooking these up with the built-in parser and pretty-printer of HOL. In this

way, the cpo parameters can be hidden in the sense that they do not appear in the terms that we look at but do appear internally in the terms that are manipulated in proofs. The current implementation of the parser is quite primitive so the parser does not always work. (Further, the problem of constructing the parameters may be undecidable in general since it seems to be similar to type checking in dependent type theory.)

Below, we present the informal notations for cpos and typable terms and give a few examples of terms which fit within the notations. Tools for fixed point induction and for proving automatically that certain predicates are inclusive are also presented.

Note

We shall keep this chapter relatively non-technical; in particular, algorithms are not considered in this chapter (cf. chapter 5). Details such as what the ML names of the functions are and how the functions are used, are presented in later chapters (see chapter 5 and 6 in particular).

2.2.1 Notations for Cpos and Pointed Cpos

The syntactic notations for cpos and pointed cpos are similar yet slightly different since certain constructions on cpos yield cpos which are not pointed. We shall therefore describe the notations separately just as there are separate proof functions to prove such facts too, called the *cpo prover* and the *pcpo prover* respectively.

The notation for cpos can be described informally as follows:

$$D ::= t \mid \text{discrete } Z \mid \text{lift } D \mid \text{cf}(D_1, D_2) \mid \text{prod}(D_1, D_2) \mid \text{sum}(D_1, D_2) \mid C(D_1, \dots, D_n)$$

where

- t is any HOL term for which a theorem is available stating it is a cpo,
- Z is some HOL set,
- C belongs to an extendable set of names of cpo constructors, and
- D, D_1, \dots, D_n are cpos ($0 \leq n$).

A cpo constructor is a constant, i.e. a nullary constructor, or a constant applied to a tuple of cpo variables such that it has been shown that the constructor yields a cpo, provided its arguments are cpos. The constructions on cpos presented in section 2.1 are built-in as it appears but new constructions can be introduced. Note that the notation allows any term as long as a theorem is available which states that the term is a cpo or a pointed cpo. Such a theorem is called a *cpo fact* below.

A constructor for pointed cpos is similar to a constructor for cpos. It is a constant which applied to a number of arguments yields a pointed cpo, provided the arguments are cpos or pointed cpos. The notation for pointed cpos can be described informally as follows:

$$D ::= t_p \mid \text{lift } D \mid \text{cf}(D, E) \mid \text{prod}(E_1, E_2) \mid C_p(D_1, \dots, D_n)$$

where

- t_p is any HOL term for which a theorem is available stating it is a pointed cpo,
- C_p belongs to an extendable set of names of constructors for pointed cpos,
- D, D_1, \dots, D_n are either cpos or pointed cpos, and
- E, E_1 and E_2 are pointed cpos.

Note that the discrete and sum constructions do not yield pointed cpos according to this notation. The sum of two cpos is never a pointed cpo whereas the discrete construction yields a pointed cpo if, and only if, the associated set is a singleton set.

The notations for cpos and pointed cpos can be extended by declaring definitions or declaring constructors. If a constant is introduced to abbreviate a term which fits within one of the notations then the notation can be extended with that constant by declaring the definition. For instance, we may define a cpo of natural numbers using the discrete construction on the universal set of all numerals in HOL, as in the previous section:

$$\vdash \text{Nat} = \text{discrete}(\text{UNIV} : \text{num} \rightarrow \text{bool})$$

Declaring this definition allows us to write `Nat` as a cpo. We can also use a derived definition tool which both defines the constant `Nat` and declares the definition. Once `Nat` has been declared one way or the other, then the term `cf(prod(Nat, Nat), lift Nat)` would fit within both notations above. Hence, the cpo prover would automatically prove the first and the pointed cpo prover the second theorem below:

$$\begin{aligned} &\vdash \text{cpo}(\text{cf}(\text{prod}(\text{Nat}, \text{Nat}), \text{lift Nat})) \\ &\vdash \text{pcpo}(\text{cf}(\text{prod}(\text{Nat}, \text{Nat}), \text{lift Nat})) \end{aligned}$$

New constructors on cpos (or pointed cpos) are introduced by theorems stating that provided their arguments are cpos or pointed cpos then they yield terms which are cpos (or pointed cpos). This way of extending the notations is used when a constructor is not equal to a term which matches the notations already. Assume for instance we have proved

$$\vdash \forall D. \text{cpo } D \Rightarrow \text{pcpo}(\text{seq } D)$$

where `seq` is a constructor for pointed cpos of lazy sequences (see section 2.4). Declaring this theorem extends the notation for pointed cpos with the constructor `seq`. Exploiting that a pointed cpo is also a cpo, the notation for cpos can also be extended.

2.2.2 Notation for Cpo-typable Terms

The parser and a proof function called the *type checker* together implement a notation for (cpo-) typable terms. It is illustrated in figure 2.3 how the parser, the type checker, and the pretty-printer interact. The parser transforms a term which fits within the notation into an internal version of the term. The main purpose of the parser is to insert cpo parameters on parameterized function constructors that have been omitted. The type checker then proves which cpo the term is an element of; more precisely, the type checker could perhaps be called a type reconstructor since it automatically constructs the cpo. The pretty-printer finally inverts the transformation performed by the parser, hence, cpo parameters are eliminated.

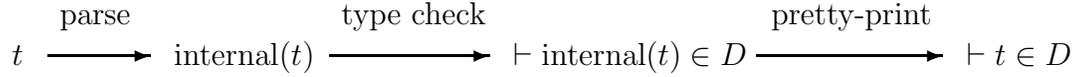


Figure 2.3: The interface and the type checker.

The (interface level) notation for typable terms can be described informally as follows:

$$e ::= t \mid x \mid c \mid \lambda xs \in D. e \mid (e_1 e_2)$$

where

- t belongs to an extendable set of basic typable terms,
- x is a variable of a (dependent) lambda abstraction,
- xs is a sequence (or tuple) of variables (x_1, \dots, x_n) and D is a product of cpos D_1, \dots, D_n , where $1 \leq n$,
- c belongs to an extendable set of parameterized function constructors, here used with or without the parameters (the parser will insert the missing ones), and
- e, e_1 and e_2 are typable terms.

A basic typable term t can be any HOL term, typically a constant or a variable, such that a fact is available which states which cpo it is an element of.

Let us illustrate by an example how the notation works and what job the parser and pretty-pretty do. Assuming **Nat** has been declared as in the previous section, consider the following term, which fits the notation of typable terms:

$$\text{Fix}(\lambda f \in \text{cf}(D, \text{lift Nat}). \lambda d \in D. \text{Ext } f (\text{Lift } d)).$$

Note that the parameterized constructors **Fix**, **Ext**, and **Lift** are used without their parameters in this term. The parser inserts the parameters and generates the following internal syntax:

$$\text{Fix}_{\text{cf}(D, \text{lift Nat})}(\text{lambda}(\text{cf}(D, \text{lift Nat}))(\lambda f. \text{lambda } D(\lambda d. \text{Ext}_{(D, \text{lift Nat})} f(\text{Lift}_D d))))).$$

If we had to write this term directly by hand, we would have to calculate all cpo parameters very carefully. Besides, reading the function is difficult due to the parameters, which are in fact not necessary for our understanding of what the function does.

Assuming now that the variable D is a cpo, the type checker proves a theorem stating which cpo the term is an element of. This theorem is pretty-printed as follows:

$$[\text{cpo } D] \vdash \text{Fix}(\lambda f \in \text{cf}(D, \text{lift Nat}), d \in D. \text{Ext } f (\text{Lift } d)) \in \text{cf}(D, \text{lift Nat})$$

Here, the interface level syntax is used again. So, the type checker reconstructs the domain that a term belongs to.

The set of basic typable terms of the notation can be extended by a declaration tool. If a term, e.g. a constant or a variable, does not fit the notation it must be declared before it is used. A declaration is simply a theorem stating which cpo a term is an element of. For instance, the theorem

$$\vdash \$+ \in \text{cf}(\text{Nat}, \text{cf}(\text{Nat}, \text{Nat}))$$

can be used to extend the notation with the built-in HOL addition. Afterwards, a strict addition can be defined as follows:

$$\vdash \text{Add} = \text{Ext}(\lambda n \in \text{Nat}. \text{Ext}(\lambda m \in \text{Nat}. \text{Lift}(n + m)))$$

The right-hand side fits within the notation (now that $+$ is in the notation) so the type checker proves automatically that **Add** is a continuous function, as stated by the theorem:

$$\vdash \text{Add} \in \text{cf}(\text{lift Nat}, \text{cf}(\text{lift Nat}, \text{lift Nat}))$$

Then this can be declared and used in other terms, and so on. A derived definition tool does these steps behind the scenes. It defines a constant, deduces which cpo it is an element of using the type checker, and finally, declares the theorem returned by the type checker. Note, by the way, that **Add** is not parameterized by any cpo variables since there are no free variables in the domain **Nat** on which it works.

The set of parameterized function constructors includes **Fix**, **Ext**, **Lift**, and many more (see chapter 5) but there is also a declaration tool to extend the set of built-in constructors. A declaration of a parameterized constructor is a theorem stating which cpo the constructor is an element of, under the assumptions that the cpo variable parameters are cpos or pointed cpos.

As an example, assume we have declared a discrete universal cpo of HOL boolean truth values by

$$\vdash \text{Bool} = \text{discrete}(\text{UNIV} : \text{bool} \rightarrow \text{bool})$$

and wish to introduce a conditional for this cpo, i.e. a continuous function which takes a boolean b and two terms t_1 and t_2 in an arbitrary cpo D and yields either t_1 or t_2 depending on whether b is true or false. The built-in conditional for booleans works well but it is not determined. Using the dependent lambda abstraction, we obtain a determined conditional as follows:

$$\vdash \forall D. \text{Condl}_D = (\lambda(b, t_1, t_2) \in \text{prod}(\text{Bool}, \text{prod}(D, D))). (b \rightarrow t_1 \mid t_2)$$

We can now prove that **Condl** is continuous but this must be done manually; the type checker cannot be used because the right-hand side of the definition does not match the notation for typable terms:

$$\vdash \forall D. \text{cpo } D \Rightarrow \text{Condl}_D \in \text{cf}(\text{prod}(\text{Bool}, \text{prod}(D, D)), D)$$

It is not difficult to prove this, but it requires deep knowledge of domain theory and of the formalization in HOL (fortunately, we do not have to step outside the notation often). Declaring **Condl** as a constructor, which is done by activating an ML function with this continuity theorem as an argument, makes **Condl** part of the notation of typable terms, just like **Fix** and **Ext**.

Again, there is a derived definition tool which can be used to define and declare function constructors when the right-hand sides of their definitions fit within the notation. So, it would not work for the conditional above which we have to introduce manually.

2.2.3 Other Syntactic-based Proof Functions

In addition to the above syntactic-based proof functions, which implement the notations for cpos and typable terms, there are two other important syntactic-based proof functions. The first one attempts to transform a cpo to a discrete universal cpo (like `Nat`) and some other kinds of trivial cpos which contain all elements of the underlying HOL type. Any term of the underlying type is trivially an element of the cpo and hence the proof function provides an alternative to the type checker in certain cases. The other proof function can be used to prove that certain predicates are inclusive such that they allow fixed point induction. This proof function exploits both the cpo provers and the type checker to deduce inclusiveness.

The syntactic-based transformation function for proving membership of trivial cpos is called the *ins-prover* since it can be used to prove that a term is in the underlying set of a cpo (recall the HOL constant for stating this is called `ins`, though we use the symbol \in here). Among others, it exploits theorems like

$$\vdash \text{cf}(\text{discrete UNIV}, \text{discrete UNIV}) = \text{discrete UNIV}$$

which states that the cpo of continuous functions between discrete universal cpos is itself a discrete universal cpo. To prove that any term is an element of a discrete universal cpo it uses

$$\vdash \forall t. t \in \text{discrete UNIV}.$$

It can also handle two cases which are universal cpos but not discrete universal cpos:

$$\begin{aligned} &\vdash \forall t. t \in \text{lift}(\text{discrete UNIV}) \\ &\vdash \forall f. f \in \text{cf}(\text{discrete UNIV}, \text{lift}(\text{discrete UNIV})) \end{aligned}$$

and with a small programming effort it could be extended to handle even more cases. It is a good alternative to the type checker. For instance, the type checker would not be able to prove the built-in addition on natural numbers is continuous:

$$\vdash \$+ \in \text{cf}(\text{Nat}, \text{cf}(\text{Nat}, \text{Nat}))$$

The symbol ‘+’ is a constant and its definition does not meet the notation of typable terms. But continuity is proved immediately by the *ins-prover* since `Nat` is a discrete universal cpo. The *ins-prover* can also be used to prove variables and any other terms are in universal cpos.

The proof function for proving that predicates are inclusive is called the *inclusive prover*. It implements syntactic checks which are similar to the ones performed by the LCF system; at least, they correspond to the syntactic checks described in the LCF book [Pa87] (see page 199–200). The inclusive prover does not check that predicates are subsets of a certain cpo, which they must be in order to be inclusive. Therefore, it requires predicates to be written using a constant `mk_pred` defined by

$$\vdash \forall DP. \text{mk_pred}_D P = \{x \mid x \in D \wedge x \in P\}$$

such that this is guaranteed. Applied to a term of the form

$$\text{inclusive}_D(\text{mk_pred}_D(\lambda x. e[x]))$$

the inclusive prover deduces this is a theorem if D has finite chains (e.g. if it is a discrete or a lifted discrete cpo) or if $e[x]$ meets one of the following syntactic conditions:

$$e ::= b \mid t_1 \sqsubseteq_E t_2 \mid t_1 = t_2 \mid t \neq \perp_E \mid t \not\sqsubseteq_E v \mid e_1 \wedge e_2 \mid e_1 \vee e_2 \mid \overline{e_1} \Rightarrow e_2 \mid \forall y \in E. e[y]$$

where

- b is a term of type boolean,
- E is a cpo or a pointed cpo (it should be pointed in the \perp_E case and non-empty in the \forall case),
- v is a term in E .
- t, t_1 and t_2 are continuous in the variable x in the sense that the lambda abstracted terms, e.g. “ $\lambda x \in D. t[x]$ ”, are continuous functions from D to E , and finally
- e, e_1 and e_2 are inclusive predicates in the variable x in the sense that e.g. the predicate $\text{mk_pred}_D(\lambda x. e[x])$ is an inclusive subset of D . Overline $\overline{e_1}$ means that $\neg e_1$ must be inclusive.

Hence, the inclusive prover is recursive and it uses the cpo provers and the type checker to prove the side conditions. (In its current prototype implementation, it does not allow the notation to be extended interactively but this could be implemented.)

2.2.4 Other Tools

Above we introduced the notations for cpos and typable terms and discussed some of the proof functions that have been implemented to support the notations. We also presented syntactic-based proof functions for inclusive predicates and for transforming cpo membership statements into trivial ones. There is almost no limit to which kinds of tools one can build as long as they are based on proved theorems.

A tactic based on the fixed point induction theorem is an example of a useful tool which combines the use of a theorem with proof functions such as the pcpo prover, the type checker and the inclusive prover. In fact, the *fixed point induction tactic* is based on a derived version of the fixed point induction theorem which is stated using `mk_pred` (for the inclusive prover).

A conversion for β -reducing dependent lambda abstractions is another example which is also based on a theorem, namely on

$$\vdash \forall D f, x \in D. \text{lambda } D f x = f x,$$

or written using λ notation

$$\vdash \forall D f, x \in D. (\lambda d \in D. f d)x = f x$$

Futhermore, it is also based on the type checker as well to prove the antecedent $x \in D$. Actually, we have extended this conversion to support all parameterized function constructors for which a similar reduction theorem is available. For instance, a reduction theorem for `Ext` is (there could be others)

$$\vdash \forall DE, f \in \text{cf}(D, E), x \in \text{lift } D. \text{Ext}_{(D, E)} f x = (x = \text{Bt} \rightarrow \perp_E \mid f(\text{unlift } x))$$

which is built-in. The conversion attempts to reduce the right-hand side conditional further by checking whether the argument of **Ext** in the lifted domain is equal to **Bt** or not (an ad hoc check). These additional manipulations make the reduction conversion behave as follows for **Ext**:

$$\begin{aligned} &\vdash \forall DE, f \in \text{cf}(D, E). \text{Ext } f \text{ Bt} = \perp_E \\ &\vdash \forall DE, f \in \text{cf}(D, E), x \in D. \text{Ext } f (\text{Lift } x) = f x \end{aligned}$$

where we have used interface level syntax (i.e. cpo parameters do not appear). Reduction theorems for user-defined constructors can be supplied via a theorem list argument of the reduction conversion.

It should be obvious why a proof function like the reduction conversion is useful. Consider for instance the strict addition, which we introduced above using function extension. Assume that we wish to prove this behaves like the built-in addition on lifted natural numbers:

$$\forall mn. \text{Add } (\text{Lift } m) (\text{Lift } n) = m + n$$

Before this can parse properly we must declare that n and m are in the cpo of natural numbers, which is obtained easily using the ins-prover twice. Stripping off the universally quantified variables, and expanding the definition of **Add**, we must prove

$$(\text{Ext}(\lambda n \in \text{Nat}. \text{Ext}(\lambda m \in \text{Nat}. \text{Lift}(n + m)))) (\text{Lift } m) (\text{Lift } n) = m + n$$

Using the reduction conversion, this automatically reduces to

$$m + n = m + n$$

which is a fact by reflexivity. Hence, we have proved the desired theorem:

$$\vdash \forall nm. \text{Add } (\text{Lift } m) (\text{Lift } n) = m + n$$

With similar ease, we can prove for example that **Add** is commutative:

$$\vdash \forall mn. \text{Add } m n = \text{Add } n m$$

First we do a case split on both arguments of **Add** which are in lift Nat . If one of the arguments is **Bt** then, due to strictness of **Ext**, both sides can be reduced to **Bt** as well. Otherwise, the theorem just proved about lifted arguments is applied and the commutativity theorem about the built-in addition

$$\vdash \forall mn. m + n = n + m$$

can be used to finish off the proof. This is the typical way of working with discrete universal and lifted discrete universal cpos, namely re-using theorems proved in the set theoretic HOL world and thereby avoiding more complicated reasoning about bottom, e.g. induction involving bottom as in LCF. The commutativity theorem about strict induction is proved by two nested inductions with a bottom case in LCF (see chapter 7).

In this chapter, we shall not go into further details concerning these and any of the other proof functions. Instead the reader is referred to chapter 5. Here, we move on and consider some examples of recursive functions.

2.3 Recursive Functions

We will now consider a few examples which explain part of the reason why it is worthwhile to develop a formalization of domain theory in HOL. To be more precise, we will consider how to define and reason about certain recursive (continuous) functions in HOL-CPO. In section 2.4 and 2.5 we consider how to define and reason about recursive domains with infinite values, which is another part of the reason why the development is worthwhile.

As mentioned above, domain theory supports an understanding of recursive specifications via least fixed points of continuous functionals on pointed cpos. It provides fixed point induction for reasoning about recursive definitions. However, fixed point induction is inadequate for certain kinds of reasoning, for instance, for reasoning about nontermination. Hence, it is important to have access to other techniques of recursion, for example Park induction and other properties of fixed points derivable in domain theory, or structural induction and well-founded induction which is an extremely general principle of induction (see e.g. [Wi93, MW90, DS90]). Well-founded induction in HOL was described in [Ag91, Ag92] (see appendix A). The examples illustrate the use of various techniques to reason about recursive functions.

We start out gently by considering the factorial function. To allow a fixed point definition we consider a function which yields a result in the lifted cpo of natural numbers. However, being primitive recursive, the factorial function is so simple in its recursion that it is also possible to define it in pure HOL. We investigate how the two definitions relate.

As a second example we consider Ackermann's function, which is a classic example of a well-founded recursive function that is not primitive recursive and therefore its definition is not supported directly in HOL. Just as any other recursive (continuous) function, Ackermann's function can be defined as a least fixed point in domain theory. Once this has been done in HOL-CPO, we can prove that Ackermann's function terminates for all inputs by well-founded induction. The example shows how a total recursive Ackermann function in pure HOL can be derived from this termination theorem. A similar approach has been used to obtain a unification algorithm in HOL (see chapter 8) and would work for many well-founded recursive functions that can be defined via a fixed point definition in HOL-CPO. Hence, the example shows in which sense HOL-CPO supports definitions of recursive HOL functions by well-founded induction.

The third and last example of this section illustrates the use of domain theory to prove the equivalence of two recursive nonterminating functions on finite lists. The example illustrates both the use of structural induction on finite lists and the use of fixed point induction on inclusive predicates, in addition to the use of other basic properties of fixed points provided by the formalization of domain theory.

2.3.1 The Factorial Function

The factorial of a natural number n is the natural number $n!$, which is calculated according to the following recursive specification:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n - 1)! & \text{if } n > 0 \end{cases}$$

In Standard ML the factorial function can be computed by a function `fac` defined by

```
fun fac n = if n = 0 then 1 else n * fac(n-1);
```

and in Miranda the factorial function can be defined as follows:

```
fac 0 = 1
fac (n+1) = (n+1) * fac n
```

Both these versions of the factorial work on integers rather than natural numbers and will not terminate when their input is a negative integer.

Below we describe how a recursive function to compute the factorial of a natural number can be defined in the framework presented here. It will be defined as a recursive function `Fac` in the function space `cf(Nat, lift Nat)` where `Nat` is the cpo of natural numbers which was used as an example several times in the previous sections. Note that the factorial is defined as a partial function though it is in fact total on the natural numbers. This is necessary in order to obtain a pointed cpo for the fixed point operator.

Previously we considered addition as an example and we saw that the built-in addition can be proved to be continuous on `Nat` by the ins-prover and then declared to extend the notation of typable terms. For the definition of the factorial function we will make use of the built-in (and continuous) operations for multiplying, subtracting and testing natural numbers for equality. These operations can be declared to extend the notation by the following three theorems:

```
⊢ $* ∈ cf(Nat, cf(Nat, Nat))
⊢ $- ∈ cf(Nat, cf(Nat, Nat))
⊢ $= ∈ cf(Nat, cf(Nat, Bool))
```

which are all proved automatically by the ins-prover (by transforming the continuous function space of discrete universal cpos into a term that syntactically is a discrete universal cpo, and therefore contains all elements of the underlying HOL type).

Next, the fixed point definition of the factorial function can be stated as follows:

```
⊢ Fac =
  Fix
  (λf ∈ cf(Nat, lift Nat), n ∈ Nat.
    Cond((n = 0), Lift 1, Ext(λm ∈ Nat. Lift(n * m))(f(n - 1))))
```

The use of `Fix` makes sense because the functional argument of `Fix` is continuous on a pointed cpo `cf(Nat, lift Nat)`. Note that we use function extension `Ext` because multiplication is an operation on `Nat` and the term $f(n - 1)$, which corresponds to the result of applying the factorial recursively, is an element of the lifted cpo `lift Nat`. One derived definition tool both defines the factorial this way and proves it is continuous:

```
⊢ Fac ∈ cf(Nat, lift Nat)
```

The tools uses the notation of typable terms, i.e. the type checker proves the continuity theorem automatically. Along the way, this proof makes use of the fact that 0 and 1 are elements of `Nat`. These membership facts are proved (automatically) and declared to extend the notation in exactly the same way as the continuity theorems above.

In order for the fixed point operator to work as desired, the cpo `cf(Nat, lift Nat)` must be a pointed cpo. For instance, this is necessary for `Fix` to yield a fixed point:

$$\vdash \forall E, f \in \text{cf}(E, E). \text{pcpo } E \Rightarrow f(\text{Fix } f) = \text{Fix } f$$

The cpo is pointed due to the use of `lift` and the fact that `cf` yields a pointed cpo if its codomain is a pointed cpo. From the fixed point property of `Fix` we can derive the following theorem about `Fac`:

$$\vdash \forall n. \text{Fac } n = \text{Cond}((n = 0), \text{Lift } 1, \text{Ext}(\lambda m \in \text{Nat}. \text{Lift}(n * m))(\text{Fac}(n - 1)))$$

where the recursion of the factorial function appears explicitly. The theorem is a bit ugly and it could be reduced further by a case analysis on whether n is zero or not. However, we cannot simplify the second branch of the conditional easily, since we do not know whether the result of the recursive call is defined or not and therefore cannot reduce the `Ext` term.

However, the factorial function is clearly total, which we can state as follows

$$\vdash \forall n. \exists m. \text{Fac } n = \text{Lift } m$$

and prove by induction on natural numbers. Hence, defining a new HOL constant `FAC` by

$$\vdash \forall n. \text{FAC } n = (@m. \text{Fac } n = \text{Lift } m)$$

we can derive the following less ugly equation from the two previous facts:

$$\vdash \forall n. \text{FAC } n = (n = 0 \rightarrow 1 \mid n * \text{FAC}(n - 1))$$

This specifies the behavior of a total recursive factorial function in pure HOL (without any domain theory), which could equivalently have been defined by primitive recursion:

$$\vdash (\text{FAC } 0 = 1) \wedge (\forall n. \text{FAC}(\text{SUC } n) = \text{SUC}(n) * \text{FAC}(n))$$

One would usually choose the approach using primitive recursion since definition by primitive recursion is supported in pure HOL (as a derived principle of definition). As a matter of fact, domain theory only complicates the definition in these cases. However, when the induction is less trivial this may suggest a useful way to derive total recursive HOL functions. This point is discussed further in the next section.

2.3.2 Ackermann's Function

The binary Ackermann function is a function $A(x, y)$ on the natural numbers which meets the following recursion equations:

$$\begin{aligned} A(0, y) &= y + 1 \\ A(x + 1, 0) &= A(x, 1) \\ A(x + 1, y + 1) &= A(x, A(x + 1, y)) \end{aligned}$$

Ackermann's function is interesting because its values grow extremely fast, for instance,

$$A(0, 0) = 1, \quad A(2, 2) = 7, \quad A(3, 3) = 61, \quad A(3, 6) = 509,$$

and then note that for instance $A(4, 1) = A(3, A(4, 0)) = A(3, 13)$ which is a very large number; SML ran for several minutes without producing a result. The function can be defined in Miranda by the recursion equations above:

$$\begin{aligned} \text{ack } 0 \ n &= n+1 \\ \text{ack } (m+1) \ 0 &= \text{ack } m \ 1 \\ \text{ack } (m+1) \ (n+1) &= \text{ack } m \ (\text{ack } (m+1) \ n) \end{aligned}$$

However, it cannot be defined automatically in a similar way using existing HOL tools since it is not primitive recursive. Ackermann's function is a classic example of a well-founded recursive function. Its proof of termination is best conducted using well-founded induction on a lexicographic combination of the natural numbers with the less-than ordering $<$.

Below, we show how Ackermann's function can be defined as a recursive partial function `ack` in HOL-CPO, using the least fixed point operator. Afterwards, it is proved to be total by well-founded induction (described in [Ag91, Ag92]). As above, this result allows us to define a recursive Ackermann function in pure HOL and derive the recursion equations via the fixed point defined `ack`.

In addition to the operations on the natural number which were introduced above we shall make use of the successor `SUC` and the predecessor `PRE` to define Ackermann's function:

$$\begin{aligned} \vdash \text{SUC} &\in \text{cf}(\text{Nat}, \text{Nat}) \\ \vdash \text{PRE} &\in \text{cf}(\text{Nat}, \text{Nat}) \end{aligned}$$

Note that `PRE` is a total function in HOL since $\vdash \text{PRE}(0) = 0$. Using domain theory we could define a partial predecessor which is undefined on 0 but we will not do that here.

The Ackermann function `ack` is defined as a fixed point of a certain functional `ack_fun`:

$$\begin{aligned} \vdash \text{ack_fun} &= \\ &(\lambda f \in \text{cf}(\text{Nat}, \text{cf}(\text{Nat}, \text{lift Nat})), \ m, \ n \in \text{Nat}. \\ &\quad \text{Cond}((m = 0), \text{Lift}(\text{SUC } n), \\ &\quad \quad \text{Cond}((n = 0), f(\text{PRE } m)1, \text{Ext}(\lambda p \in \text{Nat}. f(\text{PRE } m)p)(f \ m(\text{PRE } n)))))) \\ \vdash \text{ack} &= \text{Fix } \text{ack_fun} \end{aligned}$$

By now the reader should be familiar with the use of function extension `Ext` to extend a function to a lifted cpo in a strict way. Since `ack_fun` is a continuous function on a pointed cpo, stated by

$$\vdash \text{ack_fun} \in \text{cf}(\text{cf}(\text{Nat}, \text{cf}(\text{Nat}, \text{lift Nat})), \text{cf}(\text{Nat}, \text{cf}(\text{Nat}, \text{lift Nat}))),$$

the definition of `ack` makes sense. This was obtained automatically by using the notation of typable terms (the type checker) which was also used to prove:

$$\vdash \text{ack} \in \text{cf}(\text{Nat}, \text{cf}(\text{Nat}, \text{lift Nat}))$$

In fact, the derived tool for defining constants in cpos was used (as usual) for doing both the definitions and the proofs.

From the fixed point property of `Fix`, we can prove quite easily that `ack` satisfies the following recursion equations:

$$\begin{aligned} \vdash (\forall n. \text{ack } 0 \ n &= \text{Lift}(\text{SUC } n)) \wedge \\ (\forall m. \text{ack}(\text{SUC } m)0 &= \text{ack } m \ 1) \wedge \\ (\forall mn. \text{ack}(\text{SUC } m)(\text{SUC } n) &= \text{Ext}(\lambda p \in \text{Nat}. \text{ack } m \ p)(\text{ack}(\text{SUC } m)n)) \end{aligned}$$

From the definition of `ack`, it is not clear syntactically that `ack` is total. Therefore, we cannot get rid of the function constructors `Lift` and `Ext` associated with the lifting construction `lift` on `cpos` immediately. However, we can prove by well-founded induction (see below) that `ack` always terminates, i.e. that it always yields a lifted natural number as a result:

$$\vdash \forall mn. \exists k. \text{ack } m \ n = \text{Lift } k$$

Given this it makes sense to define a constant `ACK` by

$$\vdash \forall mn. \text{ACK } m \ n = (@k. \text{ack } m \ n = \text{Lift } k)$$

and then derive the desired equations for `ACK` from the previous theorems:

$$\begin{aligned} \vdash & (\forall n. \text{ACK } 0 \ n = \text{SUC } n) \wedge \\ & (\forall m. \text{ACK } (\text{SUC } m) \ 0 = \text{ACK } m \ 1) \wedge \\ & (\forall mn. \text{ACK } (\text{SUC } m) (\text{SUC } n) = \text{ACK } m (\text{ACK } (\text{SUC } m) \ n)) \end{aligned}$$

Hence, via domain theory and well-founded induction we have obtained a total recursive Ackermann function in pure HOL which could not have been defined in HOL directly by a primitive recursive definition. However, we should mention that another way to define Ackermann's function in HOL is via a 'hack' exploiting two primitive recursive definitions and higher order functions [Go92].

The approach to introducing recursive HOL functions by well-founded induction illustrated in this example would work for many well-founded recursive functions. Another example is presented in chapter 8 where the proof of correctness of a well-founded recursive unification algorithm is considered.

Before we conclude this example let us say a few words about how the well-founded induction mentioned above was conducted. By general induction on the natural numbers it is easy to prove that the set of all natural numbers in HOL is a well-founded set with the ordering `<`. This is stated in the theorem

$$\vdash \text{wfs}(\text{UNIV}, \$<)$$

where `wfs` states the conditions for a set and an ordering to be a well-founded set (the conditions are equivalent to saying that there are no infinite decreasing chains, see appendix A). The universal set has type `UNIV : num \rightarrow bool`. Using the lexicographic construction on well-founded sets, stated by

$$\vdash \forall BCPR. \text{wfs}(B, P) \wedge \text{wfs}(C, R) \Rightarrow \text{wfs}(\text{prod_set}(B, C), \text{lex_rel}(P, R))$$

where `prod_set` and `lex_rel` are defined as follows

$$\begin{aligned} \vdash & \forall BC. \text{prod_set}(B, C) = \{(b, c) \mid b \in B \wedge c \in C\} \\ \vdash & \forall PRbc. \text{lex_rel}(P, R)(b, c)(b', c') = P \ b \ b' \vee (b = b' \wedge R \ c \ c'), \end{aligned}$$

we obtain the desired well-founded set for the induction proof of termination:

$$\vdash \text{wfs}(\text{prod_set}(\text{UNIV}, \text{UNIV}), \text{lex_rel}(\$<, \$<))$$

A well-founded set supports induction in the following general sense:

$$\vdash \forall CR. \text{wfs}(C, R) = (\forall f. (\forall x \in C. f x) = (\forall x \in C. (\forall y \in C. R y x \Rightarrow f y) \Rightarrow f x))$$

which is derived from the definition of **wfs**. Hence, in order to prove that **ack** is total by well-founded induction we derive the following lemma from the previous two facts:

$$\begin{aligned} \vdash (\forall mn. \exists k. \text{ack } m n = \text{Lift } k) = \\ (\forall mn. \\ (\forall m'n'. m' < m \vee (m' = m \wedge n' < n) \Rightarrow \exists k. \text{ack } m' n' = \text{Lift } k) \Rightarrow \\ (\exists k. \text{ack } m' n' = \text{Lift } k)) \end{aligned}$$

Once this induction theorem has been applied in the proof of termination, the proof proceeds by cases on the arguments of **ack**. Note that fixed point induction cannot be used to prove termination since the termination predicate is not true of the bottom function.

2.3.3 Equality of Two Recursive Functions

Exercise 10.20 in section 10.5 of Winskel's book [Wi93] is a small but non-trivial exercise using several of the techniques for recursion to show the equality of two recursive functions on finite lists. Below, we present a solution to this exercise developed in HOL-CPO.

Let **List** be a discrete cpo of finite lists of natural numbers introduced by the following definition:

$$\vdash \text{List} = \text{discrete}(\text{UNIV} : (\text{num})\text{list} \rightarrow \text{bool})$$

where **list** is the built-in type constructor for finite lists. Hence, the elements of this cpo are HOL terms like the empty list **NIL**, often written as \square , and the list of the first three natural numbers **CONS** 0(**CONS** 1(**CONS** 2 **NIL**)). The list constructor **CONS** is continuous, trivially, since it is an element of the function space $\text{cf}(\text{Nat}, \text{cf}(\text{List}, \text{List}))$ which is a discrete universal cpo. Similarly, the in-built operation **APPEND** of appending two lists to form a new list is continuous—it is an element of $\text{cf}(\text{List}, \text{cf}(\text{List}, \text{List}))$. Assume we have declared the domains of \square , **CONS** and **APPEND** using the theorems

$$\begin{aligned} \vdash \square \in \text{List} \\ \vdash \text{CONS} \in \text{cf}(\text{Nat}, \text{cf}(\text{List}, \text{List})) \\ \vdash \text{APPEND} \in \text{cf}(\text{List}, \text{cf}(\text{List}, \text{List})) \end{aligned}$$

which all hold trivially and are proved immediately by the ins-prover.

The exercise can now be stated as follows: Assume (continuous) functions on natural numbers $s \in \text{cf}(\text{prod}(\text{Nat}, \text{Nat}), \text{Nat})$ and $r \in \text{cf}(\text{prod}(\text{Nat}, \text{Nat}), \text{List})$. Let f be the least function in $\text{cf}(\text{prod}(\text{List}, \text{Nat}), \text{lift Nat})$ satisfying

$$\begin{aligned} f(\square, y) &= \text{Lift } y \\ f(\text{CONS } x xs, y) &= f(\text{APPEND}(r(x, y))xs, s(x, y)). \end{aligned}$$

Let g be the least function in $\text{cf}(\text{prod}(\text{List}, \text{Nat}), \text{lift Nat})$ satisfying

$$\begin{aligned} g(\square, y) &= \text{Lift } y \\ g(\text{CONS } x xs, y) &= \text{Ext}(\lambda v \in \text{Nat}. g(xs, v))(g(r(x, y), s(x, y))). \end{aligned}$$

Prove $f = g$.

In HOL-CPO we first define functions corresponding to f and g using the fixed point operator. In order to simplify the syntax we let **r** and **s** be names of constants rather than keeping them as variables (otherwise functions would be parameterized with variables r and s). For the definitions we declare the following theorems which all hold trivially:

$\vdash s \in \text{cf}(\text{prod}(\text{Nat}, \text{Nat}), \text{Nat})$
 $\vdash r \in \text{cf}(\text{prod}(\text{Nat}, \text{Nat}), \text{List})$
 $\vdash \text{NULL} \in \text{cf}(\text{List}, \text{Bool})$
 $\vdash \text{HD} \in \text{cf}(\text{List}, \text{Nat})$
 $\vdash \text{TL} \in \text{cf}(\text{List}, \text{List})$

The function f is now defined as the least fixed point of a functional f_fun as follows:

$\vdash f_fun =$
 $(\lambda h \in \text{cf}(\text{prod}(\text{List}, \text{Nat}), \text{lift Nat}), (l, y) \in \text{prod}(\text{List}, \text{Nat}).$
 $\quad \text{Cond}(\text{NULL } l, \text{Lift } y, h(\text{APPEND}(r(\text{HD } l, y))(\text{TL } l), s(\text{HD } l, y))))$
 $\vdash f = \text{Fix } f_fun$

and similarly g is defined as the least fixed point of a functional g_fun :

$\vdash g_fun =$
 $(\lambda h \in \text{cf}(\text{prod}(\text{List}, \text{Nat}), \text{lift Nat}), (l, y) \in \text{prod}(\text{List}, \text{Nat}).$
 $\quad \text{Cond}(\text{NULL } l, \text{Lift } y, \text{Ext}(\lambda v \in \text{Nat}. h(\text{TL } l, v))(h(r(\text{HD } l, y), s(\text{HD } l, y))))))$
 $\vdash g = \text{Fix } g_fun$

We only use terms which fit within the notation of typable terms so all functions are continuous. It now follows almost immediately from the fixed point property that f and g meet the above conditions. Here are the theorems which have been proved:

$\vdash (\forall y h. f_fun h ([]), y) = \text{Lift } y) \wedge$
 $(\forall x x s y h. f_fun h (\text{CONS } x x s, y) = h(\text{APPEND}(r(x, y)) x s, s(x, y)))$
 $\vdash (\forall y. f([], y) = \text{Lift } y) \wedge$
 $(\forall x x s y. f(\text{CONS } x x s, y) = f(\text{APPEND}(r(x, y)) x s, s(x, y)))$
 $\vdash (\forall y h. g_fun h ([]), y) = \text{Lift } y) \wedge$
 $(\forall x x s y h. g_fun h (\text{CONS } x x s, y) = \text{Ext}(\lambda v \in \text{Nat}. h(x s, v))(h(r(x, y), s(x, y))))$
 $\vdash (\forall y. g([], y) = \text{Lift } y) \wedge$
 $(\forall x x s y. g(\text{CONS } x x s, y) = \text{Ext}(\lambda v \in \text{Nat}. g(x s, v))(g(r(x, y), s(x, y))))$

The proofs are based mainly on reduction (see section 2.2.3) and on built-in theorems about the operations on lists. In particular, the reduction theorems

$\vdash \forall D, x, y \in D. \text{Cond}(\text{T}, x, y) = x$
 $\vdash \forall D, x, y \in D. \text{Cond}(\text{F}, x, y) = y$

for the conditional Cond were useful.

We are now ready to consider the actual proof of $f = g$. By antisymmetry of the cpo $\text{cf}(\text{prod}(\text{List}, \text{Nat}), \text{lift Nat})$ it is enough to prove $f \sqsubseteq g$ and $g \sqsubseteq f$. Here and below we often omit the subscripts on \sqsubseteq and \perp . The first relation follows by proving g is a fixed point of f_fun and exploiting f is defined as the least prefixed point of f_fun (due to Fix). This proof technique is called Park induction (introduced in section 2.1):

$\vdash \forall E, f \in \text{cf}(E, E), d \in E. \text{pcpo } E \Rightarrow f(d) \sqsubseteq_E d \Rightarrow \text{Fix}(f) \sqsubseteq_E d$

The second relation follows by Park induction on the fixed point definition of g .

We first prove that g is a fixed point of f_fun , i.e. that the following statement holds:

$$\vdash \text{f_fun } g = g$$

By function equality of continuous functions, stated by

$$\vdash \forall D_1 D_2, f, g \in \text{cf}(D_1, D_2). (f = g) = (\forall x \in D_1. f x = g x),$$

we must prove $\text{f_fun } g(l, n) = g(l, n)$ for all lists $l \in \text{List}$ and natural numbers $n \in \text{Nat}$. From the equations for g and f_fun listed above it appears that the equality holds trivially when l is the empty list $[]$ and when l is a list $\text{CONS } h t$ (for appropriate h and t) it reduces to the goal

$$g(\text{APPEND}(r(h, n))t, s(h, n)) = \text{Ext}(\lambda v \in \text{Nat}. g(t, v))(g(r(h, n), s(h, n))).$$

We prove the slightly more general fact

$$\vdash \forall x s l y. g(\text{APPEND } l x s, y) = \text{Ext}(\lambda v \in \text{Nat}. g(x s, v))(g(l, y))$$

by structural induction on the universally quantified variable l , which has type list . By definition of APPEND and by the equations for g we must prove the following two goals:

(step)

$$\begin{aligned} & \text{Ext}(\lambda v \in \text{Nat}. g(\text{APPEND } l x s, v))(g(r(h, y), s(h, y))) = \\ & \text{Ext}(\lambda v \in \text{Nat}. g(x s, v))(\text{Ext}(\lambda v \in \text{Nat}. g(l, v))(g(r(h, y), s(h, y)))) \\ & \quad [\forall y. g(\text{APPEND } l x s, y) = \text{Ext}(\lambda v \in \text{Nat}. g(x s, v))(g(l, y))] \end{aligned}$$

(base)

$$g(x s, y) = \text{Ext}(\lambda v \in \text{Nat}. g(x s, v))(\text{Lift } y)$$

The base case of the induction follows immediately by reduction and the induction step is proved by doing a case split on whether or not $g(r(h, y), s(h, y))$ is Bt , followed by reduction.

Next, let us consider the second part of the proof. By Park induction we can derive $g \sqsubseteq f$ from $g_fun(f) \sqsubseteq f$. The ordering on continuous functions is defined pointwise so for any $l \in \text{List}$ and any $y \in \text{Nat}$ we must prove:

$$g_fun f(l, y) \sqsubseteq f(l, y)$$

Doing a case split on the list l and rewriting with the equations for f and g_fun we obtain the following two subgoals:

(non-empty)

$$(\text{Ext}(\lambda v \in \text{Nat}. f(t, v))(f(r(h, y), s(h, y)))) \sqsubseteq (f(\text{APPEND}(r(h, y))t, s(h, y)))$$

(empty)

$$\text{Lift}(y) \sqsubseteq \text{Lift}(y)$$

When l is the empty list the goal is finished off using reflexivity and when l is a non-empty list $\text{CONS } h t$ we choose to abstract over r and s by proving a slightly more general fact as above:

$$\vdash \forall x s l y. (\text{Ext}(\lambda u \in \text{Nat}. f(x s, u))(f(l, y))) \sqsubseteq (f(\text{APPEND } l x s, y))$$

The proof proceeds by fixed point induction exploiting that f is defined as a fixed point: $\vdash f = \text{Fix } f_fun$. The statement must be changed slightly to fit within the notation of inclusive predicates. An equivalent statement is:

$$\begin{aligned} & \forall(xs, l, y) \in \text{prod}(\text{List}, \text{prod}(\text{List}, \text{Nat})). \\ & (\text{Ext}(\lambda u \in \text{Nat}. f(xs, u))(f(l, y))) \sqsubseteq (f(\text{APPEND } l \ xs, y)) \end{aligned}$$

Fixed point induction on the second occurrence of f gives the following two subgoals:

$$\begin{aligned} & \text{(step)} \\ & (\text{Ext}(\lambda u \in \text{Nat}. f(xs, u))(f_fun \ h(l, y))) \sqsubseteq (f(\text{APPEND } l \ xs, y)) \\ & \quad [h \in \text{cf}(\text{prod}(\text{List}, \text{Nat}), \text{lift } \text{Nat})] \\ & \quad [\forall(xs, l, y) \in \dots (\text{Ext}(\lambda u \in \text{Nat}. f(xs, u))(h(l, y))) \sqsubseteq (f(\text{APPEND } l \ xs, y))] \\ & \text{(base)} \\ & (\text{Ext}(\lambda u \in \text{Nat}. f(xs, u))(\perp(l, y))) \sqsubseteq (f(\text{APPEND } l \ xs, y)). \end{aligned}$$

There is a conversion to calculate bottom in the continuous function space. Applied to the first subgoal (base) it yields:

$$(\text{Ext}(\lambda u \in \text{Nat}. f(xs, u))((\lambda x \in \text{prod}(\text{List}, \text{Nat}). \text{Bt})(l, y)))(f(\text{APPEND } l \ xs, y))$$

After reduction the goal is finished off using Bt is always the bottom of a lifted cpo. The proof of the second subgoal (step) first does a case split on the list l and then uses reduction and various theorems (e.g. the definition and associativity of APPEND , the equations for f and f_fun and the reflexivity property of cpos). We shall not go into the details of the proof here.

To sum up, we have proved the equality $\vdash f = g$ of recursively defined functions f and g on finite lists. Several techniques for recursion were applied: structural induction on lists, the fixed point property of Fix , Park induction and fixed point induction on an inclusive predicate. The initial goal was split up into two cases using the antisymmetry property of cpos. Finally, all cpo, continuity, cpo membership, and inclusiveness facts that were required, were proved behind the scenes automatically.

2.4 Recursive Domains

We have illustrated the use of HOL-CPO to define and reason about recursive functions. However, functions are not the only objects that may be recursively defined in functional programming languages. Often types of data are specified by recursive definitions too. Consider a type specification of the form

$$\alpha \text{ list} ::= \text{Nil} \mid \text{Cons } \alpha * \alpha \text{ list}$$

which specifies a recursive type operator list with two constructor functions Nil and Cons to build-up elements of list . With a slightly different syntax this would be a recursive datatype specification in Standard ML for introducing the type of all finite-length strict lists of any type of elements. Similarly, this corresponds to a datatype specification in Miranda where the type would consist of both finite and infinite lists, called lazy lists. ‘Strict’ and ‘lazy’ refer to whether the constructor functions (of at least one argument) are strict or lazy.

Strict and lazy datatypes denote recursive domains, i.e. cpos (or pointed cpos) which may be recursively defined. In fact, we already saw two such examples above which were introduced using the discrete construction and the universal set of some recursively

defined HOL type. First we considered the cpo of natural numbers, which corresponds to a datatype with constructors 0 and SUC. In the previous section we considered a cpo of finite lists of natural numbers which can be seen as a special case of the above specification interpreted in Standard ML.

More generally, recursive domains can be introduced as the solutions to recursive domain (isomorphism) equations of the form $X \cong F(X)$ (for some appropriate F). There are various standard techniques for this purpose. One method is the inverse limit construction which is based on embedding-projection pairs. Another method is based on the substructure relation between information systems and a third is based on retracts on universal domains like $P\omega$. Chapter 4 presents more details on solving recursive domain equations than this section does.

Our philosophy is that there should be a direct correspondence between elements of HOL types and elements of cpos via the underlying set of cpos, i.e. cpos should share elements with HOL types directly. This does not fit well with these methods of constructions. In fact, formalizing any of them in HOL would yield new ‘worlds’ different from the ‘HOL world’. For instance, the information system of natural numbers would not share any elements with the type of natural numbers in HOL. Therefore, we have chosen to study more ad hoc methods for solving recursive domain equations.

One approach for recursive domains with finite values only is presented in section 2.4.1 below. This is based directly on the recursive types provided by the type definition package of HOL [Me89] (the limitations of which are inherited). The type definition package cannot be used to produce infinite values as elements of a recursive type. One way to allow infinite values in recursive domains is to add these infinite elements (as well as partial elements) to the type returned by the type definition package. An example of this approach is presented in section 2.4.2. A third approach is based on a generalization of the way in which the type definition package works. A set of finite, partial and infinite labeled trees is constructed and shown to be a pointed cpo with a certain ordering. Trees are represented as sets of nodes. New recursive cpos with infinite values can then be defined as subsets (sub-cpos) of this set (cpo) of infinite trees. This approach is described briefly in section 2.4.3. Note, however, that the approach has not been formalized in HOL. We shall keep each of the following presentations short; more details can be found in chapter 4.

2.4.1 Finite Values

As an example, we describe how a constructor `list` D for cpos of finite lists of elements of a cpo D can be defined in HOL-CPO. The underlying type of the list construction is the recursive type of finite lists in HOL and the underlying relation relates lists of the same length if their elements are related pairwise by the ordering on D . Using primitive recursive definitions we can define the set of elements of a HOL list and the desired ordering relation on `list` as follows:

$$\begin{aligned} &\vdash (\text{list_set} [] = \{\}) \wedge (\forall ht. \text{list_set}(\text{CONS } h t) = \{h\} \cup (\text{list_set } t)) \\ &\vdash (\forall D l. \text{list_rel } D [] l = (l = [])) \wedge \\ &\quad (\forall D h t l. \\ &\quad \quad \text{list_rel } D (\text{CONS } h t) l = (\exists h' t'. l = \text{CONS } h' t' \wedge h \sqsubseteq_D h' \wedge \text{list_rel } D t t')) \end{aligned}$$

The constructor `list` is now defined by:

$$\vdash \forall D. \text{list } D = \{l \mid (\text{list_set } l) \subseteq D\}, \text{list_rel } D$$

However, it does not follow immediately that `list` yields a cpo provided its argument is a cpo:

$$\vdash \forall D. \text{cpo } D \Rightarrow \text{cpo}(\text{list } D)$$

This fact must be proved in HOL. Hence, one must prove how, for instance, chains and least upper bounds of chains are constructed in a cpo of lists. We shall leave out the details here.

The list construction has the property that if its argument cpo is discrete then the list cpo itself becomes discrete. Hence, the discrete universal cpo `List` of lists of natural numbers introduced in the previous section is equivalent (by HOL equality) to the cpo `list Nat`. However, a cpo like `list(lift Nat)` could not be defined as a discrete cpo since `lift Nat` is not discrete (and hence, the list cpo of elements in this cpo is not discrete).

The constructor functions on `list` are `NIL`, the in-built constructor for the empty list, and a determined version of the built-in list constructor `CONS`, which is defined using the dependent lambda abstraction in the straightforward way.

One might wish other variants of list cpos. For instance, a cpo of semi-strict (or tail-strict) lists could consist of a bottom list in addition to the finite lists provided by `list`. Hence, a constructor for semi-strict lists could be defined by:

$$\vdash \forall D. \text{sslist } D = \text{lift } (\text{list } D)$$

Strict lists correspond to semi-strict lists where the element cpo is a pointed cpo and all lists have only defined elements. A constructor for strict lists can be defined as follows:

$$\vdash \forall E. \text{slist } E = \{l \mid l \in \text{sslist } E \wedge \perp_E \notin (\text{lift_case}\{\}\text{list_set } l)\}, \sqsubseteq_{\text{sslist } E}$$

where `lift_case` is a cases construction on the lifted type defined by:

$$\vdash (\forall a f. \text{lift_case } a f \text{ Bt} = a) \wedge (\forall a f x. \text{lift_case } a f (\text{Lft } x) = f x)$$

It is used to extend `list_set` to lifted lists. This construction yields a pointed cpo provided its argument cpo is pointed. Proper list constructor functions can be defined for these constructions as well.

2.4.2 Lazy Sequences

Certain functional programming languages support lazy constructors for elements of recursive datatypes. If for instance the list constructor `CONS x l` is lazy then it does not evaluate its arguments `x` and `l` until this is absolutely necessary. Hence, a function like

$$f(n) = \text{CONS } n (f(n + 1))$$

may be defined in such a language to generate an infinite list of natural numbers. In a language where `CONS` is strict the above function would never terminate on any argument.

Domain theory allows an understanding of both finite and infinite values. The infinite elements are understood as least upper bounds of partial approximations which are finitely-generated. Hence, a recursive domain containing infinite elements must contain partial elements as well.

In HOL, one way to represent infinite values is as functions. For instance, an infinite list of elements of some type α can be represented by a function $f : \text{num} \rightarrow \alpha$ such that the list is $[f(0); f(1); f(2); \dots; f(n); \dots]$. This suggests an approach to defining the elements of a recursive domain of lazy lists: represent the finite and partial lists by the recursive type of lists in HOL and represent the infinite elements as HOL functions. (It would not be simpler to represent all elements by functions alone since extra information is needed to distinguish finite, partial and infinite elements.) The ordering on the recursive domain should then make sure that the partial lists can be seen as the approximating partial results of a computation of an infinite list.

The usability of this approach of distinguishing the different kinds of values in a recursive domain is rather limited. For instance, it would be considerably more difficult to apply the approach to defining a cpo with infinite binary trees, since a binary tree may contain both finite, partial and infinite branches at the same time. Below, we shall describe the approach anyway, since it was the only approach to define recursive domains with infinite values which was actually tried out in HOL. We consider the construction of a recursive domain corresponding to lazy sequences, which are equivalent to lazy lists except that the finite elements are ignored. It is easy to add these elements in the definitions below to produce a construction on lazy lists as well. In fact, this has been done in HOL such that constructions for pointed cpos of both lazy sequences and lazy lists are available in HOL-CPO.

A lazy sequence is a partial sequence or an infinite sequence. We can therefore define a type of sequences in HOL by taking the disjoint union of the type of finite lists $(\alpha)\text{list}$ interpreted as partial sequences (**pars**) and the function type $\text{num} \rightarrow \alpha$ interpreted as infinite sequences (**infs**):

$$(\alpha)\text{seq} = \text{pars } (\alpha)\text{list} \mid \text{infs } \text{num} \rightarrow \alpha$$

The underlying set of the lazy sequence cpo construction $\text{seq } D$ must consist of all sequences whose elements are elements of D :

$$\vdash \forall D. \text{seq } D = \{s \mid (\text{seq_set } s) \subseteq D\}, \text{seq_rel } D$$

where **seq_set** is defined as expected:

$$\begin{aligned} \vdash (\forall l. \text{seq_set}(\text{pars } l) &= \{x \mid \text{MEMBER } x \ l\}) \wedge \\ (\forall f. \text{seq_set}(\text{infs } f) &= \{f \ n \mid 0 \leq n\}) \end{aligned}$$

The more difficult part of defining the cpo constructor for sequences is defining the underlying ordering relation **seq_rel**:

$$\begin{aligned} \vdash \forall D s s'. \\ \text{seq_rel } D \ s \ s' = \\ (\exists l l'. s = \text{pars } l \wedge s' = \text{pars } l' \wedge \text{psrel } D \ l \ l') \vee \\ (\exists l f. s = \text{pars } l \wedge s' = \text{infs } f \wedge \text{pisrel } D \ l \ f) \vee \\ (\exists f f'. s = \text{infs } f \wedge s' = \text{infs } f' \wedge (\forall n. f(n) \sqsubseteq_D f'(n))) \end{aligned}$$

Its definition exploits that there are different kinds of sequences and that partial sequences correspond to finite lists. Hence, the sense in which a partial sequence approximates another sequence can be defined by two primitive recursive definitions:

$$\begin{aligned}
& \vdash (\forall D l. \text{psrel } D \ [] l = \top) \wedge \\
& \quad (\forall D x l_1 l_2. \text{psrel } D (\text{CONS } x l_1) l_2 = (\exists y l'_2. l_2 = \text{CONS } y l'_2 \wedge x \sqsubseteq_D y \wedge \text{psrel } D l_1 l'_2)) \\
& \vdash (\forall D f. \text{pisrel } D \ [] f = \top) \wedge \\
& \quad (\forall D x l f. \text{pisrel } D (\text{CONS } x l) f = x \sqsubseteq_D f(0) \wedge \text{pisrel } D l (\lambda n. f(n+1)))
\end{aligned}$$

The definitions say that one lazy sequence s approximates another s' if, and only if, they are both partial sequences such that s contains at most as many elements as s' and such that their elements are related pairwise; or if s is a partial sequence and s' is an infinite sequence such that their elements are related pairwise; or if they are both infinite sequences such that their elements are related pairwise.

With these definition we can prove that `seq` is a constructor for pointed cpos. It yields a pointed cpo, provided its argument is a cpo:

$$\vdash \forall D. \text{cpo } D \Rightarrow \text{pcpo}(\text{seq } D)$$

We can use this fact to extend the notations for cpos and pointed cpos by declaring `seq` by the constructor theorem for both notations (see section 2.2). It is easy to see that the bottom sequence, called `Bt_seq`, is the empty partial sequence `pars []`.

In order to define the continuous constructor function `Cons_seq` for lazy sequences we first define a function `conss` by primitive recursion as follows:

$$\begin{aligned}
& \vdash (\forall x l. \text{conss } x (\text{pars } l) = \text{pars}(\text{CONS } x l)) \wedge \\
& \quad (\forall x f. \text{conss } x (\text{infs } f) = \text{infs}(\lambda n. (n = 0 \rightarrow x \mid f(n-1))))
\end{aligned}$$

This is almost the constructor we wish but it is not determined, which a continuous function must be. Hence, we define `Cons_seq` as a determined version of the constructor using the dependent lambda abstraction:

$$\vdash \forall D. \text{Cons_seq}|_D = (\lambda x \in D, s \in \text{seq } D. \text{conss } x s)$$

This constructor is continuous:

$$\vdash \forall D. \text{cpo } D \Rightarrow \text{Cons_seq}|_D \in \text{cf}(D, \text{cf}(\text{seq } D, \text{seq } D))$$

and hence, it can be used to extend the notation of typable terms, which we assume has hereby been done. Note that the continuity theorem itself does not follow from the notation but must be proved manually in HOL-CPO.

In order to establish that `Cons_seq` behaves like a constructor, we prove it is one-one:

$$\vdash \forall D, x, x' \in D, s, s' \in \text{seq } D. (\text{Cons_seq } x s = \text{Cons_seq } x' s') = (x = x') \wedge (s = s')$$

it is distinct from the bottom sequence:

$$\vdash \forall D, x \in D, s \in \text{seq } D. \text{Cons_seq } x s \neq \text{Bt_seq}$$

and it is exhaustive on the cpo of lazy sequences:

$$\vdash \forall D s. s \in \text{seq } D = (s = \text{Bt_seq}) \vee (\exists x \in D, s' \in \text{seq } D. s = \text{Cons_seq } x s')$$

These are standard conditions that a proper constructor function should satisfy. The also standard structural induction theorem is presented in section 2.5.

It is useful to define an eliminator or cases functional for writing continuous functions on lazy sequences. The eliminator for lazy sequences is called `Seq_when` and defined by:

$\vdash \forall DE.$

$$\text{Seq_when}_{(D,E)} = (\lambda h \in \text{cf}(D, \text{cf}(\text{seq } D, E)), s \in \text{seq } D. (s = \text{Bt_seq} \rightarrow \perp_E | h(\text{hds } s)(\text{tls } s)))$$

where the constants **hds** and **tls** were defined such that the following theorems hold:

$$\begin{aligned} \vdash \forall xs. \text{hds}(\text{conss } x \ s) &= x \\ \vdash \forall xs. \text{tls}(\text{conss } x \ s) &= s \end{aligned}$$

The eliminator is continuous:

$$\vdash \forall DE. \text{cpo } D \wedge \text{pcpo } E \Rightarrow \text{Seq_when}_{(D,E)} \in \text{cf}(\text{cf}(D, \text{cf}(\text{seq } D, E)), \text{cf}(\text{seq } D, E))$$

which justifies that we drop the cpo parameters below; we can declare the continuity theorem to extend the notation of typable terms. The eliminator works as follows:

$$\begin{aligned} \vdash \forall DE, h \in \text{cf}(D, \text{cf}(\text{seq } D, E)). \text{Seq_when } h \ \text{Bt_seq} &= \perp_E \\ \vdash \forall DE, h \in \text{cf}(D, \text{cf}(\text{seq } D, E)), x \in D, s \in \text{seq } D. \text{Seq_when } h \ (\text{Cons_seq } x \ s) &= h \ x \ s \end{aligned}$$

which are called the reduction theorems for the eliminator. Note that the second cpo parameter must be a pointed cpo in order for the eliminator to be continuous but this is not a condition in order to apply the reduction theorems. The reason for this difference is that in the proof of the reduction theorems it is not necessary to check that **bottom** really does return a least element of a cpo.

The development of lazy sequences in HOL-CPO was long, tedious and relatively difficult. Every single fact about lazy sequences must be proved in several versions since the initial distinction between the different kinds of elements is inherited throughout the whole development. Hence, a more uniform treatment of the different kinds of elements would probably reduce the complexity of the development a lot. It was fairly easy to add the extra cases to obtain a pointed cpo constructor for lazy lists.

2.4.3 Infinite Values

In the following, we investigate another method for constructing recursive domains, which is based on an idea similar to the one employed in the type definition package. We did not attempt to formalize the method in HOL since there might be variations of the method, or other methods, which support more general domains without introducing complexity. We shall only sketch the method here; it is described in more detail in section 4.3. Note that we do not use the turnstyle \vdash for definitions and theorems which have not been formalized in HOL.

The idea is to derive a construction **itree** D for pointed cpos of infinite labeled trees, or more precisely, for finitely-branching labeled trees (like the trees in the type definition package) with finite, partial and infinite values. The labels must be elements of the cpo D . Certain recursive domains can then be defined as sub-cpos of the cpo **itree** D for appropriate labels D in the sense that the underlying set of the domain is a subset of **itree** D and the underlying ordering relation is inherited from **itree** D . The labels are used in exactly the same way as in the type definition package so this method supports the same kind of recursive definitions, except that here the result is a recursive domain with infinite values.

The underlying type of the infinite tree construction is sets of nodes. A node is a pair consisting of a path and a label. A path is a list of numbers indicating which branches lead to the node, starting at the root of the tree. A label can be some value or a flag stating that the node is a partial node. No value is associated with a partial node. Let $(\alpha)\text{node}$ abbreviate the type of nodes $(\text{num})\text{list} \times (\alpha)\text{label}$ where $(\alpha)\text{label} = \text{NOLBL} \mid \text{LBL } \alpha$.

Sets of paths (not exactly nodes) are also used by Gunter [Gu93] to represent finite (well-founded) but arbitrarily-branching trees (see section 4.5), and by Paulson [Pa93] to represent infinite finitely-branching trees. Their work differs from ours in various respects, for instance, they construct types, not domains, so their trees do not include the partial nodes. Paulson uses finitely-branching but infinite trees and takes greatest fixed points to obtain the infinite values; in domain theory, we would use the fixed point operator.

An infinite labeled tree can be described by some set of nodes $t : (\alpha)\text{node} \rightarrow \text{bool}$ but this type contains elements which cannot be interpreted as trees. Hence, we shall choose a certain subset of this type, called

$$\text{Is_infinite_tree} : ((\alpha)\text{node} \rightarrow \text{bool}) \rightarrow \text{bool}.$$

To define this subset we first define a subset corresponding to all finitely- and infinitely-branching trees:

$$\begin{aligned} \text{Is_tree } t = & \\ & (\exists l. ([], l) \in t) \wedge \\ & (\forall pp'l. p' \neq [] \wedge (\text{APPEND } pp', l) \in t \Rightarrow \exists v. (p, \text{LBL } v) \in t) \wedge \\ & (\forall pll'. (p, l) \in t \wedge (p, l') \in t \Rightarrow l = l') \end{aligned}$$

The conditions ensure that there is a root node, that all paths are prefix closed (such that there are no holes due to missing nodes) and that labels are unique. To exclude infinitely-branching trees, the additional conditions on infinite labeled trees are as follows:

$$\begin{aligned} \text{Is_infinite_tree } t = & \\ & \text{Is_tree } t \wedge \\ & (\forall pl. (p, l) \in t \Rightarrow \exists k. \{n \mid \exists l'. (\text{SNOC } n p, l') \in t\} = \{0, \dots, k\}) \wedge \\ & (\exists m. \forall npl. (\text{CONS } n p, l) \in t \Rightarrow n \leq m) \end{aligned}$$

The constant **SNOC** adds an element at the tail of a list. Note that in addition to requiring that each node has finitely many subtrees, we require that the subtrees are enumerated by counting from zero up to some number. The last condition says that there must be an upper limit on the number of subtrees of all nodes of a tree. If we did not have this condition then a tree might be of infinite width (even though it is finitely-branching at each node).

The construction for pointed cpos of infinite labeled trees is defined as follows:

$$\text{itree } D = \{t \mid \text{Is_infinite_tree } t \wedge (\text{label_set } t) \subseteq D\}, \text{itree_rel } D$$

where the set of labels of a tree is defined by

$$\text{label_set } t = \{v \mid \exists p. (p, \text{LBL } v) \in t\}$$

and the definition of the ordering is:

$$\begin{aligned}
\text{itree_rel } D t t' = & \\
(\forall p. (p, \text{NOLBL}) \in t \Rightarrow \exists l'. (p, l') \in t') \wedge & \\
(\forall p v. (p, \text{LBL } v) \in t \Rightarrow \exists v'. v \sqsubseteq_D v' \wedge (p, \text{LBL } v') \in t) \wedge & \\
(\forall p l. & \\
(p, l) \notin t \wedge (p, l) \in t' \Rightarrow & \\
(\exists p' p''. p = \text{APPEND } p' p'' \wedge ((p', \text{NOLBL}) \in t \vee \exists v. (p, \text{LBL } v) \in t)) &
\end{aligned}$$

The definition supports the intuition that a partial node can approximate any node. So, if some path ends at a partial node of one tree then we obtain a more defined tree by replacing that node with an arbitrary tree. A node with a label l may also approximate a node with the same path but a different label l' if l approximates l' w.r.t. the underlying ordering of the cpo of labels.

The constant `itree` is a pointed cpo constructor:

$$\forall D. \text{cpo } D \Rightarrow \text{pcpo}(\text{itree } D)$$

Here pointed cpo is essential since it allows us to write recursive functions on infinite trees using the fixed point operator. The bottom node is the singleton set $\{([], \text{NOLBL})\}$.

Finally, let us see how lazy sequences could be defined as a sub-cpo of infinite trees:

$$\text{ls_seq } s = \text{ls_infinite_tree } s \wedge (\forall p l. (p, l) \in s \Rightarrow \forall x. \text{MEMBER } x p \Rightarrow x = 0)$$

Hence, a sequence is a tree in which there are at most one branch out of each node. This is not the most general way to define the desired subset of infinite trees. Instead one can define predicates to identify which nodes correspond to trees constructed using one of the constructor functions of a recursive domain (or type) specification. The reader is referred to section 4.4 for further information.

2.5 Reasoning about Infinite Values

We have now shown methods of deriving recursive domains with infinite values in HOL-CPO. Next, we provide two proof principles of induction for reasoning about their elements. Structural induction (in the sense of Paulson [Pa84a]) is derived from fixed point induction and can be used to prove that inclusive properties hold of all elements of recursive domains, in particular the infinite ones. Co-induction, described for instance in [Pi94], can be used to prove the equality of two infinite values by inventing a bisimulation. The well-known technique of fixed point induction can sometimes be useful to reason about infinite values of recursive domains too.

2.5.1 Structural Induction

Structural induction for lazy sequences can be stated as the following theorem:

$$\begin{aligned}
& \vdash \forall DP. \\
& \text{cpo } D \wedge \text{inclusive}_D P \Rightarrow \\
& P \text{Bt_seq} \wedge (\forall s, x \in D. P s \Rightarrow P(\text{Cons_seq } x s)) \Rightarrow (\forall s \in \text{seq } D. P s)
\end{aligned}$$

The theorem is derived from fixed point induction by defining a recursive copying function and prove that it behaves as identity on all sequences:

$$\vdash \forall D, s \in \text{seq } D. \text{Copy } s = s$$

Using this on the consequent of the previous theorem and expanding the fixed point definition (see below) of the copying function, the consequent reduces to a property which is proved by fixed point induction below. The antecedents of the structural induction theorem are essentially the antecedents of the fixed point induction theorem (see section 2.1). The copying functional is defined using the ‘when’ eliminator functional for lazy sequences:

$$\begin{aligned} &\vdash \forall D. \\ &\quad \text{Copl}_D = \\ &\quad \text{Fix}(\lambda f \in \text{cf}(\text{seq } D, \text{seq } D). \text{Seq_when}(\lambda x \in D, s \in \text{seq } D. \text{Cons_seq } x (f s))) \end{aligned}$$

The definition fits within the notation of typable terms so **Copy** is a continuous function and we are justified in dropping the cpo parameter (by declaring the continuity theorem and using interface level syntax).

Let us consider an example on the use of structural induction. Define a recursive mapping functional by the following fixed point definition:

$$\begin{aligned} &\vdash \forall D_1 D_2. \\ &\quad \text{Mapsl}_{(D_1, D_2)} = \\ &\quad \text{Fix} \\ &\quad (\lambda g \in \text{cf}(\text{cf}(D_1, D_2), \text{cf}(\text{seq } D_1, \text{seq } D_2)), f \in \text{cf}(D_1, D_2). \\ &\quad \quad \text{Seq_when}(\lambda x \in D_1, s \in \text{seq } D_1. \text{Cons_seq}(f x)(g f s))) \end{aligned}$$

By the notation of typable terms it is continuous:

$$\vdash \forall D_1 D_2. \text{cpo } D_1 \wedge \text{cpo } D_2 \Rightarrow \text{Mapsl}_{(D_1, D_2)} \in \text{cf}(\text{cf}(D_1, D_2), \text{cf}(\text{seq } D_1, \text{seq } D_2))$$

and we can therefore declare it as a new constructor to extend the notation. It may not be obvious from the definition that **Maps** really behaves as a mapping functional but the following recursion equations state this clearly:

$$\begin{aligned} &\vdash \forall D_1 D_2, f \in \text{cf}(D_1, D_2). \text{Maps } f \text{ Bt_seq} = \text{Bt_seq} \\ &\vdash \forall D_1 D_2, f \in \text{cf}(D_1, D_2), x \in D_1, s \in \text{seq } D_1. \\ &\quad \text{Maps } f (\text{Cons_seq } x s) = \text{Cons_seq}(f x)(\text{Maps } f s) \end{aligned}$$

Now, we can prove that the mapping functional preserves functional composition by structural induction:

$$\begin{aligned} &\vdash \forall D_1 D_2 D_3, f \in \text{cf}(D_2, D_3), g \in \text{cf}(D_1, D_2). \\ &\quad \text{Maps}(\text{Comp}(f, g)) = \text{Comp}(\text{Maps } f, \text{Maps } g) \end{aligned}$$

where **Comp** is defined by:

$$\begin{aligned} &\vdash \forall D_1 D_2 D_3. \\ &\quad \text{Compl}_{(D_1, D_2, D_3)} = \\ &\quad (\lambda(f, g) \in \text{prod}(\text{cf}(D_2, D_3), \text{cf}(D_1, D_2)), x \in D_1. f(g x)) \end{aligned}$$

Functional composition is continuous:

$$\begin{aligned} &\vdash \forall D_1 D_2 D_3. \\ &\quad \text{cpo } D_1 \wedge \text{cpo } D_2 \wedge \text{cpo } D_3 \Rightarrow \\ &\quad \text{Compl}_{(D_1, D_2, D_3)} \in \text{cf}(\text{prod}(\text{cf}(D_2, D_3), \text{cf}(D_1, D_2)), \text{cf}(D_1, D_3)) \end{aligned}$$

and can therefore be considered to be part of the notation of typable terms as a construction on continuous functions. (The proof of continuity is presented in section 3.10.)

The goal does not allow structural induction immediately but recall that two continuous functions (with common domains) are equal if, and only if, they are equal for each element of their domain cpo:

$$\vdash \forall D_1 D_2, f, g \in \text{cf}(D_1, D_2). (f = g) = (\forall x \in D_1. f x = g x)$$

Hence, to prove the statement above it is enough to prove

$$\forall s \in \text{seq } D_1. \text{Maps}(\text{Comp}(f, g))s = \text{Comp}(\text{Maps } f, \text{Maps } g)s,$$

which allows structural induction, assuming that f and g are continuous. A tactic for structural induction is easily based on the structural induction theorem. This tactic uses the notations for cpos and inclusive predicates to prove the first two antecedents of the theorem. Hence, using the structural induction tactic the previous statement reduces to the following two cases of the induction:

$$\begin{aligned} &(\text{step}) \\ &\quad \text{Maps}(\text{Comp}(f, g))(\text{Cons_seq } x s) = \text{Comp}(\text{Maps } f, \text{Maps } g)(\text{Cons_seq } x s) \\ &\quad \quad [x \in D_1] \\ &\quad \quad [s \in \text{seq } D_1] \\ &\quad \quad [\text{Maps}(\text{Comp}(f, g))s = \text{Comp}(\text{Maps } f, \text{Maps } g)s] \\ &(\text{base}) \\ &\quad \text{Maps}(\text{Comp}(f, g))\text{Bt_seq} = \text{Comp}(\text{Maps } f, \text{Maps } g)\text{Bt_seq} \end{aligned}$$

The details of the proof are not that interesting so we skip them here.

2.5.2 Fixed Point Induction

Continuing the example above, we may now define a recursive functional $\text{Seq_of } f x$ for generating the infinite sequence

$$[x; f(x); f^2(x); \dots; f^n(x); \dots],$$

which stands for $\text{Cons_seq } x(\text{Cons_seq}(f(x))(\text{Cons_seq}(f(f(x))) \dots))$, and prove the following result about the generator and the mapping functionals by fixed point induction:

$$\vdash \forall D, f \in \text{cf}(D, D). \text{cpo } D \Rightarrow (\forall x \in D. \text{Seq_of } f (f x) = \text{Maps } f (\text{Seq_of } f x))$$

The generator functional is defined as follows:

$$\begin{aligned} &\vdash \forall D. \\ &\quad \text{Seq_ofl}_D = \\ &\quad \text{Fix} \\ &\quad (\lambda g \in \text{cf}(\text{cf}(D, D), \text{cf}(D, \text{seq } D)), f \in \text{cf}(D, D), x \in D. \text{Cons_seq } x (g f (f x))) \end{aligned}$$

and it is continuous by the notation of typable terms (which it can then be used to extend). Intuitively, the equality holds because both sequences are equal to

$$[f(x); f^2(x); \dots; f^n(x); \dots].$$

But this is no proof. Assuming $f \in \text{cf}(D, D)$ and $\text{cpo } D$, we wish to prove:

$$\forall x \in D. \text{Seq_of } f (f x) = \text{Maps } f (\text{Seq_of } f x)$$

The statement holds trivially if D is empty so we assume D is not empty. The proof proceeds by fixed point induction on both occurrences of `Seq_of`. The fixed point induction tactic, which uses the notations to prove the conditions on fixed point induction, yields the following two statements to prove:

(step)

$$\begin{aligned} \text{Cons_seq } (f x) (h f (f (f x))) &= \text{Maps } f (\text{Cons_seq } x (h f (f x))) \\ [\forall x \in D. h f (f x) = \text{Maps } f (h f x)] \end{aligned}$$

(base)

$$\perp f (f x) = \text{Maps } f (\perp f x)$$

We will skip the details of the proofs of these.

2.5.3 Co-induction

Co-induction for lazy sequences can be stated as the following theorem:

$$\vdash \forall DR. \text{cpo } D \wedge \text{bisim_seq}_D R \Rightarrow (\forall s, s' \in \text{seq } D. R s s' \Rightarrow s = s')$$

It says that in order to prove the equality of two sequences, it suffices to find a bisimulation which relates the two sequences. The notion of bisimulation is defined as follows:

$\vdash \forall DR.$

$$\begin{aligned} \text{bisim_seq}_D R &= \\ (\forall xs, xs' \in \text{seq } D. \\ R xs xs' &\Rightarrow \\ (xs = \text{Bt_seq}) \wedge (xs' = \text{Bt_seq}) &\vee \\ (\exists x \in D, s, s' \in \text{seq } D. xs = \text{Cons_seq } x s \wedge xs' = \text{Cons_seq } x s' \wedge R s s')) \end{aligned}$$

Hence, two sequences are in bisimulation with one another if they are both equal to the bottom sequence, or if they are equal to two sequences with the same head such that the two tails are in bisimulation with one another. The proof of co-induction is based on the take lemma (e.g. [BW88]) and a ‘taker’ for sequences is defined using `pow` (see section 2.1) and the same functional as was used to define the copying function (see section 2.5.1). This is certainly not the most general way to derive co-induction but it works well for lazy sequences and lazy lists. (The derivation of co-induction was inspired by looking at the Isabelle code of [Re94].)

We illustrate the use of co-induction by proving the equality of two infinite sequences of natural numbers `nats = from 0` defined as follows [Pi94]:

$$\begin{aligned} \vdash \text{nats} &= \text{Fix}(\lambda s \in \text{seq } \text{Nat}. \text{Cons_seq } 0 (\text{Maps } \text{SUC } s)) \\ \vdash \text{from} &= \text{Fix}(\lambda h \in \text{cf}(\text{Nat}, \text{seq } \text{Nat}), n \in \text{Nat}. \text{Cons_seq } n (h(\text{SUC } n))) \end{aligned}$$

where `Maps` is the mapping functional defined in section 2.5.1 above. From the fixed point property we derive the following recursion equations:

$$\begin{aligned} \vdash \text{nats} &= \text{Cons_seq } 0 \text{ (Maps SUC nats)} \\ \vdash \forall n. \text{from } n &= \text{Cons_seq } n \text{ (from(SUC } n)) \end{aligned}$$

Hence, it does not seem unreasonable to attempt to prove the equality: `nats = from 0`. The proof is by co-induction with the bisimulation

$$\vdash \text{bisim_seq}_{\text{seq Nat}}(\lambda s s'. \exists n. s = \text{Maps_iter } n \text{ nats} \wedge s' = \text{from } n)$$

where `Maps_iter n s` iterates the mapping functional n times over a sequence s of natural numbers:

$$\begin{aligned} \vdash (\text{Maps_iter } 0 &= (\lambda x \in \text{seq Nat. } x)) \wedge \\ (\forall n. \text{Maps_iter}(\text{SUC } n) &= (\lambda x \in \text{seq Nat. } \text{Maps_iter } n \text{ (Maps SUC } x))) \end{aligned}$$

To prove the suggested relation is a bisimulation, the second disjunct of the definition of bisimulation is established by choosing x to be n , s to be `Maps_iter(SUC n)nats` and s' to be `from(SUC n)`. Note that the continuity of `Maps_iter` does not follow from the notation of typable terms immediately since it takes its first argument without the use of the dependent lambda abstraction. However, the continuity is easily established since `Nat` is a discrete universal cpo. Another consequence of using a primitive recursive definition of `Maps_iter` is that we must use structural induction to derive the recursion equations:

$$\begin{aligned} \vdash (\forall x. \text{Maps_iter } 0 \ x &= x) \wedge \\ (\forall n x. \text{Maps_iter}(\text{SUC } n) \ x &= \text{Maps SUC (Maps_iter } n \ x)) \end{aligned}$$

With a fixed point definition these could be derived by appealing to the fixed point property. Note that the cpo of sequences of elements in a universal cpo is itself a universal cpo (i.e. it contains all elements of the underlying type).

Once the bisimulation has been invented the proof is straightforward. The two sequences are trivially related by the bisimulation since their first elements are the same.

2.6 Conclusion

This chapter has provided an overview of various aspects of HOL-CPO. The formalization of domain theory has been presented with some illustrative examples. The extension of HOL-CPO with new recursive domains has also been discussed, though no implemented methods for deriving recursive domains have been presented, except for an example illustrating the derivation of a pointed cpo constructor for lazy sequences. The weight has been put on showing the variety of proof techniques for recursion available with HOL-CPO. Hence, the methods of Park induction, fixed point induction, structural induction and well-founded induction have been demonstrated in use, and in addition, the principles of structural induction and co-induction were employed to reason about the recursive domain of lazy sequences which contains partial and infinite sequences. The example of section 2.3.3 brought several of these techniques into use to show the equality of two partial (non-primitive) recursive functions on finite lists.

It should be clear to the reader that HOL-CPO extends HOL with a number of useful concepts and techniques. In HOL-CPO one may reason about partial functions and nontermination and more general recursive (computable) functions may be defined, using the fixed point operator. Fixed point defined recursive functions must in general be (potentially) partial functions in order to allow taking the least fixed point. But some recursive functions can be proved to be total, for instance, by well-founded induction, and this allows the derivation of recursion equations for pure HOL recursive functions. In this way, HOL-CPO supports recursive function definitions by well-founded induction. This point was illustrated by the example on Ackermann's function (see section 2.3.2).

One of the weaknesses of the formalization is the need for the dependent lambda abstraction and cpo parameters on certain function constructions. This makes the syntax of functions inconvenient, though an interface can hide much of this inconvenience. Further, it complicates proofs since arguments of functions must be type checked before dependent lambda abstractions can be reduced. Again, this has been automated to some degree.

In the following chapters the formalization and the associated tools are presented in more details than here. We shall switch to real HOL syntax and show real HOL sessions to illustrate the use of HOL-CPO. Chapter 6 and chapter 7 provide fairly small and simple examples, some of which have been presented here too (some examples are only presented in this chapter). A larger example showing the proof of correctness of a unification algorithm is presented in chapter 8. For a more full discussion on HOL-CPO, the reader should consult the conclusion chapter 9.

Chapter 3

Basic Concepts of Domain Theory

Most computer scientists are familiar with the basic concepts of domain theory, such as complete partial orders (cpo), continuous functions and least fixed points. In denotational semantics, types of data are modeled as cpos and programs as continuous functions between cpos. Recursion is handled by taking least fixed points of continuous functionals, nontermination using a so-called bottom or undefined element of cpos. Good introductions on domain theory are provided in the textbooks by Winskel [Wi93], Schmidt [Sc86] and Gunter [Gu92].

In this chapter we provide an overview of a formalization of basic concepts of domain theory in HOL, based mainly on the book by Winskel. The notions of cpo and continuous functions are introduced as semantic constants in HOL stating a number of conditions that terms must satisfy in order to be cpos and continuous functions. Cpos are represented by HOL pairs of sets and ordering relations and continuous functions are represented as HOL functions.

It is important to note that there is a direct correspondence between elements of cpos and elements of HOL types: the underlying set of a cpo is a subset of a HOL type. As discussed in the examples (see chapter 6–8), this allows us to exploit the rich collection of theories and tools provided with the HOL system and we become able to mix domain and set theoretic reasoning in HOL. Thereby, the often quite painful reasoning involving bottom (as in LCF) can be almost eliminated and deferred until the late stages of a proof.

The semantic conditions on cpos and continuous functions can be proved to hold of terms manually in the HOL system. However, such ad hoc proofs quickly become tedious and complicated so it is desirable to have standard ways of writing terms which guarantee that they satisfy the semantic conditions. We therefore define a number of constructions on cpos and continuous functions below, which provide syntactic notations for writing cpos and continuous functions. Proof functions have been implemented in ML to support these notations in the sense that they automatically prove the desired properties of terms which fit within the notations (see chapter 5). As one example of a construction on cpos, we can mention that the continuous functions between two cpo D_1 and D_2 constitute a cpo with the pointwise ordering, usually written $[D_1 \rightarrow D_2]$. This cpo is called the continuous function space or the cpo of continuous functions.

Some might think that the set part of a cpo set and relation pair could be avoided by representing it directly as a HOL type instead of as a subset of a HOL type. This is not the case. The need for the set component arises since we wish to define the continuous function space construction on cpos. The underlying set of continuous functions cannot

be the HOL type of functions itself, because in general this type may contain functions which are not continuous. Furthermore, the desired type cannot be defined in HOL since it is a dependent subtype of the HOL function type, dependent on free term variables corresponding to `cpos` (see section 3.11). Hence, the set component of the set relation pair approach described here is used to represent this dependent type. This complicates the formalization somewhat, since many problems from dependent type theory are introduced. E.g. terms must be ‘type checked’ manually, i.e. we must prove manually which `cpos` terms belong to. This is not handled by HOL type checking alone. Furthermore, extra conditions must be introduced to ensure that functions map subsets of types to subsets of types and behave properly outside `cpo` subsets—this last condition is called *determinedness*.

In order to ensure the functions we write are determined, a kind of dependent lambda abstraction is introduced. Thus we cannot use the built-in lambda abstraction and β -conversion. A consequence is that all function constructors become parameterized by the domains on which they work; more precisely, they become parameterized by the `cpo` variables of their domain `cpos`. In addition, arguments of functions must be type checked in order to ensure they belong to the right domains. Fortunately, the disadvantages of this can be minimized by an interface consisting of a parser and a pretty-printer, which hides the `cpo` parameters when they are not ‘necessary’, and by a proof function called the type checker which can prove in certain cases which `cpo` a term belongs to from the way in which it is constructed syntactically. The interface and type checker are described in chapter 5.

Note

The presentation below is quite detailed, listing a lot of definitions and theorems of domain theory (without proofs). After reading the overview in section 2.1 of the previous chapter, it may be enough to skim this chapter. However, the discussion of section 3.11 is recommended.

3.1 Representation

A complete partial order (`cpo`) is a pair consisting of a set and a binary relation such that a number of conditions are satisfied. In the literature a `cpo` is usually thought of as a set which has an associated ordering relation [Wi93, Pa87]. Thus, it is common to say that something is an element of a `cpo` when one should really say it is an element of the underlying set, and so on. We can provide much the same useful confusion in HOL by introducing the following constants

```
"rel : (* )cpo -> (* -> * -> bool )"
"i ns : * -> (* )cpo -> bool "
```

where we use `" : (*)cpo "` to abbreviate the type representing `cpos` `" : (* -> bool)#(* -> * -> bool)"`¹. We say `" : * "` is the underlying type of a `cpo` of type `" : (*)cpo "`. Applying `rel` to a `cpo` yields the underlying ordering relation of the `cpo` and using the infix `i ns` we can express whether some term is an element of the underlying set of a `cpo`. The constant `rel` is defined by

¹Experts will know that this is not a valid HOL type abbreviation.

| - rel = SND

and ins is defined by

| - !a A. a ins A = a IN (FST A)

We use the set library of HOL which provides set notation and constants like IN and SUBSET (see the pred_sets library [Me92]).

Assume A has type " $(*)cp0$ " and assume x and y have type " $*$ " such that "rel A x y" is well-typed. Then the following sentences are used as informal ways of saying "rel A x y": "rel A" relates x to y, x is related to y by "rel A", x is less than y with respect to "rel A".

3.2 Partial Order

Assume A is a set and relation pair. Then "rel A" is called reflexive if it relates all elements of A to themselves. Reflexivity is introduced into the HOL system by the following defining theorem

| - !A. refl A = (!x. x ins A ==> rel A x x)

The relation is called transitive if for all elements x, y and z of A such that x is related to y and y is related to z it relates x to z. Transitivity is defined as follows

| - !A.
trans A =
(!x.
x ins A ==>
(!y.
y ins A ==>
(!z.
z ins A ==>
rel A x y /\ rel A y z ==> rel A x z)))

And the relation is called antisymmetric if for all elements x and y of A if x is related to y and conversely y is related to x then they are equal. Antisymmetry is defined by

| - !A.
antisym A =
(!x.
x ins A ==>
(!y.
y ins A ==>
rel A x y /\ rel A y x ==> (x = y)))

Note that the conditions only say how the relation should behave on elements of the set. The relation can be anything on elements outside the set.

A set and relation pair is called a partial order (po) if the relation is reflexive, transitive and antisymmetric on all elements of the set.

$\vdash \neg A. \text{po } A = \text{refl } A \wedge \text{trans } A \wedge \text{antisym } A$

Assume A is a partial order and B is a subset of A . An element a of A is called an upper bound (ub) of B if all elements of B are related to a . The constant `is_ub` introduces this notion

$\vdash \neg a \ B \ A. \ a \ \text{is_ub} \ (B, A) = \ a \ \text{ins } A \wedge \text{po } A \wedge B \ \text{subset } A \wedge (\!b. \ b \ \text{IN } B \ ==> \ \text{rel } A \ b \ a)$

where `subset` is defined as follows

$\vdash \neg B \ A. \ B \ \text{subset } A = B \ \text{SUBSET} \ (\text{FST } A)$

So the definition of `subset` just says that a set is a subset of a po when it is a subset of the underlying set.

A po may have a least element, i.e. an element which is related to all other elements.

$\vdash \neg a \ A. \ a \ \text{is_least } A = a \ \text{ins } A \wedge (\!b. \ b \ \text{ins } A \ ==> \ \text{rel } A \ a \ b)$

If a least element exists then it is unique,

$\vdash \neg A. \ \text{po } A \ ==> \ (\!a. \ a \ \text{is_least } A \ ==> \ (\!a'. \ a' \ \text{is_least } A \ ==> \ (a' = a)))$

due to the antisymmetry condition for partial orders.

A least upper bound (lub) is an upper bound which is related to all other upper bounds. This notion is defined as follows in HOL

$\vdash \neg a \ B \ A. \ a \ \text{is_lub} \ (B, A) = a \ \text{is_least} \ (\{b \mid b \ \text{is_ub} \ (B, A)\}, \ \text{rel } A)$

Due to the previous uniqueness of least elements, lubs of subsets of pos are unique if they exist.

The constant `lub` gives an expression for a least upper bound if a lub exists. It is defined using the choice operator (see [GM93]) as follows

$\vdash \neg B \ A. \ \text{lub}(B, A) = (@a. \ a \ \text{is_lub} \ (B, A))$

The fact that `lub` gives an upper bound and in fact the unique least upper bound is stated by the following theorem

$\vdash \neg a \ B \ A. \ a \ \text{is_lub} \ (B, A) \ ==> \ (\text{lub}(B, A)) \ \text{is_ub} \ (B, A) \wedge \ (\text{lub}(B, A) = a)$

3.3 Complete Partial Order

A chain² in a po A is a sequence of elements of A in non-decreasing order.

$$\vdash !X A. \text{chain}(X, A) = (!n. (X\ n) \text{ ins } A) \wedge (!n. \text{rel } A(X\ n)(X(n + 1)))$$

Sequences are represented as functions from numerals to elements so in the definition above the variable X has type " $\text{num} \rightarrow *$ ". All chains are infinite but an infinite chain which is constant from a certain point can be viewed as a finite chain.

The set of elements of a chain is calculated by the constant `cset`.

$$\vdash !X. \text{cset } X = \{X\ n \mid 0 \leq n\}$$

This is used to convert a chain into a subset of a cpo.

$$\vdash !X D. \text{chain}(X, D) \implies (\text{cset } X) \text{ subset } D$$

Using `cset` we can talk about for instance the least upper bound of a chain.

A partial order D is called a complete partial order when all chains in D have a least upper bound in D .

$$\vdash !D. \text{cpo } D = \text{po } D \wedge (!X. \text{chain}(X, D) \implies (?d. d \text{ is_lub } (\text{cset } X, D)))$$

It is therefore obvious that for chains in cpos the constant `lub` always calculate the least upper bound.

$$\begin{aligned} \vdash !D. \\ \text{cpo } D \implies \\ (!X. \text{chain}(X, D) \implies (\text{lub}(\text{cset } X, D)) \text{ is_lub } (\text{cset } X, D)) \end{aligned}$$

This fact is used again and again when working with cpos.

Note that a cpo is not required to have a least element. A cpo which has one is called a pointed cpo.

$$\vdash !E. \text{pcpo } E = \text{cpo } E \wedge (?e. e \text{ is_least } E)$$

A least element is unique if it exists.

Least elements are usually called bottom elements. Using the choice operator we can give an expression for the bottom (if it exists).

$$\vdash !E. \text{bottom } E = (@e. e \text{ is_least } E)$$

The fact that `bottom` yields a least element when applied to a pointed cpo is stated by

$$\vdash !E. \text{pcpo } E \implies (!e. e \text{ ins } E \implies \text{rel } E(\text{bottom } E)e)$$

If one must prove a fact about the least upper bound of a chain it is sometimes easier instead to reason about the lub of a final segment of the chain, also called a suffix of the chain.

$$\vdash !X\ n. \text{csuffix}(X, n) = (\backslash m. X(n + m))$$

The fact that this does not change the lub is stated by the following theorem

$$\begin{aligned} \vdash !D. \\ \text{cpo } D \implies \\ (!X. \\ \text{chain}(X, D) \implies (!n. \text{lub}(\text{cset}(\text{csuffix}(X, n)), D) = \text{lub}(\text{cset } X, D))) \end{aligned}$$

²The chains we consider are often called ω -chains and the cpos are often called ω -cpo or predomains because they may lack a bottom element.

3.4 Continuous Functions

We shall consider functions between cpos and in particular continuous functions which are monotonic functions that preserve lubs of chains. A function from a cpo "D1: (*1)cpo" to a cpo "D2: (*2)cpo" is represented directly as a HOL function between the underlying types "f: *1 -> *2" such that a number of conditions are satisfied.

A HOL function is called a map between cpos, or just a function between cpos, when the image of its domain is a subset of the codomain. This notion is formalized by the constant `map` which is defined as follows

```
|- !f D1 D2.
    map f(D1, D2) = (!d. d ins D1 ==> (f d) ins D2)
```

A function is said to be determined by its action on elements of a cpo `D` when it is equal to a fixed arbitrary value `ARB` (a predefined constant in HOL) on elements outside `D`.

```
|- !f D. determined f D = (!x. ~x ins D ==> (f x = ARB))
```

The notion of determined function is needed because the `map` condition above makes the functions we consider only partially specified on the underlying HOL types. Functions may be different HOL functions but equal functions on cpos. Function equality in HOL is extensional equality which means it works on all elements of a type unlike equality of functions between cpos which works on subsets of types. This equality induces equivalence classes of HOL functions and requiring functions are determined corresponds to picking a certain fixed representative in each equivalence class (see section 3.11.2).

A function is called order preserving when the image of elements of the domain are related if the elements themselves are (in the same order, of course).

```
|- !f D1 D2.
    order_pres f(D1, D2) =
    (!d.
      d ins D1 ==>
      (!d'.
        d' ins D1 ==> rel D1 d d' ==> rel D2(f d)(f d'))))
```

A monotonic function is a determined map between cpos that preserves the ordering on elements.

```
|- !f D1 D2.
    mono f(D1, D2) =
    cpo D1 /\ cpo D2 /\
    map f(D1, D2) /\ determined f D1 /\ order_pres f(D1, D2)
```

A monotonic function is called continuous if it preserves lubs of chains.

```
|- !f D1 D2.
    cont f(D1, D2) =
    mono f(D1, D2) /\
    (!X.
      chain(X, D1) ==>
      (f(lub(cset X, D1)) = lub(cset(\n. f(X n)), D2)))
```

The continuous functions between two cpos constitute a cpo with the pointwise ordering relation on functions, see section 3.6.3 below. This is called the cpo of continuous functions, or the continuous function space.

3.5 Dependent Lambda Abstraction

In order to ensure functions are determined by their actions we write functions using a dependent lambda abstraction:

$$\vdash !D f. \text{determined}(\text{lambda } D f)D,$$

defined as follows:

$$\vdash !D f. \text{lambda } D f = (\backslash x. (x \text{ ins } D \Rightarrow f x \mid \text{ARB})).$$

This is used to define all function constructors (see the following sections) which therefore become a kind of dependent constructors that depend on the domains on which they work.

For all function constructions we present the so-called reduction theorems. In these the lambda abstractions have been reduced away and they therefore state how the constructions work in a more readable way than the definitions. Besides the theorems simplify proofs because the lambda abstractions are removed once and for all and do not have to be reduced each time a construction is applied to its arguments. The reduction theorems are obtained from the reduction theorem for the lambda abstraction:

$$\vdash (!D x. x \text{ ins } D \Rightarrow (!f. \text{lambda } D f x = f x)) \wedge \\ (!D x. \neg x \text{ ins } D \Rightarrow (!f. \text{lambda } D f x = \text{ARB})).$$

This states precisely how the lambda abstraction works. If it is applied to an element of the domain then it returns the function applied to the element, otherwise it returns an arbitrary value called `ARB`. For brevity, we shall often only list the theorems derived from the first conjunct above. These are the most used ones, since functions are usually applied to arguments in the right domains.

3.6 Constructions

In the previous sections we have introduced a number of concepts of domain theory by their semantic definitions. The notions of complete partial orders and continuous functions were defined by constants stating a number of semantic conditions on terms. Many facts of domain theory can be proved directly from these definitions. However, due to the complex nature of the semantic conditions, it is desirable to have standard ways of writing cpos and continuous functions. This is obtained by defining a number of constructions on cpos and continuous functions which are proved once and for all to yield cpos and continuous functions provided their arguments are. By appealing to these theorems about the constructors, the semantic conditions for cpos and continuous functions can be proved for terms without using the complicated definitions. Furthermore, proof functions can be implemented to prove the desired facts automatically using these

theorems, as described in chapter 5. Hence, a lot of tedious work proving terms are cpos and continuous functions can be avoided in certain cases.

The lambda abstraction is used to define all function constructors which therefore become a kind of dependent constructors that depend on the (free term variables of the) domains on which they work. The dependent constructors are parameterized by the cpo variables of their domain cpos. In order to get rid of these cpo parameters, an interface has been implemented which supports two levels of syntax, called the internal and the external level respectively (see chapter 5). At the external or interface level a version of each constructor is used which does not take cpo parameters and at the internal level the ‘real’ constructor defined below is used. The interface provides a parser and a pretty-printer to translate terms from one level to the other. The last letter of each internal constructor must be the capital letter `I` in order to allow this (just a convention). The external name of the constructor is obtained by omitting this letter.

In this section we introduce five constructions on cpos, namely discrete, product, continuous function space, lifting and sum, along with the associated constructions on continuous functions. A subsection is devoted to each cpo constructor. In the following sections (see section 3.7–3.8) additional constructions on functions are introduced which are not associated with any particular cpo construction.

3.6.1 Discrete

The discrete construction associates the discrete (identity) ordering with a set. Its definition is

$$\begin{aligned} &|- !Z. \\ &\text{di scret e } Z = \\ &Z, (\lambda d1 d2. d1 \text{ IN } Z \wedge d2 \text{ IN } Z \wedge (d1 = d2)) \end{aligned}$$

and it is easy to prove that `di scret e` yields a cpo for any set

$$|- !Z. \text{cpo}(\text{di scret e } Z)$$

The `di scret e` construction is useful for making HOL sets into cpos, e.g. the sets of numerals and booleans, and it is used extensively in the examples. Note that the underlying relation is restricted to elements of the associated set. Even without this restriction it would be a cpo constructor but then a useful theorem relating the discrete and continuous function space constructions could not be proved. This point is discussed further in section 3.6.3, where the continuous function space is introduced.

All chains in discrete cpos are constant chains.

$$|- !X Z. \text{chai n}(X, \text{di scret e } Z) ==> (!n m. X n = X m)$$

Thus, the least upper bound of a chain is simply some element of the chain, for instance the first element (they are all the same).

$$|- !X Z. \text{chai n}(X, \text{di scret e } Z) ==> (\text{lub}(\text{cset } X, \text{di scret e } Z) = X 0)$$

Unless the set argument of `di scret e` is a singleton set, the discrete construction does not yield a pointed cpo.

```

|- !x. pcpo(discrete{x})
|- !Z.
  (Z = {}) \/\ (?x y. x IN Z /\ y IN Z /\ ~(x = y)) ==>
  ~pcpo(discrete Z)

```

Of course, the bottom element of a singleton discrete cpo is the one and only element of the cpo, due to reflexivity.

There are no function constructions associated with the discrete cpo construction. However, all determined functions from a discrete cpo to some cpo are continuous.

```

|- !D.
  cpo D ==>
  (!f Z.
    map f(discrete Z, D) /\ determined f(discrete Z) ==>
    cont f(discrete Z, D))

```

This fact, and instances of this fact, are useful for extending HOL functions to continuous functions. If for example we instantiate the variable `D` with a discrete universal cpo `"discrete(UNIV: *->bool)"` and instantiate `Z` with a universal set `"UNIV: **->bool"` (consisting of all elements of the underlying HOL type `"**"`) then the antecedents all hold trivially and we obtain

```

|- !f. cont f(discrete UNIV, discrete UNIV)

```

Other similar simplifications of the previous fact can be derived.

3.6.2 Product

The product construction is defined as follows

```

|- !D1 D2.
  prod(D1, D2) =
  {(d1, d2) | d1 ins D1 /\ d2 ins D2},
  (\x y. rel D1(FST x)(FST y) /\ rel D2(SND x)(SND y))

```

and the fact that `prod` is a cpo constructor is stated by

```

|- !D1. cpo D1 ==> (!D2. cpo D2 ==> cpo(prod(D1, D2)))

```

Given two cpos `D1` and `D2` it defines a product cpo `"prod(D1, D2)"` whose underlying set is the set of all pairs of elements such that the first component is in `D1` and the second component is in `D2`. The underlying relation is defined componentwise using the relations on `D1` and `D2`. The product of, for instance, three cpos is written as `"prod(D1, prod(D2, D3))"`.

A chain in a product cpo can be split into a chain for each of the components of the product as follows

```

|- !X D1 D2.
  chain(X, prod(D1, D2)) =
  chain(fst_chain X, D1) /\ chain(snd_chain X, D2)

```

where `fst_chain` and `snd_chain` are defined by

```
|- !X. fst_chain X = (\n. FST(X n))
|- !X. snd_chain X = (\n. SND(X n))
```

Since the product relation works componentwise, the least upper bound of a chain in a product cpo is calculated by taking the pair of the lubs of each individual chain.

```
|- !D1.
  cpo D1 ==>
  (!D2.
    cpo D2 ==>
    (!X.
      chain(X, prod(D1, D2)) ==>
      (lub(cset X, prod(D1, D2)) =
        lub(cset(fst_chain X), D1), lub(cset(snd_chain X), D2))))
```

The product construction can also be used to construct pointed cpos

```
|- !D1. pcpo D1 ==> (!D2. pcpo D2 ==> pcpo(prod(D1, D2)))
```

where the bottom element is calculated componentwise just as the lub.

```
|- !D1.
  pcpo D1 ==>
  (!D2. pcpo D2 ==> (bottom(prod(D1, D2)) = bottom D1, bottom D2))
```

Four function constructions are associated with the product construction. Of course we have the two projection functions

```
|- !D1 D2. Proj 1l (D1, D2) = lambda(prod(D1, D2))FST
|- !D1 D2. Proj 2l (D1, D2) = lambda(prod(D1, D2))SND
```

which are obtained as determined versions of the built-in projections `FST` and `SND`. Besides, the tupling of elements can be extended to tupling of functions and we have a function product construction.

```
|- !D1 D2 D3.
  Tuplingl (D1, D2, D3) =
  lambda
  (prod(cf(D1, D2), cf(D1, D3)))
  (\(f, g). lambda D1(\x. (f x, g x)))
|- !D1 D2 D3 D4.
  Prodl (D1, D2, D3, D4) =
  lambda
  (prod(cf(D1, D3), cf(D2, D4)))
  (\(f, g). lambda(prod(D1, D2))(\(x, y). (f x, g y)))
```

The constant `cf` is the constructor for the continuous functions space which is introduced in section 3.6.3 below. The term "`f ins cf(D1, D2)`" is equivalent to "`cont f(D1, D2)`". The following reduction theorems state in a more readable way how the constructions work and they are also useful to simplify proofs

```

|- !D1 x.
  x ins D1 ==>
  (!D2 y.
    y ins D2 ==> (Proj 1I (D1, D2) (x, y) = x))
|- !D1 x.
  x ins D1 ==>
  (!D2 y.
    y ins D2 ==> (Proj 2I (D1, D2) (x, y) = y))
|- !D1 x.
  x ins D1 ==>
  (!D2 f.
    f ins (cf(D1, D2)) ==>
    (!D3 g.
      g ins (cf(D1, D3)) ==>
      (Tupl i ngl (D1, D2, D3) (f, g) x = f x, g x)))
|- !D1 x.
  x ins D1 ==>
  (!D2 y.
    y ins D2 ==>
    (!D3 f.
      f ins (cf(D1, D3)) ==>
      (!D4 g.
        g ins (cf(D2, D4)) ==>
        (ProdI (D1, D2, D3, D4) (f, g) (x, y) = f x, g y))))

```

If just one of the conditions (antecedents) of each theorem is false then the constructions yield the arbitrary value `ARB`. These facts have also been proved in HOL though they are not stated above.

All constructors are continuous as stated by the following theorems

```

|- !D1.
  cpo D1 ==>
  (!D2. cpo D2 ==> cont (Proj 1I (D1, D2)) (prod (D1, D2), D1))
|- !D1.
  cpo D1 ==>
  (!D2. cpo D2 ==> cont (Proj 2I (D1, D2)) (prod (D1, D2), D2))
|- !D1.
  cpo D1 ==>
  (!D2.
    cpo D2 ==>
    (!D3.
      cpo D3 ==>
      cont
      (Tupl i ngl (D1, D2, D3))
      (prod (cf (D1, D2), cf (D1, D3)), cf (D1, prod (D2, D3)))))
|- !D1.
  cpo D1 ==>
  (!D2.

```

```

cpo D2 ==>
(!D3.
  cpo D3 ==>
  (!D4.
    cpo D4 ==>
    cont
    (Prod1 (D1, D2, D3, D4))
    (prod(cf(D1, D3), cf(D2, D4)), cf(prod(D1, D2), prod(D3, D4))))))

```

As a special case it can therefore easily be derived that tupling and function product preserve continuity. For instance, if tupling is applied to the pair of functions "(f, g)" where "cont f(D1, D2)" and "cont g(D1, D3)" then since tupling is a continuous function we obtain by the map property that "cont (Tupling1 (D1, D2, D3) (f, g)) (D1, prod(D2, D3))". Hence, tupling preserves continuity.

In addition we have proved the fact that a function is continuous in a product if and only if it is determined in the product and continuous in each individual argument.

```

|- !D1.
  cpo D1 ==>
  (!D2.
    cpo D2 ==>
    (!D3.
      cpo D3 ==>
      (!f.
        cont f(prod(D1, D2), D3) =
        determined f(prod(D1, D2)) /\
        (!d2.
          d2 ins D2 ==> cont(lambda D1(\d1. f(d1, d2)))(D1, D3)) /\
          (!d1.
            d1 ins D1 ==> cont(lambda D2(\d2. f(d1, d2)))(D2, D3))))))

```

It is sometimes easier to prove the right-hand side of this theorem than to prove the left-hand side directly.

3.6.3 Continuous Function Space

The continuous function space construction is defined as follows

```

|- !D1 D2.
  cf(D1, D2) =
  {f | cont f(D1, D2)},
  (\f g.
    cont f(D1, D2) /\ cont g(D1, D2) /\
    (!d. d ins D1 ==> rel D2(f d)(g d)))

```

and `cf` is indeed a `cpo` constructor

```

|- !D1. cpo D1 ==> (!D2. cpo D2 ==> cpo(cf(D1, D2)))

```

Given two cpos $D1$ and $D2$ it constructs the cpo " $cf(D1, D2)$ " of all continuous functions from $D1$ to $D2$ such that the relation on functions is defined pointwise for elements of $D1$ only, using the underlying relation of $D2$.

Note that the cf relation only relates continuous functions just as the discrete ordering is restricted to elements of the underlying set of the discrete construction. We could omit the continuity conditions in the relation and still obtain a cpo construction. However, we would not be able to prove one important theorem, namely

$$\begin{aligned} &|- !D Z. \\ &\quad cf(D, discrete Z) = discrete\{f \mid cont f(D, discrete Z)\}, \end{aligned}$$

which states that the ordering on continuous functions from any cpo to a discrete cpo is discrete. If we did not restrict the $discrete$ and cf relations to the associated sets then the proof of this fact would fail because the following does not hold

$$"(!x. x \text{ ins } D ==> (f x = g x)) = (f = g)"$$

Adding the restrictions we must prove instead

$$\begin{aligned} &"cont f(D, discrete Z) \wedge \\ &\quad cont g(D, discrete Z) \wedge (!x. x \text{ ins } D ==> (f x = g x)) = \\ &\quad cont f(D, discrete Z) \wedge \\ &\quad cont g(D, discrete Z) \wedge (f = g)" \end{aligned}$$

which does hold. It is surprising that it is necessary to add the restrictions since the cpo conditions on set and relation pairs are concerned with elements and chains of elements of the set component only. The problem is related to the equality problem for partial functions discussed in section 3.4 where we require continuous functions are determined in order to choose fixed representatives of equivalence classes of functions induced by a function equality which works on domains (subsets of types) rather than types (see also section 3.11.2).

Least upper bounds of chains of continuous functions are calculated pointwise since the underlying relation of the continuous function space is the pointwise ordering relation.

$$\begin{aligned} &|- !D1 D2 X. \\ &\quad chain(X, cf(D1, D2)) ==> \\ &\quad (lub(cset X, cf(D1, D2)) = \\ &\quad \quad \lambda d1 (\lambda d. lub(cset(\lambda n. X n d), D2))) \end{aligned}$$

Here the lub inside the lambda abstraction is justified by the following fact

$$\begin{aligned} &|- !X D1 D2. \\ &\quad chain(X, cf(D1, D2)) = \\ &\quad (!n. (X n) \text{ ins } (cf(D1, D2))) \wedge \\ &\quad (!d. d \text{ ins } D1 ==> chain((\lambda n. X n d), D2)) \end{aligned}$$

which states that a sequence of continuous functions is a chain if and only if the functions constitute a chain at each point.

In order for the continuous function space to construct a pointed cpo, only the codomain of the functions need to be a pointed cpo.

```
|- !D. cpo D ==> (!E. pcpo E ==> pcpo(cf(D, E)))
```

The bottom element is then the constant function which always returns the bottom of the codomain.

```
|- !D.
  cpo D ==>
  (!E. pcpo E ==> (bottom(cf(D, E)) = lambda D(\x. bottom E)))
```

Two well-known constructions are associated with the continuous function space, namely application and currying.

```
|- !D1 D2.
  Appl y1 (D1, D2) = lambda(prod(cf(D1, D2), D1))(\(f, x). f x)
|- !D1 D2 D3.
  Curry1 (D1, D2, D3) =
  lambda
  (cf(prod(D1, D2), D3))
  (\g. lambda D1(\x. lambda D2(\y. g(x, y))))
```

It is not necessary to use `Appl y1` to apply a function to an argument. HOL application by juxtaposition can also be used (see chapter 5). The constant `Appl y1` provides us with function application

```
|- !D1 x.
  x ins D1 ==>
  (!D2 f.
   f ins (cf(D1, D2)) ==>
   (Appl y1 (D1, D2)(f, x) = f x))
```

and `Curry1` can be used to curry a function

```
|- !D1 x.
  x ins D1 ==>
  (!D2 y.
   y ins D2 ==>
  (!D3 g.
   g ins (cf(prod(D1, D2), D3)) ==>
   (Curry1 (D1, D2, D3)g x y = g(x, y))))
```

We have proved the following continuity theorems about these constructions

```
|- !D1.
  cpo D1 ==>
  (!D2.
   cpo D2 ==>
   cont(Appl y1 (D1, D2))(prod(cf(D1, D2), D1), D2))
|- !D1.
  cpo D1 ==>
  (!D2.
```

```

cpo D2 ==>
(!D3.
  cpo D3 ==>
  cont
  (CurryI (D1, D2, D3))
  (cf(prod(D1, D2), D3), cf(D1, cf(D2, D3)))))

```

So in particular they preserve continuity.

3.6.4 Lifting

Using the lifting construction we can add a bottom element to a cpo. The lifting construction is defined as follows

```

|- !D.
  lift D =
  {Bt} UNION {Lft d | d ins D},
  (\x y.
    (x = Bt) \/\
    (?d d'. (x = Lft d) /\ (y = Lft d') /\ rel D d d'))

```

where `Bt` and `Lft` are the constructors of an abstract datatype called `lty` which can be used to extend any type with an element `Bt`. It is defined by $(*)lty = Bt \mid Lft *$. As postulated, `lift` is indeed a cpo constructor

```

|- !D. cpo D ==> cpo(lift D)

```

and it extends the underlying set of a cpo with an element which is a bottom.

```

|- !D. cpo D ==> (bottom(lift D) = Bt)
|- !D. cpo D ==> pcpo(lift D)

```

This is ensured by the definition of the underlying relation since `Bt` is less than all other elements. On non-bottom elements of the lifted cpo "`lift D`" the underlying relation of `D` is preserved.

The least upper bound of a chain in the lifted cpo is constructed by lifting the chain of unlifted elements.

```

|- !D.
  cpo D ==>
  (!X.
    chain(X, lift D) ==>
    (!n d.
      (X n = Lft d) ==>
      d ins D ==>
      (lub(cset X, lift D) =
        Lft(lub(cset(cunlift(csuffix(X, n))), D)))))

```

where

```

|- !d. unlift(Lft d) = d
|- !X. cunlift X = (\n. unlift(X n))

```

The constant `csuffix` was defined in section 3.3. We have proved the fact that if a chain in a lifted cpo contains a lifted element then the unlifted suffix of the chain is a chain.

```

|- !D.
  cpo D ==>
  (!X.
    chain(X, lift D) ==>
    (!n. (?d. X n = Lft d) ==> chain(cunlift(csuffix(X, n)), D)))

```

This ensures that the `lub` term which is lifted in the theorem above is a least upper bound. Note that due to the way the underlying relation of the lifting construction is defined we have that if one element of a chain is a lifted element then the succeeding elements are lifted too.

Of course, this strategy only works if there is an index of the chain which is equal to a lifted element. Otherwise the chain is the constant chain which is always `Bt` and therefore the `lub` will also be `Bt`.

```

|- !D.
  cpo D ==>
  (!X.
    chain(X, lift D) ==>
    (!n. X n = Bt) ==>
    (lub(cset X, lift D) = Bt))

```

There are two function constructions associated with lifting

```

|- !D. Liftl D = lambda D Lft
|- !D1 D2.
  Extl (D1, D2) =
  lambda
  (cf(D1, D2))
  (\f.
    lambda
    (lift D1)(\x. ((x = Bt) => bottom D2 | f(unlift x))))

```

where the codomain of the function argument of `Extl` must be a pointed cpo since `bottom` otherwise returns an arbitrary value. The constant `Liftl` lifts an element of some cpo to the lifted cpo

```

|- !D x. x ins D ==> (Liftl D x = Lft x)

```

and the constant `Extl` extends a function between some cpo and a pointed cpo to a function from the lifted cpo.

```

|- !D1 D2 f.
  fins (cf(D1, D2)) ==>
  (Extl (D1, D2)f Bt = bottom D2) /\
  (!x.
    x ins D1 ==> (Extl (D1, D2)f(Lft x) = f x))

```

So, `Extl` yields a strict function when applied to a function argument.

We have proved the desired continuity theorems about these constructions as well

```
|- !D. cpo D ==> cont(Liftl D)(D, lift D)
|- !D1.
  cpo D1 ==>
  (!D2.
    pcpo D2 ==>
    cont(Extl (D1, D2))(cf(D1, D2), cf(lift D1, D2)))
```

3.6.5 Sum

Finally, we have the sum construction which is defined as follows

```
|- !D1 D2.
  sum(D1, D2) =
  {INL d | d ins D1} UNION {INR d | d ins D2},
  (\d d'.
    ((ISL d /\ ISL d') =>
     rel D1(OUTL d)(OUTL d') |
     ((ISR d /\ ISR d') => rel D2(OUTR d)(OUTR d') | F)))
```

and `sum` is a cpo constructor

```
|- !D1. cpo D1 ==> (!D2. cpo D2 ==> cpo(sum(D1, D2)))
```

The sum construction is similar to the product construction but it is based on the HOL sum type. Given two cpos `D1` and `D2` it defines the underlying set of their sum "`sum(D1, D2)`" to be the subset of the HOL sum type such that elements of `D1` are injected to the left component and elements of `D2` are injected to the right component. The underlying relation relates left components that "`rel D1`" relates and right components that "`rel D2`" relates.

Since the relation never relates elements of the left components to elements of the right components of a sum, and vice versa, a chain in the sum cpo either consists of elements of the left component only, or of elements of the right component only. A sequence is therefore a chain in the sum of two cpos if and only if it consists of elements of the left component that form a chain, or of elements of the right component that form a chain. This fact is stated by

```
|- !D1 D2 X.
  chain(X, sum(D1, D2)) =
  isl_chain X /\ chain(outl_chain X, D1) \/\
  isr_chain X /\ chain(outr_chain X, D2)
```

where

```
|- !X. isl_chain X = (!n. ISL(X n))
|- !X. isr_chain X = (!n. ISR(X n))
|- !X. outl_chain X = (\n. OUTL(X n))
|- !X. outr_chain X = (\n. OUTR(X n))
```

Of course each of the two cases excludes the other.

```
|- !X. ~(i sl_chain X /\ i sr_chain X)
```

The least upper bound of a chain is calculated by injecting the lub of the left chain to the left component and the lub of the right chain to the right component.

```
|- !D1.
  cpo D1 ==>
  (!D2.
    cpo D2 ==>
    (!X.
      chain(X, sum(D1, D2)) ==>
      (lub(cset X, sum(D1, D2)) =
        (i sl_chain X =>
          INL(lub(cset(outl_chain X), D1)) |
          INR(lub(cset(outr_chain X), D2)))))))
```

The sum construction cannot be used to construct pointed cpos. The sum cpo does not have a least element since elements of the left and right components respectively are never related.

The function constructions associated with the sum construction have been defined as follows

```
|- !D. Inl I D = lambda D INL
|- !D. Inr I D = lambda D INR
|- !D1 D2 D3.
  SumI (D1, D2, D3) =
  lambda
  (prod(cf(D1, D3), cf(D2, D3)))
  (\(f1, f2).
    lambda(sum(D1, D2))(\d. (ISL d => f1(OUTL d) | f2(OUTR d))))
```

The constants `Inl I` and `Inr I` are used to inject elements to the left and right components of a sum, respectively.

```
|- !D1 x. x ins D1 ==> (Inl I D1 x = INL x)
|- !D2 x. x ins D2 ==> (Inr I D2 x = INR x)
```

The constant `SumI` applies one of two functions to an element depending on which component of a sum the element belongs to.

```
|- !D1 D3 f.
  f ins (cf(D1, D3)) ==>
  (!D2 g.
    g ins (cf(D2, D3)) ==>
    (!x.
      x ins (sum(D1, D2)) ==>
      (SumI (D1, D2, D3)(f, g)x = (ISL x => f x | g x))))
```

The desired continuity theorems about these constructions are stated as follows

```

|- !D1.
  cpo D1 ==>
  (!D2. cpo D2 ==> cont(|nl| D1)(D1, sum(D1, D2)))
|- !D1.
  cpo D1 ==>
  (!D2. cpo D2 ==> cont(|nr| D2)(D2, sum(D1, D2)))
|- !D1.
  cpo D1 ==>
  (!D2.
    cpo D2 ==>
    (!D3.
      cpo D3 ==>
      cont
      (Suml (D1, D2, D3))
      (prod(cf(D1, D3), cf(D2, D3)), cf(sum(D1, D2), D3))))

```

We can define a continous conditional using the `Suml` constructor (see section 6.1). It would also be possible to define a more general cases construction and then derive the conditional from that. However, the conditional which we use most often in the examples is obtained from the built-in conditional using the discrete cpo of booleans.

3.7 Identity and Composition

We can define continuous function constructions for the identity function and composition of functions as follows

```

|- !D. |d| D = |lambda D |
|- !D1 D2 D3.
  Compl (D1, D2, D3) =
  |lambda(prod(cf(D2, D3), cf(D1, D2)))(\ (f, g). |lambda D1(f o g))

```

where the constants `|` and `o` are the built-in identity and composition operators, respectively. Thus, we just restrict these to determined functions using the lambda abstraction. The last letter is an `|` in order to distinguish names in the internal and external level syntax of the interface (as in the previous section).

The definitions do not give us directly that identity and composition are continuous functions. Such facts must be proved in HOL.

```

|- !D. cpo D ==> cont(|d| D)(D, D)
|- !D1.
  cpo D1 ==>
  (!D2.
    cpo D2 ==>
    (!D3.
      cpo D3 ==>
      cont(Compl (D1, D2, D3))(prod(cf(D2, D3), cf(D1, D2)), cf(D1, D3))))

```

Since composition is a continuous function it also preserves continuity of continuous functions.

```

|- !D2 D3 f.
  cont f(D2, D3) ==>
  (!D1 g.
    cont g(D1, D2) ==> cont(Comp1 (D1, D2, D3) (f, g))(D1, D3))

```

Actually, this fact is proved first and then used in the proof of continuity of composition which is described in section 3.10.

Due to the lambda abstractions it may be a bit difficult to see how the constructions actually work. We have therefore proved a number of reduction theorems which also serve to simplify proofs. Among others the following reduction theorems show how identity and composition behave when applied to their arguments.

```

|- !D x. x ins D ==> (Id1 D x = x)
|- !D2 D3 f.
  f ins (cf(D2, D3)) ==>
  (!D1 g.
    g ins (cf(D1, D2)) ==>
    (Comp1 (D1, D2, D3) (f, g) = lambda D1 (f o g)))
|- !D2 D3 f.
  f ins (cf(D2, D3)) ==>
  (!D1 g.
    g ins (cf(D1, D2)) ==>
    (!x. x ins D1 ==> (Comp1 (D1, D2, D3) (f, g)x = f(g x))))

```

If one of the arguments of the constructions does not belong to the right cpo then the constructions return the constant `ARB`. Though these facts are not stated above they have been proved in HOL.

3.8 Fixed Point Operator

The least fixed point operator is used to give semantics to recursion. It constructs a fixed point of a function by iterating the function over the bottom element and then taking the least upper bound. It only works for pointed cpos.

The constant `pow` which is defined by primitive recursion is used to name the infinite number of finite iterations of a continuous function.

```

|- (!E. pow E 0 = lambda(cf(E, E))(\f. bottom E)) /\
  (!E n. pow E (SUC n) = lambda(cf(E, E))(\f. f(pow E n f)))

```

It constructs a chain of continuous functions.

```

|- !E. pcpo E ==> chain(pow E, cf(cf(E, E), E))

```

Note that for `pow` to behave in the desired way it must be applied to a pointed cpo. Otherwise the use of `bottom` in the definition will yield an arbitrary element (of the appropriate type) since this constant is defined using the choice operator.

The fixed point operator is defined as follows

```

|- !E. Fix1 E = lub(cset(pow E), cf(cf(E, E), E))

```

where we use an `l` as the last letter in order to be able to distinguish the internal and the external version of the fixed point operator. The latter is called `Fixl`. Since `pow` yields a chain for pointed cpos and `cf` constructs a cpo when applied to cpos, `lub` does indeed yield a least upper bound for pointed cpos (by the lub condition on cpos). Some readers might be used to the following characterization of the fixed point operator

```
|- !E.
    pcpo E ==>
    (!f.
      f ins (cf(E, E)) ==> (Fixl E f = lub(cset(\n. pow E n f), E)))
```

This has been derived from the previous one using that lubs in the continuous function space are calculated pointwise (see section 3.6.3).

The function for taking the least fixed point of a continuous function is itself a continuous function.

```
|- !E. pcpo E ==> cont(Fixl E)(cf(E, E), E)
```

This follows quite easily from the fact that `pow` constructs a chain.

The fact that `Fixl` indeed is a fixed point operator, i.e. that it yields a fixed point if it is applied to a continuous function on a pointed cpo, has been proved in HOL.

```
|- !E. pcpo E ==> (!f. f ins (cf(E, E)) ==> (f(Fixl E f) = Fixl E f))
```

This theorem is called the fixed point property of `Fixl`. We have also proved that it yields a prefixed point

```
|- !E.
    pcpo E ==>
    (!f. f ins (cf(E, E)) ==> rel E(f(Fixl E f))(Fixl E f))
```

and in fact the least prefixed point.

```
|- !E.
    pcpo E ==>
    (!f.
      f ins (cf(E, E)) ==>
      (!d. d ins E ==> rel E(f d)d ==> rel E(Fixl E f)d))
```

This theorem states a principle of induction which is sometimes called Park induction.

3.9 Fixed Point Induction

There are various ways to reason about recursive definitions defined as fixed points. One way is to reason about fixed points directly by their definitions, or by Park induction. Another is to use an induction on the construction of fixed points.

Based on the definition of the fixed point operator we can derive the following theorem

```

|- !E.
  pcpo E ==>
  (!f.
    f ins (cf(E, E)) ==>
    (!P.
      inclusive(P, E) ==>
      (bottom E) IN P /\ (!x. x IN P ==> (f x) IN P) ==>
      (Fixl E f) IN P))

```

called the fixed point induction theorem. This gives a method for proving properties of fixed points by looking at their finite approximations using an inductive argument. A subset P of a cpo is called inclusive when it admits induction in the sense that all chains in P must have a least upper bound in P .

```

|- !P D.
  inclusive(P, D) =
  P subset D /\
  (!X.
    chain(X, D) /\ (!n. (X n) IN P) ==>
    (lub(cset X, D)) IN P)

```

From this the fixed point induction theorem follows easily. There is no difference between sets and predicates in HOL, both are represented as functions of type " $: * \rightarrow \text{bool}$ ". We can therefore use 'inclusive subset' and 'inclusive predicate' interchangeably.

Since inclusive predicates must be subsets of cpos it is convenient to introduce the following constant

```

|- !D P. mk_pred(D, P) = {x | x ins D /\ x IN P}

```

for writing inclusive predicates. It just restricts a predicate to a cpo by taking the intersection.

3.10 Proof of Continuity of Composition

Proofs were not included above since the more interesting ones tend to get quite long. This is both due to the fact that many concepts have quite big and complicated definitions when definitions are expanded, and due to the fact that we must check manually that terms belong to the right cpos. Furthermore, HOL proofs are long since all statements and all cases must be treated, no steps can be left out because they are obvious.

In order to illustrate how a proof of continuity is conducted in HOL we prove below that composition is a continuous function (see section 3.7). First we prove composition preserves continuity then we prove it is itself a continuous function.

3.10.1 Composition Preserves Continuity

A continuous function is a monotonic function which preserves lubs of chains. So let us first prove that composition preserves monotonicity.

Assume f is a monotonic function from D_2 to D_3 and assume g is a monotonic function from D_1 to D_2 . We must prove the following statement

"mono(Compl (D1, D2, D3) (f, g)) (D1, D3)"

Using a reduction theorem for composition we obtain

"mono(lambda D1 (f o g)) (D1, D3)"

By definition of monotonicity we must prove five properties. We must prove that $D1$ is a cpo and that $D3$ is a cpo. Such statements are finished off immediately by special theorems like

| - ! f D1 D2. mono f(D1, D2) ==> cpo D1
| - ! f D1 D2. mono f(D1, D2) ==> cpo D2

which are convenient in proofs and easy to prove, since they can be proved directly from definitions. There are similar theorems for continuous functions and the other properties of both monotonic and continuous functions, e.g.

| - ! f D1 D2. cont f(D1, D2) ==> (! d. d ins D1 ==> (f d) ins D2)
| - ! f D1 D2.
cont f(D1, D2) ==>
(! X.
chain(X, D1) ==>
(f(lub(cset X, D1)) = lub(cset(\n. f(X n)), D2)))

Informally, we refer to such theorems by saying things like "by continuity f is a map" or "since f is continuous it preserves lubs".

The third property we must prove is that composition is a map. I.e., by definition,

"(lambda D1 (f o g) d) ins D3"

assuming the variable d is an element of $D1$. By the reduction theorem for λ and the theorem

| - ! f g x. (f o g)x = f(g x)

about the built-in composition o we must prove

"(f(g d)) ins D3"

under the assumption that " d ins $D1$ ". Using g is map, since it is monotonic, we obtain " $(g d)$ ins $D2$ ". Next, we use f is a map obtaining the desired statement " $(f(g d))$ ins $D3$ ".

The fourth property we must prove is that the composite of f and g is determined.

"determined(lambda D1 (f o g)) D1"

But all lambda abstractions are determined by a fact of section 3.5 so this statement is proved immediately.

Finally, we must prove the composite preserves the ordering on $D1$.

"! d.
d ins D1 ==>
(! d'.
d' ins D1 ==>
rel D1 d d' ==> rel D3(lambda D1 (f o g) d) (lambda D1 (f o g) d'))"

Reducing the lambda abstractions and the built-in composition (by definition), we must prove

"rel D3(f(g d))(f(g d'))"

where we have assumed that the variables d and d' satisfy " $d \text{ ins } D1$ ", " $d' \text{ ins } D1$ " and " $\text{rel } D1 \ d \ d'$ ". We first use that g preserves the ordering and then that f preserves the ordering. But in order to do the latter we must provide " $(g \ d) \text{ ins } D2$ " and " $(g \ d') \text{ ins } D2$ ". These follows because g is a map.

We have now proved that composition preserves monotonicity. So let us return to the original goal of this section, namely, proving that composition preserves continuity. Assuming f is a continuous function from $D2$ to $D3$ and g is a continuous function from $D1$ to $D2$, we must prove

"cont(Compl (D1, D2, D3) (f, g)) (D1, D3)"

Using a reduction theorem for composition we obtain

"cont(lambda D1 (f o g)) (D1, D3)"

which further reduces to the following two statements, from which the previous one follows, by the definition of continuity:

"mono(lambda D1 (f o g)) (D1, D3)"

and (lubs are preserved)

"lambda D1 (f o g) (lub(cset X, D1)) =
lub(cset(\n. lambda D1 (f o g) (X n)), D3)"

for any chain X in $D1$. Since continuous functions are monotonic we prove monotonicity using that composition preserves monotonicity.

Let us consider the proof of the second statement. Since X is a chain taking the lub of X yields a least upper bound. And (least) upper bounds are elements of cpos (by definition)

| - !a B A. a is_lub (B, A) ==> a ins A

so we can assume

| - (lub(cset X, D1)) ins D1

Therefore the left-hand side of our statement reduces to

"f(g(lub(cset X, D1))) = lub(cset(\n. lambda D1 (f o g) (X n)), D3)"

Since elements of chains are elements of cpos we can reduce the lambda abstraction inside the lub on the right-hand side

"f(g(lub(cset X, D1))) = lub(cset(\n. f(g(X n))), D3)"

First we use g preserves the lub of the chain X , obtaining

"f(lub(cset(\n. g(X n)), D2)) = lub(cset(\n. f(g(X n))), D3)"

and then we use f preserves the lub of the chain " $\n. g(X n)$ " in $D2$. This concludes the proof. We should justify the sequence " $\n. g(X n)$ " really is a chain in $D2$. This is obtained from the following fact

| - !f D1 D2.

mono f(D1, D2) ==> (!X. chain(X, D1) ==> chain(\n. f(X n)), D2))

since g is continuous, and therefore monotonic.

3.10.2 Composition is Continuous

Assume the variables $D1$, $D2$ and $D3$ are cpos. Again we must prove monotonicity first. Let us just sketch the proof why composition is monotonic.

"mono(Compl (D1, D2, D3))(prod(cf(D2, D3), cf(D1, D2)), cf(D1, D3))"

Of course the domain and the codomain are cpos since they involve only variables that are cpos and constructions on cpos. Composition is a map between cpos since it preserves continuity (proved above) and it is determined because it equals a `lambda` term. Finally, composition preserves the ordering, mainly due to transitivity. But before transitivity can be applied we must prove terms belong to the right cpos and make the right set up carefully using that elements of each component of the product of cpos of continuous functions preserve the ordering.

Next, we must prove that composition preserves lubs of chains in the product of cpos of continuous functions.

"!X.
chain(X, prod(cf(D2, D3), cf(D1, D2))) ==>
(Compl (D1, D2, D3) (lub(cset X, prod(cf(D2, D3), cf(D1, D2)))) =
lub(cset(\n. Compl (D1, D2, D3) (X n)), cf(D1, D3)))"

So, let us assume the variable X is a chain in the cpo "prod(cf(D2, D3), cf(D1, D2))". We wish to prove

"Compl (D1, D2, D3) (lub(cset X, prod(cf(D2, D3), cf(D1, D2)))) =
lub(cset(\n. Compl (D1, D2, D3) (X n)), cf(D1, D3))"

Since composition is monotonic and monotonic functions preserves chains we can assume the following fact

|- chain(\n. Compl (D1, D2, D3) (X n)), cf(D1, D3))

This and the fact that lubs in the continuous function space are calculated pointwise (see section 3.6.3) can then be used to reduce the lub on the right-hand side

"Compl (D1, D2, D3) (lub(cset X, prod(cf(D2, D3), cf(D1, D2)))) =
lambda D1 (\d. lub(cset(\n. Compl (D1, D2, D3) (X n)d), D3))"

Now we can use that HOL functions are equal if they are equal for all elements of a type.

"!x.
Compl (D1, D2, D3) (lub(cset X, prod(cf(D2, D3), cf(D1, D2))))x =
lambda D1 (\d. lub(cset(\n. Compl (D1, D2, D3) (X n)d), D3))x"

If x is not an element of $D1$ then both sides of the equality reduces to `ARB`. Thus they are equal. Otherwise, assume x is in $D1$ and use the reduction theorem for composition.

"FST
(lub(cset X, prod(cf(D2, D3), cf(D1, D2))))
(SND(lub(cset X, prod(cf(D2, D3), cf(D1, D2))))x) =
lub(cset(\n. Compl (D1, D2, D3) (X n)x), D3)"

A reduction tactic (see section 5.6.2) for composition introduces `FST` and `SND` when composition is not applied to a pair of functions. Since `X` is a chain the `lub` of `X` is a `lub` and therefore an element of the product `cpo`. We can also get rid of the composition on the right-hand side

```
"FST
(lub(cset X, prod(cf(D2, D3), cf(D1, D2))))
(SND(lub(cset X, prod(cf(D2, D3), cf(D1, D2))))x) =
lub(cset(\n. FST(X n)(SND(X n)x)), D3)"
```

Next we use the fact that `lubs` of chains in products are calculated component-wise, obtaining

```
"FST
(lub(cset(fst_chain X), cf(D2, D3)), lub(cset(snd_chain X), cf(D1, D2)))
(SND
(lub(cset(fst_chain X), cf(D2, D3)), lub(cset(snd_chain X), cf(D1, D2))))
x) =
lub(cset(\n. FST(X n)(SND(X n)x)), D3)"
```

which can be simplified further to

```
"lub
(cset(fst_chain X, cf(D2, D3))(lub(cset(snd_chain X), cf(D1, D2))x) =
lub(cset(\n. FST(X n)(SND(X n)x)), D3)"
```

reducing away `FST` and `SND` on the left-hand side.

Our next steps are to calculate the first `lub` of continuous functions in the left-hand side and reduce the lambda abstraction which it is equal to by results of section 3.6.3.

```
"lub(cset(\n. fst_chain X n(lub(cset(snd_chain X), cf(D1, D2))x)), D3) =
lub(cset(\n. FST(X n)(SND(X n)x)), D3)"
```

Here, the `lub` of `snd_chain` has been moved inside the `lub` of `fst_chain`. In order to do this, we also used that chains in products are calculated component-wise so that "`fst_chain X`" and "`snd_chain X`" are chains. This gave us that the `lubs` of these chains make sense. Then the second `lub` is transformed in a similar way, yielding

```
"lub
(cset(\n. fst_chain X n(lub(cset(\n'. snd_chain X n' x), D2))), D3) =
lub(cset(\n. FST(X n)(SND(X n)x)), D3)"
```

where `x` has been moved inside the `lub`.

We are now able to exploit that `fst_chain` consists of continuous functions and therefore each of its elements preserves `lubs`.

```
"lub
(cset(\n. lub(cset(\n'. fst_chain X n(snd_chain X n' x), D3)), D3) =
lub(cset(\n. FST(X n)(SND(X n)x)), D3)"
```

Folding the definitions of `fst_chain` and `snd_chain` the right-hand side becomes

```
"lub
(lub(cset(\n. lub(cset(\n'. fst_chain X n(snd_chain X n' x)), D3)), D3) =
lub(cset(\n. fst_chain X n(snd_chain X n x)), D3)"
```

This equality can be proved by observing that it is a special case of the fact that taking the lub in first one then another direction of a matrix is the same as taking the lub at the diagonal.

```
|- !D.
  cpo D ==>
  (!M.
    matrix(M, D) ==>
    (lub(cset(\n. lub(cset(\m. M(n, m)), D)), D)) is_lub
    (cset(\n. lub(cset(\m. M(n, m)), D)), D) /\
    (lub(cset(\n. lub(cset(\m. M(n, m)), D)), D) =
    lub(cset(\n. M(n, n)), D))
```

A matrix is just a kind of two dimensional chain defined as follows

```
|- !M D.
  matrix(M, D) =
  (!n m. (M(n, m)) ins D) /\
  (!n m. rel D(M(n, m))(M(n + 1, m))) /\
  (!n m. rel D(M(n, m))(M(n, m + 1))) /\
  (!n m. rel D(M(n, m))(M(n + 1, m + 1)))
```

So in a matrix an element is related to all three elements with higher indices. From this definition we can prove the theorem

```
|- !D.
  cpo D ==>
  (!M.
    matrix(M, D) =
    (!n. chain((\m. M(n, m)), D)) /\ (!m. chain((\n. M(n, m)), D)))
```

which relates matrices and chains. Using this is often the easiest way to prove some term is a matrix.

The proof of the diagonal fact and the lemmas which states that we can use this fact in the proof above are not provided here. They would probably double the number of pages used in this section.

3.11 Discussion

Other alternatives for a formalization than the one presented above were considered. Below some of these are discussed and it is argued why we chose the approach described in this and the following chapters.

3.11.1 Sets as Types

Perhaps the simplest approach to formalizing complete partial orders in HOL is to represent the set of a partial order by a HOL type and the relation by a HOL relation on elements of that type. Thus, the set is only present implicitly, in the HOL type of the relation. This approach was used by Camilieri in [Ca90] but it has one unfortunate limitation: the cpo of continuous functions cannot be defined in HOL. Below we explain why this is so.

In this ‘set as type’ approach, the notion of partial order is introduced into HOL by the constant `po1: (* -> * -> bool) -> bool` which is a predicate on binary relations on the type `:"*`. Assuming a relation `"R: * -> * -> bool"`, the term `"po1 R"` states that `R` is reflexive, transitive and antisymmetric on all elements of the underlying type `:"*`. The actual definition of `po1` is not important here. As a refinement of `po1` we introduce a constant `"cpo1: (* -> * -> bool) -> bool"` to identify ordering relations which are complete partial orders. Again, its definition is not important.

Monotonic functions between two cpos are introduced using the constant `mono1` and continuous functions using the constant `cont1`. Given a function `"f: * -> **"` and two relations `"R1: * -> * -> bool"` and `"R2: ** -> ** -> bool"` the term `"mono1 f(R1, R2)"` is true when `R1` and `R2` are cpos and `f` preserves the ordering on elements of type `:"*`. The statement `"cont1 f(R1, R2)"` requires in addition that `f` preserves least upper bounds of chains in `R1`. A chain `X` in `R1` is a HOL sequence `"X: num -> *"` such that `"R1(X n)(X(n+1))"` for all `n`.

We wish to define some standard ways of constructing cpos. First, let us define the product construction. We would like a HOL constant `prod` which takes two cpos as arguments and returns a cpo which is their product as a result. The HOL product type can be used to construct the product of the underlying types of the cpos. The relation is defined componentwise.

$$\begin{aligned} &|- !R1 R2. \\ &\quad \text{prod}(R1, R2) = \\ &\quad (\lambda x y. R1(\text{FST } x)(\text{FST } y) \wedge R2(\text{SND } x)(\text{SND } y)) \end{aligned}$$

We can prove that given cpos `"R1: * -> * -> bool"` and `"R2: ** -> ** -> bool"` their product `"prod(R1, R2)"` is a cpo on the type `:"* # **"`.

Next, let us consider the continuous function space. We would like to introduce a constant `cf` such that the term `"cf(R1, R2)"` represents the cpo of all continuous functions from the cpo `R1` to the cpo `R2`. Thus we would have to define `"cf(R1, R2)"` to be equal to the pointwise ordering on functions, i.e. to be equal to `"(\f g. !x. R2(f x)(g x))"`. But which type should these functions have? Assuming `R1` and `R2` are cpos as above, their underlying types are `:"*` and `:"**"`, respectively. Therefore the underlying type of `"cf(R1, R2)"` would be the type of all continuous functions from `:"*` to `:"**"`. We cannot use the HOL function type `:"* -> **"` itself since it may contain non-continuous functions (not all HOL functions are continuous). The predicate needed to define the desired type can be written as the set `"{f: * -> ** | cont1 f(R1, R2)}"`. However, this is not a valid predicate for defining a type in HOL. Predicates must be closed terms and this predicate is not since it depends on free term variables `R1` and `R2`. In other words, the type we are looking for is a dependent subtype of all functions of type `:"* -> **"`. The type system of HOL is not rich enough to support such types.

Note that we did not get the problem with the product construction because we can use the product type of HOL as it is. We do not need to parametrize the type with `cpos`.

Unfortunately, there does not seem to be an easy way around the problem. The first thing one could try is to define `cf` such that it only relates continuous functions of the type `" : * -> **"`, i.e. such that

```
"cf(R1, R2) =
  (\f g.
    cont1 f(R1, R2) /\ cont1 g(R1, R2) /\ (!x. R2(f x)(g x)))"
```

But this relation is not reflexive since not all functions of type `" : * -> **"` are continuous.

An alternative solution is to try to simulate dependent subtypes in HOL by defining `pos` and `cpos` to be pairs consisting of a subtype predicate and an ordering relation. Equivalently, we can let `cpos` be relations and define the underlying sets to contain exactly those elements on which the relations are reflexive³.

We chose the set (subtype) and relation pair approach because it seemed to be a more direct and intuitively simpler representation than the non-reflexive relation approach, in particular when defining constructions on `cpos`. Unfortunately it turned out that it was not always possible to separate the definition of the set and the relation of constructions entirely (see section 3.11.3 below).

3.11.2 Sets as Subtypes

In our first attempt to formalize complete partial orders as HOL pairs consisting of a subtype predicate and an ordering relation, another unexpected problem arose due to the `cpo` construction on continuous functions. We were able to define a constant for the construction but we could not prove the constant yielded a `cpo` when its arguments were `cpos`. The reason for the problem was that we did not realize that with the introduction of sets, functions became only partially specified on HOL types, conflicting HOL equality which works on all elements of a type. Hence, we could not prove the construction was antisymmetric. Below we discuss this problem and various solutions in more details. Our solution was to require functions are determined by their actions (see section 3.4).

In this first attempt, we formalize partial orders, least upper bounds, chains and `cpos` in the same way as described in section 3.1–3.3, so we will not repeat the definitions here. The difference resulting in the failure of the approach presented here appears in the definition of monotonic and continuous functions.

So, assume the constant `mono2` formalize the notion of monotonicity in HOL. Given a function `"f: *1 -> *2"` and two terms `"D1: (*1)cpo"` and `"D2: (*2)cpo"` the term `"mono2 f(D1, D2)"` states three things: `D1` and `D2` are `cpos`, `f` maps elements of `D1` to elements of `D2`, and `f` preserves the ordering on elements of `D1`.

The constant `cont2` is intended to formalize the notion of continuous function. Given HOL terms as above the term `"cont2 f(D1, D2)"` is true when `f` is a monotonic function from `D1` to `D2` that preserves all lubs of chains in `D1`.

We might now think that the following definition would be right for introducing the construction on continuous functions

³John Harrison used this non-reflexive relation approach to define a theory of well-orderings [Ha92].

```

|- !D1 D2.
  cf(D1, D2) =
  {f | cont2 f(D1, D2)},
  (λf g. !d. d ins D1 ==> rel D2(f d)(g d))

```

where the underlying set consists of all continuous functions and the underlying relation is the pointwise ordering relation on functions. We use the restricted quantifier since it is not relevant how functions relate on the elements which are outside the cpo subset of a type. Unfortunately, the definition does not yield a cpo construction. We cannot prove "cf(D1, D2)" is a cpo given cpos D1 and D2 since the construction fails to satisfy the antisymmetry condition on cpos (and pos). Knowing two functions are related both ways on all elements of some subset of a type does not make them equal on all elements of the type. This is required by antisymmetry because HOL equality of functions is defined pointwise (extensional equality).

Let us be more precise. Given two cpos "D1: (*1)cpo" and "D2: (*2)cpo" we would like to prove "cf(D1, D2): (*1->*2)cpo" is antisymmetric (see section 3.2). So, by definition we must prove that for all continuous functions f and g from D1 to D2 such that "rel (cf(D1, D2))f g" and "rel (cf(D1, D2))g f" we have "f = g". By extensionality we must prove "f x = g x" for all "x: *1". However, we can only conclude that "f x = g x" for all x in D1, using "rel (cf(D1, D2))f g", "rel (cf(D1, D2))g f" and the fact that D2 is a cpo (antisymmetric). We do not know anything about f and g outside D1.

The problem is that continuous functions are only partially specified on HOL types. Thus they are maps from subsets of types to subsets of types, and we can only prove they are equal on subsets of types. In fact, a continuous function is a representative of an equivalence class of HOL functions under this equality on subsets of types. The problem arises because HOL equality relates total functions and therefore distinguishes functions that differ outside the subsets we are interested in. The equality on subsets does not distinguish such functions.

Basically, there are two ways around this problem. Either we can replace the equality used in the definition of antisymmetry by another, or we can define continuous function in another way (and keep the equality). Let us consider the former of the two solutions first. There are two things we must do: define the equality and then prove it is an equivalence and congruence relation on HOL terms such that we can use it for substitution. In defining the equality we must be a bit careful. For instance it will not work to use the same equality as in [JM93]

```
"Eq D = (λa b. a ins D ∧ b ins D ∧ (a = b))"
```

which is a HOL formalization of the equality in dependent type theory. This approach leaves us with exactly the same problem as above (due to the use of HOL equality). Instead the obvious choice would be

```
"Eq D = (λa b. rel D a b ∧ rel D b a)"
```

Thus, antisymmetry is trivially true of any set and relation pair, in particular the continuous function space. Furthermore we are able to prove this satisfies the following equivalence and congruence laws (for brevity we state all laws as rules)

$$\begin{array}{l}
\text{----- [cpo D, x ins D]} \\
\text{Eq D x x} \\
\\
\text{Eq D x y \quad Eq D y z} \\
\text{----- [cpo D, x ins D, y ins D, z ins D]} \\
\text{Eq D x z} \\
\\
\text{Eq D x y} \\
\text{----- [cpo D, x ins D, y ins D]} \\
\text{Eq D y x} \\
\\
\text{Eq D1 x y \quad [cpo D1, cpo D2,} \\
\text{----- x ins D1, y ins D1,} \\
\text{Eq D2 (f x) (f y) \quad f ins(cf(D1,D2))]} \\
\\
\text{Eq (cf(D1,D2)) f g \quad [cpo D1, cpo D2,} \\
\text{----- x ins D1,} \\
\text{Eq D2 (f x) (g x) \quad f ins(cf(D1,D2)), g ins(cf(D1,D2))]} \\
\\
\text{Eq D2 (t1[x]) (t2[x]) \quad [cpo D1, cpo D2,} \\
\text{----- x ins D1, t1[x] ins D2,} \\
\text{Eq (cf(D1,D2)) (\x. t1) (\x. t2) \quad t2[x] ins D2]}
\end{array}$$

The terms in square brackets [...] are the side conditions. Only two of the three congruence laws can be stated within the HOL logic itself. The third one, which is the last rule above, must be derived as a meta theorem, i.e. as an inference rule. Such a congruence property for the HOL logic as the last one cannot be stated in the HOL logic itself, without formalizing terms and substitution within the logic too. Finally, note that it would not be possible to introduce an equality satisfying these rules as a new inductive definition using Melham's tool [CM92] since the specification of Eq would not be a well-typed term. In the specification Eq would be a variable and it would be used on terms of different types (in the same term). A variable can only have one type in a term.

Based on these (meta) theorems it is quite easy to obtain a substitution rule which would be needed for instance whenever referring to the uniqueness of least upper bounds and fixed points. However, this rule would only be of very limited use because of the complex and inefficient side conditions. Checking the side conditions would be a recursive process, so subterms would be treated again and again. However, if we required in the definition of pos that "rel A a b" implies "a ins A" and "b ins A" then we could get rid of some of the side conditions. But substitution would still be much more inefficient than the built-in substitution.

It is likely that other new equalities would introduce similar inefficiency problems. For instance, we cannot deduce "Eq D t[f] t[g]" from "Eq (cf(D1,D2)) f g" in general since t could apply f and g to an element outside D1. We would therefore have to check which cpos terms belong to. Thus, we prefer to keep the HOL equality in the definition of antisymmetry.

Instead of changing equality such that HOL functions in the same equivalence class

are equal we can let continuous functions be equivalence classes (sets) of HOL functions. HOL equality will then work since it is applied to sets of functions rather than functions themselves (remember two sets are equal if they contain exactly the same elements). However, though this is perhaps a theoretically appealing approach, it is quite awkward and difficult to work with functions as equivalence classes in practice (in particular when defining the function constructions) and it provides us with no advantages over the much simpler alternative of just working with one specific fixed function in each equivalence class. Therefore this approach was chosen by requiring functions are determined by their actions.

Partial functions can also be represented by relations but this approach suffers from the same disadvantage as representing functions as equivalence classes. We believe that a useful formalization should exploit features of HOL to as wide an extent as possible.

3.11.3 Comments on the Formalization

The cpo constructors are constants which are functions that take cpos as arguments and return a set and relation pair. The set specifies the elements of the cpo and the relation specifies an ordering on the elements of the set. As mentioned in section 3.2 and 3.3 the po and cpo conditions on a set relation pair are concerned with elements and chains of elements of the set only. Outside the set the behavior of the relation does not matter. Therefore it seemed obvious that the relation of each construction should not be restricted to elements of the set part of the constructions. However, in an attempt to prove a certain equality between cpos we discovered it is necessary to restrict the underlying relations of the discrete and continuous function space constructions to elements of the underlying sets (see section 3.6.3). This restriction is not necessary in order for the constructions to yield cpo constructions.

This problem was discovered at a quite late stage of the work and we therefore chose the easy solution, namely to just add the extra conditions on the discrete and continuous function space relations. A nicer solution would probably be to let partial orders be represented by relations and define the set to be the reflexive elements. Then relations and all constructions would automatically be restricted to the underlying set. Actually, this would probably not require too many changes too, due to the fact that each time we use a relation or element of a cpo we use `ins` and `rel`. These could still be used after they had been redefined.

Chapter 4

Recursive Domains

Functional programming languages like Standard ML and Miranda allow the user to define new recursive datatypes. For instance, a type of lists which are sequences of elements of the same type can be introduced by a specification of the following form

$$\alpha \textit{ list} ::= \text{Nil} \mid \text{Cons } \alpha * \alpha \textit{ list}$$

This specifies a recursive type operator *list* with two constructor functions **Nil** and **Cons** for creating elements of the type of lists. In Standard ML the type introduced will consist of strict lists, i.e. finite lists constructed using a strict version of the constructor **Cons**. In Miranda the type will consist of lazy lists, i.e. finite and infinite lists constructed using a lazy (non-strict) version of **Cons**.

Strict and lazy datatypes denote cpos. So given a specification of the form above we can introduce a cpo of strict lists and a cpo of lazy lists, or more precisely, constructors for such cpos can be introduced which take the cpo of elements as an argument. There are a number of standard techniques for this purpose which introduce the new cpos as solutions of recursive domain (isomorphism) equations. Domain equations for strict and lazy lists can be written as follows

$$\alpha \textit{ slist} \cong \textit{unit} \oplus (\alpha \otimes \alpha \textit{ slist})$$

$$\alpha \textit{ llist} \cong \textit{void} + (\alpha \times \alpha \textit{ llist})$$

where *unit* is the cpo consisting of one element apart from the bottom element and *void* is the cpo consisting of no other element than the bottom element. The cpo constructions for sum and product used in the equation for strict lists are strict. The constructions in the other equations are non-strict.

As mentioned in the introduction, the three most prominent techniques for solving recursive domain equations are the categorical inverse limit construction, information systems and via universal domains like $P\omega$. Formalizing the inverse limit construction in HOL would require a complex encoding since it seems to need a “big” set (as large as the reals) closed under ω -sequences (to capture infinite elements). Hence, its formalization in HOL would require a substantial amount of work. Information systems and $P\omega$ would be simpler to formalize since an encoding is built into their construction. Further, they seem only to need a big set closed under pairing and finite subsets.

However, due to the various encodings, formalizing these methods would extend HOL with ‘new worlds’ different from the ‘HOL world’. For instance, the information system

of natural numbers would have nothing in common with the type of natural numbers in HOL. Hence, it would become impossible to exploit HOL types and tools directly. To achieve this indirectly, one would have to define isomorphisms between the HOL world and the other worlds (to the extent to which this is possible). These would probably be painful to work with.

Our philosophy is that formalizing domain theory in HOL should support the reuse of HOL types and proof infrastructure to as large an extent as possible. In this chapter we therefore investigate more ad hoc methods to defining recursive domains. Unfortunately, this decision limits the applicability of the formalization somewhat. While on the one hand these ad hoc methods enable us to reason about certain strict and lazy recursive datatypes denoting domains, they are not strong enough to provide solutions to more (and less) difficult recursive domain equations like $E \cong A + [E \rightarrow E]$. As a further consequence, we must accept not to be able to give denotational semantics of programming languages with recursive types, though we may be able to reason about programs and types of the languages directly by their denotations in domain theory.

In section 4.1 we show how recursive domains with finite values, i.e. cpos which correspond to datatypes whose elements are obtained by a finite number of applications of constructor functions, can be defined by exploiting the type definition package [Me89] to introduce concrete recursive datatypes in HOL. By associating an ordering relation with a subset of a datatype we can obtain a cpo. In fact, from a recursive datatype in HOL various kinds of finite-valued recursive cpos can be obtained, including discrete cpos and strict cpos.

This approach cannot be used to define recursive domains with infinite values, i.e. cpos which correspond to datatypes whose elements are obtained by a finite or infinite number of applications of constructor functions. The type definition tools cannot be used to define infinite values of a datatype; it only works for well-founded types with initial models. In section 4.2, we present an approach to construct (a few) infinite-valued recursive domains which focuses on the different kinds of elements of the domain in question. This approach has been used to construct two cpos of infinite sequences of data: a cpo of lazy sequences which contains all partial and infinite sequences and a cpo of lazy lists which contains in addition all finite sequences. This method works well for sequences but it becomes very complicated if we attempt to construct, for instance, a cpo of binary trees.

The reason why the type definition package can only be used to define finite datatypes is that each new datatype is defined as a subset of a type of labeled trees which are only finite trees. In section 4.3 we present the corresponding type of (partial and) infinite labeled trees in HOL; more precisely, we present a predicate for a set of (partial and) infinite labeled trees which could be defined as a type but we do not explicitly define a type. This type (or predicate) consists of all partial, finite and infinite labeled trees which are finitely-branching. Such trees can be represented as sets of nodes. Furthermore, with an appropriate ordering relation (a subset of) this type is a pointed cpo.

It is important to note that the only purpose of the new type (predicate) of trees is to serve as the underlying type of the new recursive cpos that we wish to introduce. The type has no value at all in the set theoretic setting for defining new HOL types, e.g. the partial elements do not make sense.

Using the labels of the labeled trees in the same way as in the type definition package, we can define new lazy recursive domains with infinite (and partial and finite) elements as a sub-cpo of the cpo of labeled trees, i.e. the underlying set is a subset of infinite labeled

trees and the ordering is inherited. In order to prove a sub-cpo really is a cpo it is enough to prove it contains lubs of chains. This approach to defining cpos corresponding to lazy datatypes is described in section 4.4. These ideas about (partial and) infinite labeled trees and lazy recursive domains have not been formalized in HOL. It requires more work to investigate what the best way of solving recursive domain equations would be.

In section 4.5 we discuss a few ways (not formalized) of extending the class of cpos which can be introduced by the other methods of the chapter. One approach is based on extending Gunter's work on well-founded arbitrarily-branching trees [Gu93] in a way which is similar to the way we extended Melham's well-founded finitely-branching trees.

Note

Definitions of the last three sections of this chapter, more precisely section 4.3–4.5, have not been formalized in HOL (as mentioned above). In order to distinguish formalized and unformalized parts, we omit the turnstile \vdash when a definition or theorem has not been formalized in HOL.

4.1 Finite-valued Recursive Domains

The standard example of a finite recursive datatype is the Standard ML (SML) datatype of strict lists. A list is simply a sequence of elements of the same type. The type of lists is specified as follows

```
datatype 'a list = Nil | Cons 'a * 'a list
```

An SML list is either `Nil` or a list of the form `Cons(x, l)` where `x` is the head and `l` is the tail of the list. All lists have finite length since `Cons` is strict. If one of its arguments is the result of an undefined (non-terminating) computation the result of applying `Cons` is also undefined (SML uses eager evaluation).

In this section we discuss how such finite-valued recursive datatypes can be represented by cpos in HOL. Our approach is based on the type definition package for constructing finite concrete recursive datatypes in HOL [Me89]. Datatypes which have been defined using this package can be used as the underlying type of cpos in various ways, yielding cpos with different properties. This way of obtaining finite-valued recursive domains in HOL is illustrated on two examples below. We discuss various kinds of cpos of finite lists in section 4.1.1 and in section 4.1.2 we look at finite binary trees.

4.1.1 Lists

A list is simply a finite sequence of elements of the same type. A type of finite lists can be defined in HOL using the following type specification

```
list ::= NIL | CONS * list
```

This defines constructors `NIL` and `CONS` such that any term of type `:(*)list` is either the empty list `NIL` or the finite-length list `"CONS x l"` with head `"x: *"` and tail `"l: (*)list"`. The type definition package provides tools to prove that the two constructors are distinct

|- !h t. ~([] = CONS h t)

and CONS is one-one (actually the theorem states a fact which is a bit stronger):

|- !h t h' t'. (CONS h t = CONS h' t') = (h = h') /\ (t = t')

The type definition package also allows us to define functions on lists by primitive recursion. For instance, the length of a list can be defined by

|- (LENGTH[] = 0) /\ (!h t. LENGTH(CONS h t) = SUC(LENGTH t))

Here the empty list NIL is written as []. Besides, the list of elements x_1, x_2, \dots, x_n is often written as "[x1; x2; ...; xn]" instead of "CONS x1(CONS x2 (... (CONS xn NIL)...))". The list type is a built-in type in HOL and this syntactic sugar is provided by the HOL parser and pretty-printer [GM93].

We can use the type of lists as the underlying type of various kinds of cpos of lists. The simplest one of these is the discrete cpo of lists, for instance

|- nlist = discrete(UNIV: (num)list->bool)

which consists of all finite lists of natural numbers. The constant UNIV specifies the universal set, i.e. the set of all elements of some type, and it is defined in the pred_sets library. So, two lists "CONS x l" and "CONS x' l'" of the cpo nlist are related iff they are equal (by definition of the discrete construction, see section 3.6.1). Since CONS is one-one this is the same as

|- !x x' l l'. (CONS x l = CONS x' l') = (x = x') /\ (l = l')

By induction we conclude that two lists in nlist are related iff they have the same length and contain the same elements (in the same order).

This construction does not always work. Imagine, for instance, we wish the elements of the lists to be in the lifted cpo of natural numbers. Then it would be too strong to require elements are the same and more reasonably to allow one list to approximate another of the same length if its elements approximates, or are related to, the elements of the other list. Note that this also matches with nlist above. The elements of the lists in nlist are natural numbers which constitute a discrete cpo. Thus, two natural numbers are related iff they are the same, as in nlist. Using the discrete construction on a (subset of a) recursive type imposes the discrete ordering not only on the type itself but also on its argument types (which is the natural numbers above).

We can define a relation for finite lists of elements of some cpo using primitive recursion as follows

|- (!D l. list_rel D [] l = (l = [])) /\
 (!D h t l.
 list_rel D (CONS h t) l =
 (?h' t'. (l = CONS h' t') /\ rel D h h' /\ list_rel D t t'))

Then two lists are related iff one of the following equations holds

|- !D. list_rel D [] []
 |- !D h h' t t'.
 list_rel D (CONS h t) (CONS h' t') = rel D h h' /\ list_rel D t t'

That is, two lists are related iff they have the same length and their elements are related, just as we desired. Using this relation we can define a constant `list` as follows

```
|- !D. list D = {l | (list_set l) subset D}, list_rel D
```

where `list_set` is used to obtain the set of elements of a list

```
|- (list_set[] = {}) ∧
    (!h t. list_set(CONS h t) = h INSERT(list_set t))
```

The constant `INSERT` extends a set with an element (if the element is not in the set already). Then `list` can be proved to be a cpo constructor just like the constructors defined in section 3.6.

```
|- !D. cpo D ==> cpo(list D)
```

That is, if `D` is a cpo then "`list D`" is also a cpo.

As mentioned above, the discrete cpo of lists of natural numbers called `nlist` is just a special case of the `list` cpo. An equivalent definition to the one above is

```
|- nlist = list Nat
```

assuming `Nat` is the discrete cpo of natural numbers. The cpo of lists of lifted natural numbers is the cpo "`list(lift Nat)`". The elements of lists can be anything, for instance, continuous functions as in "`list(cf(Nat, lift Nat))`" or lists of continuous functions as in "`list(list(cf(Nat, lift Nat)))`".

Elements of a cpo "`list D`" can be constructed using `NIL` and `CONS`. We can prove the empty list is in any `list` cpo

```
|- !D. [] ins (list D)
```

and similarly for `CONS`

```
|- !D x l. x ins D ==> l ins (list D) ==> (CONS x l) ins (list D)
```

provided its arguments belong to the right cpos.

The constructor `CONS` preserves both the list ordering and lubs of chains of lists. But since `CONS` is not determined we cannot prove it is a continuous function in

```
"cf(D, cf(list D, list D))"
```

assuming `D` is a cpo. However, it is easy to define a determined and therefore continuous version of `CONS` using the dependent lambda abstraction as follows

```
|- !D. Cons1 D = lambda D(\x. lambda(list D)(\l. CONS x l))
```

We shall often use the lambda abstraction in this way to obtain continuous (determined) constructors.

The cpos of lists constructed using `list` do not contain a bottom element. Therefore `list` does not correspond to the SML type constructor for finite lists above (recall that `CONS` applied to an undefined element yields undefined). In general, SML types correspond to pointed cpos since computations may be non-terminating.

We can add a bottom element to a `list` cpo simply by lifting the cpo. In this way we obtain a cpo constructor for semi-strict (or tail-strict) lists.

```
|- !D. sslist D = lift(list D)
```

We call such lists for ‘semi-strict’ because lists are either bottom or finite lists, never partial lists. The lists are not head-strict since the elements of a lifted list may be bottom (if they are elements of a pointed cpo). This is reflected in the following definition of a constructor for semi-strict lists

```
|- !D.
  ssConsI D =
  lambda D
  (\x.
    ExtI (list D, sslist D)
    (lambda(list D)(\l. Lft(CONS x l))))
```

It is strict in its second argument due to the use of `ExtI`, not in its first. It is continuous, proved directly from the definition of continuity.

Strict lists correspond to semi-strict lists where the element cpo is a pointed cpo and where all lists have only defined elements. Thus a cpo constructor for strict lists can be defined from the constructor for semi-strict lists as follows (as a sub-cpo)

```
|- !E.
  slist E =
  {l | l ins sslist E /\ ~(bottom E) IN (lift_case{}list_set l)},
  rel (sslist E)
```

where `lift_case` is a cases construction (or an eliminator functional) for the lifted type defined by

```
|- (!a f. lift_case a f Bt = a) /\
  (!a f x. lift_case a f (Lft x) = f x)
```

It is used to extend `list_set` to lifted lists. We can prove that provided the argument of `slist` is a pointed cpo it returns a pointed cpo.

A strict list is either the bottom list `Bt`, the empty list `Lft []` or a list of the form "`sConsI E x l`" where `x` is a non-bottom element of `E` and `l` is a strict list but not `Bt`. This means the strict list constructor `sConsI` must be strict in both its first and its second argument.

```
|- !E.
  sConsI E =
  lambda E
  (\x.
    lambda(slist E)(\l. ((x = bottom E) => Bt | ssConsI E x l)))
```

Recall that `ssConsI` is strict in its second argument (corresponding to the list argument). We can prove `sConsI` is a continuous constructor for strict lists.

```
|- !E. pcpo E ==> (sConsI E) ins (cf(E, cf(slist E, slist E)))
```

This is done using the definition of continuity (and a number of lemmas).

4.1.2 Trees

The method used above to construct cpos based on a recursive type definition does not apply to lists only. As another example we consider binary trees in this section.

We can define a type of binary labeled trees applying the type definition tools to the following type specification

$$\text{btree} ::= \text{LEAF} * \mid \text{NODE} * \text{btree} \text{ btree}$$

The type we obtain contains all finite trees where non-leaf and leaf nodes have the same type of elements. Based on this type we can obtain the cpo of binary trees of natural numbers and other discrete cpos, simply by using the discrete construction.

If we wish elements to belong to cpos with a non-trivial partial order we must define a cpo constructor for binary trees

$$\mid - !D. \text{btree } D = \{t \mid (\text{btree_set } t) \text{ subset } D\}, \text{btree_rel } D$$

where `btree_set` and `btree_rel` are defined by primitive recursion as follows

$$\begin{aligned} \mid - (!x. \text{btree_set}(\text{LEAF } x) = \{x\}) \wedge \\ (!x \text{ t1 } \text{ t2}. \\ \text{btree_set}(\text{NODE } x \text{ t1 } \text{ t2}) = \\ x \text{ INSERT}((\text{btree_set } \text{t1}) \text{ UNION } (\text{btree_set } \text{t2}))) \\ \mid - (!D \ x. \\ \text{btree_rel } D(\text{LEAF } x)\text{t} = (?x'. (t = \text{LEAF } x') \wedge \text{rel } D \ x \ x')) \wedge \\ (!D \ x \ \text{t1 } \ \text{t2}. \\ \text{btree_rel } D(\text{NODE } x \ \text{t1 } \ \text{t2})\text{t} = \\ (?x' \ \text{t1}' \ \text{t2}'. \\ (t = \text{NODE } x' \ \text{t1}' \ \text{t2}') \wedge \\ \text{rel } D \ x \ x' \wedge \text{btree_rel } D \ \text{t1 } \ \text{t1}' \wedge \text{btree_rel } D \ \text{t2 } \ \text{t2}')) \end{aligned}$$

The binary tree relation defines that two binary trees `t1` and `t2` of elements of some cpo `D` are related iff the trees have the same size and shape and each label of a node (or leaf) of `t1` is related to the label of the corresponding node (or leaf) of `t2`. If we instead defined `btree` such that the type of labels at the nodes and the leafs was not the same then `btree` would become a binary operator, taking two cpos as arguments. We would define two functions constructing a set for each different kind of labels and the relation would use the underlying relation of the two cpo arguments.

A semi-strict cpo of binary trees can be obtained by lifting the cpo of finite trees (just as above for lists)

$$\mid - !D. \text{ssbtree } D = \text{lift}(\text{btree } D)$$

and continuous constructors can be obtained from the (non-determined) constructors for finite trees

$$\begin{aligned} \mid - !D. \text{ssLeafI } D = \text{lambd}a \ D(\backslash x. \text{Lft}(\text{LEAF } x)) \\ \mid - !D. \\ \text{ssNodeI } D = \\ \text{lambd}a \ D \end{aligned}$$

```
(\x.
  Ext1 (btree D, cf(ssbtree D, ssbtree D))
  (lambda(btree D)
    (\t1.
      Ext1 (btree D, ssbtree D)
      (lambda(btree D)(\t2. Lft(NODE x t1 t2))))))
```

‘Semi-strict’ means here that (the constructors of) binary trees are only strict in their recursive components such that there are no partial trees. Strictness of `ssNode1` is ensured by the use of `Ext1`.

A cpo of binary trees with strict constructors can now be obtained by defining an appropriate subset (sub-cpo) of the semi-strict cpo of binary trees.

```
|- !E.
  sbtree E =
  {t | t ins(ssbtree E) /\ ~(bottom E) IN(lift_case{}btree_set t)},
  rel (ssbtree D)
```

The constant `lift_case` was defined in the previous section. The semi-strict constructors are used to define the strict constructors as follows

```
|- !E. sLeaf1 E = lambda E(\x. ((x = bottom E) => Bt | ssLeaf1 E x))
|- !E.
  sNode1 E =
  lambda E
  (\x.
    lambda(sbtree E)
    (\t1.
      lambda(sbtree E)
      (\t2. ((x = bottom E) => Bt | ssNode1 E x t1 t2))))
```

Note we only check for strictness on the first argument of `sNode1` since `ssNode1` is strict in its second and third arguments, not counting the cpo parameter. (And similarly for `sLeaf1`.)

4.2 Infinite Sequences

In the previous section we showed how strict datatypes can be represented as finite-valued recursive domains in HOL. In lazy functional languages like Miranda datatypes may contain infinite values, e.g. an infinite-length sequence. The approach above does not allow infinite values since the type definition package only constructs finite-size datatypes. So a datatype specification in Miranda cannot be represented by a finite-valued recursive domain.

In this section we show how we can obtain constructors for cpos of lazy sequences and lazy lists in HOL by identifying what are the finite and what are the infinite values. We then use the type definition package to obtain the finite values and represent the infinite values by functions. We first present a constructor for cpos of lazy sequences which contain partial and infinite sequences of values (see section 4.2.1). Then a constructor for cpos

of lazy lists is presented (see section 4.2.2). In addition to partial and infinite sequences, lazy lists may be finite sequences. Partial sequences, and partial values in general, are finite values in the sense that they are finitely-generated.

4.2.1 Lazy Sequences

A lazy sequence is a partial or infinite sequence of elements of the same type. There is only one constructor for sequences `Cons_seq` which takes two arguments: an element of some type and a sequence of elements of this type. There is no constructor for the empty sequence and therefore no finite sequences. Partial sequences arise by applying `Cons_seq` a finite number of times to the bottom sequence, for instance

```
Cons_seq x1(Cons_seq x2(. . . (Cons_seq xn Bt_seq). . .))
```

where x_1, \dots, x_n are elements of the same type. So `Cons_seq` is non-strict in its second argument and in fact also in its first argument.

We would like to introduce a constructor for pointed cpos of lazy sequences. The elements of a sequence should all be elements of the same cpo. Thus, we wish to define a cpo constructor for lazy sequences which takes a cpo as an argument and returns a cpo of lazy sequences of elements of that cpo. Once we have proved this constructor applied to any cpo yields a pointed cpo we can write infinite sequences by taking fixed points of continuous functions which use the sequence constructor.

There are two kinds of sequences: there are partial sequences and infinite sequences. These are represented by different HOL types. Partial sequences are finite values and can be represented by lists where the empty list is interpreted as the bottom sequence. Infinite sequences are infinite values which can be represented by HOL functions from natural numbers to any type. We can introduce a type of sequences in HOL by taking the disjoint sum of these types as follows

```
seq = pars (*)list | infs num->*
```

where `pars` abbreviates ‘partial sequence’ and `infs` abbreviates ‘infinite sequence’. Such a type is most conveniently introduced using the type definition package even though it is not recursive. The bottom sequence is defined by

```
| - Bt_seq = pars[]
```

Once we have defined the cpo of lazy sequences we can prove `Bt_seq` is the bottom w.r.t. the underlying ordering relation. The constructor for sequences is defined by cases on the sequence type

```
| - (!x l. cons x(pars l) = pars(CONS x l)) /\
    (!x f. cons x(infs f) = infs(\n. ((n = 0) => x | f(n - 1))))
```

Note that the first element of an infinite sequence is at the index 0. We can prove `cons` and `Bt_seq` are distinct (for any arguments of `cons`) and `cons` is one-one. Lazy sequences are exhaustive, i.e. all elements of the sequence type can be written using `cons` and `Bt_seq`. The constructor `Cons_seq` is defined as a determined version of `cons` below and it therefore inherits these properties.

Before we can define the cpo constructor for lazy sequences we must define the underlying relation. This is done by defining three relations which state when two partial sequences are related, when a partial sequence is related to an infinite sequence and when two infinite sequences are related. The relation for partial sequences is defined by primitive recursion on lists as follows

$$\begin{aligned} &|- (!D l. psrel D [] l = T) \wedge \\ & \quad (!D x l1 l2. \\ & \quad \quad psrel D (CONS x l1) l2 = \\ & \quad \quad (?y l2'. (l2 = CONS y l2') \wedge rel D x y \wedge psrel D l1 l2')) \end{aligned}$$

So a partial sequence approximates another when their elements are related and the latter contains at least as many elements. The relation for partial and infinite sequences is also defined by primitive recursion

$$\begin{aligned} &|- (!D f. pisrel D [] f = T) \wedge \\ & \quad (!D x l f. \\ & \quad \quad pisrel D (CONS x l) f = rel D x (f 0) \wedge pisrel D l (\lambda n. f(n + 1))) \end{aligned}$$

A partial sequence of length n is related to an infinite sequence if its elements are related to the first n elements of the infinite sequence. Two infinite sequences are related if their elements are related

$$|- !D f1 f2. isrel D f1 f2 = (!n. rel D (f1 n) (f2 n))$$

Finally, the relation for arbitrary sequences is defined by cases as follows

$$\begin{aligned} &|- !D s s'. \\ & \quad seq_rel D s s' = \\ & \quad \quad (?l l'. (s = pars l) \wedge (s' = pars l') \wedge psrel D l l') \vee \\ & \quad \quad (?l f. (s = pars l) \wedge (s' = infs f) \wedge pisrel D l f) \vee \\ & \quad \quad (?f f'. (s = infs f) \wedge (s' = infs f') \wedge isrel D f f') \end{aligned}$$

and the cpo constructor for lazy sequences can be defined by the following theorem

$$|- !D. seq D = \{s \mid (seq_set s) \text{ subset } D\}, seq_rel D$$

where `seq_set` defines the set of elements of a sequence

$$\begin{aligned} &|- (!l. seq_set(pars l) = \{x \mid \text{MEMBER } x l\}) \wedge \\ & \quad (!f. seq_set(infs f) = \{f n \mid 0 \leq n\}) \end{aligned}$$

We have proved `seq` is a constructor for pointed cpos with bottom element `Bt_seq`

$$\begin{aligned} &|- !D. cpo D ==> pcpo(seq D) \\ &|- !D. bottom(seq D) = Bt_seq \end{aligned}$$

So for any cpo D the term "seq D " is a pointed cpo.

Instead of giving the details of the proof of the fact that `seq` is a cpo constructor we provide a short overview below. It is fairly straightforward to prove that "seq D " is a partial order provided D is. Then we show that a chain of sequences can have one of three different forms:

```

|- !D X.
  po D ==>
  chain(X, seq D) ==>
  (?n L.
    (csuffix(X, n) = (\m. pars(L m))) /\
    (!m. LENGTH(L m) = LENGTH(L 0))) \/\
  (?n L.
    (csuffix(X, n) = (\m. pars(L m))) /\
    (!m. ?k. m < (LENGTH(L k)))) \/\
  (?n G. csuffix(X, n) = (\m. infs(G m)))

```

Either it consists of partial sequences which have constant length from a certain point, or it consists of partial sequences such that the length of the sequences is unbounded, or it consists of infinite sequences from a certain point. In the first case the least upper bound of the chain is a partial sequence. In the second and the third case the lub is an infinite sequence, though it is constructed differently in each of the two cases. The second case is interesting because the lub of a chain of partial sequences is not only an infinite sequence but any infinite sequence is equal to the lub of such a chain. In other words, any infinite sequence can be approximated by partial sequences.

In all three cases the lub is constructed elementwise, i.e. the first element of the lub is the lub of all first elements of the sequences, and so on. The lub constructor for the first case is defined by primitive recursion on the natural numbers

```

|- (!D L. pslub D L 0 = []) /\
  (!D L n.
    pslub D L(SUC n) =
    CONS(lub(cset(\m. HD(L m)), D))(pslub D(\m. TL(L m))n))
|- !D X n L.
  cpo D ==>
  chain(X, seq D) ==>
  (csuffix(X, n) = (\m. pars(L m))) ==>
  (!m. LENGTH(L m) = LENGTH(L 0)) ==>
  (pars(pslub D L(LENGTH(L 0)))) is_lub (cset X, seq D)

```

Here we exploit the length of the sequences is constant from a certain point and that the lub of a chain is the same as the lub of any suffix of the chain. In the second case the n 'th element of the lub (which is an infinite sequence) is defined as the lub of n 'th elements of those sequences which have at least n elements

```

|- !L n.
  min_index(L, n) =
  (@m. n < (LENGTH(L m)) /\ (!k. n < (LENGTH(L k)) ==> m <= k))
|- !D L.
  pislub D L =
  (\n. lub(cset(\m. EL n(L(m + (min_index(L, n))))), D))
|- !D X n L.
  cpo D ==>
  chain(X, seq D) ==>

```

```

(csuffix(X, n) = (\m. pars(L m))) ==>
(!m. ?k. m < (LENGTH(L k))) ==>
(infs(pislub D L)) is_lub (cset X, seq D)

```

The constant `min_index` is defined to choose the least index for which the length of a sequence (i.e. of a list) is greater than some number. Finally, in the third case the n 'th element of the lub is simply the lub of the n 'th indices of all functions

```

|- !D G. islub D G = (\n. lub(cset(\m. G m n), D))
|- !D X n G.
  cpo D ==>
  chain(X, seq D) ==>
  (csuffix(X, n) = (\m. infs(G m))) ==>
  (infs(islub D G)) is_lub (cset X, seq D)

```

The fact that "seq D" is a cpo provided D is, follows immediately from these theorems using the chain cases theorem above.

Next, we define a lazy continuous constructor for sequences simply by turning the constructor `CONSS` defined above into a determined function using the dependent lambda abstraction

```

|- !D. Cons_seq1 D = lambda D(\x. lambda(seq D)(\s. conss x s))
|- !D. cpo D ==> (Cons_seq1 D) ins (cf(D, cf(seq D, seq D)))

```

This constructor behaves as desired. It is different from the bottom sequence,

```

|- !D x s. x ins D ==> s ins (seq D) ==> ~(Cons_seq1 D x s = Bt_seq)

```

it is one-one,

```

|- !D x x' s s'.
  x ins D ==>
  x' ins D ==>
  s ins (seq D) ==>
  s' ins (seq D) ==>
  ((Cons_seq1 D x s = Cons_seq1 D x' s') = (x = x') /\ (s = s'))

```

and it makes sequences satisfy the exhaustion (or cases) axiom:

```

|- !D s.
  s ins (seq D) =
  (s = Bt_seq) \/\
  (?x s'. x ins D /\ s' ins (seq D) /\ (s = Cons_seq1 D x s'))

```

Any sequence is either the bottom sequence or it can be constructed using `Cons_seq1`. These facts are derived easily from the corresponding facts about `CONSS`. Note that just as in names of the function constructors described in section 3.6 we use an `l` as the last letter of the name of the sequence constructor. This indicates that `Cons_seq1` is the internal name of the sequence constructor which at the external, or interface, level is called `Cons_seq` (see chapter 5).

We can derive the structural induction theorem (in the sense of Paulson [Pa87]) for lazy sequences from fixed point induction.

```

|- !D P.
  cpo D ==>
  inclusive(P, seq D) ==>
  P Bt_seq ==>
  (!x s'. x ins D ==> P s' ==> P(Cons_seql D x s')) ==>
  (!s. s ins (seq D) ==> P s)

```

Partial sequences are obtained by application of `Bt_seq` and a finite number of applications of `Cons_seq`. Therefore, the conclusion holds for partial sequences by the `Bt_seq` and `Cons_seq` premises. By the inclusiveness premise it also holds for infinite sequences since inclusiveness states it holds for lubs of chains of partial sequences. The proof is conducted using the reachability theorem about lazy sequences

```

|- !D. cpo D ==> (!s. s ins (seq D) ==> (Copyl D s = s))

```

and then using fixed point induction on the copying function of this theorem which is defined as a fixed point as follows

```

|- !D.
  Copy_FUNI D =
  lambda
  (cf(seq D, seq D))
  (\f.
    Seq_whenl
    (D, seq D)
    (lambda D(\x. lambda(seq D)(\s. Cons_seql D x(f s))))))
|- !D. Copyl D = Fixl (cf(seq D, seq D))(Copy_FUNI D)

```

The eliminator functional for sequences used in the definition of the copying functional above was defined by

```

|- !D E.
  Seq_whenl (D, E) =
  lambda
  (cf(D, cf(seq D, E)))
  (\h.
    lambda(seq D)
    (\s. ((s = Bt_seq) => bottom E | h(hds s)(tls s))))

```

where the constants `hds` and `tls` were defined such that the following theorems hold

```

|- !x s. hds(conss x s) = x
|- !x s. tls(conss x s) = s

```

The following theorems show in a more readable way how the eliminator works on sequences

```

|- !D E h.
  h ins (cf(D, cf(seq D, E))) ==> (Seq_whenl (D, E)h Bt_seq = bottom E)
|- !D E h x s.

```

```

h i ns (cf(D, cf(seq D, E))) ==>
x i ns D ==>
s i ns (seq D) ==>
(Seq_whenI (D, E)h(Cons_seqI D x s) = h x s)

```

And then of course it is continuous

```

|- !D E.
  cpo D ==>
  pcpo E ==>
  (Seq_whenI (D, E)) i ns (cf(cf(D, cf(seq D, E)), cf(seq D, E)))

```

which implies that the copying functional and the copying function itself are also continuous.

To sum up, we have defined a constructor for pointed cpos of lazy sequences and a continuous constructor function for lazy sequences. Further, a structural induction theorem was derived from fixed point induction for proving inclusive properties of lazy sequences, i.e. structural induction is used to prove the property holds of all finite values. The inclusiveness ensures the property holds also of all infinite values.

4.2.2 Lazy Lists

Lazy lists are a kind of lazy sequences which in addition to partial and infinite sequences contain finite sequences. Thus lazy lists have a constructor for the empty sequence and lazy sequences do not. This makes lazy lists so similar to the lazy sequences defined in the previous section that a constructor for cpos of lazy lists can be obtained simply by editing the (50 pages each consisting of 71 lines) proof script for lazy sequences and adding a new case for finite lists here and there (the resulting proof script was 70 pages long). The editing was only non-trivial in a few places.

In fact, editing was so simple not only due to the fact that lazy lists and lazy sequences are so similar. It was also because finite sequences can be represented in the same way as partial sequences, by finite HOL lists. Thus facts proved about lazy lists have similar proofs for partial and finite sequences (in most cases).

The finite values of the type of lazy lists, which we shall use as the underlying type of the cpo constructor, come from both the partial and the finite sequences. These are therefore represented by a HOL datatype, namely lists. Infinite lists are infinite values and these are represented by functions. The type of lazy lists is defined using the type definition package as follows

```

llist = par (*)list | fin (*)list | inf num->*

```

where `par`, `fin` and `inf` correspond to the three different kinds of lazy lists. The bottom lazy list is defined by

```

|- Bt_llist = par[]

```

and the constructors for empty and non-empty lazy lists are defined by

```

|- Nil_llist = fin[]
|- (!x l. ll_cons x(par l) = par(CONS x l)) /\
  (!x l. ll_cons x(fin l) = fin(CONS x l)) /\
  (!x f. ll_cons x(inf f) = inf(\n. ((n = 0) => x | f(n - 1))))

```

These definitions are similar to the definitions of the corresponding constants for lazy sequences. A determined version of `ll_cons` is introduced below.

We now proceed as for lazy sequences by defining a number of different ordering relations

```

|- (!D l. prel D[]l = T) /\
  (!D x l1 l2.
   prel D(CONS x l1)l2 =
    (?y l2'. (l2 = CONS y l2') /\ rel D x y /\ prel D l1 l2'))
|- (!D l. frel D[]l = (l = [])) /\
  (!D x l1 l2.
   frel D(CONS x l1)l2 =
    (?y l2'. (l2 = CONS y l2') /\ rel D x y /\ frel D l1 l2'))
|- (!D f. pirel D[]f = T) /\
  (!D x l f.
   pirel D(CONS x l)f = rel D x(f 0) /\ pirel D l(\n. f(n + 1)))
|- !D f1 f2. irel D f1 f2 = (!n. rel D(f1 n)(f2 n))

```

Here we define four relations which is perhaps one less than one might think. The reason is that `prel` can be used to relate a partial sequence to both a partial and a finite sequence since they are represented in the same way. The constant `frel` is used to relate finite lists and the constant `pirel` is used to relate partial and infinite lists. Finally, the constant `irel` is used to relate infinite lists. The relation for lazy lists is defined by cases as follows

```

|- !D ll ll'.
  llist_rel D ll ll' =
  (?l l'.
   (ll = par l) /\
    ((ll' = par l') \/ (ll' = fin l')) /\
    prel D l l') \/
  (?l f. (ll = par l) /\ (ll' = inf f) /\ pirel D l f) \/
  (?l l'. (ll = fin l) /\ (ll' = fin l') /\ frel D l l') \/
  (?f f'. (ll = inf f) /\ (ll' = inf f') /\ irel D f f')

```

Note that finite lists must have the same length in order to be related. Otherwise, this relation for lazy lists behaves just like the relation for lazy sequences. Defining the set of elements of a lazy list by the following theorem

```

|- (!l. llist_set(par l) = {x | MEMBER x l}) /\
  (!l. llist_set(fin l) = {x | MEMBER x l}) /\
  (!f. llist_set(inf f) = {f n | 0 <= n})

```

we can introduce a constructor for pointed cpos of lazy lists as follows

|- !D. llist D = {ll | (llist_set ll) subset D}, llist_rel D

If D is a cpo then "llist D" is a cpo with bottom Bt_llist.

|- !D. cpo D ==> pcpo(llist D)
|- !D. bottom(llist D) = Bt_llist

Thus it is a pointed cpo. The proof of this fact is similar to the proof of the cpo fact about lazy sequences. The chain cases theorem has an extra case here corresponding to the chain consisting of only finite lists from a certain point.

A continuous constructor for non-empty lazy lists is defined using the lazy list constructor ll_cons above and the dependent lambda abstraction

|- !D.
 Cons_llistl D = lambda D(\x. lambda(llist D)(\ll. ll_cons x ll))
|- !D. cpo D ==> (Cons_llistl D) ins (cf(D, cf(llist D, llist D)))

Thus it is distinct from both the bottom list and the empty list (which are also distinct) and it is one-one. Besides it makes lazy lists satisfy the exhaustion axiom, stated as follows

|- !D s.
 s ins (llist D) =
 (s = Bt_llist) \/
 (s = Nil_llist) \/
 (?x s'. x ins D /\ s' ins (llist D) /\ (s = Cons_llistl D x s'))

An eliminator functional similar to the eliminator for lazy sequences has been defined and the following theorems stating how it works have been proved

|- !D E z h.
 z ins E ==>
 h ins (cf(D, cf(llist D, E))) ==>
 (LList_whenl (D, E)z h Bt_llist = bottom E)
|- !D E z h.
 z ins E ==>
 h ins (cf(D, cf(llist D, E))) ==>
 (LList_whenl (D, E)z h Nil_llist = z)
|- !D E z h x s.
 z ins E ==>
 h ins (cf(D, cf(llist D, E))) ==>
 x ins D ==>
 s ins (llist D) ==>
 (LList_whenl (D, E)z h(Cons_llistl D x s) = h x s)

Of course the eliminator is continuous too.

|- !D E.
 cpo D ==>
 pcpo E ==>
 (LList_whenl (D, E)) ins
 (cf(E, cf(cf(D, cf(llist D, E)), cf(llist D, E))))

The structural induction theorem for lazy lists has also been proved.

```
|- !D P.
  cpo D ==>
  inclusive(P, llist D) ==>
  P Bt_llist ==>
  P Nil_llist ==>
  (!x s'. x ins D ==> P s' ==> P(Cons_llistl D x s')) ==>
  (!s. s ins (llist D) ==> P s)
```

In the same way as for lazy sequences it has been proved using a reachability result about lazy lists

```
|- !D. cpo D ==> (!s. s ins (llist D) ==> (Copy_llistl D s = s))
```

stating all lazy lists can be reached by a recursive copying function defined using the fixed point operator.

4.3 Infinite Labeled Trees

Unfortunately, this way of taking the union of the various kinds of elements of lazy datatypes works well only for fairly simple recursive types as lazy sequences "`(*)seq`". For instance, a type supporting cpos of lazy trees should allow one tree to contain both partial, finite and infinite subtrees at the same time. So it is more difficult to classify the different kind of elements.

In the following sections, we describe a more uniform approach than the previous one, defining ‘type predicates’ for lazy datatypes as subsets of a type predicate for infinite labeled trees. Note that we do not actually define types for infinite trees or new datatypes. Instead we work directly with type predicates (close terms) which could be used to define the types. Also note the difference between a recursive datatype (predicate) and the corresponding recursive domain, the underlying set of which is based on a subset of this type (predicate). The recursive datatype itself has no value in itself, since its partial elements can only be interpreted in a domain theoretic setting. Therefore the infinite trees have no value for defining recursive types in pure HOL.

Tom Melham implemented the type definition package [Me89] for defining certain concrete recursive datatypes in HOL. The type definition tool defines the user-specified datatype as a type in HOL by representing it as a subset of a type of labeled trees. The result is an axiomatization of the new type as an initiality theorem stating how to define primitive recursive functions over the type.

Labeled trees are finite trees, i.e. they are finitely-branching and all branches are finite. Therefore, the type of labeled trees cannot be used to represent any infinite values. For instance, a type of infinite sequences cannot be defined using the type definition package. If we extend the type of labeled trees with infinite trees it becomes possible to use labeled trees to define infinite-valued datatypes as well as finite-valued datatypes. However, we lose the initiality (well-foundedness) property.

So, it is not possible to define HOL functions over infinite trees by a kind of primitive recursion since the recursion might not terminate (remember all HOL functions must be total). This is where domain theory comes in useful. If we can prove some subset of a

type of infinite labeled trees equipped with a certain ordering is a pointed cpo then we can define recursive functions on trees using the fixed point operator.

In order to be a pointed cpo, a domain, and hence the underlying type, of infinite trees should contain partial trees as well finite and infinite trees. Then an infinite tree can be computed by taking the fixed point of a continuous function, i.e. by taking the lub of its finite approximations which are the partial trees. Finite trees correspond to Melham's labeled trees and partial trees are a kind of finite trees where one or more leaf nodes are partial nodes. An infinite tree is a (finitely-branching) tree which contains an infinite branch.

4.3.1 A Type of Infinite Trees

Melham first constructs a type of unlabeled trees and then defines a type of labeled trees as a subset of pairs of unlabeled trees and lists. The list associated with an unlabeled tree contains the labels on the nodes, corresponding to a depth-first (preorder) traversal of the tree.

Below we define labeled trees directly without constructing unlabeled trees first and we do not actually define a type of labeled trees. We find it is easier to have direct access to the representing type since our trees are non-wellfounded. For the same reason we do not define unlabeled trees first. If we did we would have to use infinite lists to represent all labels (our trees may be infinite). Then a depth-first traversal of trees is not possible since a tree may contain an infinite branch. Instead we could use a breath-first traversal which is more complicated.

Infinite (finitely-branching) labeled trees containing partial, finite and infinite trees can be represented as sets of nodes (similar yet different approaches to represent finite and infinite trees in set theoretic settings are presented in [Gu93, Pa93]). A node is a pair consisting of a path and a label. A path is a list of numbers indicating which branches lead to the node, starting at the root of the tree. A label can be a flag, stating the node is a partial node, or some value. Below we use the following syntactic sugar

```
(* )node == (num)list # (* )label
```

The label type is defined as follows

```
label = NOLBL | LBL *
```

using the type definition package. So the label type is really just the disjoint sum type of ":one" and ":*". It is isomorphic to the lifting of a type (defined in section 3.6.4) so this could have been used instead. We do not do this because we want to distinguish the situations where we lift and where we label a type. Besides the ordering relations of the cpo constructions which are based on the two types are different since NOLBL is not treated as a bottom (like Bt is):

```
label D =
{NOLBL} UNION {LBL a | a ins D},
\| I'.
(I = NOLBL) /\ (I' = NOLBL) \/
(?a a'. (I = LBL a) /\ (I' = LBL a') /\ rel D a a')
```

The ordering on the label construction on cpos reflects that the label type is just a kind of tagging. We will use `Label l` as the continuous version of `LBL` defined by using the dependent lambda abstraction to obtain a determined function (in the same way as we have seen several times in this chapter). It is a continuous function from any cpo to the cpo of labels of that cpo and it is parameterized by that cpo (like other function constructors). It is used together with a continuous constructor for infinite trees to define continuous constructors for new recursive domains.

If a node has the form "`(p, LBL v)`" for a path `p` and some value `v` we say the node is a labeled node. If a node has the form "`(p, NOLBL)`" for a path `p` we say the node is an unlabeled node, or a partial node.

So, our representing type of infinite labeled trees is the type of sets of nodes "`(*)node -> bool`". This type is very large. It contains all partial, finite and infinite labeled trees, both finitely- and infinitely-branching, and it contains junk elements which cannot be interpreted as trees at all. We construct a set of trees which contain the various kinds of trees just mentioned (excluding the junk elements), by defining a predicate `Is_tree` on the type above. So, the type of `Is_tree` is "`((*)node -> bool) -> bool`" and it can be defined by

```
Is_tree tr =
  (?l. ([], l) IN tr) /\
  (!p p' l. ~(p' = []) /\ (p++p', l) IN tr ==> ?v. (p, LBL v) IN tr) /\
  (!p l l'. (p, l) IN tr /\ (p, l') IN tr ==> (l = l'))
```

where we use `++` for appending lists (instead of the constant `APPEND`). The first condition says that a tree must have a root node. The second condition says that all paths must be prefix closed, i.e. for any node (different from the root) which is in the tree there is a labeled node for each branch on the path from the root to that node. The last condition ensures labels are unique so there are exactly one root node and exactly one node, and this is a labeled node, for each branch on a path to any node.

Since we wish to find a way of constructing infinite versions of Melham's concrete recursive datatypes, we do not need the trees which are infinitely-branching. Let us therefore define a type of infinite finitely-branching labeled trees as the following subset of the type of infinite labeled trees defined above

```
Is_infinite_tree tr =
  Is_tree tr /\
  (!p l.
    (p, l) IN tr ==> ?k. {n | ?l'. (SNOC n p, l') IN tr} = {0, ..., k}) /\
  (?m. !n p l. (CONS n p, l) IN tr ==> n <= m)
```

The constant `SNOC` does the opposite of `CONS`, it adds an element at the tail of a list. Note that in addition to requiring that each node has finitely many subtrees we require the subtrees are enumerated by counting from zero up to some number. The last condition says that there is an upper limit on the number of subtrees of all nodes of a tree. If we did not have this condition then trees might be of infinite width even though they are finitely-branching. It is necessary in order for infinite trees to be a cpo (lubs of chains of trees must be trees).

The predicate `Is_infinite_tree` can be used to define a type of infinite trees since it specifies a non-empty subset of an existing type, namely the type "`((*)node -> bool`)

-> bool ". However, we shall not do so here since it is convenient to use the representing type when we define the constructors below. Defining a type is just naming some subset of an existing type. This is convenient in some situations, not in others.

4.3.2 A Pointed Cpo of Infinite Trees

It is essential that a subset of the 'type' of infinite trees with some ordering constitutes a pointed cpo. This allows us to write recursive functions that compute infinite trees, using the fixed point operator. Moreover, we will prove that the new recursive datatypes we define as subsets of the type of infinite trees become pointed cpos with the same ordering relation and the same bottom; they are so-called sub-cpos. Thus, we can also define recursive functions on new recursive domains using the fixed point operator.

The ordering should support the intuition that a partial node can approximate any tree. So if some path ends at a partial node of a tree then we obtain a more defined tree by replacing that node with an arbitrary tree. A node which is not a partial node, i.e. it is a labeled node, cannot be made more defined as a tree. The definition of a partial ordering `trrel` for infinite trees can therefore be defined as follows

```
trrel tr tr' =
  (!p l.
    (p,l) IN tr ==>
    (p,l) IN tr' ∨ ((l = NOLBL) ∧ (?v. (p,LBL v) IN tr'))) ∧
  (!p l.
    (p,l) IN tr' ∧ ~(p,l) IN tr ==>
    (?p' p'' . (p = p' ++p'') ∧ (p',NOLBL) IN tr))
```

This definition says that a tree `tr` approximates another tree `tr'` (recall trees are non-empty sets) iff for all nodes of `tr` either they are also in `tr'` or they are partial nodes that have been made more defined by some labeled node in `tr'`. (The second condition above gives the 'only if'.)

We can prove this ordering is a partial order and that there exists a least upper bound for all chains of infinite trees, calculated by

```
trrel_union X = {node | ?n. !m. node IN X(n+m)}
```

Thus, the lub of a chain of infinite trees is the set of nodes which are in all trees of the chain from a certain point. Clearly, this is a tree, i.e. the set constructed satisfies the predicate `is_infinite_tree`.

The relation `trrel` reflects how one tree `tr` can approximate another tree `tr'` if `tr` has a partial node. The tree `tr'` must contain the same nodes as `tr` except for those nodes that appear in the subtree of `tr'` replacing the partial node in `tr`. In particular, the labels of the nodes in `tr'` must be the same. This corresponds to requiring that the cpo of labels is discrete, which is unfortunate of course if we wish the labels to be elements of a cpo which is not. Then a node `n1` of `tr` should be able to approximate a node `n2` of `tr'` if the label of `n1` approximates the label of `n2`. That is, we should extend the relation `trrel` above such that it allows a tree to approximate another if a label of a node of the other is more defined than the corresponding node of the first tree. This extension of `trrel` is called `itree_rel` and defined as follows

```

i tree_rel D tr tr' =
(!p. (p,NOLBL) IN tr ==> (?l'. (p,l') IN tr')) /\
(!p v. (p,LBL v) IN tr ==> (?v'. rel D v v' /\ (p,LBL v') IN tr')) /\
(!p l.
  (p,l) IN tr' /\ ~(p,l) IN tr ==>
  (?p' p'' .
    (p = p' ++p'') /\ ((p',NOLBL) IN tr \/ (?v. (p,LBL v) IN tr))))

```

Assuming variables tr and tr' as above, note that if a node of tr' is not in tr then this can be due to two situations. Either it is a node of a subtree of tr' which is approximated by an unlabeled node of tr or the label of the node is approximated by the label of the corresponding node in tr .

Now, let us define a cpo of infinite trees labeled by elements of some cpo. In other words, we will define a cpo constructor for infinite trees just as we defined constructors for continuous functions, products and lazy lists. We call this construction `i tree` and define it as follows

```

i tree D =
{tr | !s_infinite_tree tr /\ (label_set tr) subset D},
i tree_rel

```

where the set of labels of a tree is defined by

```
label_set tr = {v | ?p. (p,LBL v) IN tr}
```

The least upper bound of the cpo of infinite trees is now calculated:

```

i tree_lub D X =
{(p,NOLBL) | !m. (p,NOLBL) IN X m} UNION
{(p,LBL(lub(cset vchain, D))) |
  ?n. (?v. (p,LBL v) IN X n) /\
  (vchain = (\m. @v. (p,LBL v) IN (csuffix(X,n)m)))}

```

First we take the partial nodes that are in all trees of the chain. Then we add the least upper bounds of all labeled nodes in all trees of the chain from a certain point. Note that if a node is labeled in the n 'th element of a chain of trees then it is labeled in all trees of the n 'th suffix of the chain, but the label may be increasing with respect to the ordering on the cpo representing labels.

We can prove that `i tree` is a cpo constructor and indeed yields a pointed cpo.

```

!D. cpo D ==> pcpo(i tree D)
!D. bottom(i tree D) = {([],NOLBL)}

```

Below we call the bottom element of the cpo of infinite trees `Bt_node`.

```
Bt_node = {([],NOLBL)}
```

Note that `Bt_node` need not be parameterized by a cpo. It works for all cpos of labels.

node a [t₁; t₂; t₃]:

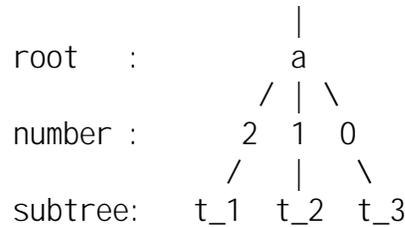


Figure 4.1: The numbering of subtrees.

4.3.3 Constructors for Infinite Trees

We already defined one ‘constructor’ for infinite trees above, namely the bottom tree `Bt_node`. This is an element of the type of infinite trees and of any cpo of infinite trees as well. Below we define another constructor for infinite trees which can be used to construct both leaf and non-leaf nodes, but not partial nodes. From this constructor we obtain a continuous constructor for the cpo of infinite trees.

A node of an infinite tree has a finite number of subtrees, and these have numbers counting from zero. We introduce the tree constructor `node` which takes a label for the root node and a list of subtrees as arguments.

$$\begin{aligned}
 \text{node } a \text{ NIL} &= \{([], \text{LBL } a)\} \\
 \text{node } a \text{ (CONS } tr \ l) &= \\
 &\{(\text{CONS}(\text{LENGTH } l)p, x) \mid (p, x) \text{ IN } tr\} \cup \{\text{node } a \ l\}
 \end{aligned}$$

The branches (paths) out of a node constructed using `node` are numbered from zero up to the number of subtrees minus one (see figure 4.1). It does not matter that this is actually in ‘reverse’ order.

A node constructed using `node` and containing no subtrees is a labeled leaf node. A partial node cannot be constructed using `node` so `node` and `Bt_node` always yield different results.

$$!a \ trl. \sim(\text{Bt_node} = \text{node } a \ trl)$$

We can also prove that `node` is one-one (injective).

$$\begin{aligned}
 &!a \ a' \ trl \ trl'. \\
 &(\text{node } a \ trl = \text{node } a' \ trl') \implies (a = a') \wedge (trl = trl')
 \end{aligned}$$

And finally, infinite trees satisfy the exhaustion axiom, i.e. any infinite tree is equal to `Bt_node` or `node` for proper arguments to `node`.

$$\begin{aligned}
 &!tr. \\
 &\text{Is_infinite_tree } tr = \\
 &(\text{tr} = \text{Bt_node}) \vee \\
 &(\exists a \ trl. \text{ EVERY Is_infinite_tree } trl \wedge (\text{tr} = \text{node } a \ trl))
 \end{aligned}$$

where `EVERY` is used to state that the predicate `Is_infinite_tree` holds for all elements of a list of subtrees.

Assuming we have a cpo of labels we know infinite tree with labels in that cpo is also a cpo. We would like a continuous constructor `Node1` for trees behaving like the `node` constructor above. So, `Node1` should be in the following continuous function space

$$\text{Node1 } D \text{ ins } cf(D, cf(\text{list}(i\text{tree } D), i\text{tree } D))$$

and it can be defined as the lambda abstracted version of `node`.

$$\text{Node1 } D = \lambda D (\lambda a. \lambda (\text{list}(i\text{tree } D)) (\lambda \text{trl}. \text{node } a \text{ trl}))$$

Note that `Node1` is parameterized by the domain `D`. The list cpo is simply a subset of the HOL list type with the ordering that relates only lists of equal length whose elements are related (see section 4.1).

4.4 Infinite-valued Recursive Domains

Many recursive datatypes can be defined as subsets of the type of infinite trees (as mentioned earlier, we do not actually define the types, merely state the type predicate). Below we describe how lazy (infinite) versions of the concrete recursive datatypes accepted by Melham's type definition package can be defined in HOL. Note however that these types are not really useful as HOL types in themselves, e.g. the 'partial' elements of the types can only be interpreted in a domain theoretic setting. The only purpose of the types is to serve as the underlying types of the recursive domains that we wish to define. Once a new datatype (predicate) has been defined, then the corresponding cpo constructor can be defined too. This is done by defining the cpo as a sub-cpo (see page 100) of the cpo of infinite trees (the type is a subset of the type predicate for infinite trees).

The motivation for defining the type (predicate) and the cpo of infinite labeled trees was that we can use these to define new recursive lazy (infinite) datatypes and cpos in HOL. We can define a type to represent lazy lists as a subset of the type of infinite trees. The cpo of lazy lists is defined as a sub-cpo of the cpo of infinite trees, based on the type of lazy lists; using type predicates rather than real types ensures that lazy lists has the same HOL type as infinite trees. Any lazy version of a type specification accepted by the type definition package can be obtained in this way. These types can be described informally by a type specification of the following form (see Melham [Me89])

$$\text{rty} ::= C_1 \text{ ty}_{1,1} \dots \text{ ty}_{1,k_1} \mid \dots \mid C_m \text{ ty}_{m,1} \dots \text{ ty}_{m,k_m}$$

where each $\text{ty}_{i,j}$ is either an existing logical type (not containing `rty`) or is the type expression `rty` itself. If the right-hand side contains n free type variables then `rty` is an n -ary type operator.

We cannot use exactly the same approach as in [Me89] to represent the new recursive datatypes since our trees may be (partial and) infinite and the finiteness (well-foundedness) of trees is exploited there. Below we first give an example showing how we can introduce a type (predicate) and a cpo of lazy lists. Then we describe how the method works in general for concrete recursive datatypes.

In defining subsets of infinite trees which correspond to infinite-valued datatypes we shall use the notion of subtree, defined by

subtree tr tr' = ?p. !p' l. (p', l) IN tr ==> (p++p', l) IN tr'

So a tree tr is a subtree of a tree tr' if the nodes of tr are present in tr' at the same position (as in tr) relative to the root of tr (in tr').

4.4.1 Example: Lazy Lists

Lazy lists can be described by the following type specification

llist = nil | cons * llist

So apart from the bottom list, lazy lists have two constructors: one for the empty list and one for adding an element to a list.

We can introduce a type predicate for lazy lists as a subset of the type predicate for infinite trees. More precisely, the representing 'type' of lazy lists is

Is_infinite_tree: ((one + *)node -> bool) -> bool

where the labels have type ": one + *". The label type is constructed from the type specification by taking the first component of the sum to be the type ": one" because nil takes no arguments and taking the second component to be ": *" because the second constructor takes elements of this existing type as arguments.

The bottom lazy list is simply the bottom node of infinite trees, i.e. Bt_node. We can define predicates to identify which trees correspond to the empty list nil and the list constructor cons as follows

Is_nil v trl = ?a. (v = LBL(INL a)) /\ (LENGTH trl = 0)
 Is_cons v trl = ?a. (v = LBL(INR a)) /\ (LENGTH trl = 1)

A node of the representing type of trees can be interpreted as the empty list if it has no subtrees (nil takes no lists as arguments) and its label is in the left component of the sum type. A node can be interpreted as a non-empty list if it has precisely one subtree (since cons takes one lazy list as an argument) and its label is in the right component of the sum type (corresponding to the first element of the resulting list).

The constructor predicates are used to define precisely which subset of infinite trees corresponds to lazy list in the following way

Is_llist ll =
 Is_infinite_tree ll /\
 (ll = Bt_node) \/
 (!v trl. (node v trl) subtree ll ==> Is_nil v trl \/\ Is_cons v trl)

In order to interpret this definition recall that infinite trees satisfy the exhaustion axiom (constructors are onto). That is, either a tree is equal to the bottom node or there exist a label and a list of subtrees, each of which is an infinite tree, such that it is equal to the node with this label and list of subtrees. The second disjunct above states that for any subtree of a tree the root node must be the empty list or a non-empty list. Thus all nodes of a tree representing a lazy list is either a partial node, an empty list node with no subtrees or a non-empty list node with precisely one subtree (the root of which is again a node of the tree and so on).

Since we identify rather than actually define types in HOL we avoid type abstraction and representation functions in terms. If we wish we can introduce a type of lazy lists without problems. It is less practical to define a type of infinite trees because we use the set representation again and again.

We can now define the cpo of lazy lists of elements of some cpo as a sub-cpo of a certain cpo of infinite trees

```
l l i s t D =
  {ll | l s _ l l i s t ll /\ ll i n s (i t r e e (s u m (O n e , D)))},
  i t r e e _ r e l (s u m (O n e , D))
```

where `One` is the discrete universal cpo "di s c r e t e U N I V" based on the type ": o n e". Note that the infinite trees used here are labeled by elements of a sum cpo the underlying type of which corresponds to the label type used above to identify the type of lazy lists.

The above definition of a type (and domain) of lazy lists guarantees that all nodes of a non-bottom tree representing a non-bottom lazy list is a `ni l` node or a `co n s` node. These constructors are defined by

```
ni l = node (L B L (I N L o n e)) []
co n s h t = node (L B L (I N R h)) [t]
```

The constant `co n s` is not determined but a determined and hence continuous version of `co n s` is obtained easily as follows

```
Co n s l D =
  l a m b d a D
  (\a.
    l a m b d a (l l i s t D)
    (\ll . N o d e l (s u m (O n e , D)) (L a b e l l (s u m (O n e , D)) (l n r l (O n e , D) a)) [ll]))
```

where we have exploited that `N o d e l` has been proved once and for all to be a continuous (determined) version of `no d e` used to define `co n s` (this saves proof commitments). The constant `L a b e l l` is a continuous version of the constant `L B L` (see page 99).

We can prove the structural induction theorem for inclusive properties of lazy lists in exactly the same way as for lazy lists in section 4.4.1 using the copying function.

4.4.2 The Method in General

We will now describe how the denotations of lazy datatypes in general can be defined using the type (predicate) and cpo of infinite labeled trees. The method is exactly the same as in the type definition package [Me89], i.e. we choose the labels and number of subtrees of infinite trees in the same way as the labels and number of subtrees of finite labeled trees are chosen there.

In general a type specification takes the following form

$$rty ::= C_1 ty_{1,1} \dots ty_{1,k_1} \mid \dots \mid C_m ty_{m,1} \dots ty_{m,k_m}$$

For the i 'th constructor let p_i stand for the number of existing types in the specification and let q_i stand for the number of occurrences of rty . Thus, k_i is equal to the sum of p_i and q_i . The existing types may be polymorphic types or not. The type predicate and

the cpo constructor corresponding to the specification will depend on the polymorphic types.

The recursive type (predicate) corresponding to this type specification can be introduced as a subset of infinite trees. The representing ‘type’ has the following form

$$\text{Is_infinite_tree: } (((\text{ty}\#\dots\#\text{ty})+\dots+(\text{ty}\#\dots\#\text{ty}))\text{node} \rightarrow \text{bool}) \rightarrow \text{bool}$$

There are m elements of the sum, one for each constructor. The products of the sum consist of p_1, \dots, p_m components. These components are the existing types in the type specification of each constructor. Thus, the name `ty` occurring in the label type is used to indicate ‘some type’ (in order to save indices). Different occurrence of `ty` may correspond to different types.

For each constructor we define a predicate to specify which trees represent values constructed by that constructor. The predicate for the i ’th constructor is defined as follows

$$\begin{aligned} \text{Is_C}_i \text{ v trl} = \\ ?x_1 \dots x_{p_i}. \\ (\text{v} = \text{LBL}(\text{INR}(\dots(\text{INR}(\text{INL}(x_1, \dots, x_{p_i})))))) \wedge (\text{LENGTH trl} = q_i) \end{aligned}$$

There are $i - 1$ occurrences of `INR`. If i is equal to m then the `INL` is omitted.

The recursive type corresponding to the specification of `rty` can now be defined as the following subset of infinite trees

$$\begin{aligned} \text{Is_rty tr} = \\ \text{Is_infinite_tree tr} \wedge \\ (\text{tr} = \text{Bt_node}) \vee \\ (!\text{v trl}. \\ (\text{node v trl}) \text{ IN tr} \Rightarrow \text{Is_C}_1 \text{ v trl} \vee \dots \vee \text{Is_C}_m \text{ v trl}) \end{aligned}$$

Keep the restrictions imposed on trees by the predicate `Is_infinite_tree` in mind when interpreting the second disjunct of this definition (see the example of section 4.4.1). It states that all nodes of a tree must correspond to a node which has been constructed using one of the constructors.

Some of the existing types of the type specification may be polymorphic types, involving say type variables $\alpha_1, \dots, \alpha_n$. For each type variable the cpo constructor for the specification becomes parameterized by a cpo variable with the type variable as the underlying type. In order to define the cpo constructor we must know the cpos corresponding to the types of the specification. The cpo of lazy elements of type `Is_rty` can be defined as follows

$$\begin{aligned} \text{rty}(D_1, \dots, D_n) = \\ \{\text{tr} \mid \text{Is_rty tr} \wedge \text{tr ins (itree(sum(prod(\dots), sum(\dots))))}\}, \\ \text{itree_rel (sum(prod(\dots), sum(\dots)))} \end{aligned}$$

where the type of each variable D_i is `" : (α_i)cpo "`, which is a piece of syntactic sugar for the type `" : ($\alpha_i \rightarrow \text{bool}$) # ($\alpha_i \rightarrow \alpha_i \rightarrow \text{bool}$) "` (introduced in chapter 3). The sum cpo of product cpos which is an argument of `itree` (and `itree_rel` above is the cpo of labels. The underlying type of this cpo was described above. Typically, sum types, product types and function types in the type of labels are replaced by the corresponding

sum, product and function space constructions on cpos, respectively. The construction introduced by this definition can be proved to be a construction on pointed cpos, by proving that it yields a sub-cpo of the pointed cpo of infinite trees for all arguments.

The constructors of the new recursive ‘type’ are defined using the tree constructor `node` and the injections `INL` and `INR` as follows

$$C_i \ x_{i,k_1} \ \dots \ x_{i,k_i} = \text{node}(\text{LBL}(\text{INR}(\dots(\text{INR}(\text{INL}(x_{i,y_{i,1}}, \dots, x_{i,y_{i,p_i}})))))) [x_{i,z_{i,1}}, \dots, x_{i,z_{i,q_i}}]$$

where $x_{i,y_{i,j}}$ corresponds to arguments which according to the type specification have an existing type. Conversely, $x_{i,z_{i,j}}$ corresponds to arguments which according to the type specification are recursive. A continuous version of each constructor can be obtained using the dependent lambda abstraction or by a definition corresponding to the one above where continuous functions are used instead of the non-determined ones (see the example of section 4.4.1).

4.5 More General Domains

The type specification accepted by the type definition package are limited in one way. If we have a specification of the form

$$\text{rty} ::= C_1 \ \text{ty}_{1,1} \ \dots \ \text{ty}_{1,k_1} \ | \ \dots \ | \ C_m \ \text{ty}_{m,1} \ \dots \ \text{ty}_{m,k_m}$$

then the $\text{ty}_{i,j}$ must be an existing type or the type `rty` which is being specified. Thus `rty` must not occur within compound types like lists and function types, or even products. However, this does not mean that it is not possible to define such types (with certain exceptions for the function space). In a mail message to `info-hol` [Me91] Melham shows how to define a recursive type `data` where the recursion is part of a compound type of lists `“(data)lists”` as a fairly simple generalization of the approach used in the type definition package.

Gunter presents a different approach based on a more general type of trees called `“(α, β)bonzai”` [Gu93] which consists of arbitrarily-branching but well-founded (finite) trees, represented as sets of nodes. Her approach allows mutual recursive definitions and function types where the recursive type occurs on the right-hand side of outer-most function arrows. The `bonzai` trees are a subset of the non-wellfounded broad trees. Below, we extend broad trees to include partial trees and then obtain a cpo of broad trees. We shall only sketch a few details below.

Before we turn to broad trees, note again that even though the methods of defining lazy datatypes and cpos described above have certain limitations inherited from the type definition package there are various ways to obtain more general types. A good place to look before one starts to consider implementing any of the ideas presented here is in Paulson’s paper [Pa93] where he shows how to obtain datatypes with finite and infinite values as least and greatest fixed points of monotone operators on sets. He also works with trees represented as sets of nodes. Note however that he does not work with domains, and therefore his trees does not contain partial nodes.

4.5.1 Broad Trees

First a type predicate for unlabeled trees is defined as a subset of sets of lists:

```
Is_unlabeled_tree tr = [] IN tr /\ (!p p'. (p++p') IN tr ==> p IN tr)
```

So, an unlabeled tree is represented by a set of nodes of type "`(*): list -> bool`" where each node is a path from the root to that node. A path is a list of branch indices.

A labeling of an unlabeled tree assigns labels to the nodes of the tree. We let labelings be partial functions of type "`(*): list -> (**)upty`" where the type specification of `upty` is

```
upty = undef | up *
```

such that the nodes for which they are defined (not equal to `undef`) correspond to unlabeled trees:

```
Is_labeling l = Is_unlabeled_tree(part_fun_domain l)
```

It is straightforward to define the constant `part_fun_domain`:

```
part_fun_domain l = {p | ~(l p = undef)}
```

Note that the type specification of the `upty` is similar to the `lty` type operator defined in section 3.6.4 and the `label` type operator defined in section 4.4. We prefer different names since they are used to do slightly different things.

The predicate `Is_labeling` defines the broad trees in the work by Gunter [Gu93]. A subset of the type of broad trees correspond to well-founded arbitrarily branching trees called 'bonsai' in [Gu93] which are used to represent a generalization of Melham's types. However, we wish to be able to distinguish partial nodes from labeled nodes. Therefore our broad trees will be a subset of an instance of labelings, defined as follows

```
Is_broad_tree (l: (*): list -> (**)label)upty) =
  Is_labeling l /\
  (!p. (l p = up NOLBL) ==> !p'. ~(p' = [])) ==> (l (p++p') = undef))
```

where the labels of trees have type "`(**): label`", not "`**`" as in Gunter's broad trees. The second conjunct makes sure that partial nodes do not have subtrees. We will use the following syntactic sugar for the 'type' of broad trees

```
(*, **)brotr == (*): list -> (**)label)upty
```

The name `brotr` abbreviates 'broad tree'.

Next, we define the ordering on broad trees which must enable us to define a `cpo` construction on broad trees:

```
brotr_rel D l l' =
  (!p. (l p = up NOLBL) ==> (?x. l p = up x)) /\
  (!p v.
    (l p = up(LBL v)) ==> (?v'. rel D v v' /\ (l' p = up(LBL v')))) /\
  (!p.
    p IN part_fun_domain l' /\ ~p IN part_fun_domain l ==>
    (?p' p''.
      (p = p' ++ p'') /\ p' IN part_fun_domain l /\ (l p' = NOLBL)))
```

Hence, two broad trees I and I' are related if, and only if, a partial node of I is also present as a node (partial or not) in I' , a labeled node of I is also a labeled node of I' such that the labels are related, and a node in I' which is not a node in I belongs to a subtree of I' replacing a partial node of I .

The construction on pointed cpos of broad trees is defined as follows

```
brotr D =
  {I | Is_broad_tree I /\ (brotr_label_set I) subset D}, brotr_rel D
```

where

```
brotr_label_set I = {v | ?p. I p = up(LBL v)}
```

Note that the cpo parameter of `brotr` corresponds to the cpo of labels whereas elements of the branching type `" : *"` are not required to be elements of a cpo. E.g. we will not take least upper bounds over the way in which trees are branching. To justify that `brotr` does yield a construction on cpos, here is the way that lubs of chains of broad trees are constructed:

```
brotr_lub D X =
  (\p.
    (?n. ?v. X n p = up(LBL v)) =>
      let n = (@n. ?v. X n p = up(LBL v)) in
        up(LBL(lub(cset(\m. @v. csuffix(X, n)m = up(LBL v)), D))) |
    (?n. X n p = up NOLBL) => up NOLBL | undef)
```

The definition constructs a broad tree, i.e. a function from paths to ‘up-lifted’ labels. For a fixed path argument of the construction, the first condition tests whether there is an tree in the chain such that the path corresponds to a labeled node in that tree. If this is the case then the rest of the trees in the chain also has a labeled node at the same position. Hence, the lub construction at this position must be the lub of these labels. Otherwise, there might be a point from which the node corresponding to the path is an unlabeled node. In this case the lub construction must be an unlabeled node at this point. Otherwise, the lub construction yields undefined because no trees of the chain has a node at the position specified by the path.

It is straightforward to define constructors for broad trees:

```
Bt_brotr = \p. ((p = []) => up NOLBL | undef)
node_brotr (a: **) (subtrees: *->((*, **)brotr)upty) =
  \p.
    (p = []) => up(LBL a) |
    (subtrees (HD p) = undef) => undef | lower(subtrees(HD p))(TL p)
```

where `lower` is defined by

```
lower(up a) = a
```

Note that subtrees of a node are represented as partial functions such that the number of subtrees may vary at each node of a tree (due to `undef`). Continuous versions of the constructors can be defined as in the previous sections.

So, we have defined a type predicate for broad trees and an ordering relation which makes (a subset of) broad trees into a cpo. The next step is to show how new recursive datatypes can be defined as subsets of broad trees and how the corresponding cpo can be defined as sub-cpos of the cpo of broad trees. Then the constructors for the new type, and continuous constructors for the cpo, must be defined.

Here, we shall only comment on how to define a type predicate for new recursive types, given a type specification of the following form

$$\text{rty} ::= C_1 \text{ ty}_{1,1} \dots \text{ ty}_{1,k_1} \mid \dots \mid C_m \text{ ty}_{m,1} \dots \text{ ty}_{m,k_m}$$

where each $\text{ty}_{i,j}$ is either an existing logical type (not containing rty) or of the form " $\text{ty} \rightarrow \text{rty}$ " for some existing type expression ty . The method described in the previous section corresponds to the case where ty is the type one . Further generalization is discussed in [Gu93].

A new type can be represented as a subset of broad tree with a certain branching type and a certain labeling type. Both of these are a sum type with a component for each constructor of the specification. The contribution of the constructor case $C_i \text{ ty}_{i,1} \dots \text{ ty}_{i,k_i}$ to the branching type is (again) a sum of existing types $\text{ety}_{i,j}$ where $\text{ty}_{i,j} = \text{ety}_{i,j} \rightarrow \text{rty}$, or one if none such exists. The contribution to the labeling type is the product of each existing type and one if there are no existing types. This is exactly the same approach to constructing the branching and labeling types as in [Gu93] which can be conferred for further details.

Chapter 5

The HOL-CPO System

Up to this point we have considered the formalization of domain theoretic concepts in HOL and not so much using this formalization. We have presented the basic definitions as well as constructions on cpos and continuous functions which give ways of writing terms that are guaranteed to be cpos and continuous functions. We have also shown how certain recursive domains can be introduced, e.g. strict and lazy lists. However, it is not practical to use the formalization as presented above directly. One must prove all the time that terms are cpos (or pointed cpos), continuous functions and inclusive predicates, e.g. each time the fixed point property of `Fix` is used (see section 3.8) or each time the fixed point induction theorem is used (see section 3.9). In particular, proofs of continuity may require a substantial amount of work when function definitions involve nested lambda abstractions and are longer than a couple of lines. Besides, function constructions are tedious and difficult to read and write since they are parameterized by the cpo variables of the domains on which they work.

So, in order to make the formalization useful for reasoning about functional programs in practice, there are at least two things we should do:

- develop an interface which supports a less tedious syntax for functions, and
- provide syntactic-based proof functions to prove automatically that certain terms are cpos, continuous functions and inclusive predicates.

Actually, it is advantageous to treat proofs of continuity facts as a special case of the more general problem of proving that a term belongs to some cpo. We can solve this problem in many cases with a syntactic-based proof function for constructing the cpo of a term. This function is called the type checker; terms that are elements of a cpo are called cpo-typable terms, or just typable terms. The syntactic-based proof functions for cpos and inclusive predicates are called the cpo prover and the inclusive prover, respectively. The notations supported by the interface and these syntactic-based proof tools are not fixed, they can be extended by declaring new constructor terms. On top of the syntactic-based proof tools and the declaration tools a number of derived definition tools have been implemented for introducing cpos and arbitrary terms in cpos as abbreviations of terms which fit within the syntactic notations.

The result of these developments is an integrated system, called HOL-CPO, where domain theory seems almost built-in to the user. Basic proof tools are applied behind the scenes so in most cases the user does not have to worry too much about proving

domain theoretic concepts in order to employ the formalization. (However, the tools are prototypes and are therefore not as optimized and powerful as they could be.)

5.1 Notations for Cpos and Pointed Cpos

The syntactic notations for cpos and pointed cpos are similar yet slightly different since certain constructions on cpos yield cpos which do not have a bottom element. We shall therefore describe the notations separately just as there are separate proof functions to prove such facts too, called the cpo prover and the pcpo prover respectively.

The notation for cpos can be described informally as follows

$$D ::= t \mid \text{discrete } Z \mid \text{prod}(D_1, D_2) \mid \text{cf}(D_1, D_2) \\ \mid \text{lift } D \mid \text{sum}(D_1, D_2) \mid \mathbb{C}(D_1, \dots, D_n)$$

where

- t is any HOL term for which a theorem is available stating it is a cpo,
- Z is some HOL set,
- \mathbb{C} belongs to an extendable set of names of cpo constructors, and
- D, D_1, \dots, D_n are cpos ($0 \leq n$).

A cpo constructor is a constant, i.e. a nullary constructor, or a constant applied to a tuple of cpo variables such that it has been shown that the constructor yields a cpo, provided its arguments are cpos. The constructions on cpos presented in section 3.6 are built-in as it appears, but new constructions can be introduced as described in section 5.4. Note that the notation allows any term as long as a theorem is available which states that the term is a cpo or a pointed cpo. Such a theorem is called a *cpo fact*.

A constructor for pointed cpos is similar to a constructor for cpos. It is a constant which applied to a number of arguments yields a pointed cpo, provided the arguments are cpos or pointed cpos. The notation for pointed cpos can be described informally as follows

$$E ::= t_p \mid \text{prod}(E_1, E_2) \mid \text{cf}(D, E) \mid \text{lift } D \mid \mathbb{C}_p(X_1, \dots, X_n)$$

where

- t_p is any HOL term for which a theorem is available stating it is a pointed cpo,
- \mathbb{C}_p belongs to an extendable set of names of constructors for pointed cpos,
- D is a cpo,
- E, E_1 and E_2 are pointed cpos, and
- X_1, \dots, X_n are either cpos or pointed cpos.

Note that the discrete and the sum constructions do not yield pointed cpos according to this notation. The sum of two cpos is never a pointed cpo whereas the discrete construction yields a pointed cpo iff the associated set is a singleton set (see section 3.6).

5.1.1 Algorithm for Proving Cpo Facts

The cpo prover takes a term t as input, which must fit within the syntactic notation for cpos. It proves and returns the theorem $\vdash \text{cpo } t$. The underlying algorithm is based on theorems of the form

$$\vdash \text{cpo}(D1) \implies \dots \implies \text{cpo}(Dn) \implies \text{cpo}(C(D1, \dots, Dn))$$

where $D1, \dots, Dn$ are variables. There must be one such theorem for each cpo constructor C . The algorithm is simple: match an input term " $C(D1', \dots, Dn')$ " against the conclusion of such a theorem and instantiate the variables. Prove each of the antecedents (left-hand sides of the implications) using the algorithm recursively, obtaining (by modus ponens) the theorem $\vdash \text{cpo}(C(D1', \dots, Dn'))$.

The algorithm for the pcpo prover is similar. However, it exploits the cpo prover to prove antecedents of the form " $\text{cpo } D$ ". It is applied recursively to prove antecedents of the form " $\text{pcpo } D$ ".

5.1.2 Proving Cpo Facts

A cpo fact is a theorem stating either some term is cpo or some term is a pointed cpo. Such facts are proved automatically by two separate programs which implement the syntactic notations for cpos and pointed cpos, respectively.

The program for automatically proving that terms are cpos is called the cpo prover and it has the following ML type.

```
#CPO_PROVER; ;  
- : (thm list -> conv)
```

It takes a theorem list of cpo facts and a term as arguments and proves a theorem stating the term is a cpo using the cpo facts and the available cpo constructors, i.e. the constructors which are part of the syntactic notation. The cpo facts may state terms are pointed cpos since the prover knows that pointed cpos satisfying pcpo also satisfy cpo.

Assuming `Nat` has been introduced as a constructor for the cpo of natural numbers (see section 5.4), the cpo prover proves immediately that the continuous functions from natural numbers to the lifted natural numbers constitute a cpo.

```
#CPO_PROVER[]"cf(Nat, lift Nat)"; ;  
|- cpo(cf(Nat, lift Nat))
```

In this case we need no additional cpo facts so the theorem list argument is the empty list. If we wish to prove that the continuous functions from `Nat` to the lifting of any cpo `D` constitute a cpo we must assume the cpo variable `D` is a cpo.

```
#cpo1; ;  
": (*1 -> bool) # (*1 -> (*1 -> bool))" : type
```

```
#let cpo_D = ASSUME "cpo(D: ^cpo1)"; ;  
cpo_D = . |- cpo D
```

```
#CPO_PROVER[cpo_D]"cf(Nat, lift(D: ^cpo1))"; ;  
. |- cpo(cf(Nat, lift D))
```

In this case, the theorem list contains the theorem stating D is a cpo and the assumption of this theorem is also an assumption of the theorem returned by the cpo prover.

Both these examples are not only cpos they are also pointed cpos. Such facts are proved by another program, called the pcpo prover,

```
#PCPO_PROVER; ;
- : (thm list -> conv)
```

which is used in the same way as the cpo prover so the examples are conducted as follows

```
#PCPO_PROVER["cf(Nat, lift Nat)"; ;
|- pcpo(cf(Nat, lift Nat))

#PCPO_PROVER[cpo_D]"cf(Nat, lift(D: ^cpo1))"; ;
. |- pcpo(cf(Nat, lift D))
```

The cpo fact in the theorem list argument could state D is a pointed cpo but in this example it does not have to. The pcpo prover as well as the cpo prover knows pointed cpos are also cpos.

5.2 Notation for Cpo-typable Terms

The parser transforms a notation for (cpo-) typable terms into an internal syntax of terms which can be type checked by the type checker (see figure 2.3 on page 23), i.e. this program can automatically construct the domain of a typable term which fits within the notation. Hence, the two programs are quite similar, both construct domains, but whereas the parser need only manipulate terms the type checker is based on proof.

The notation for typable terms can be described informally as follows

$$e ::= t \mid x \mid c \mid \backslash xs :: \text{Dom } D. e \mid (e_1 e_2)$$

where

- t belongs to a set of basic typable terms, i.e. t can be any HOL term, typically a constant or a variable, such that a fact is available which states which cpo it is an element of,
- x is a variable of a dependent lambda abstraction,
- xs is a sequence (tuple) of variables (x_1, \dots, x_n) and D is a product of cpos D_1, \dots, D_n , where $1 \leq n$,
- c belongs to an extendable set of interface level names of parameterized function constructors, and
- e, e_1 and e_2 are typable terms.

The set of basic typable terms can be extended by declaration tools (see section 5.4). If a term, e.g. a constant or a variable, does not fit the notation it must be declared before it is used. A parameterized function constructor is a function constructor similar to the

ones introduced in section 3.6 which is parameterized by the cpo variables (at least one) of the domain on which it works. The last letter of such constructors must be an 'l' (for internal). The external level names for constructors is obtained by omitting this letter.

These external names are constants in HOL which do not take cpo parameters as arguments. The external constants are only present in interface level terms since the parser replaces each interface constant by the corresponding constructor and inserts the cpo parameters, if it is able to do so. The constants are not introduced by definition, only by type (see [GM93], page 260), since we do not want them to abbreviate anything, merely to ensure interface level terms type in a certain way. The in-built constructors (i.e. interface constants) are

Proj 1 , Proj 2 , Tupling , Prod , Apply , Curry , Lift , Ext ,
Inl , Inr , Sum , Fix , Id and Comp .

New constructors can be introduced as described in section 5.4 (see also section 5.5).

In the following we first describe the algorithms for parsing and type checking and then consider a few examples.

5.2.1 Algorithm for Parsing

The standard HOL parser and pretty-printer can be extended by hooking up a special parser and pretty-printer with the built-in ones (exploiting preterms [GM93]). Employing this technique we become able to do transformations on terms after they are type checked and parsed by the HOL system and before they are pretty-printed on the screen. Hence, this allows us to implement two levels of syntax, a nice and simple one for the user to work with at the external or interface level and a more detailed and complex one for the system to work with at the internal level. At the external level it is not necessary to write the cpo parameters on function constructors. The parser attempts to calculate these parameters automatically. The pretty-printer is much simpler, it just throws away all cpo parameters of function constructors.

A secondary purpose of the parser and pretty-printer is, respectively, to unfold and fold derived definitions of cpos, i.e. definitions of constants abbreviating cpo. Such definitions must be declared to the HOL-CPO system as described in section 5.4. It is sometimes necessary to expand such abbreviations, e.g. if we define `NatFun` to be `"cf(Nat, Nat)"` and there is a function `"f ins Nat"`, then we must know how `natFun` is defined to conclude for instance `"(f x) ins Nat"`, for some `"x ins Nat"`. It is easy to implement the folding and unfolding of such definitions using rewriting so I will not comment further on this detail.

The parser inserts cpo parameters on parameterized function constructors. It takes a term of the external syntax as input and produces a transformed term of the internal syntax as output. The algorithm for constructing the cpo parameters, which is in the body of the parser, takes a term as input and returns a pair consisting of the transformed term and the domain that the term is an element of. The parser throws away the domain component of the result. A rough sketch of the algorithm can be presented as follows, by cases on the structure of the input term:

Constant/variable/any term: Find the input term in a database of terms and their domains (if possible), and return both. E.g. the database could specify that $\$+$

is in "cf(Nat, cf(Nat, Nat))", where Nat is the discrete universal cpo of natural numbers (see chapter 2 or chapter 6). In this case, the algorithm would return ($\$+$, "cf(Nat, cf(Nat, Nat))"). If the constant is a constructor, for example Fix , then it would return the internal version with cpo parameters, for example ("Fixl E", "cf(cf(E, E), E)"). The parameters are instantiated in the combination case.

Pair: The input term has the form "(t, u)". Parse t and u recursively, obtaining the results (t', D) and (u', E). Return ("(t', u')", "prod(D, E)").

Combination The input term has the form "t u". Apply the algorithm recursively to t and u, obtaining (t', "cf(D, E)") and (u', D'). The term D' must be an instance of the term D. Calculate the corresponding instance E' of E and return ("t' u'", E'). (The instance is obtained after doing a one way matching of D and D'. Generalizing the match to unification would make the algorithm more powerful.)

Abstraction: The input term has the form "\x :: Dom D. e[x]" (x could be a pair). Add (x, D) to the database while parsing e[x] recursively, obtaining the result "(e' [x], E)". Return the pair ("lambda D (\x. e' [x]", "cf(D, E)"). Strictly speaking, this is not a valid inference since a lambda abstraction need not be continuous just because it maps elements of its domain D to elements of its codomain E (see how the type checker handles lambda abstractions). However, when e[x] meets the syntactic notation, this is valid. Also note that the restricted abstraction is transformed into the lambda abstraction, a secondary purpose of the parser.

If something goes wrong in a call to the algorithm, then it returns a certain dummy domain to indicate this. The user will usually realize this when the type checker fails (only sometimes the parser fails in such a situation).

5.2.2 Algorithm for Type Checking

The type checker takes a term t of the internal syntactic notation as input and reconstructs the domain of the term in the sense that it constructs a cpo D and proves $\vdash t \text{ ins } D$. A rough sketch of the algorithm can be presented as follows, by cases on the structure of the input term:

Constant/variable/any term: Find (if possible) and return a theorem in a database of cpo membership facts stating that the input term is in some cpo. For instance, the theorem could be $\vdash \$+ \text{ ins } \text{cf}(\text{Nat}, \text{cf}(\text{Nat}, \text{Nat}))$, or indeed $\vdash t \text{ ins } D$ for any term t and domain D.

Parameterized function constructor: Input has the form "Funl (D1', ..., Dn')" for some function constructor Funl. Find Funl in a database of facts of the form

$$\vdash \text{cpo } D1 \implies \dots \implies \text{cpo } Dn \implies \text{Funl}(D1, \dots, Dn) \text{ ins } \text{cf}(\dots, \dots)$$

where D1, ..., Dn are variables. Instantiate these variables with D1', ..., Dn' respectively, prove the cpo (or pointed cpo) assumptions using the cpo (or pcpo) prover and return $\vdash \text{Funl}(D1', \dots, Dn') \text{ ins } \text{cf}(\dots, \dots)$.

Pair: The input term has the form "(t, u)". Apply the type checker recursively, obtaining $\vdash t \text{ ins } D$ and $\vdash u \text{ ins } E$. Then, use the theorem (employ modus ponens)

$$\vdash !D1 D2 x y. x \text{ ins } D1 \implies y \text{ ins } D2 \implies (x, y) \text{ ins } (\text{prod}(D1, D2))$$

to conclude and return $\vdash (t, u) \text{ ins } \text{prod}(D, E)$.

Combination The input term has the form "(t u)". Apply the type checker recursively, obtaining $\vdash t \text{ ins } \text{cf}(D, E)$ and $\vdash u \text{ ins } D$. Then, use the theorem

$$\vdash !D1 D2 e1 e2. e1 \text{ ins } (\text{cf}(D1, D2)) \implies e2 \text{ ins } D1 \implies (e1 e2) \text{ ins } D2$$

to conclude and return $\vdash (t u) \text{ ins } E$.

Lambda abstraction: The input term has the form " $\lambda D(\lambda x. e[x])$ " (which is not considered to be a combination). This is the difficult case! Ideally, we would like a theorem stating " $\forall v \text{ ins } D1 \implies f(v) \text{ ins } D2 \implies (\lambda D1 f) \text{ ins } \text{cf}(D1, D2)$ ", but this does not hold; the lambda abstraction is not necessarily continuous (nor monotonic). Instead we must proceed according to the term structure of the body of the abstraction. The local recursive algorithm for this pushes the abstraction into the body in recursive calls and pulls back continuity. The local algorithm can be described as follows, by cases on the structure of the body of the abstraction:

Constant term: The body $e[x]$ does not contain the variable x of the abstraction and the theorem $\vdash e \text{ ins } E$ is in the database of cpo membership facts (otherwise try to destruct $e[x]$ further). Prove D and E are cpos (using the cpo prover) and use the theorem (employ modus ponens)

$$\begin{aligned} &\vdash !D0 D1 e. \\ &\quad \text{cpo } D0 \implies \text{cpo } D1 \implies e \text{ ins } D1 \implies \\ &\quad (\lambda D0(\lambda v. e)) \text{ ins } (\text{cf}(D0, D1)) \end{aligned}$$

to conclude and return $\vdash (\lambda D(\lambda x. e[x])) \text{ ins } \text{cf}(D, E)$.

Variable: The body $e[x]$ equals the variable of the abstraction x . Prove D is a cpo and return $\vdash (\lambda D(\lambda x. e[x])) \text{ ins } \text{cf}(D, D)$, using

$$\vdash !D0. \text{cpo } D0 \implies (\lambda D0(\lambda v. v)) \text{ ins } (\text{cf}(D0, D0))$$

Pair: The body is a pair of the form " $(e1[x], e2[x])$ ". Apply the algorithm recursively on each component of the pair, obtaining

$$\begin{aligned} &\vdash (\lambda D(\lambda x. e1[x])) \text{ ins } \text{cf}(D, D1) \\ &\vdash (\lambda D(\lambda x. e2[x])) \text{ ins } \text{cf}(D, D2). \end{aligned}$$

Use the theorem

$$\begin{aligned} &\vdash !D0 D1 D2 e1 e2. \\ &\quad (\lambda D0 e1) \text{ ins } (\text{cf}(D0, D1)) \implies \\ &\quad (\lambda D0 e2) \text{ ins } (\text{cf}(D0, D2)) \implies \\ &\quad (\lambda D0(\lambda x. (e1 x, e2 x))) \text{ ins } (\text{cf}(D0, \text{prod}(D1, D2))) \end{aligned}$$

by matching the assumptions (left-hand sides of the implications) with the results, conclude and return

$| - (\lambda D (\lambda x. (e1[x], e2[x]))) \text{ ins } cf(D, \text{prod}(D1, D2)).$

Combination: The body is a combination of the form " $(e1[x] \ e2[x])$ ". This case is similar to the pair case, but uses instead the theorem

$| - !D0 \ D1 \ D2 \ e1 \ e2.$
 $(\lambda D0 \ e1) \ \text{ins} \ (cf(D0, cf(D1, D2))) \ ==>$
 $(\lambda D0 \ e2) \ \text{ins} \ (cf(D0, D1)) \ ==>$
 $(\lambda D0 (\lambda x. e1 \ x(e2 \ x))) \ \text{ins} \ (cf(D0, D2))$

Abstraction: The body is an abstraction of the form " $\lambda D' (\lambda y. e' [x, y])$ ". This case is a bit complicated, and gives inefficiency as well since the body $e' [x, y]$ is traversed twice. The algorithm is used recursively to conclude

$[y \ \text{ins} \ D'] \ | - (\lambda D (\lambda x. e' [x, y])) \ \text{ins} \ cf(D, E),$

assuming " $y \ \text{ins} \ D'$ ", and to conclude

$[x \ \text{ins} \ D] \ | - (\lambda D' (\lambda y. e' [x, y])) \ \text{ins} \ cf(D', E),$

assuming " $x \ \text{ins} \ D$ ". The cpo prover is then used to prove that D , D' and E are cpos (note that E is constructed by the algorithm). Using the theorem

$| - !D0 \ D1 \ D2 \ e.$
 $(!d. \ d \ \text{ins} \ D1 \ ==> (\lambda D0 (\lambda v. e \ v \ d)) \ \text{ins} \ (cf(D0, D2))) \ ==>$
 $(!v. \ v \ \text{ins} \ D0 \ ==> (\lambda D1 (e \ v)) \ \text{ins} \ (cf(D1, D2))) \ ==>$
 $\text{cpo} \ D0 \ ==>$
 $\text{cpo} \ D1 \ ==>$
 $\text{cpo} \ D2 \ ==>$
 $(\lambda D0 (\lambda v. \ \lambda D1 (e \ v))) \ \text{ins} \ (cf(D0, cf(D1, D2)))$

the algorithm derives and returns the theorem

$| - (\lambda D (\lambda x. \ \lambda D' (\lambda y. e' [x, y]))) \ \text{ins} \ cf(D, cf(D', E)).$

This concludes the description of the local algorithm for type checking lambda abstractions.

This concludes the description of the type checker. The algorithm will fail if the input term does not fit within the syntactic notation.

Lambda abstractions over tuples of variables are handled by introducing projection functions (see section 3.6.2) such that a variable can be used instead of a tuple. When the algorithm has been applied recursively this transformation is reversed by eliminating the projections and introducing the original tuple again.

5.2.3 Examples of Parsing

Let us illustrate by an example how the parser works. Consider the following term which fits within the notation above,

$"\lambda f :: \text{Dom}(cf(\text{Nat}, \text{lift Nat})). \ \text{Ext}(\text{Tupling}(f, f))"$

assuming `Nat` has been declared as the discrete cpo constructed from the set of all elements of type `": num"` (see section 5.4). The restricted abstraction is transformed into a dependent lambda abstraction as follows: a term like

```
"\x :: Dom D. e[x]"
```

becomes

```
"lambda D(\x. e[x])"
```

Furthermore, the parser works recursively on subterms so when it meets the abstraction it parses the body recursively.

In the recursive call on the body of the above abstraction, the parser assumes that the variable of the abstraction is in the cpo associated with the abstraction, i.e. `"f ins cf(Nat, lift Nat)"` is assumed. In the body term it meets `Ext` applied to the tupling term and parses this argument recursively first. Similarly for the tupling term, the product it is applied to is parsed recursively, and so on. The deepest it gets in the term is down to the variable `f`, which it meets twice. Both times it simply returns `f` and the assumption that `f` is in `"cf(Nat, lift Nat)"`. If `f` was not part of the notation then the parser would still return `f` but it would return a dummy domain and the cpo parameters on function constructors would be dummy ones too. The result of parsing the pair `"(f, f)"` is this term `"(f, f)"` and the domain `"prod(cf(Nat, lift Nat), cf(Nat, lift Nat))"`. This is used to construct the cpo parameters on the tupling construction, in turn used to construct the cpo parameters on the extension construction, yielding the final result

```
"lambda
  (cf(Nat, lift Nat))
  (\f.
    Ext1
      (Nat, prod(lift Nat, lift Nat))
      (Tupling1 (Nat, lift Nat, lift Nat) (f, f)))"
```

If we had to write this term directly by hand, we would have to calculate all cpo parameters very carefully. Besides, reading the function is difficult due to the cpo parameters on `Ext1` and `Tupling1` which are in fact not necessary for our understanding of what the function does.

In the following example, the parser deduces the wrong cpo parameter of `Lift`, namely the dummy one which is `D`, and then fails to deduce the cpo parameters of `Comp`.

```
#"Comp(Lift, ($+ 0))"; ;
evaluation failed      parse: non-matching domains of Comp
```

This problem could be solved by a better implementation which unifies the domains of arguments of constructors. The current implementation fails since the pair, which `Comp` is applied to, consists of functions in non-matching domains (the dummy domain on `Lift1` has not been instantiated). Fortunately, one is not completely lost in this situation since it is always possible to insert the right domain on `Lift` manually, using the internal version of the constant. Hence, the following works

```
#"Comp(Lift Nat, ($+ 0))";;
"Comp(Lift, ($+ 0))" : term
```

Such situations where we must insert cpo information manually are very rare in the examples we have looked at. When the parser fails to deduce the cpo parameters of some constructor (and just uses the dummy variables) the problem often is that the constants and variables used in a term have not been declared. All constants and terms must be declared before they are used, in order to become part of the notation of typable terms.

5.2.4 Type Checking

The following ML proof function is called the type checker and it can be used to deduce which cpo a typable term belongs to.

```
#TYPE_CHECK; ;
- : (thm list -> thm list -> conv)
```

It takes three arguments, a theorem list of cpo facts, a theorem list of ins facts and a term. The first theorem list is used to extend the cpo and pcpo provers with additional cpo facts about arbitrary HOL terms (e.g. variables) and the second is used to extend the set of basic typable terms. This set can also be extended by a declaration as described in section 5.4 and usually this strategy should be used since only this affects both the parser and the type checker (see below). If the term argument fits the notation for typable terms then the result of applying the type checker is a theorem stating which cpo the term belongs to.

Let us consider a couple of examples. Assuming `Nat` has been declared as a cpo and addition `$+` has been declared as a continuous function (see section 5.4), then the type checker can deduce the type of a strict double function as follows

```
#TYPE_CHECK[[]][[]]"Ext(\n :: Dom Nat. Lift(n+n))";;
|- (Ext(\n :: Dom Nat. Lift(n + n))) ins (cf(Lift Nat, Lift Nat))
```

Here, we do not need any additional facts about cpOs and other terms, hence the theorem lists are empty. Note that it is not required that the construction for applying a function to an argument is used; HOL application by juxtaposition can be used instead.

In the following example we use both a cpo variable and a function variable which have been assumed to be a pointed cpo and a continuous function, respectively.

```
#let pcpo_E = ASSUME "pcpo(E: ^cpo1)";;
pcpo_E = . |- pcpo E

#let cf_f = ASSUME"f ins cf(E: ^cpo1, Nat)";;
cf_f = . |- f ins (cf(E, Nat))

#TYPE_CHECK[pcpo_E][cf_f]
#"x :: Dom(E: ^cpo1). \n :: Dom Nat. Lift((f x)+n)";;
evaluation failed PCPO_PROVER: unknown variable D
```

The evaluation fails! The reason for this is that the parser does not know the ins fact about f (it does not know which cpo f belongs to) and therefore it cannot deduce the right cpo parameter of `Lift` (which is `Nat`, the dummy wrong one is `D`). Instead we can declare this fact to the system and then we do not need to use it as an argument of the type checker.

```
#declare cf_f; ;
. |- f ins (cf(E, Nat))

#TYPE_CHECK[pcpo_E] []
#" \x :: Dom(E: ^cpo1). \n :: Dom Nat. Lift((f x)+n)"; ;
.. |- (\x :: Dom E. \n :: Dom Nat. Lift((f x) + n)) ins
      (cf(E, cf(Nat, lift Nat)))
```

So, in most cases such facts must be declared rather than supplied as an argument since terms must be declared for the parser before they can be used. Had the parser and type checker been implemented in a more intelligent way (using unification in the combination case, for instance) they might be able to deduce the type of f from the context in this example. But in general this form of type reconstruction might be as difficult as type checking in dependent type theory, which is undecidable. I have not studied the problems more careful since the present algorithm works fine as a prototype.

After such facts have been declared they can be undeclared (as described in section 5.4)

```
#undeclare(); ;
. |- f ins (cf(E, Nat))
```

so it is just a matter of convenience how the parser and type checker work. We could also have inserted the right domain on `Lift` manually (see section 5.2.3) but using declarations are sometimes more convenient.

Finally, let us show a tactic version of the type checker.

```
#TYPE_CHECK_TAC; ;
- : (thm list -> thm list -> tactic)
```

It can be used to finish off goals stating a term is in some cpo. As an example we can apply it to the term above, set as a goal:

```
#set_goal (["pcpo(E: ^cpo1)"; "f ins cf(E: ^cpo1, Nat)"],
#" (\x :: Dom(E: ^cpo1). \n :: Dom Nat. Lift((f x)+n)) ins
# (cf(E, cf(Nat, lift Nat)))"); ;
" (\x :: Dom E. \n :: Dom Nat. Lift((f x) + n)) ins
  (cf(E, cf(Nat, lift Nat)))"
  [ "f ins (cf(E, Nat))" ]
  [ "pcpo E" ]
```

```
() : void
```

```
#e(TYPE_CHECK_TAC[] []); ;
OK..
```

```

goal proved
.. |- (lambd
      E
      (\x.
        lambda
          (di scret e UNIV)(\n. Li ftl (di scret e UNIV)((f x) + n)))) i ns
      (cf(E, cf(di scret e UNIV, Li ftl (di scret e UNIV))))

```

Previous subproof:

```

goal proved
() : voi d

```

The tactic uses the assumptions to obtain cpo and ins facts but additional facts can be supplied via the theorem list arguments as well. Note the pretty-printer is not applied to print the output from the subgoal package when a goal is proved (this is just a choice of implementation). Internally, terms look like that!

5.2.5 Switching the Interface On and Off

When something is wrong, for instance, when the parser calculates the wrong cpo argument of some constructor (see section 5.2.3), then it is useful to be able to switch off the interface, or just the parser or the pretty-printer. For this we provide the following functions

```

set_parse   : (voi d -> voi d)
unset_parse : (voi d -> voi d)
set_pp      : (voi d -> voi d)
unset_pp    : (voi d -> voi d)
unset_ppfol d: (voi d -> voi d)
set_i face  : (voi d -> voi d)
unset_i face: (voi d -> voi d)

```

The `set` functions are for switching transformations on and the `unset` functions are for switching transformations off. Note that the parser and the pretty-printer can be switched on and off independently and that we can switch off pretty-printing but keep the folding of cpo definitions, using `unset_ppfol d`. The function `set_i face` does both `set_parse` and `set_pp`, and `unset_i face` is similar.

5.3 Proving Inclusiveness

The inclusive prover is a proof function for proving that predicates for fixed point induction are inclusive, i.e. that they admit induction.

```

#I NCLUSI VE_PROVER; ;
- : (thm list -> thm list -> thm list -> conv)

```

It takes three theorem lists as arguments which contain additional cpo facts, ins facts and facts about whether domains used in universal quantifications are non-empty (in most

cases the latter kind of statements can be proved automatically). The syntactic checks performed by the inclusive prover resemble the syntactic checks performed in the LCF theorem prover by the primitive rule of the fixed point induction; at least, it is based on the description of this check in [Pa87] (see page 199–200). The inclusive prover is used on predicates stated using `mk_pred` which is defined by

$$\vdash \text{!}D P. \text{mk_pred}(D, P) = \{x \mid x \text{ ins } D \wedge x \text{ IN } P\}$$

This just guarantees that a predicate is a subset of a cpo. Predicates of the form "`mk_pred(D, \lambda x. e[x])`" can be proved to be inclusive predicates on the cpo D if all chains in D are finite, i.e. chains are constant from a certain point, or if e has the following form

- b , any term of type boolean not containing x .
- "`rel E f1[x] f2[x]`", $f1[x]$ and $f2[x]$ continuous in x .
- "`f1[x] = f2[x]`", $f1[x]$ and $f2[x]$ continuous in x .
- "`~(f[x] = bottom E)`", $f[x]$ continuous in x .
- "`~(rel E f[x] v)`", v in E and $f[x]$ continuous in x .
- "`e1[x] /\ e2[x]`", $e1[x]$ and $e2[x]$ inclusive in x .
- "`e1[x] \\/ e2[x]`", $e1[x]$ and $e2[x]$ inclusive in x .
- "`e1[x] ==> e2[x]`", $\sim e1[x]$ and $e2[x]$ inclusive in x .
- "`!a. a ins A ==> e[a, x]`", $e[a, x]$ inclusive in x for all a in a cpo pair A which must be non-empty.

This is called the notation for inclusive predicates. When we say something is continuous in a variable we mean the lambda abstracted term over the variable is continuous. Similarly when we say something is inclusive we mean the `mk_pred` term is inclusive. Examples of the use of the inclusive prover are presented in chapter 6.

The notion of chain-finiteness can be defined as follows:

$$\begin{aligned} \vdash \text{!}D. \\ \text{cfinite } D = \\ \text{cpo } D \wedge (\text{!}X. \text{chain}(X, D) ==> (\text{?}n. \text{!}m. X\ n = X(n + m))) \end{aligned}$$

Hence, a cpo is chain-finite iff all chains are constant from a certain point.

The discrete, lift and product constructions on cpos yield chain-finite cpos when their cpo arguments are chain-finite:

$$\begin{aligned} \vdash \text{!}Z. \text{cfinite}(\text{discrete } Z) \\ \vdash \text{!}D. \text{cfinite } D ==> \text{cfinite}(\text{lift } D) \\ \vdash \text{!}D1. \text{cfinite } D1 ==> (\text{!}D2. \text{cfinite } D2 ==> \text{cfinite}(\text{prod}(D1, D2))) \end{aligned}$$

In addition, the function space construction yields a chain-finite cpo for a finite domain and chain-finite codomain. The sum construction yields a chain-finite cpo if just one of the cpo components is chain-finite. These facts have not been proved.

Algorithm for inclusiveness

The algorithm for proving inclusiveness is based on theorems which correspond to each of the cases in the above notation. These are used in a backwards way, just like the theorems for type checking and proving cpo facts. Hence, each case of the above notation corresponds to the conclusion (right-hand side of the right-most implication) of one of the theorems:

```
i ncl usi ve_ const = |- !D v. cpo D ==> i ncl usi ve(mk_pred(D, (\x. v)), D)
```

```
i ncl usi ve_ rel =  
|- !D E e1 e2.  
  (lambd a D e1) i ns (cf(D, E)) ==>  
  (lambd a D e2) i ns (cf(D, E)) ==>  
  i ncl usi ve(mk_pred(D, (\x. rel E(e1 x)(e2 x))), D)
```

```
i ncl usi ve_ eq =  
|- !D E e1 e2.  
  (lambd a D e1) i ns (cf(D, E)) ==>  
  (lambd a D e2) i ns (cf(D, E)) ==>  
  i ncl usi ve(mk_pred(D, (\x. e1 x = e2 x)), D)
```

```
i ncl usi ve_ not_ bottom =  
|- !D E e.  
  pcpo E ==>  
  (lambd a D e) i ns (cf(D, E)) ==>  
  i ncl usi ve(mk_pred(D, (\x. ~(e x = bottom E))), D)
```

```
i ncl usi ve_ not_ rel =  
|- !D E e v.  
  (lambd a D e) i ns (cf(D, E)) ==>  
  v i ns E ==>  
  i ncl usi ve(mk_pred(D, (\x. ~rel E(e x)v)), D)
```

```
i ncl usi ve_ conj =  
|- !D P Q.  
  cpo D ==>  
  i ncl usi ve(mk_pred(D, P), D) ==>  
  i ncl usi ve(mk_pred(D, Q), D) ==>  
  i ncl usi ve(mk_pred(D, (\x. P x /\ Q x)), D)
```

```
i ncl usi ve_ di sj =  
|- !D P Q.  
  cpo D ==>  
  i ncl usi ve(mk_pred(D, P), D) ==>  
  i ncl usi ve(mk_pred(D, Q), D) ==>  
  i ncl usi ve(mk_pred(D, (\x. P x \/ Q x)), D)
```

```

i ncl usi ve_ i mp =
|- !D P Q.
  cpo D ==>
  i ncl usi ve(mk_ p red(D, (\x. ~P x)), D) ==>
  i ncl usi ve(mk_ p red(D, Q), D) ==>
  i ncl usi ve(mk_ p red(D, (\x. P x ==> Q x)), D)

```

```

i ncl usi ve_ f oral l =
|- !D P A.
  ~empty A ==>
  (!a. a i ns A ==> i ncl usi ve(mk_ p red(D, P a), D)) ==>
  i ncl usi ve(mk_ p red(D, (\x. !a. a i ns A ==> P a x)), D)

```

These theorems may be read as inference rules; the left-hand sides of implications (antecedents) correspond to premises and the rightmost term to the conclusion. The theorems are used in a backwards fashion, by matching against the conclusion of the theorems (inference rules). The antecedents of each theorem are proved by the inclusive prover recursively, by the cpo prover, by the type checker, or by using the following theorems to prove cpos are non-empty (again matching against conclusions):

```

|- ~empty(di s crete UNIV)
|- !D. ~empty(l i ft D)
|- !D1 D2. ~empty D1 ==> ~empty D2 ==> ~empty(prod(D1, D2))
|- !D1 D2. ~empty D1 ==> ~empty(sum(D1, D2))
|- !D1 D2. ~empty D2 ==> ~empty(sum(D1, D2))
|- !D1 D2. cpo D1 ==> cpo D2 ==> ~empty D2 ==> ~empty(cf(D1, D2))
|- !D. pcpo D ==> ~empty D

```

As mentioned above, the inclusive prover allows additional non-empty facts to be supplied via a theorem list argument. The constant `empty` is defined

```

|- !D. empty D = (set D = {})

```

The inclusive prover can also handle situations where predicates do not fit within the above notation. Then the domain which the predicate is a subset of should be easy in the sense that it is chain-finite. For chain-finite cpos all predicates are inclusive:

```

|- !D. cf i ni te D ==> (!P. i ncl usi ve(mk_ p red(D, P), D))

```

Above, we saw some rules for building chain-finite cpos. For instance, these yield the following instances of the previous fact:

```

|- !Z P. i ncl usi ve(mk_ p red(di s crete Z, P), di s crete Z)
|- !Z P. i ncl usi ve(mk_ p red(l i ft(di s crete Z), P), l i ft(di s crete Z))

```

which state that in particular discrete and lifted discrete cpos are chain-finite.

In order to transform cpos into discrete cpos, a kind of simplifier has been implemented which is based on the following theorems:

```

|- !X Y.
  prod(discrete X, discrete Y) = discrete{(x,y) | x IN X /\ y IN Y}
|- !X Y.
  sum(discrete X, discrete Y) =
  discrete({INL x | x IN X} UNION {INR y | y IN Y})
|- !D X. cf(D, discrete X) = discrete{f | cont f(D, discrete X)}

```

These transformations are also applied to the above arguments of `lift`.

5.4 Extending Notations

It is possible to extend the notations for cpos and typable terms presented in the previous section with new constructions on cpos and continuous functions. Actually, it is useful not only to be able to extend with functions but with arbitrary constants and other terms in arbitrary cpos. We call an extension for a declaration. A constant must be declared before it is used in terms, otherwise the system does not know which cpo constructor it denotes or which cpo it is an element of. Similarly, if an arbitrary term does not fit within the notation for typable terms it must be declared before it can be used in other terms.

The tools presented in this section are the basic tools for introducing new declarations and thereby extending both the notation of cpos and the notation of typable terms. Using the derived definition tools presented in section 5.5, new constants are declared automatically when they are defined.

A cpo constructor is a constant applied to a number of cpo variables such that it has been shown that the constructor yields a cpo, provided its arguments are cpos. There are two ways of declaring a new cpo constructor, by a definition or by a constructor theorem.

```

#declare_cpo_definition;
- : (thm -> thm)

```

```

#declare_cpo_constructor;
- : (thm -> thm)

```

Both these programs return their theorem argument as a result. The first tool should be used when the right-hand side of the HOL definition of the constructor belongs to the notations for cpo or pointed cpos already, assuming all variables are cpos or pointed cpos (yielding theorems of the form presented in section 5.1.2). For instance, this is the case if the right-hand side is a discrete cpo as in this example

```

#Nat_DEF;
|- Nat = discrete UNIV

#declare_cpo_definition Nat_DEF;
|- Nat = discrete UNIV

```

where `UNIV` is the universal set of HOL numerals. Hence, this tool is used to introduce abbreviations.

The second tool is used to introduce new cpo constructors which are not part of the notations for cpos. It is applied to a theorem stating some constant is a cpo or pointed

cpo constructor. For instance, it can be used to introduce the cpo constructor of lazy sequences as follows (see section 4.2)

```
#SEQ_PCPO; ;  
|- !D. cpo D ==> pcpo(seq D)
```

```
#declare_cpo_constructor SEQ_PCPO; ;  
|- !D. cpo D ==> pcpo(seq D)
```

Note that `declare_cpo_definition` is applied to a HOL definition in contrast to the program `declare_cpo_constructor` which is applied to a theorem as in this example. The difference is that definitions are unfolded and folded by the parser and pretty-printer, respectively, such that internally the right-hand sides of definitions are used (see the comment in section 5.2.1). In some cases the basic proof tools need to know how such derived cpos are constructed.

Any term can be declared to belong to some cpo once this has been proved as a theorem. Such statements are called *ins facts* and they are introduced by the program `declare`.

```
#declare; ;  
- : (thm -> thm)
```

In particular, the term can be a function term in the cpo of continuous functions but not necessarily. Assuming `Nat` has been introduced as the discrete cpo of natural numbers, we may want to declare that zero is in the cpo,

```
#ZERO_NAT; ;  
|- 0 ins Nat
```

```
#declare ZERO_NAT; ;  
|- 0 ins Nat
```

or that addition is a continuous operation on the cpo.

```
#ADD_CF; ;  
|- $+ ins (cf(Nat, cf(Nat, Nat)))
```

```
#declare ADD_CF; ;  
|- $+ ins (cf(Nat, cf(Nat, Nat)))
```

Like the declaration tools above, `declare` also returns its theorem argument as a result.

If we wish to declare a continuous function parameterized by cpos then `declare` is not always useful since theorems declared using `declare` are not instantiated by the system. As an example, consider the `when` eliminator functional for sequences (see section 4.2)

```
#Seq_when_CF; ;  
|- !D E.  
    cpo D ==>  
    pcpo E ==>  
    (Seq_whenI (D, E)) ins (cf(cf(D, cf(seq D, E)), cf(seq D, E)))
```

which is parameterized by two cpos, corresponding to the variables D and E in this continuity theorem. If we use `declare` we must fix the parameters first, for instance, as the variables D and E ,

```
#declare(UNDISCH_ALL(SPEC_ALL Seq_when_CF));
.. |- (Seq_whenI (D, E)) ins (cf(cf(D, cf(seq D, E)), cf(seq D, E)))
```

and it will then only be possible to use the eliminator in cases where it is exactly D and E which are the appropriate parameters.

The following program called `declare_constructor` is more useful to introduce such constructors that are parameterized by the cpo variables of the domains on which they work.

```
#declare_constructor Seq_when_CF;
|- !D E.
  cpo D ==>
  pcpo E ==>
  Seq_when ins (cf(cf(D, cf(seq D, E)), cf(seq D, E)))
```

This program introduces a certain new constant (or reuses an existing constant with the right name if one exists) for the interface level of syntax automatically, in this case this would be `Seq_when`, and makes sure that the parser transforms this to the internal constructor and inserts cpo parameters. The pretty-printer is also made aware of the new constructor. Hence, if pretty-printing is switched on then the external version of the constructor is printed as in the example above.

Often `declare` is used to make the parser work properly for some term containing ‘unspecified’ variables in the sense that it contains variables which are not bound by dependent lambda abstractions. Such variables need only be declared temporarily so it is advantageous to be able to get rid of declarations. The following program called `undeclare` removes the last declaration introduced by `declare` and returns the corresponding theorem.

```
#undeclare;
- : (void -> thm)
```

There has been no need for similar ‘undeclaration’ programs for the other declaration tools, only variables are introduced temporarily.

5.5 Derived Definition Tools

In this section we present a few derived definition tools which combine the basic HOL definition program, called `new_definition`, with the declaration tools and the syntactic-based proof functions presented above.

First we present a program which can be used to introduce new cpos (and pointed cpos) by definition, using `declare_cpo_definition` and the cpo prover behind the scenes.

```
#new_cpo_definition;
- : (string -> string -> thm list -> term -> (thm # thm))
```

For example, the cpo of natural numbers can be introduced as follows

```
#let Nat_DEF, NAT_CPO = new_cpo_definition 'Nat_DEF' 'NAT_CPO' []
#"Nat = discrete(UNIV:num->bool)";;
Nat_DEF = |- Nat = Nat
NAT_CPO = |- cpo Nat
```

Again the theorem list argument is used mainly when the definition contains cpo variables, in order to provide assumed facts stating these variables are cpos. Note that `Nat` has already been declared when the definition is pretty-printed since the definition which internally corresponds to the term argument of `new_cpo_definition` has been folded. The cpo fact is proved using the cpo prover. Since such cpos introduced by definition are expanded internally this theorem is not used by the system. It is proved merely as a check that the right-hand side indeed is a cpo. Therefore, even if the right-hand side had been a pointed cpo it would still only be proved to be a cpo, but it could be used as a pointed cpo.

It makes no sense to have a similar derived tool based on `declare_cpo_constructor` since this is used in situations where the cpo fact about the new constant cannot be proved automatically using the cpo prover. Hence, in such situations the HOL definition tool and this declaration tool are applied manually.

If the domain of the right-hand side of a definition can be inferred automatically using the type checker, i.e. if it fits the notation of typable terms, then the following derived definition tool can be used to introduce the corresponding constant.

```
#new_constant_definition;;
- : (string -> string -> thm list -> term -> (thm # thm))
```

It applies the type checker, defines a new constant and declares the constant using `declare`. The theorem list argument may contain assumptions about cpo variables of the definition. It is unlikely that any `ins` facts are needed, because if they were then the parser would fail (or produce dummy domains such that the type checker would fail). For instance, it can be used to introduce a strict addition as follows

```
#declare(ins_prover "$+ ins cf(Nat,cf(Nat,Nat))");;
|- $+ ins (cf(Nat,cf(Nat,Nat)))

#let Add_DEF, Add_CF = new_constant_definition 'Add_DEF' 'Add_CF' []
#"Add = Ext(\n :: Dom Nat. Ext(\m :: Dom Nat. Lift(n+m)))";;
Add_DEF = |- Add = Ext(\n :: Dom Nat. Ext(\m :: Dom Nat. Lift(n + m)))
Add_CF = |- Add ins (cf(Lift Nat,cf(Lift Nat,lift Nat)))
```

assuming `Nat` has been introduced as above. A similar tool based on `ins_prover` has not been implemented since `ins_prover` is most often used with constants that have already been defined and reasoned about in HOL and then using `declare` and `ins_prover` as above works fine (see the examples in chapter 7 and chapter 8). However, there is a similar tool to introduce new constructors.

```
#new_constructor_definition;;
- : (string -> string -> thm list -> term -> (thm # thm))
```

This is based on `declare_constructor` instead of `declare` and hence, it is more useful when the right-hand side contains free cpo variables such that the constant becomes parameterized by these domains. The last letter of the constant should be an `l` (for internal) since the tool introduces a constant for both the internal and the external level of syntax.

5.6 Other Tools

Below we describe a few of the other tools that were used for the examples presented later and developed on top of the formalization.

5.6.1 Universal Cpos

The type checker can only be used when the domain of some term can be calculated from its subterms. This is not always the case; for instance, we cannot deduce that addition on the natural numbers is in the cpo of continuous functions from its subterms (it is a constant so it does not have any subterms and its definition does not fit the above notation for typable terms either). However, in certain cases it is trivial to prove a term is in a cpo, namely if the cpo is a universal cpo which contains all elements of the underlying HOL type. Often new cpos are introduced as the discrete cpo of the universal set of elements of some HOL type. Such cpos are particularly easy to work with since the cpo constructions for products, sums and continuous functions all preserve the property of being a discrete universal cpo. The lifting of a cpo is not discrete but it is universal if the argument cpo is. These facts are exploited by the following proof function, called the `ins-prover`.

```
#i ns_prover;;
- : conv
```

The cpo of natural numbers is a discrete universal cpo so addition can be proved to be continuous by this program automatically.

```
#i ns_prover "$+ ins cf(Nat, cf(Nat, Nat))";;
|- $+ ins (cf(Nat, cf(Nat, Nat)))
```

This fact can then be declared to the system using `declare`. The following increment function is also continuous.

```
#i ns_prover "($+ 1) ins cf(Nat, Nat)";;
|- ($+ 1) ins (cf(Nat, Nat))
```

There is no need to use the type checker in this case. Note that the `ins-prover` is syntactic-based like the type checker but in a different way. The `ins-prover` looks at the way cpos are constructed, unlike the type checker which looks at the way in which a term element of a cpo is built.

As long as the specified cpo of the argument of `ins_prover` can be transformed into a discrete universal cpo the specified term can be anything, as stated by the following theorem

```
|- !x. x ins (di screte UNIV)
```

on which the implementation of `ins_prover` is based. The input of the `ins-prover` is a term of the form "`t ins D`". By equality preserving transformations, it attempts to transform `D` into a discrete universal cpo. If it succeeds, then it instantiates the universally quantified variable `x` of the above theorem to `t`, and returns `|- t ins D` (otherwise it fails). Thus, unlike the type checker, it does not look at the structure of `t` at all, `t` can be any term. It is therefore much more efficient.

The transformation of cpos into discrete universal cpos is supported by the following theorems

```
|- prod(discrete UNIV, discrete UNIV) = discrete UNIV
|- cf(discrete UNIV, discrete UNIV) = discrete UNIV
|- sum(discrete UNIV, discrete UNIV) = discrete UNIV
```

In addition to the theorems listed above, the `ins-prover` exploits the following two theorems:

```
|- !t. t ins (lift(discrete UNIV))
|- !f. f ins (cf(discrete UNIV, lift(discrete UNIV)))
```

Hence, it automatically proves that terms are in lifted discrete universal cpos and in the partial function space of two discrete universal cpos. Note, by the way, that different occurrences of constants like `discrete` and `UNIV` may have different types in a term.

5.6.2 Reduction by Definition

It is often necessary to expand the definitions of continuous functions and to use β -reduction with respect to the dependent lambda abstraction afterwards. The reduction theorems for the function constructions presented in section 3.6 are useful for doing this in one blow but they require that arguments of function constructors are in the right domains. This can be proved using the type checker but it is tedious to apply the type checker and the reduction theorems manually. Hence, a constructor reduction conversion has been implemented which reduces one constructor at a time, proving the necessary `ins` facts automatically.

```
#CONS_REDUCE_CONV; ;
- : (thm list -> thm list -> thm list -> conv)
```

It takes three theorem list arguments. As usual the first two lists must contain additional cpo and `ins` facts and in this case the third list supports extending the reduction conversion with other reduction theorems than the built-in ones for the standard constructions on functions. For instance, the reduction theorems for the sequence when eliminator could be supplied here (see section 4.2).

```
#SEQ_WHEN_BT_SEQ; ;
|- !D E h. h ins (cf(D, cf(seq D, E))) ==> (Seq_when h Bt_seq = bottom E)
```

```
#SEQ_WHEN_CONS_SEQ; ;
|- !D E h x s.
    h ins (cf(D, cf(seq D, E))) ==>
```

```

x ins D ==>
s ins (seq D) ==>
(Seq_when h(Cons_seq x s) = h x s)

```

```

#CONS_REDUCE_CONV[ ] [ ] [SEQ_WHEN_BT_SEQ; SEQ_WHEN_CONS_SEQ]
#"Seq_when(\n :: Dom Nat. \s :: Dom(seq Nat). Cons_seq n s)Bt_seq";;
|- Seq_when(\n :: Dom Nat. \s :: Dom(seq Nat). Cons_seq n s)Bt_seq =
  bottom(seq Nat)

```

```

#CONS_REDUCE_CONV[ ] [ ] [SEQ_WHEN_BT_SEQ; SEQ_WHEN_CONS_SEQ]
#"Seq_when
# (\n :: Dom Nat. \s :: Dom(seq Nat). Cons_seq n s)(Cons_seq 0 Bt_seq)";;
|- Seq_when
  (\n :: Dom Nat. \s :: Dom(seq Nat). Cons_seq n s)(Cons_seq 0 Bt_seq) =
  (\n :: Dom Nat. \s :: Dom(seq Nat). Cons_seq n s)0 Bt_seq

```

Note that only one reduction is performed and only on the whole term, it does not attempt to traverse terms to do a reduction. The usual conversion combining operators (see [GM93]) can be used to repeat the conversion and apply it in depth. There is also a similar conversion for applying each individual constructor one by one and new such conversions can be obtained by instantiating an ML function with the desired reduction theorem. Tactic versions of the conversions have also been defined which use the assumptions to obtain the cpo and ins facts argument lists. Most importantly:

```

#CONS_REDUCE_TAC; ;
- : (thm list -> thm list -> thm list -> tactic)

```

which reduces a goal as much as possible repeatedly using the cpo facts and the ins facts in the assumptions of the goal and the first two theorem list arguments. The third argument is a list of constructor reduction theorems.

5.6.3 Fixed Point Induction

Next, we present the fixed point induction tactic which is implemented on top of the fixed point induction theorem presented in section 3.8.

```

FPI_TAC : thm list -> thm list -> thm list -> term -> tactic

```

```

"P[Fix f]"
===== [pcpo E,
"P[bottom E]"          f ins cf(E, E), inclusive(mk_pred(D, P), E)]
"!x. P[x] ==> P[f x]"

```

The side conditions are checked automatically using the syntactic-based tools described above (if these fail then the underlying theorem must be applied manually). The theorem list arguments are a list of cpo facts, ins facts and non-empty facts (for the inclusive prover). There is also a basic fixed point induction tactic

```

FPI_BASIC_TAC : term -> tactic

```

which generates the side conditions as subgoals. The term argument must be of the form "Fix f" in both cases.

5.6.4 Cases on Lifted Cpos

It is often useful to do a case split on whether or not an element of a lifted cpo is bottom. For this, the following theorem is provided:

```
#LIFT_CASES;;  
|- !D d. d ins (lift D) ==> (d = Bt)  $\vee$  (?d'. d' ins D  $\wedge$  (d = lift d'))
```

and based on this theorem, the following tactic:

```
# LIFT_CASES_TAC;;  
- : thm_tactic
```

The theorem argument corresponds to the antecedent of the theorem above and the tactic generates two subgoals and substitutes the equalities of the cases theorem in the goals and their assumptions.

5.6.5 Calculating Bottom in the Function Space

Often it is necessary to make use of the fact that the bottom of a continuous function space is the constant function which is always bottom:

```
#BOTTOM_CF;;  
|- !D1.  
    cpo D1 ==>  
    (!D2. pcpo D2 ==> (bottom(cf(D1,D2)) = ( $\lambda$ x :: Dom D1. bottom D2)))
```

A conversion has therefore been implemented to apply this theorem and to prove the antecedents automatically using the cpo and pcpo provers.

```
#BOTTOM_CF_CONV;;  
- : (thm list -> conv)
```

The theorem list argument is a list of additional cpo facts. The conversion is not recursive and only works at the top level of terms. As a further simplification, which many tools do in fact, it uses the theorem

```
#BOTTOM_LIFT;;  
|- !D. cpo D ==> (bottom(lift D) = Bt)
```

to yield the bottom of lifted cpos. Hence, this theorem is used to to reduce "bottom D2" in the previous theorem, when possible.

5.6.6 Function Equality

Finally, we present a conversion and a tactic for proving equality of continuous functions between the same cpos.

```
#X_CONT_FUN_EQ_CONV;;  
- : (thm list -> thm list -> term -> conv)
```

```
#X_CONT_FUN_EQ_TAC;;  
- : (thm list -> thm list -> term -> tactic)
```

Continuous function equality is extensional equality so two continuous functions are equal if they are equal for all elements of the domain cpo.

```
|- !D1 D2 f g.  
  f i ns (cf(D1, D2)) ==>  
  g i ns (cf(D1, D2)) ==>  
  ((f = g) = (!x. x i ns D1 ==> (f x = g x)))
```

The conversion is based on this theorem and employs the type checker to get rid of the antecedents. The theorems lists are the usual ones and the first term argument is the desired name of the universally quantified variable in the result.

Chapter 6

Some Simple Examples

In this chapter we shall consider a number of simple examples to illustrate the use of the HOL-CPO system described in chapter 5 and based on the formalization described in chapter 3 (and chapter 4 in part). The examples illustrate what is proper use and what is not, and include a lot of HOL code to make as explicit as possible how the system is used and what the system does. It is shown both how new cpos and continuous functions are introduced (and declared) and how theorems about the definitions can be proved. In particular, the examples illustrate to what extent the system manages to hide the underlying formalization of domain theory from the user. In one case only, it is necessary to turn to the definition of continuity and use deep properties of the formalization. Hence, the user does not need to know the precise definitions of domain theoretic concepts in most cases; only the consequences of having these concepts available are important to know, e.g. that we can define and reason about arbitrary recursive definitions and non-termination. Hence, I claim that a HOL user (or someone familiar with HOL) who has read the previous chapter can read this chapter without knowing the precise details of chapter 3 where the formalization is presented.

This chapter is the first one in a series of chapters containing examples so it will provide more explanations and HOL-CPO details than the following chapters. A lot of sessions are shown where constants are defined or declared or where theorems are proved. The reader should be aware that one particular session may rely on earlier sessions.

6.1 Booleans and Conditionals

In this section we present two different ways of defining a cpo of booleans and the associated conditional. The cpo must contain two distinct elements for truth and falsity respectively and the elements should only be related to themselves with respect to the ordering relation on the cpo. One cpo is obtained using the sum construction and the other is obtained using a discrete universal cpo of HOL booleans.

6.1.1 A Sum Cpo of Truth Values

A cpo of booleans can be defined as a sum cpo of two copies of a singleton cpo, interpreted as the cpos of truth and falsity respectively (as described in section 8.3.5 of [Wi93]). The singleton cpo is obtained as the discrete cpo of just one element, hence the underlying

type is the HOL type `": one"`. We can use `new_cpo_definition` to introduce this cpo as follows

```
#let One_DEF, One_CPO = new_cpo_definition 'One_DEF' 'One_CPO' []
# "One = discrete(UNIV: one->bool)";;
One_DEF = |- One = One
One_CPO = |- cpo One
```

Executing this program defines a new constant `One` and declares this as a cpo constructor by its definition. Hence, we can use this constant as a cpo in terms without stating each time that it is a cpo. Recall this was implemented by the parser which expands the definition of `One` such that internally the right-hand side of the above definition appears. Similarly the pretty-printer folds the definition which is the reason why the definition above is printed as `|- One = One`. Thus, the cpo fact proved by the definition tool is actually not used; it is proved merely as a syntactic check that the right-hand side meets the cpo notation presented in section 5.1. Only definitions of cpos are unfolded and folded in this way, not definitions of continuous functions or other terms. The string arguments of the program are the names under which the definition and the cpo theorem are stored and the (empty) theorem list argument is used in general to assume that cpo variables which occur in the left-hand side of the definition are cpos. These assumptions will appear in the cpo fact proved by the program.

The cpo of booleans can now be defined using the sum construction and the cpo constructor `One` just defined. Again `new_cpo_definition` is used, in exactly the same way as above.

```
#let bool_DEF, bool_CPO = new_cpo_definition 'bool_DEF' 'bool_CPO' []
# "bool = sum(One, One)";;
bool_DEF = |- bool = bool
bool_CPO = |- cpo bool
```

This cpo has two elements, namely `"INL one"` and `"INR one"` respectively. The constant `one` is the single inbuilt element of type `": one"` (constants and types may have the same names in HOL). We can now interpret this left element of the sum cpo as the truth value `true` and the right element as the truth value `false`.

```
#let true_DEF = new_definition('true_DEF', "true = INL one:one+one");;
true_DEF = |- true = INL one
```

```
#let false_DEF = new_definition('false_DEF', "false = INR one:one+one");;
false_DEF = |- false = INR one
```

These new constants are introduced by the standard HOL definition tool and it is still left to be proved that there is a connection between these new constants and elements of the cpo `bool`. This connection is established by proving the constants are indeed elements of `bool` and then declaring these facts to the system.

```
#declare(INS_prover "true ins bool");;
|- true ins bool
```

```
#declare(INS_prover "false ins bool");;
|- false ins bool
```

The ins-prover can be used to prove the constants are booleans since `bool` is a discrete universal cpo which in turn is true because the sum of discrete universal cpos is a discrete universal cpo (see section 5.6.1). The consequence of declaring the facts returned by the ins-prover, and returned by the declaration program in turn, is that the notation for typable terms is extended with the constants `true` and `false` (see section 5.4). Hence, the parser and the type checker know these constants and we can use the constants in terms like `"Lift false"` (a lifted boolean for falsity). If `false` was not declared this would not parse correctly, as the following session shows, since the parser would not be able to deduce the right cpo parameter of `Liftl`, the internal version of `Lift` (see section 5.2).

```
#"Lift false";;
"Lift false" : term

#unset_ppfold();;
() : void

#"Lift false";;
"Liftl bool false" : term

#undeclare();;
|- false ins bool

#"Lift false";;
"Liftl D false" : term

#set_pp();;
() : void
```

The program `unset_ppfold` switches off the pretty-printer but keeps the folding of cpo definitions. The program `set_pp` switches pretty-printing on again. The right cpo parameter of `Liftl` is the constant `bool` or in fact, this constant expanded by its definition. The wrong parameter is the (dummy) cpo variable `D`. In order to check whether a term has been parsed correctly it is often easier to look at the free variables of a term using the program `frees` than to unset pretty-printing and look at the entire term. Hence, continuing the session above we can perform the `frees` check as follows.

```
#"Lift false";;
"Lift false" : term

#frees"Lift false";;
["D"] : term list

#declare(ins_prover "false ins bool");;
|- false ins bool

#frees"Lift false";;
[] : term list
```

In the term list returned by `frees` one should look for any unexpected variables, typically `D`, `D1`, `D2`, etc., or `E`, `E1`, etc.

Next, we introduce a continuous conditional for this cpo of booleans. It takes three arguments, one boolean and two terms in an arbitrary cpo `D` corresponding to each branch of the conditional. This arbitrary cpo is present in the definition of the conditional as a cpo variable by which the conditional is parameterized. Hence, it is a parameterized constructor similar to the constructors presented in section 3.6 and we therefore best introduce it using the program `new_constructor_definition` (see section 5.5).

```
#cpo1;;
": (*1 -> bool) # (*1 -> (*1 -> bool))" : type

#|let cond_DEF,cond_CF = new_constructor_definition' cond_DEF' ' cond_CF'
# [ASSUME"cpo(D: ^cpo1)"]
# "cond1 D =
#   \ (b, t, t') :: Dom(prod(bool, prod(D: ^cpo1, D))).
#   Sum(\x :: Dom One. t), (\x :: Dom One. t'))b";;
cond_DEF =
|- !D.
  cond =
  (\ (b, t, t') :: Dom(prod(bool, prod(D, D))).
    Sum(\x :: Dom One. t), (\x :: Dom One. t'))b)
cond_CF = |- !D. cpo D ==> cond ins (cf(prod(bool, prod(D, D)), D))
```

This introduces a continuous conditional as a parameterized function constructor. This means that the conditional is introduced in two versions, one called `cond1` for the internal level of syntax and another called `cond` for the external level of syntax. The difference is that `cond` does not take cpo parameters whereas `cond1` does. When we use `cond` in terms it is transformed by the parser to the internal version and cpo parameters are inserted automatically. The pretty-printer does the opposite, it removes the cpo parameters and prints `cond` instead of `cond1`. Hence, `cond` appears in the pretty-printed results above. The ML variable `cpo1` is bound to a polymorphic type of cpo pairs and is used as the type of the cpo variable `D` which appears in the definition. The arguments of `new_constructor_definition` is similar to the arguments of `new_cpo_definition` used above. Note, we assume that the cpo variable `D` is a cpo. This is not necessary for the definition but for the proof that the new constant introduced is a continuous function (i.e. is in the cpo of continuous functions). The assumption appears in this theorem.

The above definition is complicated by the fact that the `Sum` constructor function takes two arguments which are functions from each component of the boolean sum cpo to the arbitrary cpo `D`. This introduces the ‘unnecessary’ lambda abstractions over the variable `x`. The following reduction theorem is easier to understand and it states how the conditional works on elements of the right cpos.

```
cond_REDUCE_THM =
|- (!D t t'. cpo D ==> t ins D ==> t' ins D ==> (cond(true, t, t') = t)) /\
  (!D t t'. cpo D ==> t ins D ==> t' ins D ==> (cond(false, t, t') = t'))
```

This theorem has a simple proof

```

REPEAT STRIP_TAC
THEN REWRITE_TAC[cond_DEF; true_DEF; false_DEF]
THEN CONS_REDUCE_TAC[] [] []
THEN REFL_TAC

```

which breaks the statement down (assuming left-hand sides of implications), expands definitions, reduces away the constructors by the reduction theorems and finishes off the proof using reflexivity; both conjuncts of the statement are of the form "x = x" at this point. The assumption about the variable `D` is needed in order to prove the function arguments of `Sum` are continuous before `Sum` is reduced. This `cpo` variable is the codomain of these functions and hence it must be a `cpo` to allow the functions to be continuous.

We shall not consider this `cpo` of booleans further. Instead we turn our attention towards another one which is based on the HOL type of booleans.

6.1.2 A Discrete Universal Cpo of HOL Booleans

Above we defined a `cpo` of booleans using the sum construction on `cpo`s. This was simple and it works well too but we do not have any operations on this `cpo` of booleans, like e.g. conjunction and negation. These must be introduced separately. If we choose instead to introduce a `cpo` of booleans as the discrete universal `cpo` of HOL boolean values then we inherit all operations on booleans from HOL and these are automatically continuous. Besides, we can exploit all built-in theorems and tools for the HOL booleans, e.g. the tautology checker (see the `taut` library of HOL [Bo91]). The disadvantage of this approach is that the conditional is much more difficult to introduce than above; but it is equally easy to use.

The `cpo` of booleans is introduced in exactly the same way as we introduced the `cpo`s above, using the definition tool for introducing derived `cpo` definitions.

```

#let Bool_DEF, Bool_CPO = new_cpo_definition 'Bool_DEF' 'Bool_CPO' []
# "Bool = discrete(UNIV: bool -> bool)";;
Bool_DEF = |- Bool = Bool
Bool_CPO = |- cpo Bool

```

The important advantage of using discrete universal `cpo`s, in this example as well as in general, is that we inherit all operations on the underlying types which become continuous operations on the `cpo`s. Thus, for instance, conjunction and negation are continuous operations.

```

#declare(ins_prover "$/\ ins cf(Bool, cf(Bool, Bool))");;
|- $/\ ins (cf(Bool, cf(Bool, Bool)))

```

```

#declare(ins_prover "$~ ins cf(Bool, Bool)");;
|- $~ ins (cf(Bool, Bool))

```

Besides of course the HOL boolean truth values `T` and `F` are in this `cpo` of booleans.

```

#declare(ins_prover "T ins Bool");;
|- T ins Bool

```

```
#declare(ins_prover "F i ns Bool");;
|- F i ns Bool
```

Since any element of this cpo is a term of type ": bool " in HOL we can use the built-in theorems, inference rules and other programs to reason about elements of the cpo of booleans.

This is all very good. We have obtained many operations on booleans for free and a lot of proof infrastructure is inherited as well. However, it is not as easy to introduce the conditional for this cpo of booleans. It is easy enough to define it, just let it be a determined version of the built-in conditional COND

```
|- !D.
  Cond1 D = (\(x, y, z) :: Dom(prod(Bool , prod(D, D))). (x ==> y | z))
```

where " $(x ==> y | z)$ " is HOL's syntactic sugar for "COND x y z". The problem is instead to prove it is continuous; this cannot be proved by the ins-prover due to the use of the cpo variable D. Besides, continuity does not follow automatically as in the previous section since the body of the lambda abstraction in the above definition does not fit within the notation of typable terms. In the previous section the conditional was introduced using the Sum construction on continuous functions and therefore continuity could be proved automatically there.

Here, we must prove the conditional is continuous from the definition of continuity manually. In fact, this is not really difficult, but it requires that we use the formalization of domain theory presented in chapter 3 more extensively than we really want to. We must use theorems stating, for instance, that lubs of chains in the product cpo are calculated componentwise and that lubs of chains in discrete cpos are just the first elements of the chains since these chains are all constant (contain one element only). The theorem of continuity of the conditional is stated as follows

```
Cond_CF = |- !D. cpo D ==> (Cond1 D) i ns (cf(prod(Bool , prod(D, D)), D))
```

We can use this to declare the conditional as a constructor function.

```
#declare_constructor Cond_CF;;
|- !D. cpo D ==> Cond i ns (cf(prod(Bool , prod(D, D)), D))
```

The external version of the conditional called Cond is introduced automatically here (it is just a 'place holder' for Cond1, i.e. an arbitrary element of the appropriate type without a specific definition).

Finally, the obligatory reduction theorems:

```
Cond_REDUCE_THM =
|- !D x y b. x i ns D ==> y i ns D ==> (Cond(b, x, y) = (b ==> x | y))
```

```
Cond_T_REDUCE_THM =
|- !D x y. x i ns D ==> y i ns D ==> (Cond(T, x, y) = x)
```

```
Cond_F_REDUCE_THM =
|- !D x y. x i ns D ==> y i ns D ==> (Cond(F, x, y) = y)
```

We have chosen to provide three theorems of which the last two are obtained easily from the first one. These last two theorems can be used instead of the first one to avoid simplifying the HOL conditional after a reduction has been performed. Examples of the use of the reduction theorems with the program `CONS_REDUCE_TAC` are provided in section 6.3.

6.2 Natural Numbers

In the previous section we saw it was straightforward to introduce a cpo of HOL boolean truth values using the discrete construction. The operations on HOL booleans were inherited and proved to be continuous on the new cpo automatically. We can use exactly the same approach to introduce a cpo of natural numbers and the associated continuous operations for addition, subtraction, multiplication, predecessor, and so on.

The cpo of natural numbers is introduced as the discrete universal cpo of all elements of the HOL type `" : num"` of numerals (the same as natural numbers).

```
#let Nat_DEF, Nat_CPO = new_cpo_definition 'Nat_DEF' 'Nat_CPO' []
# "Nat = discrete(UNIV: num->bool)";;
Nat_DEF = |- Nat = Nat
Nat_CPO = |- cpo Nat
```

Hence, all terms of this type are in the cpo trivially. In particular, the constants `0` and `1`, which we shall use below, are in `Nat`.

```
#declare(INS_prover "0 ins Nat");;
|- 0 ins Nat

#declare(INS_prover "1 ins Nat");;
|- 1 ins Nat
```

Similarly, the built-in addition, subtraction and multiplication operations on numerals are continuous operations on the cpo of natural numbers.

```
#declare(INS_prover "$+ ins (cf(Nat, cf(Nat, Nat)))");;
|- $+ ins (cf(Nat, cf(Nat, Nat)))

#declare(INS_prover "$- ins (cf(Nat, cf(Nat, Nat)))");;
|- $- ins (cf(Nat, cf(Nat, Nat)))

#declare(INS_prover "$* ins cf(Nat, cf(Nat, Nat))");;
|- $* ins (cf(Nat, cf(Nat, Nat)))
```

The function `INS_prover` proves its term argument is a theorem by transforming the right-hand side of `INS` to a discrete universal cpo. Then the left-hand side is trivially an element of the cpo. The transformation works above because the function space of discrete universal cpos is itself a discrete universal cpo.

In an example below we define a denotational semantics of a simple language for writing recursive functions on natural numbers where addition, subtraction and multiplication are

strict operations on the lifted cpo of natural numbers (see section 6.4). Strict versions of the above built-in constants are obtained easily using the construction, called function extension, for extending a function to a lifted cpo in a strict way (see section 3.6.4).

```
Add_DEF = |- Add = Ext(\n :: Dom Nat. Ext(\m :: Dom Nat. Lift(n + m)))
Add_CF = |- Add ins (cf(lift Nat, cf(lift Nat, lift Nat)))
```

```
Sub_DEF = |- Sub = Ext(\n :: Dom Nat. Ext(\m :: Dom Nat. Lift(n - m)))
Sub_CF = |- Sub ins (cf(lift Nat, cf(lift Nat, lift Nat)))
```

```
Mult_DEF = |- Mult = Ext(\n :: Dom Nat. Ext(\m :: Dom Nat. Lift(n * m)))
Mult_CF = |- Mult ins (cf(lift Nat, cf(lift Nat, lift Nat)))
```

All three continuous functions are introduced in the same way using the derived definition program `new_constant_definition`, which proves continuity behind the scenes. This program can be used since the right-hand sides of the definitions fit within the notation for typable terms. The declarations above, of the built-in operations, extends this notation and they are necessary both for the proofs of continuity and for the function definitions to parse too. For instance, subtraction is introduced as follows:

```
#let Sub_DEF, Sub_CF = new_constant_definition' Sub_DEF' ' Sub_CF' []
# "Sub = Ext(\n :: Dom Nat. Ext(\m :: Dom Nat. Lift(n-m)))";;
Sub_DEF = |- Sub = Ext(\n :: Dom Nat. Ext(\m :: Dom Nat. Lift(n - m)))
Sub_CF = |- Sub ins (cf(lift Nat, cf(lift Nat, lift Nat)))
```

Recall from section 5.5 that this definition program defines a new constant and then derives a fact stating which cpo it is an element of using the type checker. Finally, this fact is declared to the system using `declare` behind the scenes.

In section 6.4 we shall also use a strict test for whether a term is zero. In order to introduce this test we shall exploit the cpo of booleans introduced in section 6.1.2 above. First we introduce a test by a standard HOL definition:

```
|- iszero = (\n :: Dom Nat. Lift(n = 0))
```

This can be declared to be a continuous function using `declare` and `ins_prover` as usual.

```
#declare(ins_prover "iszero ins cf(Nat, lift Bool)");;
|- iszero ins (cf(Nat, lift Bool))
```

The ins-prover can be used since the cpo `"cf(Nat, lift Bool)"` is a universal cpo, even though it is not discrete. This is the only kind of universal non-discrete cpo that the ins-prover can handle at the moment. Then a strict test is introduced using `new_constant_definition` as above.

```
#let Iszero_DEF, Iszero_CF = new_constant_definition
# 'Iszero_DEF' 'Iszero_CF' []
# "Iszero = Ext iszero";;
Iszero_DEF = |- Iszero = Ext iszero
Iszero_CF = |- Iszero ins (cf(lift Nat, lift Bool))
```

Note how simple the definition is. Since `iszero` already returns a lifted boolean it can be extended to a strict function simply using `Ext` and nothing more.

The built-in predecessor `PRE` is also a continuous function on the cpo of natural numbers. However, its definition in HOL may seem unsatisfactory to some people since it defines the predecessor of `0` to be `0`:

$$\vdash (\text{PRE } 0 = 0) \wedge (!m. \text{PRE}(\text{SUC } m) = m)$$

In domain theory we have the choice of defining a predecessor which is undefined (bottom) when it is applied to `0`.

```
pred_DEF = \n. pred = (\n :: Dom Nat. ((n = 0) => Bt | Lft(PRE n)))
```

This is introduced by an ordinary HOL definition and then declared to be continuous using `declare` and `ins_prover` in the usual way.

```
#declare(ins_prover "pred ins cf(Nat, lift Nat)");;
\pred ins (cf(Nat, lift Nat))
```

A strict predecessor can be obtained as above using `new_constant_definition`.

6.2.1 Reduction Theorems

The main aim of a reduction theorem is to reduce away as many of the function constructions (including lambda abstractions) in the right-hand side of the definition of a function as possible. Using such derived theorems makes the reduction of functions by definitions more efficient since the simplifications just mentioned have been done once and for all.

Let us consider a few of the reduction theorems for the operations introduced above. The theorems for addition, subtraction and multiplication are all similar, they have the following form

```
Add_REDUCE_THM =
\ ( !n. Add Bt n = Bt ) \
  ( !n. Add n Bt = Bt ) \
  ( !nn mm. Add(Lift nn)(Lift mm) = Lift(nn + mm))
```

Hence, they state the lifted operations are strict in both arguments and behave as the built-in operations on lifted arguments. Note that it is not necessary to assume the universally quantified variables are in the cpo of natural numbers or in the lifted cpo of natural numbers since these are both universal cpos. Thus, the reduction theorems can be used with rewriting tactics instead of with the reduction tactic; rewriting is of course much faster.

The proof of such facts proceed by expanding definitions and reducing using the reduction tactic called `CONS_REDUCE_TAC` (see section 5.6.2). A case split on the lifted natural number variable `n` using the tactic `LIFT_CASES_TAC` (see section 5.6.4) is necessary to prove the second conjunct, in order to get rid of the `Ext` construction. In section 6.2.3 below we describe the proof in full detail.

The operation of testing whether a term is zero was introduced in two versions. The first of these called `iszero` is a function from `Nat` to "`lift Nat`" which behaves as follows

```

i_szero_REDUCE_THM =
|- (i_szero 0 = Li ft T) /\ (!nn. i_szero(nn + 1) = Li ft F)

```

This theorem is stated more ‘precisely’ than `Add_REDUCE_THM` above, in the sense that it tells what the final result of a test is (as a ground/constant term), since testing for zero is so simple. It would complicate the above theorem considerably if we attempted to build in how the addition `$+` works. The proof of the theorem about `i_szero` uses rewriting with built-in facts like

```

#REWRITE_RULE[ADD1]PRE;;
|- (PRE 0 = 0) /\ (!m. PRE(m + 1) = m)

```

```

#REWRITE_RULE[ADD1]NOT_SUC;;
|- !n. ~(n + 1 = 0)

```

after the definition have been expanded and the function constructions have been reduced away.

The reduction theorem for the strict test `!szero` just states how it behaves on bottom and lifted elements.

```

!szero_REDUCE_THM =
|- (!szero Bt = Bt) /\ (!nn. !szero(Li ft nn) = i_szero nn)

```

From this and the previous reduction theorem the following derived equations can be obtained easily (using rewriting)

```

!szero_EQS =
|- (!szero Bt = Bt) /\
  (!szero(Li ft 0) = Li ft T) /\
  (!nn. !szero(Li ft(nn + 1)) = Li ft F)

```

It depends on the situation which of the two theorems are the most useful one for rewriting. Of course, if the latter can be applied, then this is the most useful one, since it is a special case of the former.

6.2.2 The Factorial Function

In a typical functional programming language like ML, the factorial function might be defined as follows

```

#letrec fac n = if n = 0 then 1 else n * (fac(n-1));;
fac = - : (int -> int)

```

Below we describe how this function (restricted to the natural numbers) can be defined in the framework presented above.

Using the extensions of the notations of `cpos` and `typable` terms presented in the previous sections we can define a recursive factorial function as a fixed point of some functional. Actually, we shall need one more extension to allow the use of equality on natural numbers (we do not use `i_szero` since it returns a lifted boolean).

```
#declare(INS_prover "$= ins cf(Nat, cf(Nat, Bool))");;
|- $= ins (cf(Nat, cf(Nat, Bool)))
```

The factorial we define in domain theory is a function from the natural numbers to the lifted natural numbers in order to allow to take the least fixed point. It is introduced using the derived definition program `new_constant_definition` described above.

```
#let Fac_DEF, Fac_CF = new_constant_definition' Fac_DEF' ' Fac_CF' []
#   "Fac =
#   Fix
#   (\f :: Dom(cf(Nat, lift Nat)).
#     \n :: Dom Nat.
#       Cond((n = 0), Lift 1, Ext(\m :: Dom Nat. Lift(n*m))(f(n-1))))";;
Fac_DEF =
|- Fac =
  Fix
  (\f :: Dom(cf(Nat, lift Nat)).
    \n :: Dom Nat.
      Cond((n = 0), Lift 1, Ext(\m :: Dom Nat. Lift(n * m))(f(n - 1))))
Fac_CF = |- Fac ins (cf(Nat, lift Nat))
```

Hence, introducing a recursive function is just as simple as introducing other functions and arbitrary terms. We use `Ext` in the definition since the term "`f(n-1)`" corresponding to the result of the recursive application of the factorial function is in the lifted cpo of natural numbers and multiplication `$*` works on natural numbers only. In order for the fixed point operator to work as desired and yield a least fixed point:

```
|- !E. pcpo E ==> (!f. f ins (cf(E, E)) ==> (f(Fix f) = Fix f))
```

the cpo `cf(Nat, lift Nat)` must be a pointed cpo, which it is. This is the reason why we use lifted natural numbers.

The reduction theorem we have proved for the factorial was derived using this fixed point property of `Fix`.

```
|- !nn.
  Fac nn =
  Cond
  ((nn = 0), Lift 1, Ext(\m :: Dom Nat. Lift(nn * m))(Fac(nn - 1)))
```

The theorem is a bit ugly and it could be reduced further by making a cases on whether `nn` is zero or not. However, we cannot simplify the second branch of the conditional easily because we do not know the result of the recursive call and therefore cannot reduce the `Ext` term. However, on actual natural number arguments we would be able to calculate the result of the factorial function, first as a sequence of multiplications and then as a natural number by simplifying these multiplications.

A simple recursive total function like the factorial can also be defined in pure HOL by a primitive recursive definition. Domain theory is not necessary and as a matter of fact only complicates its definition. However, later we shall see examples of recursive functions which are recursive in a non-trivial way. The definitions of such functions are not supported in HOL but can be defined easily using domain theory.

6.2.3 Proof of a Reduction Theorem

We describe the simple proof of the reduction theorem about addition started by setting the desired goal term as follows (cf. the subgoal package [GM93]).

```
#g "(!n. Add Bt n = Bt) /\
#  (!n. Add n Bt = Bt) /\
#  (!nn mm. Add(Li ft nn)(Li ft mm) = Li ft(nn+mm))";;
"(!n. Add Bt n = Bt) /\
(!n. Add n Bt = Bt) /\
(!nn mm. Add(Li ft nn)(Li ft mm) = Li ft(nn + mm))"
```

```
() : void
```

```
#frees(snd(top_goal ()));;
["D"] : term list
```

The `frees` test on the goal term tells us that something is wrong. The parser have not been able to deduce the `cpo` parameters of `Li ft` since it does not know that the variables `nn` and `mm` are elements of the `cpo Nat`. We declare this trivial fact and try setting the goal again.

```
#declare(ins_prover "mm ins Nat");;
|- mm ins Nat
```

```
#declare(ins_prover "nn ins Nat");;
|- nn ins Nat
```

```
#g "(!n. Add Bt n = Bt) /\
#  (!n. Add n Bt = Bt) /\
#  (!nn mm. Add(Li ft nn)(Li ft mm) = Li ft(nn+mm))";;
"(!n. Add Bt n = Bt) /\
(!n. Add n Bt = Bt) /\
(!nn mm. Add(Li ft nn)(Li ft mm) = Li ft(nn + mm))"
```

```
() : void
```

```
#frees(snd(top_goal ()));;
[] : term list
```

This time it worked. The proof is straightforward. We strip the goal apart and expand the definition.

```
#e(REPEAT STRIP_TAC THEN REWRITE_TAC[Add_DEF]);;
OK. .
```

```
3 subgoals
```

```
"Ext(\n :: Dom Nat. Ext(\m :: Dom Nat. Li ft(n + m)))(Li ft nn)(Li ft mm) =
Li ft(nn + mm)"
```

```
"Ext(\n :: Dom Nat. Ext(\m :: Dom Nat. Lift(n + m)))n Bt = Bt"
```

```
"Ext(\n :: Dom Nat. Ext(\m :: Dom Nat. Lift(n + m)))Bt n = Bt"
```

```
() : void
```

The third of the three goals (enumerated from top to bottom) must be proved first. The reduction tactic can be used to reduce the first `Ext` term since it is applied to `Bt`. This should give us the bottom of the codomain of the function that `Ext` is applied to. This codomain is the cpo "`cf(Lift Nat, Lift Nat)`" which indeed is a pointed cpo (otherwise the bottom would not exist).

```
#e(CONS_REDUCE_TAC[][][]);;
```

```
OK.
```

```
"(\x :: Dom(Lift Nat).
```

```
  ((x = Bt) =>
```

```
    bottom(cf(Lift Nat, Lift Nat)) |
```

```
    (\n :: Dom Nat. Ext(\m :: Dom Nat. Lift(n + m)))(unlift x)))
```

```
Bt
```

```
n =
```

```
Bt"
```

```
() : void
```

This looks terrible and is not what we expected! We forgot to provide the fact that `Bt` is in the cpo of lifted natural numbers. Back up and try again using this fact.

```
#b();;
```

```
3 subgoals
```

```
"Ext(\n :: Dom Nat. Ext(\m :: Dom Nat. Lift(n + m)))(Lift nn)(Lift mm) =  
  Lift(nn + mm)"
```

```
"Ext(\n :: Dom Nat. Ext(\m :: Dom Nat. Lift(n + m)))n Bt = Bt"
```

```
"Ext(\n :: Dom Nat. Ext(\m :: Dom Nat. Lift(n + m)))Bt n = Bt"
```

```
() : void
```

```
#let Bt_Nat = ins_prover "Bt ins Lift Nat";;
```

```
Bt_Nat = |- Bt ins (Lift Nat)
```

```
#e(CONS_REDUCE_TAC[][Bt_Nat][]);;
```

```
OK.
```

```
"bottom(cf(Lift Nat, Lift Nat))n = Bt"
```

```
() : void
```

Next we can calculate the bottom in this function space using `BOTTOM_CF_CONV` (see section 5.6.5).

```
#e(CONV_TAC(ONCE_DEPTH_CONV(BOTTOM_CF_CONV[]))));;
OK.
"(\x :: Dom(Lift Nat). Bt)n = Bt"
```

```
() : void
```

Reducing the lambda abstraction and using reflexivity solves the goal.

```
#e(CONS_REDUCE_TAC[][ins_prover"n ins Lift Nat"[][] THEN REFL_TAC]);;
OK.
goal proved
|- Lambda(Lift(discrete UNIV))(\x. Bt)n = Bt
|- bottom(cf(Lift(discrete UNIV), Lift(discrete UNIV)))n = Bt
|- Ext1
  (discrete UNIV, cf(Lift(discrete UNIV), Lift(discrete UNIV)))
  (Lambda
    (discrete UNIV)
    (\n.
      Ext1
        (discrete UNIV, Lift(discrete UNIV))
        (Lambda(discrete UNIV)(\m. Lift1(discrete UNIV)(n + m)))))
  Bt
  n =
  Bt
```

Previous subproof:

2 subgoals

```
"Ext(\n :: Dom Nat. Ext(\m :: Dom Nat. Lift(n + m)))(Lift nn)(Lift mm) =
Lift(nn + mm)"
```

```
"Ext(\n :: Dom Nat. Ext(\m :: Dom Nat. Lift(n + m)))n Bt = Bt"
```

```
() : void
```

Note that we have provided a fact for the reduction tactic stating n is in the cpo of lifted natural numbers for this to work (it is not in the database/notation of typable terms already). Also note that our pretty-printer does not work on the output theorems from the subgoal package (a choice of implementation) which therefore corresponds to the internal syntax used.

The proof of the next goal is similar but requires a case split on n in order to get rid of the first occurrence of `Ext`.

```
#e(LIFT_CASES_TAC(ins_prover"n ins Lift Nat"));;
OK.
2 subgoals
"Ext(\n :: Dom Nat. Ext(\m :: Dom Nat. Lift(n + m)))(Lift d')Bt = Bt"
  [ "d' ins Nat" ]
  [ "n = Lift d'" ]
```

```
"Ext(\n :: Dom Nat. Ext(\m :: Dom Nat. Lift(n + m)))Bt Bt = Bt"
  [ "n = Bt" ]
```

```
() : void
```

The cases tactic is based on the following theorem (see section 5.6.4)

```
#LIFT_CASES;;
|- !D d. d ins (Lift D) ==> (d = Bt) \\/ (?d'. d' ins D /\ (d = Lift d'))
```

The first of the two subgoals is similar to the previous goal proved and we can therefore use the same tactic (except that the fact `Bt_Nat` is used for the last reduction).

```
#e(CONS_REDUCE_TAC[][Bt_Nat][])
# THEN CONV_TAC(ONCE_DEPTH_CONV(BOTTOM_CF_CONV[]))
# THEN CONS_REDUCE_TAC[][Bt_Nat][]) THEN REFL_TAC);;
OK..
goal proved
.
.
.
```

Previous subproof:

```
"Ext(\n :: Dom Nat. Ext(\m :: Dom Nat. Lift(n + m)))(Lift d')Bt = Bt"
  [ "d' ins Nat" ]
  [ "n = Lift d'" ]
```

```
() : void
```

We have replaced the output theorems from the subgoal package by dots. Since the assumptions say that `d'` is in `Nat` we can finish off this goal as follows

```
#e(CONS_REDUCE_TAC[][Bt_Nat][]) THEN REFL_TAC);;
OK..
goal proved
.
.
.
```

Previous subproof:

```
"Ext(\n :: Dom Nat. Ext(\m :: Dom Nat. Lift(n + m)))(Lift nn)(Lift mm) =
  Lift(nn + mm)"
```

```
() : void
```

This is the final subgoal. It is finished off in the same way as the previous subgoals by the reduction tactic and reflexivity.

```

#e(CONS_REDUCE_TAC[][][] THEN REFL_TAC);;
OK.
goal proved
|- Ext1
  (discrete UNIV, cf(lift(discrete UNIV), lift(discrete UNIV)))
  (lambda
    (discrete UNIV)
    (\n.
      Ext1
        (discrete UNIV, lift(discrete UNIV))
        (lambda(discrete UNIV)(\m. lift1(discrete UNIV)(n + m))))))
  (lift1(discrete UNIV)nn)
  (lift1(discrete UNIV)mm) =
  lift1(discrete UNIV)(nn + mm)
|- (!n. Add Bt n = Bt) /\
  (!n. Add n Bt = Bt) /\
  (!nn mm.
    Add(lift1(discrete UNIV)nn)(lift1(discrete UNIV)mm) =
    lift1(discrete UNIV)(nn + mm))

```

Previous subproof:

```

goal proved
() : void

```

We do not need any facts about `mm` and `nn` in the `ins fact` argument list of the reduction tactic since these fact have been declared and therefore are part of the notation of typable terms already. We have proved

```

|- (!n. Add Bt n = Bt) /\
  (!n. Add n Bt = Bt) /\
  (!nn mm. Add(lift nn)(lift mm) = lift(nn + mm))

```

The reduction theorems for subtraction and multiplication have exactly the same form and proofs.

6.3 Using Fixed Point Induction

In this section we define a double recursive function as a fixed point and prove a theorem about the function by fixed point induction. This example is taken from [Sc86] (see section 6.6.3 and proposition 6.27). We assume the following sessions continues the previous ones such that all declarations are still valid.

The double recursive function which is called `gg` is defined as the fixed point of the following functional introduced by `new_constant_definition`.

```

gg_fun_DEF =
|- gg_fun =
  (\g :: Dom(cf(Nat, lift Nat)).
    \n :: Dom Nat.

```

```

Cond
((n = 0), Lift 1, Ext(\m :: Dom Nat. Lift((m + m) - 1))(g(n - 1)))
gg_fun_CF = |- gg_fun ins (cf(cf(Nat, lift Nat), cf(Nat, lift Nat)))

```

These two theorems were returned by this program which declares `gg_fun` such that it becomes part of the notation for typable terms. Hence, the function `gg` can be introduced as follows

```

#let gg_DEF, gg_CF = new_constant_definition 'gg_DEF' 'gg_CF' []
# "gg = Fix gg_fun";;
gg_DEF = |- gg = Fix gg_fun
gg_CF = |- gg ins (cf(Nat, lift Nat))

```

The constant `gg` is a recursive function which always returns one when it terminates.

```

|- !nn. ~(gg nn = Bt) ==> (gg nn = Lift 1)

```

In fact, it always terminates but we shall not prove that here. The rest of this section is devoted to describe the proof of this theorem using the subgoal package. The goal is stated as follows.

```

#g "!nn. ~(gg nn = Bt) ==> (gg nn = Lift 1)";;
"!nn. ~(gg nn = Bt) ==> (gg nn = Lift 1)"

```

```

() : void

```

In order to prove this goal using fixed point induction we must ensure it is an inclusive predicate as a function of a variable replacing `gg`. The inclusive prover can prove this if we add an antecedent stating `nn` is in the cpo of natural numbers.

```

#e(GEN_TAC THEN MP_TAC(ins_prover"nn ins Nat"))
# THEN_SPEC_TAC("nn: num", "nn: num");;
OK.
"!nn. nn ins Nat ==> ~(gg nn = Bt) ==> (gg nn = Lift 1)"

```

```

() : void

```

Expanding the definition of `gg` to yield a fixed point term and applying the basic fixed point induction tactic generates five subgoals.

```

#e(REWRITE_TAC[gg_DEF] THEN FPI_BASIC_TAC"Fix gg_fun");;
OK.
5 subgoals
"!x.
  x ins (cf(Nat, lift Nat)) ==>
  (!nn. nn ins Nat ==> ~(x nn = Bt) ==> (x nn = Lift 1)) ==>
  (!nn. nn ins Nat ==> ~(gg_fun x nn = Bt) ==> (gg_fun x nn = Lift 1))"
"!nn.
  nn ins Nat ==>

```

```

~(bottom(cf(Nat, lift Nat))nn = Bt) ==>
(bottom(cf(Nat, lift Nat))nn = Lift 1)"

```

```

"i ncl usi ve
(mk_pred
(cf(Nat, lift Nat),
(\GEN%VAR%3260.
!nn.
nn ins Nat ==>
~(GEN%VAR%3260 nn = Bt) ==>
(GEN%VAR%3260 nn = Lift 1))), cf(Nat, lift Nat))"

```

```

"gg_fun ins (cf(cf(Nat, lift Nat), cf(Nat, lift Nat)))"

```

```

"pcpo(cf(Nat, lift Nat))"

```

```

() : void

```

The first three ones (counting from the bottom) can be proved automatically using the pointed cpo prover, the tactic version of the type checker and the inclusive prover respectively. These are applied behind the scenes by the fixed point induction tactic `FPI_TAC` so let us back up the goal and use this tactic instead.

```

#b();;
"!nn. nn ins Nat ==> ~(gg nn = Bt) ==> (gg nn = Lift 1)"

```

```

() : void

```

```

#e(REWRITE_TAC[gg_DEF] THEN FPI_TAC[][Bt_Nat][]"Fix gg_fun");;

```

```

OK.

```

```

2 subgoals

```

```

"!x.
x ins (cf(Nat, lift Nat)) ==>
(!nn. nn ins Nat ==> ~(x nn = Bt) ==> (x nn = Lift 1)) ==>
(!nn. nn ins Nat ==> ~(gg_fun x nn = Bt) ==> (gg_fun x nn = Lift 1))"

```

```

"!nn.
nn ins Nat ==>
~(bottom(cf(Nat, lift Nat))nn = Bt) ==>
(bottom(cf(Nat, lift Nat))nn = Lift 1)"

```

```

() : void

```

Note that we must provide a theorem stating that `Bt` is in the cpo of lifted natural numbers. This theorem is used by the inclusive prover. The theorem list arguments of the fixed point induction tactic are used by the underlying proof tools. They consist of a theorem list of additional cpo facts, ins facts and non-empty facts about cpos used in universal quantifications for the inclusive prover (see section 5.3).

The proof proceeds as follows. The first subgoal is proved easily by first calculating the bottom in the cpo of continuous functions

```
#e(CONV_TAC(ONCE_DEPTH_CONV(BOTTOM_CF_CONV[])));;
OK.
"!nn.
  nn ins Nat ==>
  ~((\x :: Dom Nat. Bt)nn = Bt) ==>
  ((\x :: Dom Nat. Bt)nn = Li ft 1)"
```

() : void

and then reducing the lambda abstractions.

```
#declare(ins_prover "nn ins Nat");;
|- nn ins Nat

#e(CONS_REDUCE_TAC[][][]);;
OK.
"!nn. nn ins Nat ==> ~(Bt = Bt) ==> (Bt = Lft 1)"
```

() : void

Actually, it is enough to reduce the first one but the reduction tactic reduces both. We could instead use the reduction conversion and other conversions to make sure it is applied at the right spot. The reduction tactic does not need to know via the theorem list argument that `nn` is in `Nat` since this is declared to the system for later use too. The resulting subgoal can be proved using rewriting.

```
#e(RT[]);;
OK.
goal proved
.
.
.
```

Previous subproof:

```
"!x.
  x ins (cf(Nat, Li ft Nat)) ==>
  (!nn. nn ins Nat ==> ~(x nn = Bt) ==> (x nn = Li ft 1)) ==>
  (!nn. nn ins Nat ==> ~(gg_fun x nn = Bt) ==> (gg_fun x nn = Li ft 1))"
```

() : void

Here, the theorem output from the subgoal package is replaced by dots. This finishes the basis step of the induction proof.

Next we consider the induction step. The induction hypothesis is the second (the large) assumption. First we strip this subgoal apart and assume all antecedents except the last one.

```
#e(REPEAT STRIP_TAC THEN POP_ASSUM MP_TAC);;
```

```
OK. .
```

```
"~(gg_fun x nn = Bt) ==> (gg_fun x nn = Lift 1)"  
  [ "x ins (cf(Nat, lift Nat))" ]  
  [ "!nn. nn ins Nat ==> ~(x nn = Bt) ==> (x nn = Lift 1)" ]  
  [ "nn ins Nat" ]
```

```
() : void
```

Now it is time to expand the definition of `gg_fun` and reduce the lambda abstractions.

```
#e(REWRITE_TAC[gg_fun_DEF] THEN CONS_REDUCE_TAC[[]][[]]);;
```

```
OK. .
```

```
"~(Cond  
  ((nn = 0), Lft 1, Ext(\m :: Dom Nat. Lift((m + m) - 1))(x(nn - 1))) =  
  Bt) ==>  
(Cond  
  ((nn = 0), Lft 1, Ext(\m :: Dom Nat. Lift((m + m) - 1))(x(nn - 1))) =  
  Lft 1)"  
  [ "x ins (cf(Nat, lift Nat))" ]  
  [ "!nn. nn ins Nat ==> ~(x nn = Bt) ==> (x nn = Lift 1)" ]  
  [ "nn ins Nat" ]
```

```
() : void
```

Note that the "Lift 1" terms have also been reduced, namely to "Lft 1". Next we do a case split on `nn` using the theorem

```
#REWRITE_RULE[ADD1](ISPEC"nn: num"num_CASES);;
```

```
|- (nn = 0) \/\ (?n. nn = n + 1)
```

in order to be able to reduce the conditional.

```
#e(STRIP_SUBST1_TAC(REWRITE_RULE[ADD1](ISPEC"nn: num"num_CASES)));;
```

```
OK. .
```

```
2 subgoals
```

```
"~(Cond  
  ((n + 1 = 0), Lft 1,  
  Ext(\m :: Dom Nat. Lift((m + m) - 1))(x((n + 1) - 1))) =  
  Bt) ==>  
(Cond  
  ((n + 1 = 0), Lft 1,  
  Ext(\m :: Dom Nat. Lift((m + m) - 1))(x((n + 1) - 1))) =  
  Lft 1)"  
  [ "x ins (cf(Nat, lift Nat))" ]  
  [ "!nn. nn ins Nat ==> ~(x nn = Bt) ==> (x nn = Lift 1)" ]  
  [ "(n + 1) ins Nat" ]  
  [ "nn = n + 1" ]
```

```

"~(Cond((0 = 0), Lft 1, Ext(\m :: Dom Nat. Lift((m + m) - 1))(x(0 - 1))) =
  Bt) ==>
(Cond((0 = 0), Lft 1, Ext(\m :: Dom Nat. Lift((m + m) - 1))(x(0 - 1))) =
  Lft 1)"
  [ "x ins (cf(Nat, lift Nat))" ]
  [ "!nn. nn ins Nat ==> ~(x nn = Bt) ==> (x nn = Lift 1)" ]
  [ "0 ins Nat" ]
  [ "nn = 0" ]

```

() : void

The tactic `STRIP_SUBST1_TAC` is home-made; it strips a theorem apart before assuming it and substitutes equalities in both the goal and the assumptions. The first subgoal to be proved is simplified using rewriting to get rid of "0 = 0".

```

#e(REWRITE_TAC[EQT_INTRO(REFL"0")]);;
OK.
"~(Cond
  (T, Lft 1, Ext(\m :: Dom Nat. Lift((m + m) - 1))(x(0 - 1))) = Bt) ==>
(Cond
  (T, Lft 1, Ext(\m :: Dom Nat. Lift((m + m) - 1))(x(0 - 1))) = Lft 1)"
  [ "x ins (cf(Nat, lift Nat))" ]
  [ "!nn. nn ins Nat ==> ~(x nn = Bt) ==> (x nn = Lift 1)" ]
  [ "0 ins Nat" ]
  [ "nn = 0" ]

```

() : void

Here we note that the antecedent is not necessary to prove the goal. Therefore it is discharged before the conditional is reduced.

```

#e(DISCH_THEN(\th. ALL_TAC)
# THEN CONS_REDUCE_TAC[]
# [ins_prover"Lft 1 ins lift Nat"][Cond_T_REDUCE_THM]);;
OK.
"Lft 1 = Lft 1"
  [ "x ins (cf(Nat, lift Nat))" ]
  [ "!nn. nn ins Nat ==> ~(x nn = Bt) ==> (x nn = Lift 1)" ]
  [ "0 ins Nat" ]
  [ "nn = 0" ]

```

() : void

Note we must provide the fact that "Lft 1" is a lifted natural number since the constant `Lft` is not part of the notation for typable terms (`Lift` is). This goal is finished off using reflexivity.

```

#e(REFL_TAC);;
OK.

```

goal proved

.
. .
.

Previous subproof:

```
"~(Cond
  ((n + 1 = 0), Lft 1,
  Ext(\m :: Dom Nat. Lift((m + m) - 1))(x((n + 1) - 1))) =
  Bt) ==>
(Cond
  ((n + 1 = 0), Lft 1,
  Ext(\m :: Dom Nat. Lift((m + m) - 1))(x((n + 1) - 1))) =
  Lft 1)"
[ "x ins (cf(Nat, lift Nat))" ]
[ "!nn. nn ins Nat ==> ~(x nn = Bt) ==> (x nn = Lift 1)" ]
[ "(n + 1) ins Nat" ]
[ "nn = n + 1" ]
```

() : void

This concludes the case when `nn` is zero.

The next subgoal corresponds to the case when `nn` is "`n + 1`". First we observe that adding one never yields zero.

```
#REWRITE_RULE[ADD1]NOT_SUC;;
|- !n. ~(n + 1 = 0)
```

so we can simplify the conditionals as follows

```
#e(REWRITE_TAC[REWRITE_RULE[ADD1]NOT_SUC]
# THEN CONS_REDUCE_TAC[]
# [ins_prover"Lft 1 ins lift Nat"] [Cond_F_REDUCE_THM]
OK.
"~(Ext(\m :: Dom Nat. Lift((m + m) - 1))(x((n + 1) - 1)) = Bt) ==>
(Ext(\m :: Dom Nat. Lift((m + m) - 1))(x((n + 1) - 1)) = Lft 1)"
[ "x ins (cf(Nat, lift Nat))" ]
[ "!nn. nn ins Nat ==> ~(x nn = Bt) ==> (x nn = Lift 1)" ]
[ "(n + 1) ins Nat" ]
[ "nn = n + 1" ]
```

() : void

which is simplified further using the fact that adding a number and then subtracting the number is identity.

```
#ADD_SUB;;
|- !a c. (a + c) - c = a
```

```
#e(REWRITE_TAC[ADD_SUB]);;
```

```
OK.
```

```
"~(Ext(\m :: Dom Nat. Lift((m + m) - 1))(x n) = Bt) ==>
(Ext(\m :: Dom Nat. Lift((m + m) - 1))(x n) = Lft 1)"
  [ "x ins (cf(Nat, lift Nat))" ]
  [ "!nn. nn ins Nat ==> ~(x nn = Bt) ==> (x nn = Lift 1)" ]
  [ "(n + 1) ins Nat" ]
  [ "nn = n + 1" ]
```

```
() : void
```

In order to get rid of the construction `Ext` we must do a case split on "`x n`".

```
#e(LIFT_CASES_TAC(INS_PROVER"x (n: num) ins lift Nat"));;
```

```
OK.
```

```
2 subgoals
```

```
"~(Ext(\m :: Dom Nat. Lift((m + m) - 1))(Lift d') = Bt) ==>
(Ext(\m :: Dom Nat. Lift((m + m) - 1))(Lift d') = Lft 1)"
  [ "x ins (cf(Nat, lift Nat))" ]
  [ "!nn. nn ins Nat ==> ~(x nn = Bt) ==> (x nn = Lift 1)" ]
  [ "(n + 1) ins Nat" ]
  [ "nn = n + 1" ]
  [ "d' ins Nat" ]
  [ "x n = Lift d'" ]
```

```
"~(Ext(\m :: Dom Nat. Lift((m + m) - 1))Bt = Bt) ==>
(Ext(\m :: Dom Nat. Lift((m + m) - 1))Bt = Lft 1)"
  [ "x ins (cf(Nat, lift Nat))" ]
  [ "!nn. nn ins Nat ==> ~(x nn = Bt) ==> (x nn = Lift 1)" ]
  [ "(n + 1) ins Nat" ]
  [ "nn = n + 1" ]
  [ "x n = Bt" ]
```

```
() : void
```

The first subgoal is finished off by the reduction tactic (recall `Ext` is strict) and rewriting. We can prove the second subgoal as follows. Clearly we should reduce the `Ext` terms since `Ext` is applied to `Lift` terms which makes this possible. The antecedent is not necessary so let us get rid of this before we reduce the goal.

```
#e(DISCH_THEN(\th. ALL_TAC) THEN CONS_REDUCE_TAC[[]][[]]);;
```

```
OK.
```

```
"Lift((d' + d') - 1) = Lft 1"
  [ "x ins (cf(Nat, lift Nat))" ]
  [ "!nn. nn ins Nat ==> ~(x nn = Bt) ==> (x nn = Lift 1)" ]
  [ "(n + 1) ins Nat" ]
  [ "nn = n + 1" ]
  [ "d' ins Nat" ]
```

```
[ "x n = Li ft d' " ]
```

```
() : void
```

To proceed we derive the fact that d' is equal to 1 from the assumptions. First recall that Bt and $Li ft$ are distinct.

```
#BT_LIFT_DISTINCT;;  
|- !D d. d ins D ==> ~(Bt = Li ft d)
```

Hence, "x n" is not bottom.

```
#e(TOP_ASSUM(\th. ASSUME_TAC(SUBS[SYM th]  
# (GSYM(MATCH_MP BT_LIFT_DISTINCT(ASSUME"d' ins Nat")))))));;  
OK..  
"Lft((d' + d') - 1) = Lft 1"  
[ "x ins (cf(Nat, li ft Nat))" ]  
[ "!nn. nn ins Nat ==> ~(x nn = Bt) ==> (x nn = Li ft 1)" ]  
[ "(n + 1) ins Nat" ]  
[ "nn = n + 1" ]  
[ "d' ins Nat" ]  
[ "x n = Li ft d' " ]  
[ "~(x n = Bt)" ]
```

```
() : void
```

We can use this and the induction hypothesis to conclude that "x n" is equal to "Li ft 1".

```
#e(FIRST_ASSUM(\th. IMP_RES_TAC(MATCH_MP th(ins_prover"n ins Nat"))));;  
OK..  
"Lft((d' + d') - 1) = Lft 1"  
[ "x ins (cf(Nat, li ft Nat))" ]  
[ "!nn. nn ins Nat ==> ~(x nn = Bt) ==> (x nn = Li ft 1)" ]  
[ "(n + 1) ins Nat" ]  
[ "nn = n + 1" ]  
[ "d' ins Nat" ]  
[ "x n = Li ft d' " ]  
[ "~(x n = Bt)" ]  
[ "x n = Li ft 1" ]
```

```
() : void
```

Now we have that "x n" is equal to both "Li ft 1" and "Li ft d'" so since $Li ft$ is injective (one-one)

```
#LIFT_ONE_ONE;;  
|- !D x y. x ins D ==> y ins D ==> ((Li ft x = Li ft y) = (x = y))
```

we can deduce that d' is equal to one.

```

#e(POP_ASSUM(\th1. POP_REMOVE THEN POP_ASSUM(\th2. SUBST1_TAC
#   (SUBS[(MATCH_MP(MATCH_MP LIFT_ONE_ONE(ASSUME"d' ins Nat"))
#     (ins_prover"1 ins Nat"))](SUBS[th2]th1)))));;
OK.
"Lft((1 + 1) - 1) = Lft 1"
  [ "x ins (cf(Nat, lift Nat))" ]
  [ "!nn. nn ins Nat ==> ~(x nn = Bt) ==> (x nn = Lift 1)" ]
  [ "(n + 1) ins Nat" ]
  [ "nn = n + 1" ]
  [ "d' ins Nat" ]

```

() : void

Now rewriting can prove the goal.

```

#e(RT[ADD_SUB]);;
OK.
goal proved
.
.
.

```

Previous subproof:

```

goal proved
() : void

```

Hence, we have prove the desired fact

$$GG_ONE = |- !nn. \sim(gg\ nn = Bt) ==> (gg\ nn = Lift\ 1)$$

stating `gg` is always equal to one (lifted) when it terminates.

6.4 A Simple Language and Its Semantics

As another simple example we will show how the HOL-CPO system can be used to give a denotational call-by-value semantics of a language called REC which supports the recursive definition of functions on natural numbers. This example is taken from [Wi93] (see chapter 9) where it is treated more thoroughly than here. The presentation below assumes the extensions of the notations of `cpo`s and `typable` terms which are presented in section 6.1 and section 6.2 above.

The language is represented by an abstract datatype of syntax in HOL. This type is defined to meet the following type specification.

```

rec = N num | V var
      | ADD rec rec | SUB rec rec | MULT rec rec
      | IF rec rec rec
      | FUN1 rec
      | FUN2 rec rec

```

The type `:"var"` is introduced as an abbreviation of the type of strings `:"string"` [GM93]. The conditional term `"IF t0 t1 t2"` in REC tests whether its argument `t0` is zero. If this is the case it returns `t1` otherwise it returns `t2`. The constants `FUN1` and `FUN2` are the function variables of the language. These are given a meaning by a declaration of the following form

$$\begin{aligned} \text{FUN1 } (V' x') &= t1 \\ \text{FUN2 } (V' x') (V' y') &= t2 \end{aligned}$$

where `t1` and `t2` must meet the above syntax and therefore may contain both `FUN1` and `FUN2`. Hence, mutual recursion is allowed. The term `t1` should contain no other variables than `"V' x' "` and `t2` should contain no other variables than `"V' x' "` and `"V' y' "`. The functions are unary and binary operations respectively.

In fact, there could be any number of functions of different arities in REC. The language we consider here is just an example with two functions of the arities just mentioned. If we were going to use this language we would write a declaration first and then introduce the version of REC which had the right function variables. This could be automated by an ML program.

The semantics of REC is stated using a function `den` which says which domain theoretic function or value a term denotes. It is defined by primitive recursion on the above datatype of syntax. The denotation of terms of REC is stated with respect to environments for variables and function variables. Defining the cpo of names of variables as follows

```
#let Var_DEF, Var_CPO = new_cpo_definition 'Var_DEF' 'Var_CPO' []
# "Var = discrete(UNIV: var->bool)";;
Var_DEF = |- Var = Var
Var_CPO = |- cpo Var
```

we can define the cpo of variable environments by

```
#let Env_DEF, Env_CPO = new_cpo_definition 'Env_DEF' 'Env_CPO' []
# "Env = cf(Var, Nat)";;
Env_DEF = |- Env = Env
Env_CPO = |- cpo Env
```

and the cpo of function environments by

```
#let Fenv_DEF, Fenv_CPO = new_cpo_definition 'Fenv_DEF' 'Fenv_CPO' []
# "Fenv = prod(cf(Nat, lift Nat), cf(Nat, cf(Nat, lift Nat)))";;
Fenv_DEF = |- Fenv = Fenv
Fenv_CPO = |- cpo Fenv
```

This is a product of two function spaces of which the first component is determined by the type of `FUN1` and the second component is determined by the type of `FUN2`. A declaration determines a particular function environment. Finally, the primitive recursive definition of `den` can be stated as follows

```
den_DEF =
|- (!n. den(N n) = (\f :: Dom Fenv. \s :: Dom Env. lift n)) /\
```

```

(!v. den(V v) = (\f :: Dom Fenv. \s :: Dom Env. Lift(s v))) /\
(!t1 t2.
  den(ADD t1 t2) =
    (\f :: Dom Fenv. \s :: Dom Env. Add(den t1 f s)(den t2 f s))) /\
(!t1 t2.
  den(SUB t1 t2) =
    (\f :: Dom Fenv. \s :: Dom Env. Sub(den t1 f s)(den t2 f s))) /\
(!t1 t2.
  den(MULT t1 t2) =
    (\f :: Dom Fenv. \s :: Dom Env. Mult(den t1 f s)(den t2 f s))) /\
(!t0 t1 t2.
  den(IF t0 t1 t2) =
    (\f :: Dom Fenv.
     \s :: Dom Env.
      Ext
        (\b :: Dom Bool. Cond(b,den t1 f s,den t2 f s))
        (Iszero(den t0 f s)))) /\
(!t.
  den(FUN1 t) =
    (\(f1, f2) :: Dom Fenv. \s :: Dom Env. Ext f1(den t(f1, f2)s))) /\
(!t1 t2.
  den(FUN2 t1 t2) =
    (\(f1, f2) :: Dom Fenv.
     \s :: Dom Env.
      Ext
        (\v1 :: Dom Nat. Ext(\v2 :: Dom Nat. f2 v1 v2))
        (den t1(f1, f2)s)
        (den t2(f1, f2)s)))

```

Note that we use the strict versions of addition, subtraction and multiplication presented above. Besides, we use the strict test for zero and furthermore extends the conditional `Cond` to a strict conditional using `Ext`. The above semantics corresponds to the one presented on page 144 in [Wi93].

We wish the denotation of a term to be a continuous function since this allows us to define the actual function environment determined by a declaration. Indeed it is continuous as stated by the following theorem.

$$\vdash !t. (\text{den } t) \text{ ins } (\text{cf}(\text{Fenv}, \text{cf}(\text{Env}, \text{Lift Nat})))$$

This is proved by structural induction on terms and the type checker, or more precisely, the tactic version of the type checker, called `TYPE_CHECK_TAC`. Structural induction is stated by a theorem `rec_INDUCT` which can be proved automatically by a tool associated with the type definition package [Me89]. The proof of the above theorem is so simple that we will list it here:

```

INDUCT_THEN rec_INDUCT STRIP_ASSUME_TAC
THEN REPEAT GEN_TAC
THEN REWRITE_TAC[den_DEF]
THEN TYPE_CHECK_TAC[][ins_prover "n ins Nat"; ins_prover "s ins Var"]

```

Behind the scenes, the type checker tactic uses the induction hypotheses which are present in the assumptions after induction has been applied.

We have also proved the following theorem about the semantics

$$\begin{aligned} &|- !s s' t. \\ &\quad (!v. v \text{ IN } (\text{vars } t) ==> (s \ v = s' \ v)) ==> \\ &\quad (!f. f \text{ ins Fenv } ==> (\text{den } t \ f \ s = \text{den } t \ f \ s')) \end{aligned}$$

which states that the result of a denotation of a term in a function environment only depends on variables in the term. In particular, the denotation "den t f s" of a closed term t is independent of the environment state variable s. The variables function is defined straightforwardly by

$$\begin{aligned} &|- (!n. \text{vars}(N \ n) = \{\}) \wedge \\ &\quad (!v. \text{vars}(V \ v) = \{v\}) \wedge \\ &\quad (!t \ t'. \text{vars}(\text{ADD } t \ t') = (\text{vars } t) \text{ UNION } (\text{vars } t')) \wedge \\ &\quad (!t \ t'. \text{vars}(\text{SUB } t \ t') = (\text{vars } t) \text{ UNION } (\text{vars } t')) \wedge \\ &\quad (!t \ t'. \text{vars}(\text{MULT } t \ t') = (\text{vars } t) \text{ UNION } (\text{vars } t')) \wedge \\ &\quad (!t \ t' \ t''. \\ &\quad \quad \text{vars}(\text{IF } t \ t' \ t'') = \\ &\quad \quad (\text{vars } t) \text{ UNION } ((\text{vars } t') \text{ UNION } (\text{vars } t''))) \wedge \\ &\quad (!t. \text{vars}(\text{FUN1 } t) = \text{vars } t) \wedge \\ &\quad (!t \ t'. \text{vars}(\text{FUN2 } t \ t') = (\text{vars } t) \text{ UNION } (\text{vars } t')) \end{aligned}$$

Note, by the way, that although Fenv is in fact a universal cpo we include the antecedent above stating "f ins Fenv". We do this to ease the proof since such a term cannot be proved using ins_prover. However, ins_prover could of course be extended to handle such situations.

Finally, we shall see how a declaration determines a function environment by considering a simple example of the use of REC to define the factorial function. The declaration we consider is the following

$$\begin{aligned} &"(\text{FUN1}(V \ 'x') = \\ &\quad \text{IF}(V \ 'x')(N \ 1)(\text{MULT}(V \ 'x')(\text{FUN1}(\text{SUB}(V \ 'x')(N \ 1)))))) \wedge \\ &\quad (\text{FUN2}(V \ 'x')(V \ 'y') = \text{FUN2}(V \ 'x')(V \ 'y'))" \end{aligned}$$

where FUN1 is defined as the factorial function and FUN2 is a recursive functions which keeps on calling itself with the same arguments and therefore runs forever. A functional for the function environment is introduced using new_constant_definition which returns the following theorems

$$\begin{aligned} &\text{fenv_fun_DEF} = \\ &|- \text{fenv_fun} = \\ &\quad (\backslash s :: \text{Dom Env.} \\ &\quad \quad \backslash f :: \text{Dom Fenv.} \\ &\quad \quad ((\backslash n :: \text{Dom Nat.} \\ &\quad \quad \quad \text{den} \\ &\quad \quad \quad (\text{IF}(V \ 'x')(N \ 1)(\text{MULT}(V \ 'x')(\text{FUN1}(\text{SUB}(V \ 'x')(N \ 1)))))) \\ &\quad \quad \quad f \end{aligned}$$

```

      (bind(n, 'x', s))),
    (\n m :: Dom Nat.
      den(FUN2(V 'x')(V 'y'))f(bind(n, 'x', bind(m, 'y', s))))))
fenv_fun_CF = |- fenv_fun ins (cf(Env, cf(Fenv, Fenv)))

```

where the constant `bind` is used to associate a value with a variable in a state.

```

|- bind =
  (\(n, x, s) :: Dom(prod(Nat, prod(Var, Env))).
    \y :: Dom Var. ((x = y) => n | s x))
|- bind ins (cf(prod(Nat, prod(Var, Env)), Env))

```

Now, the function environment determined by the above declaration can be introduced by a fixed point definition as follows (using the same program as above)

```

fenv_DEF = |- fenv = (\s :: Dom Env. Fix(fenv_fun s))
fenv_CF = |- fenv ins (cf(Env, Fenv))

```

Hence, the denotation of the factorial function in a variable environment `S` corresponds to the first component of "`fenv S`" which is a pair (the tiring calculation of this is left to the reader). One must admit that this is not the most neat and convenient definition of the factorial function that one can think of (compare for instance with the factorial of section 6.2.2).

Chapter 7

LCF Examples

In chapter 10 of Paulson’s book on the Cambridge LCF system [Pa87] a number of examples are presented. These illustrate the use of LCF for reasoning about natural numbers, recursive functions and infinite sequences. In this chapter we describe how the examples can be developed in HOL-CPO, the extension of HOL with domain theory.

The LCF system and the HOL system have many similarities since HOL is based on the ‘LCF approach’ to theorem proving (due to Robin Milner, the originator of LCF). Both systems have a meta-language ML and their logics are implemented in ML. Inference rules and tactics are ML functions. Extensions are organized in hierarchies of theories. In fact, the central difference between HOL and LCF is in their logic parts. Where HOL is based on a version of Church’s higher order logic the LCF system implements a version of Scott’s Logic of Computable Functions, a first order logic of domain theory.

In a way, extending HOL with domain theory corresponds to embedding the logic of the LCF system within HOL. Thus, any proof conducted in LCF can be conducted in HOL-CPO as well, and axioms of LCF theories can be defined, or derived from definitions, provided of course they are consistent extensions of LCF. This correspondence breaks down for difficult recursive domains with infinite values. It is not easy to define such domains in HOL-CPO and LCF could just axiomatize the domains; this has been automated in certain cases [Pa84a].

However, HOL-CPO is not just another LCF system. Ignoring the problems with recursive domains, we claim it is more powerful and usable than LCF since (1) it inherits the underlying logic and proof infrastructure of the HOL system, and (2) it provides direct access to domain theory. These two points are the consequences of *embedding* semantics rather than *implementing* logic.

Experience with LCF shows that the continual fiddling with bottom is very annoying. Its presence in all types makes LCF less suited for proofs about strict (finite-valued) datatypes than for proofs about lazy datatypes [Pa84a, Pa85, Pa84b]. Paulson says the ugliest reasoning in LCF involves flatness; a flat type denotes a cpo with a bottom element but no partial elements (strict datatypes are flat). Further, the so-called definedness assumptions stating arguments of functions are not bottom makes reasoning about strict functions difficult, e.g. constructor functions of strict datatypes are strict. However, LCF seems to be well-suited to reason about lazy evaluation, for instance about constructor functions of lazy datatypes. Here, such tests for definedness do not occur.

One historical advantage of (1) is that we can exploit the rich collection of built-in theorems, tools and libraries provided with the HOL system. LCF has almost nothing

like that, but could have of course. The main advantage is that we become able to mix domain and set theoretic reasoning in HOL. Hence, reasoning about bottom and strict functions can often be (almost) eliminated from proofs, or deferred until the late stages of a proof.

In contrast to (2), domain theory is only present in the logic of LCF through axioms and primitive rules of inference. Therefore fixed point induction is the only way to reason about recursive definitions and testing that a predicate admits induction can only be performed in ML by an incomplete syntactic check. By exploiting the semantic definitions of these concepts in domain theory, HOL-CPO does not impose such limitations. Fixed point induction can be derived as a theorem and syntactic checks for admissibility, also called inclusiveness, can be implemented, just as in LCF. But using other techniques for recursion or reasoning directly about fixed points allow more theorems to be proved than with just fixed point induction. Inclusive predicates not accepted by the syntactic checks can be proved to be inclusive directly from the semantic definition.

The examples will show that we can define and reason about arbitrary recursive functions and non-termination in domain theory and about finite-valued types and total ('strict') functions in set theory (higher order logic) before turning to domain theory. The natural number example illustrates how we can mix set and domain theoretic reasoning and thereby ease reasoning about strict LCF datatypes. The two other examples about recursive definitions and infinite sequences respectively illustrate that we can conduct LCF proofs by fixed point induction and structural induction on inclusive predicates in HOL-CPO.

7.1 The LCF System

The LCF system is very similar to the HOL system. It has a meta-language ML (or Standard ML) in which the logic and theorem proving tools are implemented. Theorems are implemented by an abstract datatype for security and axioms and primitive inference rules are constructors of this datatype. Derived inference rules are ML functions. The subgoal package allows proofs in a backwards fashion using tactics. Constants, axioms, theorems and so on are organized in hierarchies of theories. In fact, the HOL system is a direct descendant of LCF. However, the two systems are quite different on some points. The main properties of LCF may be summarized as follows:

- LCF supports a first order logic of domain theory.
- The use of LCF to reason about recursive definitions (fixed points) is restricted, for instance fixed point induction is part of the primitive basis of the underlying logic (so there is no access to domain theory) and employs an incomplete syntactic check for admissibility of a predicate for induction (performed in ML).
- Extending theories in LCF is done by an axiomatic approach and is therefore unsafe. Checking whether an axiom is safe is difficult since it must be done in domain theory (outside LCF).
- Rewriting is relatively powerful in LCF, which makes final (and edited) proof in LCF quite compact.

In the following sections these points and the most important differences between the two systems are discussed further.

7.1.1 The Logic $PP\lambda$

The central difference between LCF and HOL is in their logic parts. The logic of the HOL system is an implementation of a version of Church's higher order logic. The logic of the LCF system is an implementation of a version of Scott's Logic of Computable Functions, usually abbreviated LCF. In order to be able to distinguish the logic and the system the logic was renamed to $PP\lambda$, an acronym of Polymorphic Predicate λ -calculus.

$PP\lambda$ is a first order logic of domain theory; it has a domain theoretic semantics. It differs from higher order logic since it is a first order logic and since types denote cpos with bottom. The function type denotes the continuous function space. Types of the HOL logic just denote sets (cpo's may be viewed as sets with structure) and functions are total functions of set theory.

Because $PP\lambda$ is a first order logic there is a distinction between terms and formulae and quantification is only allowed for terms. Formulae are just the usual ones of predicate calculus and terms are similar to HOL terms; either an LCF term is a constant, a variable, an abstraction or a combination (application). Every such term has a type in LCF and there are both formulae and terms of type boolean. Therefore, there are also two equality tests, the computational one for terms and the logical one for formulae, written as \equiv .

In HOL there is no distinction between terms and formulae. Roughly speaking, the formulae of LCF correspond to HOL terms of type boolean. LCF terms and types can be denoted by using the extension of HOL with domain theory. Therefore, HOL-CPO can be seen as kind of shallow embedding of LCF in HOL.

Note that types of LCF correspond to pointed cpos in HOL-CPO. It is a real advantage to allow cpos without bottom since this makes it easier to view (subsets of) HOL types as cpos and eliminate reasoning about bottom. The presence of bottom in all types is very annoying in LCF (as mentioned above and demonstrated by the natural number example).

Since types of $PP\lambda$ denote pointed cpos and all functions are continuous, it is a fact of domain theory that recursive functions can be defined by using a fixed point operator. Reasoning about recursive definitions can be conducted using fixed point induction. The fixed point theory of computation is provided in $PP\lambda$ through axioms and primitive rules of inference. A constant denotes the fixed point operator of domain theory due to an axiom which states it yields a fixed point and due to the primitive rule of fixed point induction which states it yields the least fixed point. In the logic there is no domain theoretic definition of the fixed point operator. Therefore, this axiom and primitive rule of inference provide the only ways for reasoning about recursive definitions. Hence, proofs that reason about recursive functions in other ways, e.g. by their fixed point definition, cannot be mechanized in LCF. Besides, the concept of admissibility (inclusiveness) of predicates for induction is only present as a syntactic check performed by the rule of fixed point induction. This check is not complete and examples of inclusive predicates exist that are not accepted for fixed point induction in LCF. Paulson gives an example in [Pa84a].

Such limitations are not present in HOL-CPO. Domain theory is embedded in HOL so we have direct access to all semantic definitions of concepts of fixed point theory. In

particular, the fixed point operator and inclusiveness are defined semantically and fixed point induction is derived as a theorem from these definitions. Since the embedding is shallow [BGH92], it allows mixing any set theoretic reasoning in HOL with domain theoretic reasoning in HOL-CPO. There is no immediate difference between ordinary HOL terms and terms that are cpos or continuous functions. We can therefore do proofs about recursive functions by fixed point induction, by the definition of the fixed point operator or by any induction technique that can be derived in HOL, e.g. well-founded induction (see appendix A). Besides inclusiveness can be checked syntactically by an ML function which implements a syntactic check based on theorems derived from the definition of inclusiveness (see section 5.3). This check is very similar to the LCF one (or they are the same).

7.1.2 Extending Theories

There are quite different traditions of extending theories in LCF and HOL. In HOL there is a sharp distinction between purely definitional extensions and axiomatic extensions. Definitional extensions are conservative (or safe), i.e. they always preserve consistency of the logic. Stating a new axiom may not be a conservative extension, it might introduce inconsistency (or not). In LCF there is no such distinction between axioms and definitions. The only way to extend theories with new concepts is by introducing new axioms. First a new constant or type is given a name. Then a number of properties of the constant or type are stated as axioms. It is not possible to state axioms about a type directly, they must be stated about constants as e.g. constructor functions of a datatype or abstraction and representation functions (see below).

It is not always easy to know whether an LCF axiom is safe or not since this must be justified in domain theory. In particular, an axiom should not violate the continuity of a function. All functions are assumed to be continuous in $PP\lambda$ since the function type denotes the continuous function space. Paulson shows how easy it is to go wrong in example 4.11 of his book [Pa87]. Instead of defining recursive functions using the fixed point operator an axiom can be introduced which is a recursion equation, i.e. an equation where the function constant appears on both sides of the equality sign. Such axioms are justified in domain theory since the equation follows from a property stating that the fixed point operator always yields a fixed point. Fixed point induction can only be used if a function is defined using the fixed point operator.

As mentioned above, recursive types are also introduced by axioms. There is no recursion operator for types corresponding to the fixed point operator for functions so new types cannot be defined (since there is no access to domain theory). Given a recursive domain equation of the form $\tau \cong F(\tau)$ where F is a continuous domain constructor the most general approach is to state the existence of abstraction and representation functions $ABS : F(\tau) \rightarrow \tau$ and $REP : \tau \rightarrow F(\tau)$ that are bijective. This approach is justified in domain theory since the isomorphism equation follows from a limit construction of the new domain, which is said to be a solution to the recursive domain equation (see chapter 4).

Another more direct approach works for certain recursive datatypes which can be described by a set of constructor functions, each taking a number of arguments of specified types where the function type is not used in an essential way [Pa84a]. Provided such a set as an argument an ML program can introduce constants and axioms to define a new recursive datatype in LCF. The program supports both strict, lazy and mixed (mutual)

recursive datatypes. Some of these types we can derive manually in HOL using the methods described in chapter 4. If we decided to axiomatize recursive domains as in LCF, we could probably introduce recursive domains easier than with the methods of that chapter. But axiomatizing theories does not fit in well with the HOL tradition. Hence, HOL-CPO is weaker than LCF on this point concerning defining recursive domains.

In LCF, the axioms of the new datatype(s) state a number of properties about the constructor functions which are defined as constants. For instance, there is an axiom which states what the partial ordering on elements of the new type is and there is a exhaustion (or cases) axiom which states that the constructors are exhaustive, i.e. any term of that type is equal to one of the constructors for appropriate arguments of the constructor. From the axioms it is derived that the constructors are distinct and that structural induction with inclusive predicates is valid on the new type. Structural induction is derived from fixed point induction.

7.1.3 Rewriting

Finally, there is a small difference in the implementation of rewriting in the two systems. In addition to standard HOL rewriting LCF performs conditional rewriting where it tries to simplify antecedents of conditional rewrite theorems to truth recursively. And it implements rewriting with local assumptions where the antecedent are used to rewrite the consequence of implicative formulae. It also does β -conversion whenever possible and it employs various simplifications not used in HOL, e.g. DeMorgan's Laws. Therefore proofs can be written in a very compact way in LCF. But the price to pay for powerful rewriting is of course efficiency. Besides a proof which use rewriting too heavily is almost impossible to read since it has no structure. If a proof is written to reflect the way it was invented it is more accessible for later inspection.

7.2 Natural Numbers

In this section we define a cpo of natural numbers and consider a few properties about the operations addition and equality.

We start up a new theory called `lcf_nat` and define a discrete universal cpo of natural numbers using the function `new_cpo_definition`.

```
#new_theory' lcf_nat' ;;
(): void

#new_cpo_definition' Nat_DEF' ' Nat_CPO' []
#"Nat = discrete(UNIV: num->bool)";;
|- Nat = Nat
|- cpo Nat
```

The facts stating that the two built-in constants `0` and `SUC` construct elements in `Nat` can be proved automatically and then added to the system using `ins_prover` and `declare`.

```
#declare(ins_prover "0 ins Nat");;
```

```
| - 0 ins Nat
```

```
#declare( ins_prover "SUC ins(cf(Nat, Nat))" ); ;  
| - SUC ins (cf(Nat, Nat))
```

This cpo does not have a bottom but using the lifting construction we can obtain a pointed cpo "lift Nat" when we need it. A strict successor for the lifted cpo can be obtained from SUC using the function construction called extension associated with the lifting construction (see section 3.6.4).

```
#new_constant_definition' Suc_DEF' ' Suc_CF' []  
#"Suc = Ext(\nn: : Dom Nat. Lift(SUC nn))" ; ;  
| - Suc = Ext(\nn : : Dom Nat. Lift(SUC nn))  
| - Suc ins (cf(lift Nat, lift Nat))
```

So this makes SUC a strict continuous constructor for the lifted cpo of natural numbers "lift Nat". The bottom of this cpo is: | - Bt ins (lift Nat) . The zero element is: | - (Lift 0) ins (lift Nat) . Both facts can be proved using the ins-prover (though the latter would not need to be declared since 0 has already been declared).

The cpo "lift Nat" corresponds to the LCF type of natural numbers and SUC corresponds to the strict constructor function for this type. In LCF natural numbers are introduced as a recursive datatype axiomatically. Names of constants for the type and constructor functions are declared and then axioms about the new constants are postulated. The axioms specify the partial ordering on natural numbers and state strictness and definedness of the constructors. The exhaustion (or cases) axiom is also postulated. It states there are three possible values of a natural number, namely bottom, zero and the successor of some natural number. Distinctness of the constructors and the structural induction rule are then derived from these axioms and fixed point induction. This is performed automatically by a few ML programs.

All such axioms, theorems and rules can be derived as theorems in HOL and in fact, most are built-in theorems or they can be derived very easily. For instance, it is a built-in fact that 0 and SUC are distinct for all arguments of SUC . Lifting natural numbers and extending SUC to a strict function, these constructors yield elements which are distinct from bottom (and still distinct from each other).

Addition on the cpo of natural numbers is simply the built-in addition of natural numbers. This is so because Nat is discrete and the underlying set is the whole type of natural numbers. Then the determinedness condition on continuous functions is satisfied trivially, which is exploited by the ins-prover. It was used to prove that the built-in successor is continuous above and it can be used to prove the built-in addition is continuous too.

```
#declare( ins_prover "$+ ins (cf(Nat, cf(Nat, Nat)))" ); ;  
| - $+ ins (cf(Nat, cf(Nat, Nat)))
```

Again we make the system aware of this fact by declaring it.

Since addition is strict in LCF, let us introduce a strict version of addition by extending the built-in addition to the lifted natural numbers.

```
#new_constant_definition' Add_DEF' ' Add_CF' []
#"Add = Ext(\nn :: Dom Nat. Ext(\mm :: Dom Nat. Lift(nn+mm)))";;
|- Add = Ext(\nn :: Dom Nat. Ext(\mm :: Dom Nat. Lift(nn+mm)))
|- Add ins (cf(Lift Nat, cf(Lift Nat, Lift Nat)))
```

The second fact returned by `new_constant_definition` states that the strict addition `Add` is continuous. In order to provide this fact it uses the continuity prover which proves the fact automatically from the subterms of the definition of `Add`.

An equality test for natural numbers is obtained in two versions in the same way as the successor and addition functions. The strict equality is called `Eq` and defined by the following theorem

```
|- Eq = Ext(\nn :: Dom Nat. Ext(\mm :: Dom Nat. Lift(nn = mm)))
```

The continuity theorems for the two equalities look as follows

```
|- $= ins (cf(Nat, cf(Nat, Bool)))
|- Eq ins (cf(Lift Nat, cf(Lift Nat, Lift Bool)))
```

The constant `Bool` is the discrete universal cpo of booleans defined in section 6.1. Note that it is only one particular instance of the polymorphic HOL equality that we have proved is continuous, namely the equality of type "`num -> num -> bool`".

The LCF functions for addition and equality corresponds to the functions `Add` and `Eq` that we defined above. However, these operations are introduced quite differently in LCF since there are no ‘meta-logical’ operations they can be identified with. They must be defined by a recursive definition. This is done by first introducing axioms for an eliminator functional called *NAT_WHEN*

$$\begin{aligned} NAT_WHEN\ x\ f\ \perp &\equiv \perp \\ NAT_WHEN\ x\ f\ 0 &\equiv x \\ \forall m. m \neq \perp &\Rightarrow NAT_WHEN\ x\ f\ (SUCC\ m) \equiv f\ m \end{aligned}$$

which can be used to define continuous functions on natural numbers by cases. Then axioms for addition and equality are postulated in terms of *NAT_WHEN*. The axioms are recursive definitions in the sense that the constant appearing on the left-hand side also appears on the right-hand side. These axioms must not violate the continuity of addition and equality, and they do not due to the use of *NAT_WHEN* (but there is not guarantee for this).

We could define addition and equality in HOL by introducing an eliminator functional but our approach is simpler. There is one major disadvantage of using when eliminator functions for strict datatypes as natural numbers. Since constructors are strict it is necessary to assume their arguments are defined. Otherwise there might be a conflict between the undefined case of an eliminator and a constructor case (see above). A consequence of this is that most theorems stated about functions defined in terms of a when eliminator must assume some of their arguments are defined (not necessarily all). This also makes all theorems about addition and equality in LCF more complicated to state, prove, and use than the corresponding theorems in HOL-CPO. In general, such definedness assumptions makes reasoning about total functions difficult since computations (terms) must be proved to terminate (to yield a value different from bottom) and functions must be proved to be total (to terminate for all arguments).

In HOL, functions are guaranteed to be total and using function extension we can extend functions to lifted domains in a strict way. This way we eliminate a lot of reasoning involving bottom. Furthermore, we also defer reasoning about bottom until as late as possible in a proof.

Since addition and equality are defined by recursion in LCF, proofs must be conducted using techniques as natural number induction. In HOL we can reuse built-in theorems about addition and equality. For flat types as strict datatypes we can always do the set theoretic developments in HOL before adding bottom. In general, LCF reasoning about total functions is difficult since functions must be proved to be total and computations must be proved to terminate. In HOL functions are guaranteed to be total and using lifting we can extend these to flat types.

In the next two sections we discuss a number of theorems about addition and equality. In general, the proofs in HOL are much simpler (in an intuitive sense) than the corresponding ones in LCF, since the theorems can be proved by similar inductions as in LCF, but without considering the bottom element as in LCF induction proofs. For strict datatypes we can do the set theoretic developments in HOL before adding bottom. It is very advantageous to eliminate and defer reasoning about bottom, e.g. definedness assumptions tend to accumulate.

7.2.1 Theorems about Addition

In this section we present a few theorems about the strict addition called `Add`. Proofs are always by cases on the lifted type of numbers due to the use of `Ext` in the definition of addition (no induction). The usual recursion equations for addition are stated first but in fact they are of no use with this definition of addition. A certain reduction theorem for different kinds of arguments of addition (bottom or lifted) is more useful. In LCF, the recursion equations are important since proofs use induction. Three perhaps more interesting theorems about addition than the reduction theorem are provided: addition is total, associative and commutative.

The recursion equations for the strict addition can be proved in HOL. However they are not very interesting since we shall never do an induction proof about the strict addition. Induction proofs are conducted about the built-in addition which is in fact defined by the corresponding recursion equations on the HOL type of natural numbers. Anyway, the equations are stated as follows

$$\begin{aligned} &|- (!n. \text{Add } Bt \ n = Bt) \wedge \\ &\quad (!n. \text{Add } (Lift \ 0) \ n = n) \wedge \\ &\quad (!n \ m. \text{Add}(\text{Suc } n)m = \text{Suc}(\text{Add } n \ m)) \end{aligned}$$

In LCF the third equation would assume the variable `n` is not bottom (cf. the comment on when eliminators above, the successor case must not collapse to bottom). The proof of this fact is longer but just as simple as the LCF proof. It is performed by cases on the lifted type using that the functions `Ext` and `Suc` are strict in the bottom case and using the corresponding clauses about `$+` in the lifted case. The proof is longer because we did not use an eliminator to define addition (and therefore do not have the equations provided by this eliminator) and because LCF proofs are compact due to the fact that many situations are handled by rewriting.

The more useful reduction theorem states that addition is strict in both arguments and behaves as the built-in addition on lifted arguments.

$$\begin{aligned} &|- (!n. \text{Add } Bt \ n = Bt) \wedge \\ & \quad (!n. \text{Add } n \ Bt = Bt) \wedge \\ & \quad (!nn \ mm. \text{Add}(\text{Li ft } nn)(\text{Li ft } mm) = \text{Li ft}(nn+mm)) \end{aligned}$$

The proof was described in details in section 6.2.3.

The next fact we consider states that strict addition is total. That is, provided the arguments of `Add` are not bottom the result of applying `Add` will not be bottom. In LCF this fact is stated by a theorem of the following form

$$|- !n \ m. \sim(n=Bt) ==> \sim(m=Bt) ==> \sim(\text{Add } n \ m = Bt)$$

Since we use lifting this fact can be stated equivalently by the following theorem in HOL

$$|- !nn \ mm. \sim(\text{Add}(\text{Li ft } nn)(\text{Li ft } mm) = Bt)$$

This can be derived immediately from the third clause of the above cases theorem for addition using the fact that `Bt` and `Li ft` are distinct and exhaustive on the lifted type.

The proof of termination of strict addition requires much more thought in LCF though it can be proved by a one-line LCF tactic, induction then rewriting. (However, in general inventing such a proof probably results in a larger tactic which can then be edited to become shorter.) Induction yields three subgoals which can all be solved by rewriting with the assumptions, the recursion equations for addition and a termination (definedness) property of the successor function.

Finally, let us consider two theorems stating that strict addition is associative and commutative.

$$\begin{aligned} &|- !k \ m \ n. \text{Add } (\text{Add } k \ m) \ n = \text{Add } k(\text{Add } m \ n) \\ &|- !m \ n. \text{Add } m \ n = \text{Add } n \ m \end{aligned}$$

Their proofs are almost exactly the same in HOL; do a case split on the universally quantified variables (lifted numbers) one by one and reduce using the reduction theorem for addition after each case split. We end up with goals stating that the properties we wish to prove must hold for the built-in addition. So we finish off the proofs by using the desired built-in HOL facts. Such proofs by cases could be automated easily. The LCF proofs use induction (in fact two nested inductions for commutativity) and rewriting. However, the main point is that these inductions have a bottom case, the corresponding ones in HOL would not have this case.

7.2.2 Theorems about Equality

The strict equality was defined by extending the built-in equality on the numbers to a strict function. Recall that we obtained a strict addition and a strict successor in the same way. Proofs of properties about equality is therefore similar to the proofs about addition discussed above. So a reduction theorem for different cases of arguments of equality is the basic component of proofs together with built-in properties about equality.

```

|- (!n. Eq Bt n = Bt) /\
   (!n. Eq n Bt = Bt) /\
   (!nn mm. Eq(Li ft nn)(Li ft mm) = Li ft(nn=mm))

```

Again we have a quite different situation than in LCF where equality is defined using nested applications of the eliminator function. There, a number of in total six recursion equations are important since proofs are done using induction. The recursion equations have also been proved for our equality but we shall not show the theorem here (since it is not useful).

From the reduction theorem it is easy to see that two lifted natural numbers are equal precisely when they are the same numbers (in the HOL sense).

```

|- !mm nn. (Eq(Li ft mm)(Li ft nn) = Li ft T) = (mm = nn)

```

This states the partial correctness of equality; if it is applied to (lifted) numbers which are not bottom it calculates the right result. The proof uses that `Li ft` is one-one. The corresponding theorem in LCF has antecedents which assume that the arguments of `Eq` are not bottom. The proof uses a nested induction.

7.3 A Recursive Function

In this section we give an example of the use of fixed point induction. We define a function which is recursive in a quite general way and show it is strict and idempotent, i.e. composing the function with itself yields the function itself. The function is continuous on any pointed cpo so we cannot exploit the discrete construction and lifting as in the previous section. Thus we cannot extend any built-in recursive HOL function to the desired continuous function either. Further, the recursion of the function is too complicated. We must define the function in the domain theoretic way, as a fixed point. The definitions of this section are very close to the definitions in LCF except that we prefer to keep certain names as variables instead of defining them as constants. The proofs of the theorems presented below are therefore the same as in LCF ignoring the few extra tactics we must introduce in order to type check arguments of dependent functions.

We start a new session and a new theory (see [GM93]). First we assume a variable `D` is a pointed cpo. This assumption is used to prove that the functional `G` which is defined next is a continuous function.

```

#new_theory' lcf_fpi ';;
() : void

#cpo1;;
":(*1 -> bool) # (*1 -> *1 -> bool)" : type

#let D_PCPO = ASSUME "pcpo(D: ^cpo1)";;
D_PCPO = . |- pcpo D

#let G_DEF, G_CF = new_constant_definition
# 'G_DEF' 'G_CF' [D_PCPO]
# "G (D: ^cpo1) =

```

```

#   \ (p, g) :: Dom(prod(cf(D, lift Bool), cf(D, D))).
#   \ h :: Dom(cf(D, D)).
#   \ x :: Dom D.
#   Ext(\ b :: Dom Bool . Cond(b, x, h(h(g x))))(p x)"; ;
G_DEF =
|- !D.
  G D =
    \ (p, g) :: Dom(prod(cf(D, lift Bool), cf(D, D))).
      \ h :: Dom(cf(D, D)).
        \ x :: Dom D. Ext(\ b :: Dom Bool . Cond(b, x, h(h(g x))))(p x)
G_CF =
. |- (G D) ins (cf(prod(cf(D, lift Bool), cf(D, D)), cf(cf(D, D), cf(D, D))))

```

Hence the single assumption in the `G_CF` theorem says that `D` must be a pointed cpo. Otherwise the theorem could not be proved due to the use of `Ext` (the codomain of its function argument must be a pointed cpo). The HOL definition, which as usual is just an abbreviation, holds for any `D`. The conditional `Cond` is straightforward, if its first argument is true it returns the second argument, if it is false it returns the third argument. It was introduced together with the cpo of booleans `Bool` in section 6.1. Note that the boolean test above is made strict in its first argument due to the use of `Ext`.

After this definition our system considers "`G D`" to be a 'constant term', though `G` is the constant in HOL. This means that we must always use `D` as the cpo argument of `G`. The system will fail for any other argument. Perhaps a better choice would be to define `G` as a constructor function since it would then be possible to give it any cpo as an argument. But for the purpose of the example of this section the choice above works fine.

The functional `G` is used to define the recursive function `H` and the continuity of `G` is used behind the scenes to prove that `H` is continuous.

```

##| let H_DEF, H_CF = new_constant_definition
#   'H_DEF' 'H_CF'
#   [D_PCPO]
#   "H(D: ^cpo1) =
#   \ (p, g) :: Dom(prod(cf(D, lift Bool), cf(D, D))). Fix(G D(p, g))"; ;
H_DEF =
|- !D.
  H D = \ (p, g) :: Dom(prod(cf(D, lift Bool), cf(D, D))). Fix(G D(p, g))
H_CF = . |- (H D) ins (cf(prod(cf(D, lift Bool), cf(D, D)), cf(D, D)))

```

This is a straightforward fixed point definition of a recursive function. Note we again assume that `D` is a pointed cpo and define "`H D`" to be seen as a constant term from the viewpoint of our system.

Both `G` and `H` are parameterized by two functions, corresponding to the variables `p` and `q`. Here we differ from LCF where these functions are declared to be constants. We could do the same and then it would not be necessary to parameterize `G` and `H` by the functions. However, defining constants is the same as fixing the functions to be these constants. We would like to leave the choice of the functions open (contradicting ourselves a bit, since `D` is fixed). The following theorem shows this parameterization more explicitly than the definition above

$$\begin{aligned} &|- !D \ p \ g. \\ &\quad p \ ins \ (cf(D, \text{lift Bool})) \ ==> \\ &\quad g \ ins \ (cf(D, D)) \ ==> \\ &\quad (H \ D(p, g) = \text{Fix}(G \ D(p, g))) \end{aligned}$$

Note that the antecedents only assume D is a cpo. It need not be a pointed cpo in order for the equality to hold.

We saw in section 3.8 that the fixed point operator defines a fixed point of a continuous function on a pointed cpo.

$$|- !E. \ pcpo \ E \ ==> \ (!f. \ f \ ins \ (cf(E, E)) \ ==> \ (f(\text{Fix } f) = \text{Fix } f))$$

From this fact the following unfolding theorem about H follows immediately

$$\begin{aligned} &|- !D \ p \ g. \\ &\quad pcpo \ D \ ==> \\ &\quad p \ ins \ (cf(D, \text{lift Bool})) \ ==> \\ &\quad g \ ins \ (cf(D, D)) \ ==> \\ &\quad (H \ D(p, g) = G \ D(p, g)(H \ D(p, g))) \end{aligned}$$

It states that H is equal to one unfolding of G applied to H . Remember that H is defined as the fixed point of G so the left-hand side of this theorem corresponds to the right-hand side of the previous theorem, and vice versa.

Assuming the p argument of H is strict we can prove that H is strict too.

$$\begin{aligned} &|- !D \ p \ g. \\ &\quad pcpo \ D \ ==> \\ &\quad p \ ins \ (cf(D, \text{lift Bool})) \ ==> \\ &\quad (p(\text{bottom } D) = \text{Bt}) \ ==> \\ &\quad g \ ins \ (cf(D, D)) \ ==> \\ &\quad (H \ D(p, g)(\text{bottom } D) = \text{bottom } D) \end{aligned}$$

This follows easily by first unfolding H once and then reducing by definition of G . The assumption about p is used to reduce the Ext of the definition of G to the bottom element of D .

We can now prove the fact that H is idempotent using fixed point induction and the theorems above.

$$\begin{aligned} &|- !D \ p \ g. \\ &\quad pcpo \ D \ ==> \\ &\quad p \ ins \ (cf(D, \text{lift Bool})) \ ==> \\ &\quad (p(\text{bottom } D) = \text{Bt}) \ ==> \\ &\quad g \ ins \ (cf(D, D)) \ ==> \\ &\quad (!x. \ x \ ins \ D \ ==> \ (H \ D(p, g)(H \ D(p, g)x) = H \ D(p, g)x)) \end{aligned}$$

The overall idea is to use fixed point induction on the second and third occurrences of H once we have stripped off the first four antecedents (keeping the inner-most universal quantification).

```

#H_THM; ;
|- !D p g.
  p ins (cf(D, lift Bool)) ==>
  g ins (cf(D, D)) ==>
  (H D(p, g) = Fix(G D(p, g)))

#e(REPEAT GEN_TAC THEN REPEAT DISCH_TAC
# THEN SUBST_OCCS_TAC([[2; 3], UNDISCH_ALL(SPEC_ALL H_THM)])); ;
OK. .
"!x. x ins D ==> (H D(p, g)(Fix(G D(p, g))x) = Fix(G D(p, g))x)"
  [ "pcpo D" ]
  [ "p ins (cf(D, lift Bool))" ]
  [ "p(bottom D) = Bt" ]
  [ "g ins (cf(D, D))" ]

() : void

#e(FPI_TAC[] [] [] "Fix(G(D: ^cpo1)(p, g))"); ;
OK. .
2 subgoals
"!x.
  x ins (cf(D, D)) ==>
  (!x. x ins D ==> (H D(p, g)(x x) = x x)) ==>
  (!x. x ins D ==> (H D(p, g)(G D(p, g)x x) = G D(p, g)x x))"
  [ "pcpo D" ]
  [ "p ins (cf(D, lift Bool))" ]
  [ "p(bottom D) = Bt" ]
  [ "g ins (cf(D, D))" ]

"!x. x ins D ==> (H D(p, g)(bottom(cf(D, D))x) = bottom(cf(D, D))x)"
  [ "pcpo D" ]
  [ "p ins (cf(D, lift Bool))" ]
  [ "p(bottom D) = Bt" ]
  [ "g ins (cf(D, D))" ]

() : void

```

The fixed point induction tactic proves inclusiveness behind the scenes, employing the inclusive prover presented in section 5.3. Note in particular that since D is a pointed cpo it is non-empty. This is required and proved by the inclusive prover.

The base case of the induction proof holds because H is strict. The induction case is first simplified using the definition of G and reducing the dependent lambda abstractions. Then we do a case split on the lifted booleans. Again the bottom case is solved using H is strict. The true and false cases are proved by unfolding H , reducing function constructors and conditionals and using the induction hypothesis.

Our proof is not as compact as the LCF proof though the tactics must do almost exactly the same things to the goals. The difference is that LCF rewriting is quite powerful

(see section 7.1) and at the same time we must use reduction tactics in order to type check arguments of functions before reduction is conducted. Here LCF simply use rewriting which does β -conversion behind the scenes.

7.4 A Mapping Functional for Lazy Sequences

In this section we define a mapping functional `Maps` for lazy sequences which were presented in section 4.2. It is introduced a bit differently than the other functions of this chapter since it is declared as a constructor using `new_constructor_definition`. We present a few theorems about the mapping functional, proved e.g. using structural induction on sequences. An infinite sequence constructor is defined by a fixed point definition and a theorem is presented which relates this constructor and the mapping functional. The proof is conducted by fixed point induction.

The main difference between our approach of introducing a cpo of sequences and the approach used in LCF to define a type of sequences is that we formalize the cpo and its constructors from first principles, which is difficult and time-consuming, whereas LCF uses an axiomatic approach. Lazy sequences are introduced in LCF by declaring constant names for the type and the constructor functions (corresponding to the ones below) and postulating various axioms about the constructors and the partial ordering of the cpo which the type denotes. Axioms for an eliminator functional are then stated and structural induction and other theorems as distinctness of constructors are derived. The same approach is used to introduce both natural numbers and lazy sequences, in fact all recursive datatypes (satisfying certain syntactic restrictions).

Despite the differences in introducing lazy sequences, we work with sequences in the same way as in LCF. This makes sense compared to the natural number example, where we do not, since LCF is useful to reason about lazy types and less useful to reason about strict (flat) datatypes. Using the same approach as for natural numbers would not be easy since HOL has no support for defining recursive functions over non-wellfounded types.

Let us recall the cpo of sequences and its associated constructor and eliminator functions introduced in section 4.2. The constructor for pointed cpos of sequences is called `seq`. We can make `seq` available as a constructor by executing

```
#declare_cpo_constructor seq_PCPO;;
|- !D. cpo D ==> pcpo(seq D)
```

There are two constructors for sequences `Bt_seq` and `Cons_seq` which satisfy the following cases theorem

```
|- !D s.
  s ins (seq D) =
  (s = Bt_seq) \/\
  (?x s'. x ins D /\ s' ins (seq D) /\ (s = Cons_seq x s'))
```

Hence, they are exhaustive on the cpo of sequences. The constructors are distinct too and `Cons_seq` is one-one. The eliminator satisfies the following theorems

```
|- !D E h.
  h ins (cf(D, cf(seq D, E))) ==> (Seq_when h Bt_seq = bottom E)
```

```

|- !D E h x s.
  h i ns (cf(D, cf(seq D, E))) ==>
  x i ns D ==>
  s i ns (seq D) ==>
  (Seq_when h(Cons_seq x s) = h x s)

```

which have been derived from its definition. The continuity theorems about `Cons_seq` and `Seq_when` are used to declare these functions as constructors as follows

```

#declare_constructor Cons_seq_CF;;
|- !D. cpo D ==> Cons_seq i ns (cf(D, cf(seq D, seq D)))

#declare_constructor Seq_when_CF;;
|- !D E.
  cpo D ==>
  pcpo E ==>
  Seq_when i ns (cf(cf(D, cf(seq D, E)), cf(seq D, E)))

```

We also showed that we can do structural induction proofs on sequences.

```

|- !D P.
  cpo D ==>
  i ncl u si ve(P, seq D) ==>
  P Bt_seq ==>
  (!x s'. x i ns D ==> P s' ==> P(Cons_seq x s')) ==>
  (!s. s i ns (seq D) ==> P s)

```

A structural induction tactic has been implemented on top of this theorem which uses the `cpo` prover and the inclusive prover behind the scenes.

Again we start up a new session and a new theory, assuming the above declarations. Then we assume the variables `D` and `E` are cpos, binding these assumptions to ML variables for later use.

```

#new_theory' lcf_seq';;
() : void

#cpo1;;
": (*1 -> bool) # (*1 -> *1 -> bool)" : type

#cpo2;;
": (*2 -> bool) # (*2 -> *2 -> bool)" : type

#let D_CPO = ASSUME"cpo(D: ^cpo1)";;
D_CPO = . |- cpo D

#let E_CPO = ASSUME"cpo(E: ^cpo2)";;
E_CPO = . |- cpo E

```

We shall use the assumptions to define the functional for the mapping function and the mapping function `Maps` itself, or more precisely, to prove these functions are continuous. The functional used to define `Maps` is introduced as a constructor as follows

```

#new_constructor_definition' Maps_FUN_DEF' ' Maps_FUN_CF' [D_CPO; E_CPO]
#"Maps_FUNI (D: ^cpo1, E: ^cpo2) =
# (\g: : Dom(cf(cf(D, E), cf(seq D, seq E))).
#   \f: : Dom(cf(D, E)).
#   \s: : Dom(seq D).
#     Seq_when(\x: : Dom D. \t: : Dom(seq D). Cons_seq(f x)(g f t))s)";;
|- !D E.
  Maps_FUN =
  (\g :: Dom(cf(cf(D, E), cf(seq D, seq E))).
   \f :: Dom(cf(D, E)).
   \s :: Dom(seq D).
   Seq_when(\x :: Dom D. \t :: Dom(seq D). Cons_seq(f x)(g f t))s)
|- !D E.
  cpo D ==>
  cpo E ==>
  Maps_FUN i ns
  (cf(cf(cf(D, E), cf(seq D, seq E)), cf(cf(D, E), cf(seq D, seq E))))

```

The program `new_constructor_definition` defines two constants, one for the internal level of syntax which is the one used in the term argument, and one for the external level of syntax which is the one used in the pretty-printed definition and continuity theorem output. The latter is not parameterized by `cpo`.

We can now use `Maps_FUN` whenever the `cpo` arguments of the internal version can be calculated from the context (i.e. from the arguments of `Maps_FUN`) and it will always occur in pretty-printed results. The mapping functional is defined as the fixed point of this functional.

```

#new_constructor_definition' Maps_DEF' ' Maps_CF' [D_CPO; E_CPO]
#"MapsI (D: ^cpo1, E: ^cpo2) = Fix(Maps_FUNI (D, E))";;
|- !D E. Maps = Fix Maps_FUN
|- !D E.
  cpo D ==> cpo E ==> Maps i ns (cf(cf(D, E), cf(seq D, seq E)))

```

In this definition we have a situation where we cannot determine the `cpo` arguments of the internal version of `Maps_FUN` from the context (it has no arguments) and we must therefore write the `cpo`s explicitly (hence we use `Maps_FUNI`).

It is not immediately obvious from the definition of `Maps` how it actually works on the different kinds of sequences. We have therefore proved two reduction theorems (recursion equations).

```

|- !D E f. f i ns (cf(D, E)) ==> (Maps f Bt_seq = Bt_seq)
|- !D E f x s.
  f i ns (cf(D, E)) ==>
  x i ns D ==>
  s i ns (seq D) ==>
  (Maps f(Cons_seq x s) = Cons_seq(f x)(Maps f s))

```

These are proved easily by using that `Maps` is a fixed point of the functional `Maps_FUN`.

We can now prove that the mapping functional preserves functional composition (defined in section 3.7).

```
|- !D1 D2 D3 f g.
    f i ns (cf(D2, D3)) ==>
    g i ns (cf(D1, D2)) ==>
    (Maps(Comp(f, g)) = Comp(Maps f, Maps g))
```

Setting this statement as a goal in the subgoal package, we first strip the antecedents and then observe that the two continuous functions are equal iff they are equal for all sequences of elements in $D1$, i.e. iff the following term holds

```
"!s. s i ns (seq D1) ==> (Maps(Comp(f, g))s = Comp(Maps f, Maps g)s)"
```

This goal is obtained from a goal stating equality of the functions using the program `X_CONT_FUN_EQ_TAC`, which was described in section 5.6.6.

Next, we use a structural induction tactic for sequences based directly on the structural induction theorem.

```
#e(SEQ_INDUCT_TAC[] [] []);;
```

```
OK.
```

```
2 subgoal s
```

```
"!x s.
```

```
  x i ns D1 ==>
```

```
  s i ns (seq D1) ==>
```

```
  (Maps(Comp(f, g))s = Comp(Maps f, Maps g)s) ==>
```

```
  (Maps(Comp(f, g))(Scons x s) = Comp(Maps f, Maps g)(Scons x s))"
```

```
  [ "cpo D1" ]
```

```
  [ "cpo D2" ]
```

```
  [ "cpo D3" ]
```

```
  [ "f i ns (cf(D2, D3))" ]
```

```
  [ "g i ns (cf(D1, D2))" ]
```

```
"Maps(Comp(f, g))Sbt = Comp(Maps f, Maps g)Sbt"
```

```
  [ "cpo D1" ]
```

```
  [ "cpo D2" ]
```

```
  [ "cpo D3" ]
```

```
  [ "f i ns (cf(D2, D3))" ]
```

```
  [ "g i ns (cf(D1, D2))" ]
```

```
() : void
```

This uses the inclusive prover behind the scenes so we do not have to worry about proving the above equation admits induction. Note that "seq D1" is non-empty since it is a pointed cpo. The proof is finished off using the reduction tactic with the reduction theorems for `Maps` and `Comp`.

Finally, we can define a functional `Seq_of` which given a continuous function `f` and any starting point value `x` generates an infinite sequence of the form

```
"Cons_seq x(Cons_seq(f x)(Cons_seq(f(f x))...))"
```

or written in a more readable way `[x; f(x); f(f(x)); ...]`. This function is the fixed point of a functional called `Seq_of_FUN` (the external name).

```

|- !D.
  Seq_of_FUN =
  (\sf :: Dom(cf(cf(D, D), cf(D, seq D))).
   \f :: Dom(cf(D, D)). \x :: Dom D. Cons_seq x(sf f(f x)))
|- !D. Seq_of = Fi x Seq_of_FUN

```

Both are defined using `new_constructor_definition`. The continuity theorems returned by this program are stated as follows

```

|- !D.
  cpo D ==>
  Seq_of_FUN i ns
  (cf(cf(cf(D, D), cf(D, seq D)), cf(cf(D, D), cf(D, seq D))))
|- !D. cpo D ==> Seq_of i ns (cf(cf(D, D), cf(D, seq D)))

```

It should be apparent from the fixed point definition of `Seq_of` that it always computes an infinite sequence.

We can now prove the following statement about `Seq_of` and `Maps`

```

|- !D f.
  cpo D ==>
  f i ns (cf(D, D)) ==>
  (!x. x i ns D ==> (Seq_of f(f x) = Maps f(Seq_of f x)))

```

Informally, the two sequences are equal since they are both equal to a term corresponding to $[f\ x; f(f\ x); \dots]$. The proof of the theorem is conducted by fixed point induction on both occurrences of `Seq_of`. But first the two first antecedents are stripped off and a case split is performed on whether `D` is empty or not. If it is then an induction is not necessary since "`x i ns D`" must be false. Otherwise, the remaining predicate is proved to be inclusive (as a function of a variable replacing `Seq_of`) and fixed point induction is conducted. Examples of the use of fixed point induction has been given above. We shall not consider the details of this induction proof here.

Again the proofs in our formalization and LCF are based on the same overall idea but tend to be longer in HOL since we do the simplifications explicitly whereas LCF rewriting handles many situations. As mentioned above this probably makes LCF rewriting inefficient at least compared to HOL rewriting but our simplifications are quite inefficient too, since they do more than just rewriting; e.g. they also do type checking.

7.5 Conclusion

The LCF system provides a logic of fixed point theory and is useful for reasoning about nontermination, arbitrary recursive definitions and infinite-valued types as lazy lists. It is unsuitable for reasoning about finite-valued types and strict functions. On the other hand, the HOL system provides set theory and supports reasoning about finite-valued types and total functions well. In this chapter a number of examples conducted in LCF in [Pa87] have been used to demonstrate that an extension of HOL with domain theory combines the benefits of both systems. The examples illustrate reasoning about infinite values and non-terminating functions and show how mixing domain and set theoretic reasoning eases

reasoning about strict LCF datatypes and functions. The extension is more useful than ‘pure’ HOL for reasoning about infinite-valued types and arbitrary recursive functions and more useful than LCF for reasoning about finite-valued types and strict functions. In a way, HOL-CPO can be seen as an embedding of the LCF system in HOL which is performed in such a way that the benefits of the HOL world are not lost.

The natural number example illustrates how we can mix set and domain theoretic reasoning and thereby ease reasoning about finite-valued LCF types and strict functions. We avoid definedness assumptions (complicating statements and proofs in LCF) and natural number inductions involving bottom completely. Another example illustrating this point is presented in chapter 8 (the unification algorithm).

The example on a fixed point definition shows that we can define an arbitrary recursive function in HOL-CPO and reason about it using fixed point induction. This development follows the development in LCF closely and would not be easy in pure HOL.

Finally, the example on lazy sequences gives a definition of an infinite sequence constructor functional as a fixed point and illustrates that we can conduct LCF proofs by fixed point induction and structural induction on lazy recursive domains in HOL-CPO. Again we follow the LCF definitions and proofs closely since pure HOL does not support this kind of reasoning directly.

Some disadvantages of the embedding of domain theory in HOL have also been mentioned. One main problem is that it is time-consuming and not at all straightforward to introduce new recursive domains (see chapter 4). Another problem is that constructors must be parameterized by the domains on which they work. This inconvenience is handled by the interface in most cases but the problem also makes proofs inconvenient since type checking must be performed quite often. LCF rewriting uses β -conversion after function definitions have been expanded to handle this. Rewriting is fairly powerful in LCF, providing conditional rewriting and rewriting with local assumptions, and it handles many situations in a compact way. Therefore LCF ‘induction then rewriting’ proofs about infinite-valued types and lazy evaluation tend to be longer but just as simple in HOL where different tactics are used for each specific task.

HOL-CPO is a semantic embedding of domain theory in a powerful theorem prover. It was an important goal of this embedding that there should be a direct correspondence between elements of domains and elements of HOL types. This allows us to exploit the types and tools of HOL directly and hence, to benefit from mixing domain and set theoretic reasoning as discussed above. The fact that we use predomains, i.e. cpos which do not necessarily have a bottom, also supports this.

A semantic embedding does not always have this property. The formalization of $P\omega$ in [Pe93] builds a separate $P\omega$ world inside HOL so there is no direct relationship between for instance natural numbers in the $P\omega$ model and in the HOL system. The same thing would be true about a HOL formalization of information systems [Wi93].

Chapter 8

Verifying the Unification Algorithm

The problem of finding a common instance of two expressions is called *unification*. The unification algorithm generates a substitution to yield this instance, and returns a failure if a common instance does not exist. The algorithm has played a central role in logic programming and theorem proving after it was used by Robinson in 1965 in his resolution principle for automatic theorem proving [Ro65].

Manna and Waldinger (MW) synthesized the unification algorithm by hand using their deductive tableau system [MW81]. Their informal presentation of the proof was unusually complete providing all lemmas (without proof) and the entire main body of the proof which covered more than 15 pages (the paper itself covered more than 40 pages). The constructive proof from which they extract the algorithm relies on a substantial theory of expressions and substitutions and employs the very general principle of well-founded induction for the proof of correctness. They work in an ordinary untyped first-order logic where variables range over sets and functions are total.

Paulson translated MW's proof to a mechanical proof in LCF [Pa85]. Paulson's proof follows their proof closely. However, he did not deduce the algorithm from the proof as they did; he stated the algorithm first and then proved it was correct. Mechanizing the proof in LCF was rather difficult, both due to the logic, the limited range of available induction strategies and the lack of proper proof infrastructure.

Coen did yet another version of the proof in his implementation of CCL (Classical Computational Logic) in the Isabelle theorem prover [Co92]. His logic supports the proof of correctness well, in particular, because it provides well-founded induction, which is needed for the termination proof of the algorithm.

The underlying logic of the LCF system is entirely different from the logic used by MW since variables range over domains rather than sets and functions may be partial. The consequence of this was that definedness assertions of the form $t \equiv \perp$ appeared everywhere, complicating both statements and proofs considerably, and it was necessary again and again to use and prove the totality of functions which were obviously total. These problems arose since all datatypes and functions were strict (cf. section 7.2 on the natural numbers).

Further, the well-founded induction used by MW was not used in the LCF proof; it is not clear whether or not it is possible to derive well-founded induction in LCF. Instead this was inconveniently translated into two nested structural inductions on the natural numbers and terms, respectively. The algorithm contains an unusual and non-trivial recursion so a simple induction argument was not enough to prove termination.

Finally, the LCF system contained very little proof infrastructure when Paulson started this project. The present tools to define recursive datatypes and to perform proof by structural induction (see section 7.1) were developed by Paulson as part of the project.

In this chapter we describe an implementation of MW's proof in HOL-CPO which is based directly on Paulson's proof in LCF¹. The HOL system seemed to provide a much more appropriate framework for implementing the proof. Its underlying logic is based on set theory and total functions and well-founded induction is available due to the development presented in [Ag91, Ag92]. It also provides substantial theories of sets and lists, among others, which may be useful to formalize the various datatypes in the proof. It supports the definition of new abstract recursive datatypes and primitive recursive functions and it allows proof by structural induction. However, it is not easy to define the unification algorithm directly in pure HOL, since it is not primitive recursive. The unusual recursion in the algorithm makes a proof of termination non-trivial. Here, domain theory appears useful. We can define the recursive algorithm as a fixed point of an appropriate functional and prove termination afterwards, using well-founded induction. Hence, combining the theory of well-founded sets and the formalization of domain theory in HOL, we become able to extend the methods for defining recursive functions in HOL to allow defining functions by well-founded induction (see also the Ackermann example of chapter 2).

In the following sections we describe the implementation of the correctness proof of the unification algorithm in HOL-CPO. The first three sections do not use any domain theory at all!

8.1 Terms

The LCF type of expressions, also called *terms*, is a strict recursive datatype, just like the type of natural numbers discussed in section 7.2. Hence, it denotes a cpo with bottom and the constructor functions are strict. This is a major disadvantage of the LCF formalization; definitions and theorems are infested with the bottom element. Furthermore, all functions on terms except unification itself are primitive recursive and obviously total. Hence, for most parts of the proof the presence of bottom is not even necessary.

In HOL-CPO we can therefore reason in set theory until the point where the unification algorithm must be defined. Terms are introduced as a type in HOL and the theories of substitution and unifiers are developed in pure HOL. Later a cpo of terms is introduced as the discrete universal cpo of elements of type `term`. This approach simplifies the HOL proof of correctness considerably compared with the LCF proof.

The type of terms is introduced using the type definition package. The type specification which is the argument of the `define_type` program is

```
term = Const name | Var name | Comb term term
```

where `name` is a type abbreviation for `num`. Any type could be used for the type of names, and further, names of constants and variables could even be different. We choose `num` for convenience; if we used a polymorphic type we would have to type terms explicitly. The type definition package provides the theorems and tools we need to

¹Larry Paulson was so kind to send me the actual LCF proof. It was useful to have access to a version of the proof which was already structured for a machine.

reason about terms. It supports definition by primitive recursion and proof by cases and induction. Furthermore, it provides theorems stating the constructors are distinct and one-one. Paulson had to implement the corresponding LCF tools for the LCF proof of correctness.

8.1.1 Occurrence Relation

A notion of subterm is introduced by the occurs-in relation. This relation appears in both an irreflexive version for proper subterm, defined as the infix $\$<<$, and a reflexive version which allows terms to be equal, defined as the infix $\$<<=$.

$$\begin{aligned} &|- !t\ u. \ t \ll = u = (t = u) \ \vee \ t \ll u \\ &|- (!t\ c. \ t \ll (\text{Const } c) = F) \ \wedge \\ &\quad (!t\ v. \ t \ll (\text{Var } v) = F) \ \wedge \\ &\quad (!t\ t_1\ t_2. \ t \ll (\text{Comb } t_1\ t_2) = t \ll = t_1 \ \vee \ t \ll = t_2) \end{aligned}$$

The second of these two theorems is not the actual definition of the occurs-in relation; mutual recursive definitions are not supported in HOL. The relation is introduced by a proper primitive recursive definition using $\$=$ and $\$<<$ instead of $\$<<=$. Below we shall usually think of $\$<<$ as the occurs-in relation and not use $\$<<=$ often.

The occurrence ordering is a partial ordering on terms. The following theorems state it is irreflexive, transitive and anti-symmetric.

$$\begin{aligned} &|- !t. \ \sim t \ll t \\ &|- !t\ u. \ t \ll u \implies (!v. \ u \ll v \implies t \ll v) \\ &|- !t\ u. \ \sim(t \ll u \ \wedge \ u \ll t) \end{aligned}$$

In fact, it is a well-founded ordering on terms similar to the less-than ordering $<$ on natural numbers (see section 8.6.1). Transitivity is useful to prove the irreflexivity together with the following lemmas.

$$\begin{aligned} &|- (!t\ t'. \ \sim(\text{Comb } t\ t' = t)) \ \wedge \ (!t'\ t. \ \sim(\text{Comb } t\ t' = t')) \\ &|- (!t\ t'. \ t \ll (\text{Comb } t\ t')) \ \wedge \ (!t\ t'. \ t' \ll (\text{Comb } t\ t')) \end{aligned}$$

The former states that the combination of two terms is always different from either term, a kind of idempotence fact. The latter follows immediately from the definition of the occurrence relation.

In order to illustrate the mess that the presence of the bottom element makes of LCF definitions and theorems, we show the LCF axiom for the occurrence relation

$$\begin{aligned} &t\ OCCS\ \perp \equiv \perp \wedge \\ &(\forall c. \ c \neq \perp \implies t\ OCCS\ (\text{CONST } c) \equiv FF) \wedge \\ &(\forall v. \ v \neq \perp \implies t\ OCCS\ (\text{VAR } v) \equiv FF) \wedge \\ &(\forall t_1\ t_2. \ t_1 \neq \perp \implies t_2 \neq \perp \implies \\ &\quad t\ OCCS\ (\text{COMB } t_1\ t_2) \equiv (t\ OCCS_EQ\ t_1) \ \text{OR} \ (t\ OCCS_EQ\ t_2)) \end{aligned}$$

which is universally quantified with the variable t . Compare this with the definition above. All other concepts that we define easily by primitive recursion or as simple HOL definitions are similarly complicated in LCF. Besides, LCF proofs about functions over terms might use that functions are strict and total all the time. We avoid this completely!

8.1.2 Variable Set

Finite sets of variables play a central role in the proof, in particular to prove termination by well-founded induction. The set of variables of a term is defined by primitive recursion as follows

$$\begin{aligned} &|- (!c. \text{vars}(\text{Const } c) = \{\}) \wedge \\ & \quad (!v. \text{vars}(\text{Var } v) = \{v\}) \wedge \\ & \quad (!t1\ t2. \text{vars}(\text{Comb } t1\ t2) = (\text{vars } t1) \text{ UNION } (\text{vars } t2)) \end{aligned}$$

It follows immediately that a variable v is in the set of variables of "Var v ".

$$|- !v. v \text{ IN } (\text{vars}(\text{Var } v))$$

The set is just the singleton containing the variable itself. To reason about sets we use the predicate sets library [Me92] of the HOL system, which provides a kind of typed sets represented as subsets of HOL types. Again, this kind of set theory had to be developed for the proof in LCF. This caused some trouble since sets had to allow proof by structural induction, which was derived from fixed point induction. This was accomplished by representing sets as equivalence classes of lists [Pa84a]. In particular, it seems strange that there is a bottom set, namely the bottom element of the type of sets. In HOL, the predicate set theory is less awkward and set induction is provided by the library.

The variables of a term are precisely the variables that occur in or are equal to the term.

$$|- !v\ t. v \text{ IN } (\text{vars } t) = (\text{Var } v) \ll= t$$

This fact is called *variables* below. As a corollary of the variables theorem we obtain the following theorem

$$|- !v\ t. \sim(\text{Var } v) \ll t = \sim v \text{ IN } (\text{vars } t) \vee (\text{Var } v = t)$$

which states that if a variable does not occur in a term then it is equal to that term or it is not in the set of variables of the term, and vice versa. Finally, the variables function is monotonic with respect to the reflexive occurrence relation.

$$|- !t\ t'. t \ll= t' ==> (\text{vars } t) \text{ SUBSET } (\text{vars } t')$$

Note that using \ll would not allow us to strengthen the statement to proper subsets.

8.2 Substitutions

In [MW81], a substitution is a set of replacement pairs $\{x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n\}$ where the x_1, \dots, x_n are distinct variables and t_1, \dots, t_n are terms such that $t_i \neq x_i$. Hence, trivial substitutions such as $\{x \leftarrow x\}$ and ambiguous substitutions such as $\{x \leftarrow t, x \leftarrow t'\}$ are not allowed.

In HOL, a substitution can be represented by a list of pairs of names and terms. This type is called `rpl list` which stands for list of replacement pairs. A substitution "[v, t]" consisting of a single replacement pair is called a *replacement*. The list type is built-in and HOL provides primitive recursive definitions, induction and many useful

theorems. In the formalization, trivial substitutions like "[v, Var v]" where v is a name behave in the same way as the empty substitution [], and ambiguous substitutions like "[(v, t); (v, t')]" where t and t' are terms behave like "[v, t]". Hence, HOL equality of lists is not feasible for substitutions and we must define another equality which corresponds to the informal 'behaves in the same way as'.

Paulson says it is impractical to use lists of pairs for substitutions because, I think, he insists on using the logical equality of LCF (denoted by \equiv) instead of defining a new equality. So he defines a new type, corresponding to lists of pairs but with additional axioms to get rid of trivial and ambiguous substitutions, and states an axiom for equality of substitutions which is different from the axiom for equality of lists.

Below, we define a number of concepts associated with substitutions. Equality of substitutions is based on the notion of agreement which in turn is based on the notion of applying a substitutions to a term. The domain and range of a substitution are the set of variables affected and introduced by the substitution, respectively. A substitution can be composed with another, an (inefficient) operation used when unifying combinations in the algorithm, and we can define a notion of generality on substitutions. Finally, we will define idempotent substitutions. The unification algorithm constructs most-general and idempotent substitutions.

8.2.1 Application

A substitution is represented by a list of pairs of names and terms. The type of substitutions is called `rplst` which abbreviates the type "`(name#term)list`".

A substitution is applied to a term by simultaneously replacing every instance of variables of the substitution by the corresponding terms.

$$\begin{aligned} &|- (!c s. (Const c) subst s = Const c) \wedge \\ & \quad (!v s. (Var v) subst s = lookup v s) \wedge \\ & \quad (!t1 t2 s. (Comb t1 t2) subst s = Comb(t1 subst s)(t2 subst s)) \end{aligned}$$

Hence, a constant is left unchanged and a combination becomes a combination where the substitution is applied to each of the two subterms. The result of applying a substitution to a variable is defined using an auxiliary function called `lookup`.

$$\begin{aligned} &|- (!v. lookup v[] = Var v) \wedge \\ & \quad (!v x t s. lookup v(CONS(x, t)s) = ((v = x) => t | lookup v s)) \end{aligned}$$

The `lookup` function searches a substitution `s` for the term associated with the first occurrence of a specified variable `v`. If `v` is not found it returns "`Var v`", making application of a substitution to a variable behave as identity. Note that the definition of `lookup` identifies "[(v, t); (v, t')]" with "[v, t]" and "[v, Var v]" with [].

It follows directly by definitions and induction on terms that the empty substitution behaves as identity when applied to a term.

$$|- !t. t subst [] = t$$

Another useful theorem is the following cases theorem for applying a substitution to a variable.

$$\begin{aligned} &|- (!v\ t\ s.\ (\text{Var } v)\ \text{subst } (\text{CONS}(v, t)s) = t) \wedge \\ &\quad (!v\ w\ t\ s.\ \\ &\quad \quad \neg(v = w) \implies ((\text{Var } v)\ \text{subst } (\text{CONS}(w, t)s) = (\text{Var } v)\ \text{subst } s)) \end{aligned}$$

Finally, we have proved that the application of a substitution is monotonic with respect to the occurrence relation.

$$|- !s\ t\ u.\ t \ll u \implies (t\ \text{subst } s) \ll (u\ \text{subst } s)$$

This follows by a simple induction proof on terms. The reflexive occurrence relation is also preserved by application of a substitution.

8.2.2 Domain and Range

The domain of a substitution is the set of variables affected by the substitution.

$$\begin{aligned} &|- (\text{domain}[] = \{\}) \wedge \\ &\quad (!v\ t\ s.\ \\ &\quad \quad \text{domain}(\text{CONS } (v, t) s) = \\ &\quad \quad ((t = \text{Var } v) \implies (\text{domain } s)\ \text{DELETE } v \mid v\ \text{INSERT } (\text{domain } s))) \end{aligned}$$

This notion has a relatively complicated definition because substitutions may contain a trivial replacement pair of the form " $(v, \text{Var } v)$ " as the first pair with v as a variable. Such a pair has no affect on a term. The following theorem states that the informal description of the domain above holds,

$$|- !v\ s.\ v \text{ IN } (\text{domain } s) = \neg((\text{Var } v)\ \text{subst } s = \text{Var } v)$$

namely that a variable is in the domain of a substitution iff the substitution does not leave that variable unchanged. This fact is called *domain characterization* below.

The range of a substitution is the set of variables that a substitution may introduce. Hence the range should consist of the variables of the terms associated with variables of the domain.

$$\begin{aligned} &|- !s.\ \\ &\quad \text{range } s = \text{SET_UNION}(\text{IMAGE}(\backslash v.\ \text{vars}((\text{Var } v)\ \text{subst } s))(\text{domain } s)) \end{aligned}$$

The constant `SET_UNION` is a generalized version of the binary `UNION`.

$$|- !X.\ \text{SET_UNION } X = \{e \mid \exists x.\ e \text{ IN } x \wedge x \text{ IN } X\}$$

Note that in this definition the variable X is a set of sets. The following characterization of the range may be a bit easier to read than the definition.

$$\begin{aligned} &|- !v\ s.\ \\ &\quad v \text{ IN } (\text{range } s) = \\ &\quad (\exists w.\ w \text{ IN } (\text{domain } s) \wedge v \text{ IN } (\text{vars}((\text{Var } w)\ \text{subst } s))) \end{aligned}$$

That is, the range is the set of all variables that may be introduced by a substitution. This theorem is called *range characterization* below.

Finally, a couple of properties that relate the variables of the domain and range of a substitution with the variables of terms.

The first theorem is called *variable elimination*.

$$\begin{aligned} &|- !v s. \\ &\quad v \text{ IN } (\text{domain } s) \implies \\ &\quad \sim v \text{ IN } (\text{range } s) \implies (!t. \sim v \text{ IN } (\text{vars}(t \text{ subst } s))) \end{aligned}$$

If a variable is in the domain but not in the range of a substitution then it is eliminated from any term that substitution is applied to. The following lemma follows from variable elimination and range characterization.

$$\begin{aligned} &|- !s. \\ &\quad \text{DISJOINT}(\text{domain } s)(\text{range } s) = \\ &\quad (!t. \text{DISJOINT}(\text{domain } s)(\text{vars}(t \text{ subst } s))) \end{aligned}$$

This statement is called the *disjoint* theorem for domain and range.

The second theorem is called *variable introduction*.

$$|- !v s t. v \text{ IN } (\text{vars}(t \text{ subst } s)) \implies v \text{ IN } (\text{range } s) \vee v \text{ IN } (\text{vars } t)$$

If a variable is in the result of applying a substitution to a term then it occurred in the term originally or it was introduced by the substitution (i.e. it is in the range of the substitution).

8.2.3 Agreement and Equality

Two substitutions are said to agree on a term if applying the substitutions to the term yields the same result. The following proposition, called *agreement*, states that two substitutions agree on a term iff they agree on all variables of the term.

$$\begin{aligned} &|- !r s t. \\ &\quad (t \text{ subst } r = t \text{ subst } s) = \\ &\quad (!v. v \text{ IN } (\text{vars } t) \implies ((\text{Var } v) \text{ subst } r = (\text{Var } v) \text{ subst } s)) \end{aligned}$$

This theorem has many consequences. For instance, we can prove the *invariance* property

$$|- !s t. (t \text{ subst } s = t) = \text{DISJOINT}(\text{vars } t)(\text{domain } s)$$

which states that a term is invariant under a substitution iff the set of variables of the term and the domain of the substitution are disjoint. Another invariance property, called *replacement invariance*, is

$$|- !v t. \sim v \text{ IN } (\text{vars } t) \implies (!u s. t \text{ subst } (\text{CONS}(v,u)s) = t \text{ subst } s)$$

which states that when applying a substitution to a term the replacement pairs where the variable does not occur in the term can be disregarded.

Equality of two substitutions should state that the substitutions behave in the same way. Hence, we define two substitutions to be equal when they agree on all terms.

$$|- !r s. r == s = (!t. t \text{ subst } r = t \text{ subst } s)$$

Among the basic properties that this equality enjoys we have proved reflexivity, symmetry and transitivity

$$\begin{aligned} &|- !s. s == s \\ &|- !r s. r == s = s == r \\ &|- !s1 s2. s1 == s2 ==> (!s3. s2 == s3 ==> s1 == s3) \end{aligned}$$

which follow immediately by definition using properties of HOL equality.

In order to prove two substitutions are equal it suffices to prove they agree on all variables.

$$|- !r s. r == s = (!v. (Var v) subst r = (Var v) subst s)$$

This is yet another consequence of the agreement proposition. In fact, it suffices to prove the substitutions agree on all variables in their domains.

$$\begin{aligned} &|- !r s. \\ &\quad r == s = \\ &\quad (!v. \\ &\quad \quad v \text{ IN } ((\text{domain } r) \text{ UNION } (\text{domain } s)) ==> \\ &\quad \quad ((Var v) subst r = (Var v) subst s)) \end{aligned}$$

This is a corollary of the previous fact. The two theorems are called the *equality* theorem and corollary for substitutions, respectively.

The proof of the following *decomposition* theorem is based on the equality theorem for substitutions.

$$|- !v s. s == (\text{CONS}(v, (Var v) subst s)s)$$

In particular we have the theorem

$$|- !v. [v, Var v] == []$$

which states that trivial substitutions are identified with the empty substitution under the equality of substitutions. We can also prove that the order in which replacement pairs appear in substitutions does not matter provided the variables are different.

$$\begin{aligned} &|- !v w t t' s. \\ &\quad \neg(v = w) ==> \text{CONS}(v, t)(\text{CONS}(w, t')s) == \text{CONS}(w, t')(\text{CONS}(v, t)s) \end{aligned}$$

Another consequence of the equality theorem is the theorem

$$|- !v t t' s. \text{CONS}(v, t)(\text{CONS}(v, t')s) == \text{CONS}(v, t)s$$

which states how ambiguous substitutions are interpreted—all replacement pairs with the same variable component are disregarded, except the first one. These theorems justify the choice of equality of substitutions.

8.2.4 Composition

Composition of substitutions is defined by primitive recursion on lists.

$$\begin{aligned} &|- (!s. [] \text{ thens } s = s) \wedge \\ &\quad (!v t r s. (\text{CONS}(v, t)r) \text{ thens } s = \text{CONS}(v, t \text{ subst } s)(r \text{ thens } s)) \end{aligned}$$

A useful lemma in proofs is the following *application-composition* theorem

$$\vdash \! \lambda t r s. t \text{ subst } (r \text{ thens } s) = (t \text{ subst } r) \text{ subst } s$$

which relates application of a substitution to composition. It is proved by induction over terms. Associativity of composition which is stated by the theorem

$$\vdash \! \lambda s_1 s_2 s_3. (s_1 \text{ thens } s_2) \text{ thens } s_3 = s_1 \text{ thens } (s_2 \text{ thens } s_3)$$

is proved using list induction, and the previous fact as well. Note that this theorem is stated using HOL equality. It could be stated using substitution equality but this would be a weaker statement. List induction also proves the second conjunct of the following theorem

$$\vdash (\! \lambda s. [] \text{ thens } s = s) \wedge (\! \lambda s. s \text{ thens } [] = s)$$

which states that the empty substitution is both a left and a right identity of composition. The first conjunct follows immediately from the definition.

Finally, let us show the *addition-composition* theorem

$$\vdash \! \lambda v t s. ([v, t] \text{ thens } s) == (\text{CONS}(v, t \text{ subst } s)s)$$

which is just a special case of the second conjunct in the definition of composition.

8.2.5 Generality

A substitution s_1 is *more general* than a substitution s_2 if s_2 can be obtained from s_1 by composition with some substitution.

$$\vdash \! \lambda s_1 s_2. \text{more_gen}(s_1, s_2) = (\?r. s_2 == (s_1 \text{ thens } r))$$

In this case, we also say that s_2 is an instance of s_1 . Clearly, generality is a reflexive notion

$$\vdash \! \lambda s. \text{more_gen}(s, s)$$

and the empty substitution is more general than any substitution.

$$\vdash \! \lambda s. \text{more_gen}([], s)$$

Roughly speaking, a more general substitution makes fewer changes to a term.

8.2.6 Idempotent Substitutions

A substitution is called idempotent if composing the substitution with itself yields the substitution itself. This statement is equivalent to the statement that the domain and range of the substitution are disjoint.

$$\vdash \! \lambda s. (s \text{ thens } s) == s = \text{DISJOINT}(\text{domain } s)(\text{range } s)$$

This theorem is called *idempotence*. Its proof is based on the invariance theorem and the disjoint theorem about domains and ranges. A consequence of idempotence and range characterization states that replacements are idempotent provided the variable does not occur in the term of the replacement pair.

$$\vdash \! \lambda v t. \sim(\text{Var } v) \ll t ==> ([v, t] \text{ thens } [v, t]) == [v, t]$$

This fact is called *replacement idempotence* below.

8.3 Unifiers

In the previous section we presented a number of definitions and lemmas about substitutions which are directly related to the proof of correctness of the unification algorithm. In this section we shall continue this development by considering certain substitutions which can be used to produce a common instance of two terms by application. Such a substitution is called a unifier.

$$\vdash \! \exists s \ t \ u. \text{ unifier}(s, t, u) = (t \text{ subst } s = u \text{ subst } s)$$

A unifier does not always exist (see below). If it does we say the terms are unifiable. The following properties of the `unifier` predicate follows immediately from the definition. The theorem

$$\vdash \! \exists s \ t. \text{ unifier}(s, t, t)$$

states that any substitution unifies a term with itself and the theorem

$$\vdash \! \exists s \ t \ u. \text{ unifier}(s, t, u) = \text{ unifier}(s, u, t)$$

states that the `unifier` predicate is commutative. The following clauses state when a substitution unifies constants and combinations.

$$\begin{aligned} \vdash \! \exists s \ c1 \ c2. \text{ unifier}(s, \text{Const } c1, \text{Const } c2) &= (c1 = c2) \ \wedge \\ \! \exists s \ t1 \ t2 \ u1 \ u2. & \\ \text{ unifier}(s, \text{Comb } t1 \ u1, \text{Comb } t2 \ u2) &= \\ \text{ unifier}(s, t1, t2) \ \wedge \text{ unifier}(s, u1, u2) & \end{aligned}$$

Any substitution unifies two constants iff they are equal and a substitution is a unifier of two combinations iff it unifies the subterms pairwise. This follows from the definition of application of substitutions. It also follows that a constant and a combination cannot be unified, stated by

$$\vdash \! \exists c \ t \ u. \text{ cant_unify}(\text{Const } c, \text{Comb } t \ u)$$

where the constant `cant_unify` is defined as follows

$$\vdash \! \exists t \ u. \text{ cant_unify}(t, u) = (\! \exists s. \sim \text{ unifier}(s, t, u))$$

From the unifier clauses theorem above it follows immediately that different constants cannot be unified.

$$\vdash \! \exists c1 \ c2. \sim(c1 = c2) \implies \text{ cant_unify}(\text{Const } c1, \text{Const } c2)$$

If names `c1` and `c2` are different then so are "`Const c1`" and "`Const c2`" since `Const` is one-one. The constant `cant_unify` is commutative

$$\vdash \! \exists t \ u. \text{ cant_unify}(t, u) = \text{ cant_unify}(u, t)$$

since `unifier` is commutative.

8.3.1 Most-general and Idempotent Unifiers

A unifier is a substitution so a unifier may be more general than another unifier. In fact, we have that any substitution which is an instance of a unifier is also a unifier.

$$\vdash !s\ t\ u. \text{unifier}(s, t, u) \implies (!r. \text{more_gen}(s, r) \implies \text{unifier}(r, t, u))$$

A most-general unifier is a unifier which is more general than all other unifiers, defined by

$$\vdash !s\ t\ u. \\ \text{most_gen_unifier}(s, t, u) = \\ \text{unifier}(s, t, u) \wedge (!r. \text{unifier}(r, t, u) \implies \text{more_gen}(s, r))$$

The following *characterization* theorem about most-general unifiers follows from the previous fact.

$$\vdash !s\ t\ u. \\ \text{most_gen_unifier}(s, t, u) = (!r. \text{unifier}(r, t, u) = \text{more_gen}(s, r))$$

So a substitution S is a most-general unifier iff any substitution is a unifier precisely when it is an instance of S . By commutativity properties presented above we have the theorem

$$\vdash !s\ t\ u. \text{most_gen_unifier}(s, t, u) = \text{most_gen_unifier}(s, u, t)$$

which states that also most-general unifier is a commutative notion.

The following theorem, called *variable unifier*, is important since it is the main lemma of the proof that the unification algorithm correctly unifies a variable and a term.

$$\vdash !v\ t. \sim(\text{Var } v) \ll t \implies \text{most_gen_unifier}([v, t], \text{Var } v, t)$$

It states that if a variable v does not occur in a term t the replacement " $[v, t]$ " is a most-general unifier. This is the first of the theorems listed above which does not have a straightforward proof. After expanding definitions in the consequent of the implication recursively down to a term which does not contain other constants than `subst` we have two statements to prove (recall `most_gen_unifier` is a conjunction). The first branch of the proof uses the corollary of the variables theorem and the replacement invariance theorem. The second branch uses the addition-composition theorem and the decomposition theorem.

Most-general unifiers are not unique since the more general property of substitutions is not anti-symmetric; two substitutions may be more general than each other without being equal. We shall consider most-general unifiers which are also idempotent since such unifiers have useful properties for the proof of correctness of the unification algorithm. A most-general, idempotent unifier is called a best unifier.

$$\vdash !s\ t\ u. \\ \text{best_unifier}(s, t, u) = \text{most_gen_unifier}(s, t, u) \wedge (s \text{ thens } s) == s$$

The first useful property states that a unifier S is a best unifier iff composing it with any unifier makes the action of S disappear.

$$\begin{aligned} &|- !s \ t \ u. \\ &\quad \text{uni fi er}(s, t, u) ==> \\ &\quad (\text{best_uni fi er}(s, t, u) = (!r. \text{uni fi er}(r, t, u) ==> (s \ \text{thens} \ r) == r)) \end{aligned}$$

This fact is proved from definitions and application-composition. The theorem is also useful in the following variant, called *best unifier characterization*.

$$\begin{aligned} &|- !s \ t \ u. \\ &\quad \text{best_uni fi er}(s, t, u) = \\ &\quad \text{uni fi er}(s, t, u) \wedge (!r. \text{uni fi er}(r, t, u) ==> (s \ \text{thens} \ r) == r) \end{aligned}$$

Two important properties of best unifiers which are not at all obvious are stated as follows

$$\begin{aligned} &|- !s \ t \ u. \\ &\quad \text{best_uni fi er}(s, t, u) ==> \\ &\quad (\text{domain } s) \text{ SUBSET } ((\text{vars } t) \text{ UNION } (\text{vars } u)) \\ &|- !s \ t \ u. \\ &\quad \text{best_uni fi er}(s, t, u) ==> \\ &\quad (\text{range } s) \text{ SUBSET } ((\text{vars } t) \text{ UNION } (\text{vars } u)) \end{aligned}$$

These state that a best unifier may only contain variables that appear in the terms it unifies. This is used to prove termination of the unification algorithm, i.e. to establish the well-founded ordering holds in certain cases. The theorems are called the *domain* and the *range* properties of best unifiers, respectively. Their proofs are among the most difficult ones, relying on many theorems presented above as e.g. idempotence, invariance, replacement invariance and the characterization theorems for domain and range.

8.3.2 Best Unifiers and their Existence

Below we list a number of theorems concerned with the unification problem stating properties of best unifiers as well as which terms cannot be unified. They bear directly on the correctness proof of the unification algorithm. That is, if terms cannot be unified the algorithm should fail and otherwise it should return a substitution which is a best unifier as stated by the theorems. A few of the theorems listed below have been presented above but are mentioned again for completeness.

In the proof of the unification algorithm we shall make a case split on whether the two terms to be unified are equal or not. If they are the same, then the empty substitution is a best unifier.

$$|- !t. \ \text{best_uni fi er}([], t, t)$$

So the algorithm should return the empty substitution, or some equivalent substitution due to the following theorem

$$|- !r \ s. \ r == s ==> (!t \ u. \ \text{best_uni fi er}(r, t, u) = \text{best_uni fi er}(s, t, u))$$

If the terms are not equal we shall make a case split on both term arguments. Distinct constants and constants and combinations cannot be unified as we saw above.

|- !c1 c2. ~(c1 = c2) ==> cant_uni_fy(Const c1, Const c2)
 |- !c t u. cant_uni_fy(Const c, Comb t u)

As a corollary of the variable unifier theorem we obtain the following theorem for unifying a variable and a term

|- !v t. ~(Var v) << t ==> best_uni_fier([v, t], Var v, t)

It states that if a variable v does not occur in a term t then the replacement " $[v, t]$ " is a best unifier. For the opposite case, it has been proved that if a term occurs in another term then the terms cannot be unified.

|- !t u. t << u ==> cant_uni_fy(t, u)

This follows from monotonicity of application of a substitution and the irreflexivity of the occurrence relation.

We now only need to consider how two combinations are unified because the notions of best unifier and 'cannot be unified' are commutative.

|- !s t u. best_uni_fier(s, t, u) = best_uni_fier(s, u, t)
 |- !t u. cant_uni_fy(t, u) = cant_uni_fy(u, t)

Since a substitution is a unifier of a combination iff it unifies the subterms pairwise, stated by the unifier clauses theorem, it is obvious that if the operators of two combinations cannot be unified then the combinations cannot be unified either.

|- !t u.
 cant_uni_fy(t, u) ==> (!t' u' . cant_uni_fy(Comb t t' , Comb u u'))

It is less obvious but still straightforward to prove the following theorem

|- !s t1 u1.
 best_uni_fier(s, t1, u1) ==>
 (!t2 u2.
 cant_uni_fy(t2 subst s, u2 subst s) ==>
 cant_uni_fy(Comb t1 t2, Comb u1 u2))

which states that a combination cannot be unified if the operands cannot be unified after a best unifier of the operators have been applied to the operands. Finally, if we have best unifiers for the operators and the corresponding instances of the operands, then their composition is a best unifier of the combination.

|- !s1 t1 u1.
 best_uni_fier(s1, t1, u1) ==>
 (!s2 t2 u2.
 best_uni_fier(s2, t2 subst s1, u2 subst s1) ==>
 best_uni_fier(s1 thens s2, Comb t1 t2, Comb u1 u2))

These theorems directly reflect the way in which the unification algorithm works. First it attempts to unify the operators and, if it succeeds, it applies the resulting substitution to the operands and attempts to unify these instances.

8.4 The Unification Algorithm

In this section we define an algorithm to produce a unifier of two terms, if a unifier exists. The algorithm tries to construct a best unifier. If this attempt is successful it returns the unifier and otherwise it returns a failure. From the theorems listed above, we already have a pretty good idea about how the algorithm should work, at least for variables and constants. Unifying two combinations involves an unusual recursion so the unification algorithm cannot be defined by primitive recursion. Instead it is defined as a fixed point in domain theory.

Below we first describe a type of attempts with two constructor functions for failure and success, respectively. Then we introduce discrete universal cpos for terms, substitutions and attempts. A number of continuous functions used to define the unification algorithm are also introduced. Finally, the algorithm is defined and its most important properties are presented.

8.4.1 The Type of Attempts

An attempt is either a failure or a success in the form of a substitution.

```
attempt = Failure | Success rplist
```

This type could be defined using the type definition package but it is also easy to define as an abbreviation for the sum type `": unit + rplist"` where `unit` is an abbreviation for the type `": one"`. The constructor for failure is then defined by

```
|- Failure = INL one
```

and the constructor for success is defined by

```
|- !s. Success s = INR s
```

Note that in the first definition `one` is an element of type `": unit"`. From properties of the injection functions we immediately obtain

```
|- !s. ~(Failure = Success s)
|- !s s'. (Success s = Success s') = (s = s')
```

which state the constructors are distinct and one-one.

The unification algorithm is correct if an attempt to unify two terms satisfies the predicate `best_unify_try`, defined as follows:

```
|- !a t u.
  best_unify_try(a, t, u) =
  (a = Failure) /\ cant_unify(t, u) \/\
  (?s. (a = Success s) /\ best_unifier(s, t, u))
```

Hence, if the algorithm fails it should not be possible to unify the term arguments and if it succeeds it should return a best unifier. From this definition and the properties of the constructors for attempts we can easily prove the following clauses

```
|- (!t u. best_unify_try(Failure, t, u) = cant_unify(t, u)) /\
  (!s t u. best_unify_try(Success s, t, u) = best_unifier(s, t, u))
```

Besides the predicate is commutative

```
| - ! a t u. best_unify_try(a, t, u) = best_unify_try(a, u, t)
```

since `best_unifier` and `cant_unify` are both commutative.

8.4.2 Domain Theory

All developments above have been conducted in pure HOL only, the need for domain theory never arose. However, in order to define the unification algorithm as a fixed point we shall now turn our attention towards domain theory. The algorithm will be a recursive function taking two arguments in the cpo of terms and yielding a result in the lifted cpo of attempts. Hence, the unification algorithm is a partial function by definition but we shall prove it always terminates, i.e. it is total.

We introduce a number of discrete universal cpos using the same approach as in the previous chapter.

```
| - cpo name
| - cpo term
| - cpo rplist
| - cpo unit
```

In all cases the underlying types of the cpos are the types which have the same names as the cpos. The cpo of attempts is defined a bit differently, as a sum of two cpos:

```
| - attempt = sum(unit, rplist)
| - cpo attempt
```

The cpo of attempts is in fact a discrete universal cpo too, since the sum of discrete universal cpos is itself a discrete universal cpo. It is defined as a sum cpo because we can then use the `Sum` function to construct continuous functions. The same program `new_cpo_definition` is used to introduce all cpos.

Since the cpos above are discrete universal cpos and since the continuous function space of discrete universal cpos is itself a discrete universal cpo, constants belong to such cpos trivially. Hence, the constructor functions for terms are continuous

```
| - Const ins (cf(name, term))
| - Var ins (cf(name, term))
| - Comb ins (cf(term, cf(term, term)))
```

and application of a substitution and composition are continuous.

```
| - $subst ins (cf(term, cf(rplist, term)))
| - $thens ins (cf(rplist, cf(rplist, rplist)))
```

The constructor for failure is in the cpo of attempts,

```
| - Failure ins attempt
```

and the constructor for success is a continuous function from the cpo of substitutions (called `rplist`) to the cpo of attempts.

```
|- Success ins (cf(rplist, attempt))
```

Such facts are proved automatically using the program `ins_prover` and HOL-CPO is made aware of the facts if we use the program `declare`.

In order to be able to define the unification of combinations conveniently, we define a kind of composition function for attempts. This function is introduced using the program `new_constant_definition` which returns the following definition and continuity theorem.

```
|- athen =
  (\a :: Dom(lift attempt).
   \s :: Dom(cf(rplist, lift attempt)).
   Ext
   (\a' :: Dom attempt.
    Sum((\x :: Dom unit. Lift Failure), s)a' )a)
|- athen ins
  (cf(lift attempt, cf(cf(rplist, lift attempt), lift attempt)))
```

The following clauses are proved automatically from the definition

```
|- (!f. f ins (cf(rplist, lift attempt))) ==> (athen Bt f = Bt)) /\
  (!f.
   f ins (cf(rplist, lift attempt)) ==>
   (athen(Lift Failure)f = Lift Failure)) /\
  (!f s.
   f ins (cf(rplist, lift attempt)) ==>
   (athen(Lift(Success s))f = f s))
```

using the constructor reduction tactic. This tactic requires that the argument of the `Sum` constructor function is in a sum cpo. Therefore, `attempt` was defined as a sum cpo and not as a discrete universal cpo. The continuous function space of which `athen` is an element is not a discrete universal cpo since the cpo for the second argument is not a discrete universal cpo, or even a lifted discrete universal cpo. The reason for this is that the result of unifying two terms is in a pointed cpo "`lift attempt`" in order to allow a fixed point definition of the unification algorithm. This means that `athen` cannot be defined as a strict extension of some simple primitive recursive HOL function.

Another function which is also a bit special is the 'when' eliminator functional for terms. The following clauses are derived from its definition.

```
|- (!D f g h c.
   f ins (cf(name, D)) /\
   g ins (cf(name, D)) /\
   h ins (cf(term, cf(term, D))) ==>
   (term_when f g h(Const c) = f c)) /\
  (!D f g h v.
   f ins (cf(name, D)) /\
   g ins (cf(name, D)) /\
   h ins (cf(term, cf(term, D))) ==>
   (term_when f g h(Var v) = g v)) /\
```

```
(!D f g h t u.
  f i ns (cf(name, D)) /\
  g i ns (cf(name, D)) /\
  h i ns (cf(term, cf(term, D))) ==>
  (term_when f g h(Comb t u) = h t u))
```

It is defined as a new constructor using `new_constructor_definition` which returns the following theorems

```
|- !D.
  term_when =
  (\f g :: Dom(cf(name, D)).
    \h :: Dom(cf(term, cf(term, D))).
    \t :: Dom term. term_cases D t f g h)
|- !D.
  cpo D ==>
  term_when i ns
  (cf
    (cf(name, D), cf(cf(name, D), cf(cf(term, cf(term, D)), cf(term, D))))))
```

where `term_cases` is a primitive recursive HOL function defined by

```
|- (!D c.
  term_cases D(Const c) =
  (\f g :: Dom(cf(name, D)).
    \h :: Dom(cf(term, cf(term, D))). f c)) /\
  (!D v.
  term_cases D(Var v) =
  (\f g :: Dom(cf(name, D)).
    \h :: Dom(cf(term, cf(term, D))). g v)) /\
  (!D t1 t2.
  term_cases D(Comb t1 t2) =
  (\f g :: Dom(cf(name, D)).
    \h :: Dom(cf(term, cf(term, D))). h t1 t2))
```

The point here is that the domain and range of `term_when` are not discrete universal cpos due to the use of the `cpo` variable. Hence, it is advantageous to define the function such that continuity can still be proved automatically. After a case split on terms the continuity of `term_cases` is proved automatically

```
|- !D.
  cpo D ==>
  (term_cases D) i ns
  (cf
    (term,
      cf(cf(name, D), cf(cf(name, D), cf(cf(term, cf(term, D)), D))))))
```

but it takes the arguments in the wrong order. The definition of `term_when` solves this problem.

8.4.3 Defining the Algorithm

Finally, we are ready to define the unification algorithm which is a recursive function that takes two terms as arguments and returns a (lifted) attempt as a result. It is defined as the fixed point of a certain functional on a pointed cpo and it works by cases on both term arguments using the when eliminator functional for terms.

It is convenient to have the following assignment function to unify variables with other terms (and vice versa).

```
|- !v t. v assign t = ((Var v) << t => Failure | Success[v, t])
|- $assign ins (cf(name, cf(term, attempt)))
```

Since a variable cannot be unified with a term if it occurs in the term assignment returns a failure in this case. Otherwise, it returns the replacement consisting of the variable and the term which we saw above is a best unifier. Assignment only involves discrete universal cpos so it is continuous trivially.

It is also convenient to define the constant case separately since it can be defined by primitive recursion on terms as follows

```
|- (!c c'. unifyC c(Const c') = ((c = c') => Success[] | Failure)) /\
  (!c v. unifyC c(Var v) = v assign (Const c)) /\
  (!c t u. unifyC c(Comb t u) = Failure)
|- unifyC ins (cf(name, cf(term, attempt)))
```

Note, however, that it is not recursive. The constant `unifyC` is continuous trivially.

Using the program `new_constant_definition` the functional for defining the unification algorithm can now be introduced as follows

```
|- unify_FUN =
  (\g :: Dom(cf(term, cf(term, lift attempt))).
    term_when
      (\c :: Dom name. \t :: Dom term. Lift(unifyC c t))
      (\v :: Dom name. \t :: Dom term. Lift(v assign t))
      (\t1 t2 :: Dom term.
        term_when
          (\c :: Dom name. Lift Failure)
          (\v :: Dom name. Lift(v assign (Comb t1 t2)))
          (\u1 u2 :: Dom term.
            athen
              (g t1 u1)
              (\s1 :: Dom rplist.
                athen
                  (g(t2 subst s1)(u2 subst s1))
                  (\s2 :: Dom rplist. Lift(Success(s1 thens s2))))))))))
|- unify_FUN ins
  (cf
    (cf(term, cf(term, lift attempt)), cf(term, cf(term, lift attempt))))
```

It is a continuous function on a pointed cpo because if `D2` is a pointed cpo then so is "`cf(D1, D2)`" for any cpo `D1`. Therefore we can take the fixed point of the functional. This fixed point defines the recursive unification algorithm, and it is called `unify`.

```

|- unify = Fix unify_FUN
|- unify ins (cf(term, cf(term, lift attempt)))

```

Again the program `new_constant_definition` was used. From the fixed point property of `Fix` we derive the following specification of `unify` immediately.

```

|- unify =
  term_when
  (\c :: Dom name. \t :: Dom term. Lift(unifyC c t))
  (\v :: Dom name. \t :: Dom term. Lift(v assign t))
  (\t1 t2 :: Dom term.
    term_when
    (\c :: Dom name. Lift Failure)
    (\v :: Dom name. Lift(v assign (Comb t1 t2)))
    (\u1 u2 :: Dom term.
      athen
      (unify t1 u1)
      (\s1 :: Dom rplist.
        athen
        (unify(t2 subst s1)(u2 subst s1))
        (\s2 :: Dom rplist. Lift(Success(s1 thens s2)))))))

```

Note the recursion here, `unify` appears inside the right-hand side term in two places when unifying two combinations. The algorithm is not primitive recursive since the second time `unify` is applied recursively the arguments are not subterms of the original two combinations and may even be larger terms. Note how the recursion match the theorems about unifiers (or non-existence of unifiers) listed in section 8.3.2 above.

In LCF, the unification algorithm is introduced as a collection of axioms which correspond to the clauses presented next. However, since the LCF types of names and terms contain bottom definedness assertions occur everywhere and complicate the statements unnecessarily. In HOL-CPO, the theorems were derived immediately from the specification theorem above.

```

|- (!c c'.
  unify(Const c)(Const c') =
    ((c = c') => Lift(Success[]) | Lift Failure)) /\
  (!c v. unify(Const c)(Var v) = Lift(v assign (Const c))) /\
  (!c t u. unify(Const c)(Comb t u) = Lift Failure)

|- !v t. unify(Var v)t = Lift(v assign t)

|- (!t1 t2 c. unify(Comb t1 t2)(Const c) = Lift Failure) /\
  (!t1 t2 v.
    unify(Comb t1 t2)(Var v) = Lift(v assign (Comb t1 t2))) /\
  (!t1 t2 u1 u2.
    unify(Comb t1 t2)(Comb u1 u2) =
      athen
      (unify t1 u1)

```

```
(\s1 :: Dom rplist.
  athen
  (uni fy(t2 subst s1)(u2 subst s1))
  (\s2 :: Dom rplist. Li ft(Success(s1 thens s2))))))
```

The proofs are quite slow since large terms are type checked again and again.

The case corresponding to unifying two combinations can be simplified further using the clauses for `athen`. The algorithm may fail to unify the operators

```
|- !t1 u1.
  (uni fy t1 u1 = Li ft Fai lure) ==>
  (!t2 u2. uni fy(Comb t1 t2)(Comb u1 u2) = Li ft Fai lure)
```

or it may be successful and fail to unify the instances of the operands obtained by applying the successful substitution.

```
|- !t1 u1 s.
  (uni fy t1 u1 = Li ft(Success s)) ==>
  (!t2 u2.
    (uni fy(t2 subst s)(u2 subst s) = Li ft Fai lure) ==>
    (uni fy(Comb t1 t2)(Comb u1 u2) = Li ft Fai lure))
```

In case it unifies both it also unifies the combination and the successful substitution is the composition of the two unifiers.

```
|- !t1 u1 s1.
  (uni fy t1 u1 = Li ft(Success s1)) ==>
  (!t2 u2 s2.
    (uni fy(t2 subst s1)(u2 subst s1) = Li ft(Success s2)) ==>
    (uni fy(Comb t1 t2)(Comb u1 u2) = Li ft(Success(s1 thens s2))))
```

Note that these theorems match the non-existence and best unifier theorems of section 8.3.2 very well.

Hence, it is obvious now that provided the unification algorithm terminates it is correct, i.e. it returns a failure if the two term arguments are non-unifiable and returns a best unifier if a unifier exists. Termination and correctness are proved as one statement:

```
|- !t u. ?a. (uni fy t u = Li ft a) /\ best_uni fy_try(a, t, u)
```

In order to prove that the algorithm always terminates, an inductive argument is needed which is most conveniently conducted by well-founded induction. The algorithm terminates because for every recursive call it either reduces the set of variables contained in its arguments or the sets of variables are the same and its first term argument is reduced to a proper subterm. The proof of correctness and termination is discussed in section 8.6.

8.5 A HOL Unification Function

Though the unification algorithm is a total function as stated by the correctness theorem above, it is not straightforward to define it in ‘pure’ HOL since it is not primitive recursive. However, going via domain theory and well-founded induction to prove termination it is possible to introduce a pure HOL unification function. We can simply define this function using the choice operator as follows:

$\vdash \! \text{!t u. Uni fy t u} = (@a. \text{uni fy t u} = \text{Li ft a})$

Furthermore, we can prove this function yields a best unifier for terms of type `" : term"`:

$\vdash \! \text{!t u. best_uni fy_try(Uni fy t u, t, u)}$

From its definition, the recursion equations stating how it behaves on various kinds of arguments can be derived.

This approach to derive a pure HOL unification function via domain theory and well-founded induction may be seen as a recursive definition by well-founded induction.

8.6 Proof of Correctness

The proof of correctness is both a proof of the fact that the unification algorithm always terminates and of the fact that it always returns the right result. This correctness statement has been formulated as follows

$\vdash \! \text{!t u. ?a. (uni fy t u} = \text{Li ft a}) \wedge \text{best_uni fy_try(a, t, u)}$

Below we first describe the well-founded ordering we use for the proof of termination and then describe the actual proof of correctness.

8.6.1 The Well-founded Ordering

A theory of well-founded sets in HOL was described in [Ag91, Ag92] and is printed in appendix A of this thesis. The notion of well-founded set is introduced as a HOL predicate `wfs`. A well-founded set is a set and an ordering such that the ordering is well-founded on all elements of the set. Well-foundedness can be defined in various ways but essentially means that there can be no infinite decreasing chain of elements in the set with respect to the ordering. The theory provides a theorem stating that any well-founded set allows proof by mathematical induction:

$\vdash \! \text{!C R.}$
 $\text{wfs}(C, R) =$
 $(\! \text{!f.}$
 $(\! \text{x. x IN C} \implies \text{f x}) =$
 $(\! \text{x. x IN C} \wedge (\! \text{y. R y x} \wedge \text{y IN C} \implies \text{f y}) \implies \text{f x}))$

Once we have introduced a suitable well-founded ordering we shall use this theorem to prove the unification algorithm always terminates.

Fortunately, we do not have to define an ordering and prove it is well-founded from scratch. The theory of well-founded sets provides a number of constructions which can be used to derive well-foundedness from simpler well-founded orderings very easily.

The well-founded ordering for the proof of the unification algorithm is called `un_ord` and it is defined as follows (the actual definitions of `inv_rel` and `lex_rel` can be found in appendix A):

$\vdash \text{un_ord} =$
 inv_rel
 $(\text{lex_rel} (\$ \text{PSUBSET}, \$ \text{<<}), (\! \text{(t, u). ((vars t) UNION (vars u), t)))$

Let us immediately show how it works:

```
|- !t t' u u'.
    un_ord(t, t')(u, u') =
      ((vars t) UNION (vars t')) PSUBSET ((vars u) UNION (vars u')) \/\
      ((vars t) UNION (vars t') = (vars u) UNION (vars u')) /\ t << u
```

So, two pairs of terms are related if the variables of the first pair are a proper subset of the variables of the second pair, or if the sets of variables are the same and the first component of the first pair is a proper subterm of the first component of the second pair.

The `un_ord` ordering is well-founded on pairs of terms:

```
|- wfs(prod_set(UNIV, UNIV), un_ord)
```

where the product set is defined as expected:

```
|- !B C. prod_set(B, C) = {(b, c) | b IN B /\ c IN C}
```

The proof can be reduced to proving

```
|- wfs(FINITE, $PSUBSET)
|- wfs(UNIV, $<<)
```

using the constructions for inverse images and lexicographic ordering relations, stated by

```
|- !B C P R. wfs(B, P) /\ wfs(C, R) ==> wfs(prod_set(B, C), lex_rel (P, R))
|- !B C G R.
    wfs(C, R) /\ (!b. b IN B ==> (G b) IN C) ==> wfs(B, inv_rel (R, G))
```

One can prove that the proper subset relation on finite sets is well-founded by reducing this to the well-foundedness of the less-than ordering on the natural numbers

```
|- wfs(UNIV, $<)
```

using the mapping construction, stated by

```
|- !B C G P R.
    wfs(C, R) /\
    (!b. b IN B ==> (G b) IN C) /\
    (!b1 b2. b1 IN B /\ b2 IN B /\ P b1 b2 ==> R(G b1)(G b2)) ==>
    wfs(B, P)
```

and the monotonic cardinality function on finite sets (provided by the `pred_sets` library [Me92]).

```
|- !s. FINITE s ==> (!t. t PSUBSET s ==> (CARD t) < (CARD s))
```

The mapping construction is also used to prove the occurrence relation is well-founded on the set of all terms, using a monotonic size function on terms:

```
|- (!c. size(Const c) = 1) /\
    (!v. size(Var v) = 1) /\
    (!t u. size(Comb t u) = 1 + ((size t) + (size u)))
|- !t u. t << u ==> (size t) < (size u)
```

8.6.2 The Induction Proof

Below we give an overview of the well-founded induction. We wish to prove the following statement

$$"!t\ u.\ ?a.\ (\text{uni fy } t\ u = \text{Li ft } a) \wedge \text{best_uni fy_try}(a, t, u)"$$

By well-founded induction this reduces to

$$\begin{aligned} &"?a.\ (\text{uni fy } t\ u = \text{Li ft } a) \wedge \text{best_uni fy_try}(a, t, u)" \\ &\quad ["!t'\ u' . \\ &\quad \quad \text{un_ord}(t', u')(t, u) ==> \\ &\quad \quad (?a.\ (\text{uni fy } t'\ u' = \text{Li ft } a) \wedge \text{best_uni fy_try}(a, t', u'))"] \end{aligned}$$

The proof is now split into various cases. The induction hypothesis is not considered until the case where two combinations are unified, called the comb-comb case below. First we do a case split on whether t and u are equal. If they are equal the correctness of `uni fy` follows from the lemma

$$|- !t.\ ?s.\ (\text{uni fy } t\ t = \text{Li ft}(\text{Success } s)) \wedge s == []$$

which states that `uni fy` returns a substitution which is equivalent to the empty substitution. This is proved easily by induction on terms and the properties of `uni fy`. The algorithm does not generate the empty substitution in the variable case of the induction. Instead it generates a trivial replacement which we saw is equivalent to the empty substitution by equality of substitution.

Assuming the variables are distinct terms we do a cases split on t first. Another case split is done on u in case t is a constant and if t is a variable we do a case split on whether it occurs in u or not. In case t is a combination we also do a case split on u . All cases except the comb-comb case can be proved immediately using the theorems of section 8.3.2 and section 8.4.3 where the properties that a correct algorithm should satisfy and the properties of `uni fy` are listed.

The theorems of those two sections are also required for the comb-comb case but this in addition requires an appeal to the induction hypothesis. Assume t is "`Comb t1 t2`" and u is "`Comb t1' t2'`". Then the following lemma

$$|- \text{un_ord}(t1, t1')(\text{Comb } t1\ t2, \text{Comb } t1'\ t2')$$

allows us to use the induction hypothesis on the first recursive call of `uni fy`. So `uni fy` terminates in this case and returns a correct result. If this call returns a failure then the combinations cannot be unified and `uni fy` correctly fails to unify the combination. Otherwise, we appeal to the induction hypothesis again using the lemma

$$\begin{aligned} &|- \text{best_uni fier}(s, t1, t1') ==> \\ &\quad \text{un_ord}(t2\ \text{subst } s, t2'\ \text{subst } s)(\text{Comb } t1\ t2, \text{Comb } t1'\ t2') \end{aligned}$$

Hence, the second call also terminates with the correct result. If the result is a failure then `uni fy` correctly fails to unify the combination and if it is a success then `uni fy` correctly returns composition of the two unifiers it has constructed in the recursive calls. This concludes the proof.

8.7 Discussion

In this chapter we have described a mechanization of the correctness proof of a unification algorithm based on the proof by Paulson in LCF [Pa85], which in turn is based on the proof by Manna and Waldinger [MW81]. The theories of expressions, substitutions and most-general, idempotent unifiers were developed in pure HOL and then ‘extended’ to domain theory automatically, using discrete universal cpos. Expressions, also called terms, were introduced by a new abstract datatype and substitutions were represented using the built-in type of finite lists. Most functions and concepts were defined by primitive recursion, or simply as HOL definitions (abbreviations). The formalization of domain theory was employed for defining the recursive unification algorithm since in pure HOL there is no support for defining the unusual recursion of the algorithm. It is not obvious that the algorithm is total, this was proved using well-founded induction. The ordering for the induction proof was defined using constructions on well-founded orderings and proved to be well-founded almost automatically.

The two implementations of MW’s proof in LCF and HOL-CPO respectively are similar and yet radically different. Both proofs closely resembles MW’s proof though both use different representations of terms and substitutions than in their proof. However, the presence of bottom in all types in LCF had such a big influence on statements and proofs that it is tempting to conclude that the logic is inappropriate for the proof. Having the LCF proof available, it was clear what a big difference the absence of bottom makes in our proof where most of the development was done in the set theoretic HOL world (cf. section 8.1). This was possible since all datatypes and functions are strict. Further, the HOL-CPO proof shows that it is convenient to use well-founded induction as in MW’s proof rather than the nested structural inductions on natural numbers and terms as in the LCF proof.

Proofs in HOL and LCF look different. In LCF most proofs use conditional rewriting extensively, with long lists of theorems. Such proofs are split up into more tactics in HOL, stripping goals apart and using resolution and (simple) rewriting. The result is much more readable. The outcome of a conditional rewrite often depends on the order in which the theorems are applied and often strange hacks appear to avoid loops and unexpected behavior.

Chapter 9

Conclusion

A HOL formalization of central concepts of domain theory has been presented with syntactic-based proof functions and other tools to support reasoning about functional programs. The formalization may be seen as a semantic embedding of the LCF system in HOL which is conducted in such a way that it inherits and extends the advantages of both systems. The extension of HOL is called HOL-CPO.

Our philosophy was that there should be a direct correspondence between elements of complete partial orders and elements of HOL types, in order to allow the reuse of higher order logic and proof infrastructure already available with the HOL system. Hence, we were able to mix set and domain theoretic reasoning to advantage and exploit HOL types and tools directly to reason about denotations of functional programs (as continuous functions).

Examples illustrated the use of HOL-CPO and demonstrated many ways in which HOL-CPO extends both the LCF and the HOL system (though, LCF supports the axiomatization of more recursive domains than HOL-CPO).

The main results of the work are:

- HOL-CPO: a conservative extension of HOL to support reasoning about the denotations of functional programs in domain theory.
- A comparison of LCF and HOL-CPO.
- A method for introducing derived definitions of recursive well-founded functions in HOL via domain theory and well-founded induction.
- A larger example which shows the proof of correctness of a unification algorithm.
- Methods and ideas for defining some recursive domains with finite and infinite values.

As part of HOL-CPO, informal extendable notations for cpos, typable terms and inclusive predicates were implemented by an interface and a number of syntactic-based proof functions. A term is called (cpo-) typable when it is an element of some cpo. The purpose of the notations was in part to automate proofs of semantic properties. The work load of the user of HOL-CPO was reduced considerably by the notations. Proof obligations induced by the formalization could be proved automatically in most examples.

9.1 Extension of HOL

HOL-CPO extends the possibilities of the HOL user in various ways. In HOL, there is no notion of non-termination and all functions are total functions of set theory. Only primitive recursive functions on concrete datatypes of syntax can be defined directly using tools available with the HOL system. In order to define a more general recursive function, one must first show its existence in higher order logic, or perform some trick.

Firstly, HOL-CPO supports the concepts and techniques of fixed point theory to reason about non-termination, partial functions and arbitrary recursive (computable) functions. Partial functions and non-termination are handled via the bottom element of pointed cpos; such cpos are built, for instance, using the lifting construction on cpos. The fixed point operator allows the definition of recursive continuous functions on pointed cpos. Fixed point induction, which is derived as a theorem, can be used to prove that inclusive properties hold of recursive definitions. Other techniques for recursion derivable in HOL, e.g. well-founded induction, can also be used and it is possible to reason about fixed points directly from their definition in HOL.

Secondly, HOL-CPO supports reasoning about infinite values of recursive domains like lazy sequences and lazy lists, providing techniques like fixed point induction, structural induction and co-induction based on the notion of bisimulation. Structural induction is used to prove an inclusive property holds of finite elements—the inclusiveness guarantees that the property holds of infinite elements too.

Thirdly, HOL-CPO supports a method for introducing total recursive functions in HOL by well-founded induction: A recursive function is defined using the fixed point operator in domain theory. Using well-founded induction the function is then proved to terminate for all input. Hence, it allows a recursive HOL function to be defined.

9.2 Embedding Semantics vs. Implementing Logic

HOL-CPO may be seen as a semantic embedding of the LCF system in HOL. Domain theoretic concepts like complete partial orders, continuous functions and inclusive predicates were formalized as semantic constants in HOL. LCF, on the other hand, is an implementation of a logic with a domain theoretic semantics. Hence, LCF only provides access to domain theory via a few axioms and primitive inference rules.

HOL-CPO extends the LCF system in the following ways:

- It provides direct access to domain theory.
- It inherits the underlying higher order logic and proof infrastructure of the HOL system.

These are the consequences of embedding semantics in a powerful theorem prover rather than implementing logic.

The last point may be supported to various degrees in a semantic embedding. Our goal has been to develop a formalization that provides a direct correspondence between elements of domains and elements of HOL types in order to allow the many built-in types and tools of the HOL system to be exploited. A formalization of ‘real’ domain theory based on information systems or universal domains like $P\omega$ would not support this directly, due to the encoding performed in these theories.

So, although the last point is a consequence of the present embedding rather than semantic embeddings in general, it is a central advantage of HOL-CPO over LCF. Experience with LCF shows that the continual fiddling with bottom is very annoying. Its presence in all types makes LCF unsuitable for reasoning about finite-valued types and strict functions. In HOL-CPO, we are able to mix domain and set theoretic reasoning. This means that reasoning about bottom often can be eliminated or deferred until the late stages of a proof. We can also exploit the rich collection of built-in theorems, tools and libraries provided with the HOL system in applications. LCF has almost nothing like that, though this is more a consequence of history than of differences in approaches.

A disadvantage of implementing logic as in LCF is that one is restricted to the reasoning provided by the axioms and primitive inference rules of the logic. Since domain theoretic concepts are not present by their semantic definitions in LCF, it is not possible to reason directly about fixed points, for instance, though this allows more theorems to be proved than with just fixed point induction. Further, testing that a predicate admits fixed point induction, i.e. testing that it is inclusive, can only be performed in ML by an incomplete syntactic check.

These problems were solved in HOL-CPO where the semantic definitions are available. For instance, fixed point induction was derived as a theorem from the semantic definition of the fixed point operator. Further, syntactic checks for inclusiveness were implemented as a syntactic-based proof function. Inclusive predicates not accepted by the syntactic checks can be proved to be inclusive from the semantic definition of inclusiveness.

On the other hand, a disadvantage of embedding semantics and having direct access to domain theory is that this introduces new proof obligations. In order to use theorems of domain theory, one must prove all the time that terms are cpos, continuous functions and inclusive predicates. The proof functions for the notations for cpos, typable terms and inclusive predicates could prove these obligations in most cases.

In LCF, types denote (pointed) cpos and the function type denotes the cpo of continuous functions. Hence, the proof obligations are not present, except for inclusiveness. (Not all predicates of first order logic are inclusive so this condition on fixed point induction cannot be avoided.)

As a further disadvantage of the formalization presented here, it is necessary to prove that terms are elements of the right cpos before functions are applied to terms. This was called type checking. The problem arose due to the determinedness condition on continuous functions which introduced the need for the dependent lambda abstraction. This was used to write determined functions that became parameterized by free cpo variables of the domains on which they worked. An interface could hide this annoying extra information in most cases.

One may compare the problems in LCF due to bottom to the problems in HOL-CPO due to the parameters on the dependent lambda abstraction and some function constructions. An interface could also be implemented in LCF to hide bottom in many cases. But it would always appear in proofs whereas we often avoid type checking in HOL-CPO. For instance, in the unification example where the bottom element was a major nuisance in LCF we worked most of the time in the set theoretic HOL world where the problem of dependent functions (or bottom) does not exist. Domain theory was only used to define the recursive unification algorithm at a late stage of the proof. Further, at that stage it was only present temporarily since we worked with universal cpos which consisted of all elements of the underlying HOL types.

Finally, let us address the complex issue of constructing solutions to recursive domain equations. HOL-CPO does not support this well. A recursion operator for recursive domains similar to the fixed point operator for recursive functions has not been developed. We have seen some examples on how certain recursive domains with finite values can be derived using recursive types in HOL, introduced by the type definition package. This approach does not work for recursive domains with infinite values since all types introduced by the package are well-founded. Theories of lazy sequences and lazy lists were provided with induction principles like structural induction and co-induction. Some ideas were presented for introducing other more general recursive domains with infinite values.

However, LCF does not provide a recursion operator for recursive domains either and since it does not provide direct access to domain theory, it is impossible to derive solutions to recursive domain equations. Instead, LCF just axiomatizes recursive domains. As pointed out by Paulson [Pa84a, Pa87], an axiomatization is difficult in even quite simple cases; it can be both arduous and tedious to develop a useful theory for a new recursive type. This motivated a number of tools to axiomatize certain recursive types with finite and infinite elements (denoting recursive domains) which can be described by a set of constructor functions, each taking a number of arguments of specified types where the function type is not used in an essential way.

Axiomatizing even simple theories is against the traditions in HOL and it would be difficult to provide a derived tool which was similar in strength to the LCF tools.

9.3 Alternative Formalizations

In section 3.11, a number of alternative formalizations were discussed. An approach where the set component of cpos is represented directly as a HOL type does not work since the continuous function space construction on cpos cannot be defined. This is a dependent subtype of the type of HOL functions (depending on free term variables ranging over cpos). Furthermore, the disadvantageous determinedness condition on continuous functions is necessary to prove that the continuous function space satisfies the antisymmetry condition on cpos.

For further discussion the reader should consult section 3.11.

9.4 Limited Treatment of Recursive Domains

Our philosophy has been that formalizing domain theory in HOL should support the reuse of HOL types and proof infrastructure to as large an extent as possible. Hence, a formalization should provide a direct correspondence between elements of HOL types and elements of cpos. We have seen some concrete advantages of this. Any recursive type in HOL like natural numbers, finite lists and terms (for unification) can be reasoned about in the set theoretic HOL world before turning to domain theory where reasoning may involve bottom and complex notions as complete partial orders and continuous functions. We can mix set and domain theoretic reasoning to advantage.

A direct consequence of this philosophy is that solutions of recursive domain equations are not easily defined in HOL-CPO. Some recursive domains have been considered. Certain recursive domains with finite values can be defined by exploiting recursive types

in HOL, introduced by the type definition package. However, this approach has not been automated and requires tedious proofs about basic domain theoretic concepts.

Further, when recursive domains contain infinite values we have no implemented methods or examples which work in more general cases. We have implemented constructions for pointed cpos of lazy sequences and lazy lists but the method which was used for this purpose is not easily generalized. Quite detailed ideas were presented on a method for introducing more general recursive domains with infinite values, based on finitely-branching labeled trees with finite, partial and infinite elements. This method borrows ideas from the type definition package and are imposed by the same limitations. Hence, constructor functions are only allowed to take arguments of simple types or the recursive type itself; e.g. pairs, lists and functions are not allowed.

The advantage of methods based on trees is that they leave only a few proof obligations to be proved when a new recursive domain is introduced. Roughly speaking, the only thing to prove is that the new domain contains least upper bounds of chains; we even know what the least upper bounds look like since they have been proved to be in the cpos of labeled trees. Further, defining constructor functions which fit within the notation of typable terms, we can prove they are continuous automatically.

So, we have only considered how to define recursive domains denoted by fairly simple recursive type specifications. Attempts to solve real recursive domain equations which involve the function space in an essential way, has not been considered at all; Gunter's approach deals with recursive type specifications where the function type is permitted to a limited extent, only at the right-hand sides of outermost function spaces.

Paulson has done some interesting work on constructing recursive types (not domains) by taking least and greatest fixed points of monotone operators [Pa93]. He provides co-induction to reason about infinite elements. However, this does not directly apply to a domain theoretic setting.

One could put deep thought into the question of how to allow the solution of recursive domain equations. The inverse limit construction is a well-known method for constructing solutions to recursive domain equations as ω -colimits of continuous functors on the category of cpos with embedding projection pairs. A formalization in HOL would be rather complex and probably not fit in well with our philosophy since it seems to need a "big" set closed under ω -sequences to capture infinite elements. This would require some kind of encoding.

9.5 Related Work

Other formalizations of domain theory exist. Petersen [Pe93] has formalized the $P\omega$ model such that all recursive domain equations can be solved (this has not been fully implemented). However, domains live in $P\omega$ only and it is not clear how to lift HOL types and functions to $P\omega$ and back. Therefore very few of HOL's facilities can be exploited directly.

Camilleri mechanized a theory of cpos and fixed points which he used to define recursive operators in CSP trace theory [Ca90]. However, he did not consider constructions on cpos and continuous functions but proved continuity in an ad hoc way in the HOL system. A major problem with his approach is that it does not allow the continuous function space construction which is fundamental to our work. The problem is that the set

of continuous functions between two cpos is a dependent subset of all HOL functions, as we mentioned above. We have solved this problem using dependent subtypes, also called term parameterized types. Though these are not provided by the HOL logic itself they can be simulated by predicates denoting subsets of types. The same approach is used in [JM93].

Franz Regensburger¹ is working on a very similar project in Isabelle HOL but the formalizations seem to be quite different. Pointed cpos are introduced using type classes and continuous functions constitute a type. Type checking arguments of functions seems not to be necessary but before β -reduction can be performed functions must be shown to be continuous (unlike in our formalization). Recursive domains can be axiomatized in a similar way as in LCF, though this has not been automated as in LCF. He is currently writing a Ph.D. thesis about the work [Re94] (in German unfortunately).

Bernhard Reus² works on synthetic domain theory [RS94] in the LEGO system which implements a strong type theory (ECC) with dependent sums and products. Dependent families can be exploited for the inverse limit construction. This is work in progress for a Ph.D. and nothing has been published on the formalization yet.

9.6 Evaluation of HOL-CPO

Developing the formalization of domain theory was extremely tedious. Introducing the continuous function space as a ‘dependent subtype’ of the HOL function type was the main reason for this, since working with subsets of types introduced a lot of extra work in proofs. All the time one must prove that terms are in the right subsets. For example, before one can deduce that the result of applying a continuous function to an argument is in a certain subset, one must prove that the argument is in the right subset. To take the lub of chains, each elements of the chains must be in some subset (cpo) before we can deduce that the lub is an element of this subset too. This type checking, which must be done manually here in an ad hoc way, was also one of the reasons why it was so tough to develop the cpos of lazy sequences and lazy lists. It will make any domain theoretic extension of the formalization arduous.

Fortunately, it seemed to be less tedious to use the formalization because one does not need to reason about the deeper properties of domain theory. The notations for cpos and typable terms helped a lot. Further, most of the time we worked with universal cpos which contained all elements of underlying HOL type; the subsets were the whole types. Therefore, type checking could be avoided in many cases.

The prototype proof functions and tools supports reasoning about denotations of functional programs pretty well. The user does not have to worry about proving deep properties in domain theory. This is handled in most cases by the proof functions. It is only necessary to turn to the formalization of domain theory itself when one wish to introduce new recursive domains. Though, some recursive domains can be introduced via the discrete construction and some recursive HOL type (e.g. see section 2.3.3).

In a wider perspective, the limited possibilities of constructing solutions of recursive domain equations means that we cannot give semantics to realistic programming languages,

¹Technical University, Munich. Email: regensbu@informatik.tu-muenchen.de

²Ludwig-Maximilian University, Munich. Email: reus@informatik.uni-muenchen.de

for instance, languages with recursive types. We can still reason about denotations of some recursive types and functions directly in domain theory.

9.7 Future Work: ‘Real’ Domain Theory

Before we develop the prototype proof functions and tools further, we must consider how to support solutions to recursive domain equations in a much better way than now. One should make ones goals clear and try to investigate whether solving recursive domain equations based on trees would be enough, or whether the additional power obtained by turning to representations of domains, for instance as information systems, would be required.

If we chose to formalize information systems, at least the following limitations of the current approach could be eliminated: We would be able to

- solve recursive domain equations fairly easily,
- give semantics to realistic programming languages and
- prove adequacy results relating operational and denotational semantics,

using the concrete nature of information systems to advantage.

We could hope to solve all problems of the present formalization without introducing new ones. Earlier we said that a formalization of information systems would introduce a kind of new world different from the HOL world; for instance, the information system of natural numbers would not share any elements with the HOL type of natural numbers directly. Perhaps isomorphisms or suitable mappings between information systems and HOL types, hidden by an interface, would allow us to reuse HOL types and tools as now, and mix set and domain theoretic reasoning.

There would be benefits and drawbacks of a formalization of information systems. Only deeper thought and trying out the ideas in practice would be able to answer whether the benefits or the drawbacks would win.

Instead of insisting on working only in HOL, one could also try to employ more powerful theories. One approach would be to move to a system supporting a stronger dependent type theory. But such theories are usually not easy to learn and there are quite a lot of different new systems to choose among. Set theory might provide a simple alternative to type theories.

As an alternative to the many new theorem provers based on type theories, Mike Gordon has implemented an experimental extension of the HOL system with a ZF-like set theory [Go94]. In this system, called HOL-ST, I have recently formalized the inverse limit construction (which is not possible directly in pure HOL without a complex encoding) and used this to derive a non-trivial solution D_∞ of the equation

$$D_\infty \cong [D_\infty \rightarrow D_\infty],$$

which provides a model of the untyped λ -calculus [Ag94b]. Any recursive domain equation could be solved in a similar way in HOL-ST, but it is still not clear whether this approach is really useful or not.

Appendix A

Well-founded Sets

This is the theory of well-founded sets presented in [Ag91, Ag92].

```
#print_theory' wfs' ;;
The Theory wfs
Parents -- HOL      pred_sets
Constants --
  decr_chain ": (num -> *) -> ((* -> bool) # (* -> (* -> bool))) -> bool"
  min_e     ": * -> ((* -> bool) # (* -> (* -> bool))) -> bool"
  strict_ordering ": (* -> (* -> bool)) # (* -> bool) -> bool"
  wfs      ": (* -> bool) # (* -> (* -> bool)) -> bool"
  gen_seq  ": (* -> bool) -> ((* -> (* -> bool)) -> (num -> *))"
  prod_set ": (* -> bool) # (** -> bool) -> (* # ** -> bool)"
  prod_rel
    ": (* -> (* -> bool)) # (** -> (** -> bool)) ->
      (* # ** -> (* # ** -> bool))"
  lex_rel
    ": (* -> (* -> bool)) # (** -> (** -> bool)) ->
      (* # ** -> (* # ** -> bool))"
  inv_rel  ": (** -> (** -> bool)) # (* -> **) -> (* -> (* -> bool))"
  pnum    ": num -> bool"
Infixes --
  decr_chain ": (num -> *) -> ((* -> bool) # (* -> (* -> bool))) -> bool"
  min_e     ": * -> ((* -> bool) # (* -> (* -> bool))) -> bool"
Definitions --
  decr_chain_DEF
    |- !X A R.
      X decr_chain (A, R) = (!n. (X n) IN A) /\ (!n. R(X(n + 1))(X n))
  strict_ordering_DEF
    |- !R A. strict_ordering(R, A) = (!x. x IN A ==> ~R x x)
  min_e_DEF   |- !x A R. x min_e (A, R) = x IN A /\ ~(?y. R y x /\ y IN A)
  wfs_DEF
    |- !C R.
      wfs(C, R) = (!A. ~(A = {})) /\ A SUBSET C ==> (?x. x min_e (A, R))
  gen_seq_DEF
    |- (!A R. gen_seq A R 0 = (@x. x IN A)) /\
```

$(!A R n. \text{gen_seq } A R(\text{SUC } n) = (@x. x \text{ IN } A \wedge R x(\text{gen_seq } A R n)))$
 $\text{prod_set_DEF} \quad |- !B C. \text{prod_set}(B, C) = \{(b, c) \mid b \text{ IN } B \wedge c \text{ IN } C\}$
 prod_rel_DEF
 $|- !P R x y. \text{prod_rel}(P, R)x y = P(\text{FST } x)(\text{FST } y) \wedge R(\text{SND } x)(\text{SND } y)$
 lex_rel_DEF
 $|- !P R b c.$
 $\quad \text{lex_rel}(P, R)b c =$
 $\quad P(\text{FST } b)(\text{FST } c) \vee (\text{FST } b = \text{FST } c) \wedge R(\text{SND } b)(\text{SND } c)$
 $\text{inv_rel_DEF} \quad |- !R f b b'. \text{inv_rel}(R, f)b b' = R(f b)(f b')$
 $\text{pnum_DEF} \quad |- \text{pnum} = \{x \mid 0 < x\}$

Theorems --

WFS_DEFS_EQUIV

$|- !C R.$

$\text{wfs}(C, R) =$

$(!A. (?x. x \text{ IN } (C \text{ INTER } A)) = (?x. x \text{ min_e } (C \text{ INTER } A, R)))$

DC

$|- !A R.$

$\sim(A = \{\}) \wedge (!a. a \text{ IN } A \implies (?b. b \text{ IN } A \wedge R b a)) \implies$

$(?X. X \text{ decr_chain } (A, R))$

$\text{WFS_FIN_CHAINS} \quad |- !C R. \text{wfs}(C, R) = \sim(?X. X \text{ decr_chain } (C, R))$

$\text{WFS_STRICT_ORD} \quad |- !C R. \text{wfs}(C, R) \implies \text{strict_ordering}(R, C)$

WFS_MATH_INDUCT

$|- !C R.$

$\text{wfs}(C, R) \implies$

$(!f.$

$(!x. x \text{ IN } C \implies f x) =$

$(!x. x \text{ IN } C \wedge (!y. R y x \wedge y \text{ IN } C \implies f y) \implies f x))$

WFS_EQ_INDUCT

$|- !C R.$

$\text{wfs}(C, R) =$

$(!f.$

$(!x. x \text{ IN } C \implies f x) =$

$(!x. x \text{ IN } C \wedge (!y. R y x \wedge y \text{ IN } C \implies f y) \implies f x))$

$\text{WFS_SUBSET} \quad |- !B C R. B \text{ SUBSET } C \wedge \text{wfs}(C, R) \implies \text{wfs}(B, R)$

WFS_PROD

$|- !B C P R.$

$\text{wfs}(B, P) \wedge \text{wfs}(C, R) \implies \text{wfs}(\text{prod_set}(B, C), \text{prod_rel}(P, R))$

WFS_LEX_PROD

$|- !B C P R.$

$\text{wfs}(B, P) \wedge \text{wfs}(C, R) \implies \text{wfs}(\text{prod_set}(B, C), \text{lex_rel}(P, R))$

WFS_INV_IMAGE

$|- !B C G R.$

$\text{wfs}(C, R) \wedge (!b. b \text{ IN } B \implies (G b) \text{ IN } C) \implies \text{wfs}(B, \text{inv_rel}(R, G))$

WFS_MAPPING

$|- !B C G P R.$

$\text{wfs}(C, R) \wedge$

```
(!b. b IN B ==> (G b) IN C) /\
(!b1 b2. b1 IN B /\ b2 IN B /\ P b1 b2 ==> R(G b1)(G b2)) ==>
wfs(B, P)
NUM_IS_WFS |- wfs(UNIV, $<)
PNUM_IS_WFS |- wfs(pnum, $<)
LEX_NUM_IS_WFS |- wfs(prod_set(pnum, UNIV), lex_rel ($<, $<))
```

```
***** wfs *****
```

```
() : void
```


Bibliography

- [Ag91] S. Agerholm, ‘Mechanizing Program Verification in HOL’. In the *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and Its Applications*, Davis California, August 28–30, 1991. IEEE Computer Society Press, 1992.
- [Ag92] S. Agerholm, *Mechanizing Program Verification in HOL*. M.Sc. Thesis, Aarhus University, Computer Science Department, Report IR-111, April 1992.
- [Ag93] S. Agerholm, ‘Domain Theory in HOL’. In the *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, Jeffrey J. Joyce and Carl-Johan H. Seger (eds.), Vancouver, B.C., Canada, August 11–13 1993, LNCS 780, Springer-Verlag 1994.
- [Ag94a] S. Agerholm, ‘LCF Examples in HOL’. In *Proceedings of the 7th International Workshop on Higher Order Logic Theorem Proving and its Applications*, Thomas F. Melham and Juanito Camilleri (Eds.), Malta, September 1994, LNCS 859, Springer-Verlag 1994. A revised version will appear in *Computer Journal*, May 1995 (provisionally).
- [Ag94b] S. Agerholm, ‘Formalising a Model of the λ -calculus in HOL-ST’. Technical Report no. 354, University of Cambridge, Computer Laboratory, November 1994.
- [An92] F. Andersen, *A Theorem Prover for UNITY in Higher Order Logic*. Ph.D. Thesis, Technical University of Denmark, 1992. Also published as TFL RT 1992–3.
- [APP93] F. Andersen, K.D. Petersen and J.S. Pettersson, ‘Program Verification using HOL-UNITY’. In the *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, Jeffrey J. Joyce and Carl-Johan H. Seger (eds.), Vancouver, B.C., Canada, August 11–13 1993, LNCS 780, Springer-Verlag 1994.
- [Ba84] H.P. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.
- [BW88] R. Bird and P. Wadler, *Introduction to Functional Programming*. Prentice-Hall International, 1988.
- [Bo91] R.J. Boulton, ‘The HOL taut Library’. University of Cambridge, Computer Laboratory, July 1991. (Appears in [Un94].)

- [BGH92] R.J. Boulton, A.D. Gordon, J.R. Harrison, J.M.J. Herbert, and J. Van Tassel, ‘Experience with Embedding Hardware Description Languages in HOL’. In V. Stavridou, T.F. Melham, and R.T. Boute (Eds), *Theorem Provers in Circuit Design: Theory, Practice and Experience: Proceedings of the IFIP TC10/WG 10.2 International Conference*, IFIP Transactions A-10. North-Holland, June 1992.
- [Ca90] A.J. Camilleri, ‘Mechanizing CSP Trace Theory in Higher Order Logic’. *IEEE Transactions on Software Engineering*, Vol. 16, No. 9, September 1990.
- [CM92] J. Camilleri and T.F. Melham, ‘Reasoning with Inductively Defined Relations in the HOL Theorem Prover’. University of Cambridge, Computer Laboratory, Technical Report No. 265, August 1992.
- [Co89] A.J. Cohn, ‘The Notion of Proof in Hardware Verification’. *Journal of Automated Reasoning*, Vol. 5, No. 2, 1989.
- [Co92] M.D. Coen, *Interactive Program Derivation*. Ph.D. Thesis, University of Cambridge, Computer Laboratory, Technical Report No. 272, November 1992.
- [DS90] E.W. Dijkstra and C. Scholten, *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.
- [Go92] A.D. Gordon, ‘Re: Ackermann’s Function’. Info-hol mail message, May 14, 1992.
- [Go89a] M.J.C. Gordon, ‘HOL: A Proof Generating System for Higher Order Logic’. In G. Birtwistle and P.A. Subrahmanyam (eds.), *Current Trends in Hardware Verification and Theorem Proving*, Springer-Verlag, 1989.
- [Go89b] M.J.C. Gordon, ‘Mechanizing Programming Logics in Higher Order Logic’. In G. Birtwistle and P.A. Subrahmanyam (eds.), *Current Trends in Hardware Verification and Theorem Proving*, Springer-Verlag, 1989.
- [Go94] M.J.C. Gordon, ‘Merging HOL with Set Theory: preliminary experiments’. Technical Report no. 353, University of Cambridge, Computer Laboratory, November 1994.
- [GM93] M.J.C. Gordon and T.F. Melham (Eds.), *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
- [GMW79] M.J.C. Gordon, R. Milner and C.P. Wadsworth, *Edinburgh LCF: A Mechanised Logic of Computation*. Springer-Verlag, LNCS 78, 1979.
- [Gu92] C.A. Gunter, *Semantics of Programming Languages: Structures and Techniques*. The MIT Press, 1992.
- [GS90] C.A. Gunter and D.S. Scott, ‘Semantic Domains’. In J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science, Vol. B*, North-Holland, Amsterdam, 1990.
- [Gu93] E. Gunter, ‘A Broader Class of Trees for Recursive Type Definitions for HOL’. In the *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, Jeffrey J. Joyce and Carl-Johan H. Seger (eds.), Vancouver, B.C., Canada, August 11–13 1993, LNCS 780, Springer-Verlag 1994.

- [Ha92] J.R. Harrison, ‘The HOL wellorder Library’. University of Cambridge, Computer Laboratory, May 1992. (Appears in [Un94].)
- [JM93] B. Jacobs and T. Melham, ‘Translating Dependent Type Theory into Higher Order Logic’. In the *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, Utrecht, 16–18 March 1993. Springer-Verlag, LNCS 664, 1993.
- [LW91] K.G. Larsen and G. Winskel, ‘Using Information Systems to Solve Recursive Domain Equations’. *Information and Computation*, Vol. 91, No. 2, April 1991.
- [MW81] Z. Manna and R. Waldinger, ‘Deductive Synthesis of the Unification Algorithm’. *Science of Computer Programming*, Vol. 1, 1981, pp. 5–48.
- [MW90] Z. Manna and R. Waldinger, *The Logical Basis for Computer Programming. Volume 2: Deductive Systems*. Addison-Wesley, 1990.
- [Me89] T.F. Melham, ‘Automating Recursive Type Definitions in Higher Order Logic’. In G. Birtwistle and P.A. Subrahmanyam (eds.), *Current Trends in Hardware Verification and Theorem Proving*, Springer-Verlag, 1989.
- [Me91] T.F. Melham, ‘Recursive Data Types’. Info-hol mail message, November 9, 1991.
- [Me92] T.F. Melham, ‘The HOL pred_sets Library’. University of Cambridge, Computer Laboratory, February 1992. (Appears in [Un94].)
- [MTH90] R. Milner, M. Tofte and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
- [Mo90] P.D. Mosses, ‘Denotational Semantics’. In J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science, Vol. B*, North-Holland, Amsterdam, 1990.
- [Pa84a] L.C. Paulson, ‘Structural Induction in LCF’. Springer-Verlag, LNCS 173, 1984. Also in Technical Report No. 44, University of Cambridge, Computer Laboratory, February 1984.
- [Pa84b] L.C. Paulson, ‘Lessons Learned from LCF’. Technical Report No. 54, University of Cambridge, Computer Laboratory, August 1984.
- [Pa85] L.C. Paulson, ‘Verifying the Unification Algorithm in LCF’. *Science of Computer Programming*, Vol. 5, 1985, pp. 143–169. Also in Technical Report No. 50, University of Cambridge, Computer Laboratory, March 1984.
- [Pa87] L.C. Paulson, *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge Tracts in Theoretical Computing 2, Cambridge University Press, 1987.
- [Pa90] L.C. Paulson, ‘Isabelle: The Next 700 Theorem Provers’. In P. Odifreddi (ed.), *Logic and Computer Science*, Academic Press, 1990.
- [Pa91] L.C. Paulson, *ML for the Working Programmer*. Cambridge University Press, 1991.

- [Pa93] L.C. Paulson, ‘Co-induction and Co-recursion in Higher-order Logic’. Technical Report No. 304, University of Cambridge, Computer Laboratory, 1993.
- [Pe93] K.D. Petersen, ‘Graph Model of LAMBDA in Higher Order Logic’. In the *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, Jeffrey J. Joyce and Carl-Johan H. Seger (eds.), Vancouver, B.C., Canada, August 11–13 1993, LNCS 780, Springer-Verlag 1994.
- [Pi94] A.M. Pitts, ‘A Co-induction Principle for Recursively Defined Domains’. *Theoretical Computer Science*, Vol. 124, 1994.
- [Pl83] G. Plotkin, *Domains*. Course notes, Department of Computer Science, University of Edinburgh, 1983.
- [Re94] F. Regensburger, *HOLCF: A Conservative Extension of HOL with LCF*. Ph.D. Thesis (in German), Technical University, Munich, 1994 (to appear).
- [RS94] B. Reus and T. Streicher, ‘Naive Synthetic Domain Theory—A Logical Approach’. Draft, 1994.
- [Ro65] J.A. Robinson, ‘A Machine-oriented Logic Based on the Resolution Principle’. *J. ACM*, Vol. 12, No. 1, 1965.
- [Sc86] D.A. Schmidt, *Denotational Semantics*. Allyn and Bacon, 1986.
- [Sc76] D.S. Scott, ‘Data Types as Lattices’. *SIAM J. Comput.*, Vol. 5, No. 3, September 1976.
- [Sc82] D.S. Scott, ‘Domains for Denotational Semantics’. In *Proc. of ICALP’82*, Springer-Verlag, LNCS 140, 1982.
- [SP82] M. Smyth and G.D. Plotkin, ‘The Category-theoretic Solution of Recursive Domain Equations’. *SIAM Journal of Computing*, Vol. 11, 1982.
- [St77] J.E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.
- [Un94] University of Cambridge Computer Laboratory, *The HOL System: Libraries*. Version 2 (for HOL88.2.02), March 1994. (This documentation is distributed with the HOL system.)
- [Wi93] G. Winskel, *The Formal Semantics of Programming Languages*. The MIT Press, 1993.

Recent Publications in the BRICS Report Series

- RS-94-44 Sten Agerholm. *A HOL Basis for Reasoning about Functional Programs*. December 1994. PhD thesis. 233 pp.
- RS-94-43 Luca Aceto and Alan Jeffrey. *A Complete Axiomatization of Timed Bisimulation for a Class of Timed Regular Behaviours (Revised Version)*. December 1994. 18 pp. To appear in *Theoretical Computer Science*.
- RS-94-42 Dany Breslauer and Leszek Gąsieniec. *Efficient String Matching on Coded Texts*. December 1994. 20 pp.
- RS-94-41 Peter Bro Miltersen, Noam Nisan, Shmuel Safra, and Avi Wigderson. *On Data Structures and Asymmetric Communication Complexity*. December 1994. 17 pp.
- RS-94-40 Luca Aceto and Anna Ingólfssdóttir. *CPO Models for GSOS Languages — Part I: Compact GSOS Languages*. December 1994. 70 pp. An extended abstract of the paper will appear in: *Proceedings of CAAP '95, LNCS, 1995*.
- RS-94-39 Ivan Damgård, Oded Goldreich, and Avi Wigderson. *Hashing Functions can Simplify Zero-Knowledge Protocol Design (too)*. November 1994. 18 pp.
- RS-94-38 Ivan B. Damgård and Lars Ramkilde Knudsen. *Enhancing the Strength of Conventional Cryptosystems*. November 1994. 12 pp.
- RS-94-37 Jaap van Oosten. *Fibrations and Calculi of Fractions*. November 1994. 21 pp.
- RS-94-36 Alexander A. Razborov. *On provably disjoint NP-pairs*. November 1994. 27 pp.
- RS-94-35 Gerth Stølting Brodal. *Partially Persistent Data Structures of Bounded Degree with Constant Update Time*. November 1994. 24 pp.
- RS-94-34 Henrik Reif Andersen, Colin Stirling, and Glynn Winskel. *A Compositional Proof System for the Modal μ -Calculus*. October 1994. 18 pp. Appears in: *Proceedings of LICS '94, IEEE Computer Society Press*.