Treating Partiality in a Logic of Total Functions

Olaf Müller and Konrad Slind

Institut für Informatik, Technische Universität München, 80290 München, Germany
Email: {mueller,slind}@informatik.tu-muenchen.de

The need to use partial functions arises frequently in formal descriptions of computer systems. However, most proof assistants are based on logics of total functions. One way to address this mismatch is to invent and mechanize a new logic. Another is to develop practical workarounds in existing settings. In this paper we take the latter course: we survey and compare methods used to support partiality in a mechanization of a higher order logic featuring only total functions. The techniques we discuss are generally applicable and are illustrated by relatively large examples.

Received December 10, 1996; revised November 17, 1997

1. INTRODUCTION

In any setting where there are operations acting on elements, the following fundamental consideration arises: how should an operation treat an element that lies outside of its domain? This is known as *partiality*. In many cases, matters can be arranged such that the question simply does not arise, but in many others partiality must be addressed. The problem of partiality has received a great deal of attention in the fields of logic and formal methods; in fact, there is a deep mismatch between modelling problems in formal methods and the tools used to solve such problems, since the tools are often based on logics of total functions, while the modelling problems often demand treatment of partiality. One way to repair this mismatch is to bring partiality into the logic: as can be seen in *e.g.* [1, 2], there are a range of possibilities. Unfortunately, few have been mechanized, and none have yet proved to be clearly better than the others. On the other hand, tools based on total logics have prospered and are beginning to be used in realistic industrial applications, see *e.g.* [3]. Hence, in this paper, we will argue for staying in a logic of total functions and using various 'tricks of the trade' to model partiality.

To start, we first provide a sketchy survey of these tricks before moving on to examine some approaches in more detail. We will relate various approaches by describing, in a not completely serious way, what they mean for a familiar function: a *shoe shop*. Customers come in to the shop, ask for shoes, and either go away with shoes, or go away empty handed (when, *e.g.*, shoes in the requested size are not available).

Total. A total function corresponds to a shoe shop that always has *exactly* the desired shoes for each customer.

Since such perfection can rarely, if ever, be achieved, a shoe shop should naturally be modelled with a partial function. In the following we first survey some approaches that use total functions in order to represent partial functions.

Conditional. The function is applied only if the argument x meets a predicate P. Thus the usual way of using a function's value $\mathsf{f}(\mathsf{x})$ is in the *conditional style*: $\mathsf{P}(\mathsf{x}) \longrightarrow \mathsf{Q}(\mathsf{f}(\mathsf{x}))$.

You phone the store to see if the shoes are in.

Relational. Explicitly use the set-theoretic graph of a function, i.e. the relation representing the function. (In many interesting cases, such a relation can be defined inductively.)

You have to look through the storeroom yourself for your shoes.

Underspecification. When a function is defined, one neglects to give values for some arguments. The function still *has* values at those arguments, but only the existence of the values is guaranteed: the user will be unable to prove much about them.

You get a pair of shoes but can't be sure they're going to fit.

Lifting. A tag is attached to each result of the function, saying whether the input was in the domain of the function or not.

You either get your shoes (and know you got them) or you don't get your shoes (and know you didn't get them).

Default value. Each invocation of the function must provide a dummy value to return at undefined arguments.

When you ask for a pair of shoes, you must hand the clerk a pair of shoes. You get some shoes back, but if they didn't have the requested shoes, you get the shoes you handed in.

Dependent types. In this, the conditional approach has (roughly) been integrated into the type system of the logic [4, 5, 6]. Types can capture the domain of functions precisely, so that partial functions can often be modelled as total functions on a dependent type.

A heavyset Swede at the door of the shop allows entry only if you promise to ask for shoes they have in stock.

Instead of using these workarounds it is also possible to provide a tailored logic which explicitly supports partial functions. We can distinguish between two different approaches here.

Logical solutions. These lift concerns about partiality up to the level of the logical rules. Examples are **LPT** [7] or **PF** [8].

You ask for a pair of shoes and you get something from the clerk. It might not be shoes.

Domain theory. Functions over complete partial orders (so-called *domains*) are used to represent the computable subset of partial functions. This approach is due to Scott and is called LCF (Logic of Computable Functions) [9].

You ask for a pair of shoes and the clerk goes to search for them. If this search terminates, you get your shoes. Otherwise, you do not know if either the search ends in an infinite loop or the requested shoes are not in stock.

In this paper, we will not discuss a new approach in order to extend the list above; instead, we will survey and compare the techniques available in an existing logic: the higher-order logic instantiation of the generic theorem prover Isabelle [10]. Isabelle/HOL mechanizes a logic similar to Church's formulation of Higher Order Logic and is conceptually close to Gordon's HOL system [11]. (In the sequel HOL is short for Isabelle/HOL.) In HOL, all terms are typed and a function of type $\alpha \to \beta$ can only be applied to a term of type α. All functions are total: an function of type $\alpha \to \beta$ is defined for every element of type α. Since HOL is a logic of total functions, most of the known workarounds for partiality apply. We will examine some of these; however, there is another arrow in our quiver: Isabelle/HOL has a semantic embedding of Scott's LCF, called HOLCF, due to Regensburger [12, 13]. When partiality problems become particularly difficult, one can move smoothly to the extension and use the machinery that HOLCF makes available. Arguing about formalizations that contain both the total functions of HOL and the partial functions of HOLCF often requires extra work, e.g. proofs of continuity; we will explain a methodology developed for supporting this mixture of different types of function.

The approaches are illustrated by examples, some quite substantial, like an unification algorithm, operational semantics of an imperative language and modelling of finite and infinite sequences. Different examples are used for different approaches to partiality in order to clarify in what applications which approach is of most use.

1.1. Overview

The structure of the paper is given by several packages that are incorporated into Isabelle/HOL. Each package is used to illustrate one (or more) techniques used to treat partiality. We first examine how `TFL`, a package for the definition of total recursive functions can be adjusted to handle *underspecification* without loss of reasoning power (Section 2). In the same section we examine a larger example, a unification algorithm, where another standard method, *lifting*, is used to advantage. Then we move on to illustrate how *inductive definitions* can be used in place of partial functions, by examining a recent formalization of programming language semantics by Nipkow (Section 3). In Section 4 we discuss HOLCF and examine its use in solving a troublesome modelling question. Although this modelling problem has a solution using purely total functions, the HOLCF solution is much simpler. We discuss a few of the advantages and disadvantages of using HOLCF and discuss our approach to formalizations featuring a mix of total and partial functions.

1.2. Notation and Basic Definitions

The HOL logic offers the standard connectives and quantifiers. The following provides a short introduction to HOL's surface syntax:

Formulae The syntax is standard, except that there are two implications ($\longrightarrow$ and $\Longrightarrow$) and two equalities ($=$ and $\equiv$) which stem from object and metalogic, respectively. The distinction can be ignored while reading this paper. In parsing logical expressions, earlier members of the following list of infixes (denoting, in order, conjunction, disjunction, implication, and equality) have stronger binding power than later members: $=, \&, \vee, \longrightarrow, \equiv$. All infixes associate to the right. Suc denotes the successor function on the natural numbers. The Hilbert Choice operator $\varepsilon \mathsf{x}.\mathsf{P}(\mathsf{x})$ is used to implement underspecification: its behaviour is characterized by the following version of the Axiom of Choice:

$$\forall P\ x.\ P(x) \longrightarrow P(\varepsilon x.P(x))$$

Types follow the syntax for ML types, except that the Isabelle function arrow is written as $\Rightarrow$.

Theories introduce constants with the keyword **consts** and non-recursive definitions with **defs**. Further constructs are explained as we encounter them.

2. GENERAL RECURSION

TFL [14] is a package for defining total recursive functions described via ML-style pattern matching and for reasoning about them via recursion induction. The system is portable and has been instantiated to Gordon's HOL [11] and also to Isabelle-HOL. The interface to the system is that the user supplies recursion equations along with a termination relation and TFL will then define the function corresponding to the recursion equations and automatically derive a principle of *recursion induction*. For example, the following description defines the greatest common denominator algorithm (we will ignore termination relations throughout this paper):

```
function (termination relation)
  gcd(0,y) = y
  gcd(Suc x, 0) = Suc x
  gcd(Suc x, Suc y) =
    if (y≤x) then gcd(x-y,Suc y)
             else gcd(Suc x, y-x)
```

A proof of the *pattern completeness* theorem for this function

$$\begin{array}{l}\forall x.\ (\exists y.\ x = (0,y)) \quad \vee \\ \quad (\exists y.\ x = (\mathsf{Suc}\ y, 0)) \vee \\ \quad \exists y\ z.\ x = (\mathsf{Suc}\ y, \mathsf{Suc}\ z)\end{array}$$

is also carried out in this process. (The pattern completeness theorem shows that the patterns used in the definition of gcd are exhaustive and non-overlapping.) As a consequence the following induction theorem is automatically derived for *gcd*:

$$\begin{array}{l}\forall P.\ (\forall y.\ P\ (0,y)) \wedge \\ \quad (\forall x.\ P\ (\mathsf{Suc}\ \ x,0)) \wedge \\ \quad (\forall x\ y.\ (\neg(y\leq x) \longrightarrow P\ (\mathsf{Suc}\ x,y\text{-}x)) \wedge \\ \qquad (y\leq x \longrightarrow P\ (x\text{-}y,\mathsf{Suc}\ y)) \\ \qquad \longrightarrow P\ (\mathsf{Suc}\ x,\mathsf{Suc}\ y)) \\ \quad \longrightarrow \forall v\ w.\ P\ (v,w).\end{array}$$

Although the intent of TFL is to provide a nice environment for reasoning about *total* functions, partiality has been a recurrent subject in its development:

- partial functions are used in the statement and proof of the recursion theorem that TFL is based on;
- TFL has recently been extended to accept partial descriptions of functions; and
- interesting examples of partial functions defined through lifting have been defined and reasoned about.

We will discuss these in turn.

2.1. Wellfounded recursion

TFL bases itself on the notion of wellfoundedness (denoted WF). A general induction theorem applies to relations enjoying this property. Also, the following recursion theorem can be proven (see [14] for details):[2]

$$\begin{array}{l}(f \equiv \mathsf{WFREC}\ R\ M) \longrightarrow \mathsf{WF}(R) \\ \longrightarrow \forall x.\ f(x) = M\ (f|R,x)\ x.\end{array}$$

The proof of this theorem, as usual in proofs of recursion theorems, constructs a function by taking the union of a set of *partial* functions—making this work out properly in a logic of total functions takes a little bit of care. Also in the statement of the recursion theorem we find a use of underspecification to describe *function restriction*, a ternary operator that restricts a function to a certain set of values:

$$(f|R,y) \equiv \lambda x.\ \mathsf{if}\ R\ x\ y\ \mathsf{then}\ f\ x\ \mathsf{else}\ \varepsilon z.\mathsf{True}.$$

In this definition, the expression $\varepsilon z.\mathsf{True}$ uses the Hilbert choice operator to denote an arbitrary element of the range of f. Thus, we are using partiality (underspecification) to define a total function (more precisely, one in which *no* underspecification occurs): when TFL processes a definition, it traverses the recursion equations looking for recursive calls. If it can be established that the argument to each recursive call becomes smaller in a wellfounded relation, then the function is total. But what does this mean when all functions are already total? Merely, as already mentioned, that no underspecification occurs. This means that the recursion equations, as initially given by the user, can be validly used.

2.2. Underspecification and induction

Now we discuss how TFL deals with function descriptions which are missing some patterns. Suppose we give the following ML-style description to the system:

```
function (termination relation)
  (nth(0,h::t) = h) ∧
  (nth(Suc n,h::t) = nth(n,t))
```

This definition is a partial description of a function: it neglects to say what values nth has when the list argument is empty. However, we would still like to be able to derive an induction theorem for this function, and to do that we need full coverage of the domain,

[2] WFREC is a recursion operator; M (roughly) represents the body of the function, from which recursive occurrences have been λ-abstracted.

i.e., we must prove the pattern completeness theorem. In TFL, this is handled by underspecifying nth and also automatically generating the full set of patterns so that the induction theorem can still be derived. We get the following rules (an echo of the input, with the difference that the input is a *term* and the output a *theorem*):

$$(\mathsf{nth}(0,\mathsf{h}::\mathsf{t}) = \mathsf{h}) \wedge$$
$$(\mathsf{nth}(\mathsf{Suc\ n},\mathsf{h}::\mathsf{t}) = \mathsf{nth}(\mathsf{n},\mathsf{t}))$$

Notice that only the rules given by the user are returned. The values of the function for the unspecified clauses are derivable (and are equal to εz.True), but are not easily accessible: the user would have to burrow under the level of abstraction provided by TFL and that would be painful. What is more noteworthy is that TFL is able to return the following customized induction theorem, which contains the full cases:

$$\begin{aligned}&\forall \mathsf{P}.\ \mathsf{P}(0,[]) \qquad\qquad \wedge\\ &\quad(\forall \mathsf{n}.\ \mathsf{P}(\mathsf{Suc\ n},[])) \ \wedge\\ &\quad(\forall \mathsf{h\ t}.\ \mathsf{P}(0,\ \mathsf{h}::\mathsf{t})) \ \wedge\\ &\quad(\forall \mathsf{n\ h\ t}.\ \mathsf{P}(\mathsf{n},\mathsf{t}) \longrightarrow \mathsf{P}\ (\mathsf{Suc\ n},\ \mathsf{h}::\mathsf{t}))\\ &\quad\longrightarrow \forall \mathsf{v\ w}.\ \mathsf{P}(\mathsf{v},\mathsf{w}).\end{aligned}$$

This is only possible because TFL contains an adaptation of a standard pattern matching algorithm [15], which generates the complete set of patterns for the type the function is being defined over, as well as returning a nested case expression (its usual functionality). With this induction theorem, proofs of inductive properties of nth are easy, for example the following theorem:

$$\forall \mathsf{n\ l}.\ \mathsf{n}<\mathsf{length\ l} \longrightarrow \mathsf{mem}\ (\mathsf{nth}(\mathsf{n},\mathsf{l}))\ \mathsf{l}$$

where mem denotes membership of an element in a list. Notice the condition in this theorem: it restricts the domain of nth so that the only base cases considered in the induction are those in the initally given recursion equations. To summarize, underspecification for recursively defined functions requires extra steps to be taken so that handy induction theorems can still be automatically derived. Even then, properties of underspecified functions must be restricted so that underspecified portions of the domain of the function are ruled out. We will return to this point at the end of the next section.

Default Values and Lifting

Now we show how the lifting approach is used in a relatively large example: a unification algorithm [16, 17]. This illustrates a computer science model of partiality: failure. When two terms are not unifiable, the algorithm is required to fail. Thus, when unification is used in another function (*e.g.*, a type inference algorithm), failure (partiality) of type inference arises directly from failure (partiality) in unification. To begin the formalization we define a simple type of terms.

$$\begin{aligned}\textbf{datatype}\ &(\alpha)\mathsf{uterm} = \mathsf{Var}(\alpha)\\ &\quad\ |\ \mathsf{Const}(\alpha)\\ &\quad\ |\ \mathsf{Comb}(\alpha\ \mathsf{uterm})(\alpha\ \mathsf{uterm})\end{aligned}$$

The type of substitutions is represented by lists of pairs (v,t) where v has type α and t has type (α uterm). Thus the unification algorithm will have the (naive) type

$$(\alpha)\mathsf{uterm}\ *\ (\alpha)\mathsf{uterm} \Rightarrow (\alpha\ *\ \alpha\ \mathsf{uterm})\mathsf{list}$$

Now we define the substitution function (infix $\lhd$) by primitive recursion. It is implemented in terms of the well-known assoc function, the partiality of which is perfectly accomodated by a default value: when assoc is called by the substitution operation $\lhd$ and cannot find a replacement, the default value d is used instead.

$$\begin{aligned}&\mathsf{assoc\ v\ d}\ [] = \mathsf{d}\\ &\mathsf{assoc\ v\ d}\ ((\mathsf{a},\mathsf{b})\#\mathsf{al}) = \mathsf{if\ v{=}a\ then\ b}\\ &\qquad\qquad\qquad\qquad\mathsf{else\ assoc\ v\ d\ al}\\ &(\mathsf{Var\ v} \lhd \mathsf{s}) \quad = \mathsf{assoc\ v\ (Var\ v)\ s}\\ &(\mathsf{Const\ c} \lhd \mathsf{s}) \ = \mathsf{Const\ c}\\ &(\mathsf{Comb\ M\ N} \lhd \mathsf{s}) = \mathsf{Comb}\ (\mathsf{M} \lhd \mathsf{s})\ (\mathsf{N} \lhd \mathsf{s})\end{aligned}$$

Composition of substitutions (infix <>) and the occurs check (infix $\prec$) are also defined by primitive recursion, but we omit their definitions. Now we come to unification. First, we define a type of answers: either the algorithm fails or it returns a substitution.

$$\textbf{datatype}\ (\alpha)\mathsf{subst} = \mathsf{Fail}\ |\ \mathsf{Subst}\ (\alpha\ \mathsf{list})$$

Thus the algorithm has the following lifted type:

$$(\alpha)\mathsf{uterm}\ *\ (\alpha)\mathsf{uterm} \Rightarrow (\alpha\ *\ \alpha\ \mathsf{uterm})\mathsf{subst}$$

and is given to TFL in the form shown in Fig. 1.

The proof of termination of Unify is a difficult exercise in its own right, and we omit it. After the termination proof, one can prove the following correctness statement by induction (we also omit the definition of what an MGUnifier is).

$$\forall\theta.\ \mathsf{Unify}\ (\mathsf{P},\mathsf{Q}) = \mathsf{Subst}\ \theta \longrightarrow \mathsf{MGUnifier}\ \theta\ \mathsf{P}\ \mathsf{Q}.$$

An important point that this example brings out is that the success or failure of the algorithm is explicit in the returned value. In contrast to a function defined by underspecification, one can use Unify without knowing what inputs it is defined on. There may be a useful methodological point here, so we repeat it: knowledge of the domain of the function (required to prove properties about underspecified functions) is replaced by knowledge of the result of calling the function (lifting). As invocations of a function are found further and further from its definition, one would like to be able to forget about its domain. Thus the lifting approach may scale up better. However, we also note that use of the properties of Unify can be a bit clumsy to deal with because of the 'pipefitting' that must be done to handle success and failure. This is an instance of employing a monad [18].

3. INDUCTIVE DEFINITIONS

The graph of an n-ary function, total or partial, is easily represented as an $(n+1)$-ary relation. Hence partial functions can be specified and reasoned about in

```
function (termination relation)
   (Unify(Const m, Const n)  = if (m=n) then Subst[] else Fail) ∧
   (Unify(Const m, Comb M N) = Fail) ∧
   (Unify(Const m, Var v)    = Subst[(v,Const m)]) ∧
   (Unify(Var v, M) = if (Var v ≺ M) then Fail else Subst[(v,M)]) ∧
   (Unify(Comb M N, Const x) = Fail) ∧
   (Unify(Comb M N, Var v) = if (Var v ≺ Comb M N) then Fail else Subst[(v,Comb M N)]) ∧
   (Unify(Comb M1 N1, Comb M2 N2) =
      (case Unify(M1,M2)
        of Fail ⇒ Fail
         | Subst(θ) ⇒ (case Unify(N1 <| θ, N2 <| θ)
                        of Fail ⇒ Fail
                         | Subst σ ⇒ Subst (θ <> σ)))).
```

FIGURE 1. A unification algorithm as input to TFL

terms of their graphs. Reasoning about such relations is natural and powerful if one uses inductive definitions and the induction proof principle. In HOL the keyword `inductive` together with a set of rules defines the least relation closed under the rules. HOL automatically derives the corresponding induction principles, called *rule induction* in [9].

We will demonstrate this approach by the inductive definition of the operational semantics of the simple imperative programming language IMP with WHILE-loops, taken from Nipkow [19]. Clearly, a total function can never capture the semantics of infinitely looping programs in this language.

3.1. Syntax of IMP

Datatypes in HOL resemble those in functional programming languages and allow a direct representation of the abstract syntax of the commands of IMP:

```
datatype com = SKIP
  | ":="  loc aexp      (infixl)
  | ";"   com com       (infixl)
  | Cond  bexp com com  ("IF _ THEN _ ELSE _")
  | While bexp com      ("WHILE _ DO _")
```

The annotations in brackets define the concrete syntax.[3] For simplicity we identify the syntax of arithmetic expressions (aexp) and boolean expressions (bexp) with their semantics. The central semantic concept is that of a **state**, i.e. a mapping from locations to **values**. We formalize both locations loc and values val as unspecified types and define state, aexp and bexp as function spaces:

```
types  state = loc ⇒ val
       aexp  = state ⇒ val
       bexp  = state ⇒ bool
```

[3] We have omitted the priority of binding

3.2. Operational Semantics of IMP

We consider a **natural semantics** for IMP, expressing the evaluation of commands as a relation between a command, an initial state, and a final state. In HOL we declare a constant evalc as a set of such triples

```
consts  evalc :: (com * state * state)set
```

n and add some syntactic sugar for better readability:

```
translations  <c,s> ⟶ t  ≡  (c,s,t)∈evalc
```

This means we read and write <c,s> ⟶ t instead of (c,s,t)∈evalc. The relation evalc is defined inductively by a set of inference rules, which are represented by implications in HOL. Fig. 2 displays the definition. The assignment command is defined in terms of an auxiliary function on states:

```
consts assign :: state ⇒ val ⇒ loc ⇒ state  ("_[_/_]")
defs   s[m/x]  ≡  (λy. if y=x then m else s y)
```

where the suffix ("_[_/_]") in the definition introduces a specific infix syntax for the command.

Reasoning about inductive definitions. In [19] a number of other semantics of IMP are defined in HOL, an operational transition semantics and a denotational fixpoint semantics, for example. All these semantics are shown to be equivalent. The proofs make heavy use of rule induction and convincingly show the power of this proof principle. For further details we refer to [19].

3.3. Discussion

Tackling partiality by inductive definitions is particularly interesting, because many notions of mathematics are defined inductively, so that inductive definitions and proofs are natural and familiar principles for the reader of general mathematics textbooks. Of course, one has to distinguish between the inductive definition principle as such, which is often used to define arbitrary sets

```
inductive evalc
    <SKIP,s> ⟶ s
  <x := a,s> ⟶ s[a(s)/x]
    <c1,s0> ⟶ s1  ∧
    <c2,s1> ⟶ s2  ⟹  <c1;c2, s0> ⟶ s2

   b s ∧  <c1,s> ⟶ t  ⟹  <IF b THEN c1 ELSE c2, s> ⟶ t
  ¬b s ∧  <c2,s> ⟶ t  ⟹  <IF b THEN c1 ELSE c2, s> ⟶ t

  ¬b s  ⟹  <WHILE b DO c, s> ⟶ s
  b s ∧  <c,s> ⟶ s1  ∧
 <WHILE b DO c, s1> ⟶ s2  ⟹  <WHILE b DO c, s> ⟶ s2
```

FIGURE 2. Operational semantics: evalc as inductive definition in HOL

in mathematics, and its applications to the definition of partial functions. In the latter case it has to be guaranteed that the relation is indeed a (partial) function, *i.e.* that there is at most one y in relation to every x. For the former case, there are many examples in mathematics, e.g. the Borel hierarchy of subsets of the real numbers, can be described inductively.

4. PARTIAL FUNCTIONS IN HOLCF

The collections of techniques we have seen so far *avoid* partiality or code around it somehow; but sometimes it is necessary to have *real* partial functions. The HOL extension HOLCF formalizes domain theory [20], our theory for partial functions. HOLCF can, of course, be used to model semantics of sequential programming languages – the original motivation for developing domain theory. It has also been used to formalize FOCUS [21], a specification and verification methodology for distributed, reactive systems. In FOCUS the system requirements are described in HOL, and refinement steps end up with a system described in HOLCF as a set of computable partial functions.

In 4.1 we provide a brief introduction to HOLCF. In 4.2. we discuss the following interesting point: Sometimes, it is easier to model mathematical objects by use of partiality, even when there is a solution in the total setting. We illustrate this phenomenon with an abstract datatype of finite and infinite sequences.

On the other hand, partiality often complicates proofs; therefore, we prefer to stay in HOL as long as possible and switch only to HOLCF when really required. We explain how we deal with mixtures of partial and total objects in Subsection 4.3.

4.1. Introduction to HOLCF

HOLCF [12, 13] extends HOL conservatively with concepts of domain theory such as complete partial orders, continuous partial functions and a fixed point operator. As a result, the original LCF logic [22] constitutes a proper sublanguage of HOLCF. HOLCF uses Isabelle's type classes to distinguish HOL and LCF types. More precisely, a type class pcpo which is equipped with a complete partial order $\sqsubseteq$ and a least element $\bot$ is introduced. pcpo becomes the default type class of HOLCF and is a subclass of term, the default type class of HOL.

There is a special type for partial, continuous functions between pcpos. Elements of this type are called *operations*, the type constructor is denoted by $\rightarrow$, in contrast to the standard HOL function type constructor $\Rightarrow$. Special syntax is introduced for abstractions (Λ instead of λ) and applications (f‘t instead of (f t)). For continuous functions the fixpoint operator fix exists. HOLCF comes with several standard domains. The truth values tr, which are HOLCF's counterpart to HOL's bool, are modeled by a flat domain with the elements TT, FF and $\bot$. Operations on them include andalso, orelse and neg, which are strict extensions of the standard predicates $\wedge$,$\vee$ and $\neg$ on bool. The counterpart of the conditional expression if A then B else C is written If A then B else C fi.

4.2. A Modelling Example

Sometimes it is simpler to use partial functions, even if there is a solution using total functions. As an example we describe the problem of modelling finite and infinite sequences. The following requirements are posed on the abstract datatype sequence:

- Sequences are finite or infinite.
- A predicate is_finite selects the finite ones.
- List operations like hd, tl, map, filter, concat, length, last must be provided.
- An indexing function nth should be available.

Such sequences are often used to model communication histories (traces) of distributed systems. For abstraction and modularity purposes, internal messages are allowed in histories. Therefore, a filter operation is needed to remove internal messages from histories. In the following we will focus on how filter for sequences is defined

in total and partial settings. The main problem is how elements removed by filter are treated: they do not really disappear if they are replaced by explicit "gaps", which will be the cause of difficulties in the total setting.

Sequences as total functions. In HOL, infinite sequences can be described by functions of type nat⇒α. However, if we want to incorporate finite sequences into the model, we have to define values for the infinite tail of a finite sequence. This can be done by using nat⇒(α)option[4], where None is used to denote a nonexisting element. To avoid the case in which None appears *within* a sequence – otherwise the representation would not be unique – the predicate

$$\text{is\_sequence s} \equiv \forall \text{i.s(i)=None} \longrightarrow \text{s(Suc(i))=None}$$

is introduced, which has to hold of every sequence. At first glance it seems to be easy to define filter in this context: Just replace Some(a) by None if P(a) does not hold:

```
filter P s ≡ λi.
   case s(i) of
    None     ⇒ None
| Some(a) ⇒ if P(a) then Some(a)
                    else None
```

But the problem with this coding of partiality is that such a filter generates Nones all over the sequence; therefore, to satisfy the predicate is_sequence, filter has to compute a normal form NF, where Nones are not allowed within a sequence. However, to remove Nones in this setting is very awkward, because shifting means "redefinition" of all succeeding values. NF can be defined by claiming a monotone function f between sequences that serves as an index transformation (see Fig. 3):

$$\begin{aligned}\text{NF(s)} \equiv\ & \varepsilon\text{nf.}\ \exists\text{f.}\ \text{mono(f)} \wedge\ (\forall\text{i.}\ \text{nf(i)=s(f(i)))} \wedge \\ & (\forall\text{j.}\ \text{j} \notin \text{range(f)} \longrightarrow \text{s(j)= None}) \wedge \\ & \text{is\_sequence nf}\end{aligned}$$

Hilbert's choice operator εx.P(x) and the index transformation make proofs about NF hard and clumsy. Other operations, like infinite concatenation, are also very complicated to realize with this representation. The reader may think that there is an easier means of directly defining sequences in a total setting, for example by an inductive definition, but we have not been able to find such, despite considerable effort.

Sequences as disjoint sum of finite and infinite sequences. Another modelling possibility is to use a disjoint sum of finite and infinite sequences:

$$(\alpha)\text{seq} \equiv \text{FinSeq}((\alpha)\text{list}) \mid \text{InfSeq}(\text{nat}\Rightarrow(\alpha))$$

[4]We use the standard *option* datatype which is defined as (α)option= Some(α) | None

Here (α)list stands for the inductively defined finite lists of HOL. Note that filter can produce both finite and infinite sequences from an infinite sequence. Therefore filter in this setting is defined using limits of projections of finite sequences, which are in turn defined using recursion on lists. Thus, the notion of a limit of an ascending chain of sequences (acording to the prefix ordering) has to be formalized. This approach has been taken by Chou and Peled [23] to model sequences as a prerequisite for the formal verification of a partial-order reduction technique in the HOL theorem prover [11]. However, this seems to be an ad hoc modelling of domain concepts tailored for a specific datatype, which is more generally supported in HOLCF, as we will see in the sequel.

Sequences as domains. In HOLCF finite and infinite sequences are defined by the simple recursive domain equation

$$\textbf{domain}\ (\alpha)\text{sequence} = \text{nil} \mid (\alpha)\#(\text{lazy}\ (\alpha)\text{sequence})$$

where nil and the "cons"-operator # are the constructors of the datatype. # is strict in its first argument and lazy in the second. The datatype package of HOLCF [24] automatically proves a number of user-relevant theorems concerning the constructors, discriminators, and selectors of the datatype, as well as induction and co-induction principles. For example, hd(nil)=$\bot$ and tl($\bot$)=$\bot$ are generated automatically. See the next section for a short account of the induction principles.

The definition above means that elements of type sequence come in 3 flavours:

- Finite total sequences: $a_1\#\ldots\#a_n\#\text{nil}$.
- Finite partial sequences: $a_1\#\ldots\#a_n\#\bot$.
- Infinite sequences: $a_1\#\ldots\#a_n\#\ldots$.

All the operations known from functional programming with lazy lists, e.g. map, filter and the concatenation operator , are easily defined over (α)sequence. For example, filter has type

$$(\alpha \to \text{tr}) \to (\alpha)\text{sequence} \to (\alpha)\text{sequence}$$

and is defined as a fixpoint. The following rewrite rules can immediately be deduced from the definition:

```
filter‘P‘⊥=⊥
filter‘P‘nil=nil
x≠⊥ ⟹ filter‘P‘(x#xs)= If P‘x
                        then x#(filter‘P‘xs)
                        else filter‘P‘xs fi
```

To us, this seems to be the simplest implementation of filter. The other operations of the abstract datatype are likewise easy to define. In [25] this representation of sequences has been used to model finite and infinite behaviors of I/O automata, a specification and verification methodology for reactive, distributed systems.

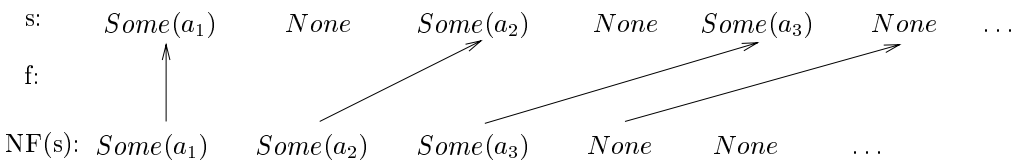

FIGURE 3. Generating Normal Forms using Index Transformation

Comparison. The second and last solution *have in common* that recursive definitions construct a completely new sequence, where the removed elements are really absent, whereas the total solution just changes the old sequence a bit by replacing elements by Nones. Then the main problem is to remove these Nones.

The second and last solution *differ* in the handling of infinite sequences. The second solution has to represent infinite sequences separately, while the HOLCF approach just extends the recursive construction to infinity. Thus, in this example, the advantage of partiality is the representation of infinite computations.

The interested reader is referred to [26] where a deeper comparison of these three approaches to model sequences in higher order logic is presented.

Related Work. An important approach to modelling infinite datatypes is the theory of coalgebras. This theory has recently been mechanized in higher order theorem provers, namely in Isabelle/HOL by Paulson [27] and in PVS by Jacobs and Hensel [28]. Both implementations have been applied to sequences. It is known that the definition of *filter* in a pure coalgebraic manner is more complicated than the inductive definition. Paulson very recently found a mixed inductive-coinductive definition of *filter* on sequences that allows the derivation of some common properties. It remains to see whether such a definition is as easy to use as that of HOLCF.

Feferman [29] has recently also addressed the problem of defining a common type of finite and infinite sequences; significantly, his solution also models sequences by partial functions. One difference with our approach is that his solution does not require continuity of functions, and has not, as far as we know, been mechanized in a proof tool.

4.3. Moving between HOL and HOLCF

Although we have seen that some *modelling* questions have nice solutions in HOLCF, there is also a disadvantage in that *proofs* may become more complicated. Why? First, $\bot$ as least element of every domain often creates an additional proof case, which cannot always be discharged automatically. However, *admissibility* proofs cause even more problems. In HOLCF the main proof principles, structural induction and fixpoint induction, demand that the predicate to be proven is *admissible*, denoted by adm P. Because of their importance we mention induction theorems here explicitly: fixpoint induction reduces properties of fixpoints to properties of function application:

$$\begin{array}{l}\forall P.\ \mathsf{adm}\ P \wedge \\ \quad P\ \bot \wedge \\ \quad (\forall x.\ P\ x \longrightarrow P\ (f`x)) \\ \quad \longrightarrow P\ (\mathsf{fix}`f)\end{array}$$

whereas structural induction (here for the example of the sequences of the last section)

$$\begin{array}{l}\forall P.\ \mathsf{adm}\ P \wedge \\ \quad P\ \bot \wedge \\ \quad P\ \mathsf{nil} \wedge \\ \quad (\forall x\ xs.\ x \neq \bot \wedge\ P\ xs \longrightarrow P\ (x\#xs) \\ \quad \longrightarrow \forall y.\ P\ y\end{array}$$

reduces properties of an (infinite) data element to local properties of the constructors of that datatype. As noted, both rules need admissibility, which means that P holds for the least upper bound of every chain satisfying P. If P (reduced to conjunctive normal form) contains no existential quantifier or negation, admissibility can be proved automatically by reducing it to the continuity of every operation occurring in P, for which an automatic tactic exists. Otherwise admissibility has to be shown interactively by arguing explicitly about least upper bounds of chains. As experience shows, this can be very clumsy and often represents the most difficult part of the entire proof.

Therefore we propose using HOLCF only when the general model becomes much simpler, as in the case of (α)sequence, or when modelling programming languages in Scott's style. Even in the latter case, it pays to 'stay total' for as long as possible.

Lifting HOL types. Domain definitions, like (α)sequence, require that the argument type α be a domain type, too. Suppose however, we want to model sequences of natural numbers. Then it would not be appropriate according to the argumentation above to model the natural numbers by a domain, thus dragging undefined elements and partial orders into them. This is emphasized by the fact that often theorem provers provide a theory of natural numbers with total operators like Suc, +, * and $\geq$, together with specific proof procedures, which should not be developed twice. Instead we propose "lifting" the existing HOL type nat to a domain *a posteriori*. Then computations on nat are done in HOL, and HOLCF is only used for the *sequences* of natural numbers.

As example, we could then define an operation

$$\mathsf{gt9} : ((\mathsf{nat})\mathsf{lift})\mathsf{sequence} \to ((\mathsf{nat})\mathsf{lift})\mathsf{sequence}$$

that filters every element greater than 9 in a sequence by

$$\mathsf{gt9} \equiv \mathsf{filter}`(\mathsf{flift2}\ (\lambda x.x \geq 10)).$$

Here filter is a HOLCF operation, $\geq$ a HOL predicate, and flift2 one of the lifting constructs we will introduce in the sequel.

We define a type constructor lift of arity (term)pcpo which lifts every HOL-datatype to a pcpo type:

$$\textbf{datatype}\ (\alpha)\mathsf{lift} \equiv \mathsf{Undef} \mid \mathsf{Def}(\alpha)$$

The least element and the approximation ordering are defined very easily:

$$\begin{array}{l} \bot \equiv \mathsf{Undef} \\ x \sqsubseteq y \equiv (x{=}y) \mid x{=}\mathsf{Undef} \end{array}$$

This is known as a *flat* domain. Note that $\bot$ and $\sqsubseteq$ are overloaded and this definition only fixes their meaning at type (α)lift. Futhermore, Undef is completely hidden from the user who deals with $\bot$ as least element from now on.

Lifting HOL functions. If in an operation on ((nat)lift)sequence a total function on nat is involved, it is also necessary to lift this total function to an partial operation. For this purpose we introduce two functionals that transform HOL functions to HOLCF operations using lift. The type variables α,α_1 and α_2 are of class term, whereas β is of class pcpo.

$$\begin{array}{ll} \mathsf{flift1} & (\alpha \Rightarrow \beta) \Rightarrow ((\alpha)\mathsf{lift} \to \beta) \\ \mathsf{flift2} & (\alpha_1 \Rightarrow \alpha_2) \Rightarrow ((\alpha_1)\mathsf{lift} \to (\alpha_2)\mathsf{lift}) \end{array}$$

The former lifts only the argument type of a HOL function, the latter both argument and result type. Lifting essentially means strict extension. Formally:

$$\begin{array}{l} \mathsf{flift1\ f} \equiv \Lambda x.\ \mathsf{case\ x\ of} \\ \qquad\qquad \mathsf{Undef} \Rightarrow \bot \\ \qquad\qquad \mid \mathsf{Def(y)} \Rightarrow \mathsf{f(y)} \\ \mathsf{flift2\ f} \equiv \Lambda x.\ \mathsf{case\ x\ of} \\ \qquad\qquad \mathsf{Undef} \Rightarrow \bot \\ \qquad\qquad \mid \mathsf{Def(y)} \Rightarrow \mathsf{Def(f(y))} \end{array}$$

Notice that these two functionals indeed suffice: Since the truth values tr are defined as (bool)lift, a special lifting for booleans or predicates on booleans is not needed.

Using the above lifting functionals instead of lifting argument or result types in an *ad hoc* fashion has the following advantages:

- First, these concepts are frequently used, and abbreviating them increases readability.
- More importantly, these functionals enable automated proof support for continuity proofs. In HOLCF β-reduction on domains is subject to the following continuity restriction:

$$\mathsf{cont}(\lambda x.t(x)) \longrightarrow (\Lambda x.t(x))`u = t(u)$$

where $\mathsf{cont}(\lambda x.t(x))$ means that t is continuous in x. These continuity proof obligations are discharged automatically for all terms of the LCF sublanguage (Λ-abstractions and f`t-applications). But for normal HOL terms (λ-abstractions and (f t)-applications) these proof obligations have to be discharged manually. Here the lifting functionals serve as a "continuity interface" to HOL. By proving them to be continuous and adding these theorems to the automatic proof tactic, we get automatic continuity proofs also for the combination of HOL and LCF terms. More precisely, the following three theorems are proved:

$$\begin{array}{l} \mathsf{cont}(\lambda x.\mathsf{flift1\ f\ x}) \\ \mathsf{cont}(\lambda x.\mathsf{flift2\ f\ x}) \\ \forall a.\ \mathsf{cont}(\lambda y.((\mathsf{f\ y})\ \mathsf{a})) \Longrightarrow \mathsf{cont}(\lambda y.\mathsf{flift1\ (f\ y)\ x}) \end{array}$$

The first two theorems hold, since strict functions from a flat domain are always continuous. The last one handles the case when continuity of the function f to lift is claimed not in its argument x, but in another variable y. It is easily proved, as flift1 behaves as a constant in this context. An equivalent theorem need not be stated for flift2, as continuity requires a partial order $\sqsubseteq$ on the range. Therefore these theorems suffice to establish automatic proof support for HOL and LCF terms. Note that this proof support is particularly useful for admissibility requirements.

5. ANALYSIS AND DISCUSSION

This section addresses the issue of proof support in the various approaches to partiality. In particular it is argued for our solution of taking an existing platform supporting total functions instead of a building a new system tailored only for partial functions. As well, we discuss a few remaining technical points.

The major benefit of our approach is *reuse*. Higher-order logic theorem provers, like Isabelle-HOL, often provide many user-invocable proof procedures. Following our approach, one has the immense pragmatic benefit that existing proof procedures need not be modified. However, if partial functions were allowed in the logic, some important proof techniques might not work. For example, totality of uninterpreted function symbols is assumed in the Nelson-Oppen method for combining decision procedures [30].

In applications that truly need partial functions, our methodology, as explained in section 4.3, allows further instances of reuse. Our experience in proving facts about sequences has shown that many proof obligations in the mixed HOL/HOLCF setting can be broken down into pure HOL propositions. For example, since the type of truth values tr in HOLCF was defined as (bool) lift, there is no need for (say) a tableaux prover

for three valued logic, which would be a major undertaking, see for example [31].

Another example is conditional rewriting, which is heavily used in proof. The HOLCF simplifier (an instantiation of Isabelle's generic simplifier) must solve frequent (and usually trivial) conditions showing that function arguments are not equal to bottom. Using the lift interface to HOL explicitly documents the intuition that elements are not undefined – they are modeled in HOL and lifted to HOLCF only *a posteriori* to fit the context. This can often be used to rule out the $x \neq \bot$ conditions statically, i.e. before rewriting commences.

Another important point is that most work to date on proof support for partiality has focused only on how to define partial functions and the semantics of partial function application. This is not enough: proof principles for the properties of partial functions must also be accounted for; this explains our emphasis on induction throughout the paper. To re-capitulate, for those total functions defined by recursion, induction principles can be straightforwardly derived. However, as we have shown with our TFL examples, extra steps must be taken to construct induction principles for partially described (but total) functions. For partial recursive functions, fixpoint induction is the main reasoning principle. For partial functions represented by inductively defined relations, rule induction is indispensable. The kind of partiality to adopt when defining a function is an important matter, and experience is required in order to make the right choice.

Finally, we remark a more technical point: the observant reader will have noticed that there have been several occurrences of lifting in this paper, namely (α)option, (α)subst and (α)lift. From the point of view of uniformity, it seems that only one lifted type, *e.g.* option should be defined. On the other hand, such an approach would mean that the type of the unification algorithm would be

$$(\alpha)\mathsf{uterm} * (\alpha)\mathsf{uterm} \Rightarrow (\alpha * \alpha\ \mathsf{uterm})\ \mathsf{list\ option}$$

rather than

$$(\alpha)\mathsf{uterm} * (\alpha)\mathsf{uterm} \Rightarrow (\alpha * \alpha\ \mathsf{uterm})\mathsf{subst}$$

which we feel is more readable. Using type abbreviations, which are available in Isabelle/HOL, should solve the problem. However, the constructors Some and None will also need to be aliased in some way. Furthermore, (α)lift distinguishes itself from the two other liftings as it is the only one which is interpreted as a flat domain.

6. RELATED WORK

There has been a lot of theoretical work on free logics and logics of partial functions, a small amount of which has been mechanized. We have neglected this research in our discussion; however, the following citations should serve as useful entrypoints to the interested reader: [1, 2, 32, 33, 8, 34].

Turning to mechanizations of partial functions, the work most closely related to our work is probably [35] which provides a subtle analysis of partiality and recursive function definition as presented in the LAMBDA system, which is used in commercial hardware verification. Mechanizations of other logics are provided by the following tools:

- The IMPS system [36] implements a simple type theory of partial functions and it has been used to formalize some interesting mathematics.
- Dependent types can be well utilized in both constructive and classical logic, as shown by the LEGO, Coq, and PVS systems [37, 38, 6].
- Recently, implementations of classical ZF set theory, which treats functions as certain sets of ordered pairs, have become available [39, 40, 41, 42, 43].

It will be interesting to see whether the availability of partial functions in these implementations will provide objectively better verification environments than those based on total functions.

7. CONCLUSION

In this paper we have attempted no new theory, rather we have surveyed methods for dealing with partiality in a well-established higher-order logic of total functions. The methods range from underspecification, lifting, and default values, to more special-purpose (although quite powerful) formalization styles, such as inductively defined relations, to a formalization of domain theory. For each of these, we have shown, by example, how proof support, e.g. induction theorems, is provided.

We have also discussed the advantages and disadvantages of lifting and underspecification. Lifting provides clarity and modularity at the price of clutter, whereas underspecification allows clean formalizations, but burdens the user with the task of correctly constraining goals involving underspecified functions.

Our broad spectrum approach allows total functions to be employed as much as possible; partiality requires extra work, as it does in any setting. However, when partiality problems must be addressed, a range of different solutions can be applied, from lightweight to heavyweight. Our heavyweight solution, domain theory, is a restricted model, where continuity of functions becomes a requirement. For modelling issues dealing with programs, this is suitable. Our Section 4 modelling example shows, somewhat surprisingly, that some issues resolvable with total functions can be more easily solved in domain theory. Moreover, we presented a smooth integration of domain theory into the total logic HOL, so that the theory libraries developed for HOL can be reused in a hybrid HOL/HOLCF formalization.

Computer Science shows that there are many flavours of partiality. For example:

- total functions (zero partiality)

- intended infinite loops (reactive systems, e.g. our finite and infinite sequences)
- unintended infinite loops (programmer mistake)
- error states in programs
- total functions of high computational complexity.

Our approach uses standard methods in classical logic to deal with all of these distinctions except the last. To summarize, we feel that the convenience of staying in a framework of total functions is a definite advantage, since standard techniques, when well-supported by tools, often suffice in treating partiality.

ACKNOWLEDGEMENTS

This work has been partially funded by "Bundesministerium für Forschung und Technologie" within the project *KorSys*, and by "Deutsche Forschungsgesellschaft" within the project *Deduktion*. The authors like to thank the anonymous referees for their helpful comments.

REFERENCES

[1] J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J. C. P. Woodcock, editors, *Proc. 3rd Refinement Workshop*, pages 51 – 69. Springer-Verlag, 1991.

[2] Solomon Feferman. Definedness. *Erkenntnis*, 43:295–320, 1995.

[3] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[4] R. Constable, S. Allen, H. Bromly, W. Cleaveland, J. Cremer, and R. Harper et al. *Implementing Mathematics With the Nuprl Proof Development System*. Prentice-Hall, New Jersey, 1986.

[5] Thierry Coquand and Gerard Huet. The Calculus of Constructions. *Information and Computation*, 76(2-3):95–120, February-March 1988.

[6] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proc. 11th Conf. on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752, 1992.

[7] Michael Beeson. *Foundations of Constructive Mathematics*. Springer-Verlag, 1985.

[8] William Farmer. A partial functions version of Church's simple theory of types. *J. Symbolic Logic*, 55(3):1269–1291, 1990.

[9] G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, 1993.

[10] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[11] M.C.J. Gordon and T.F. Melham. *Introduction to HOL: a theorem-proving environment for higher-order logic*. Cambridge University Press, 1993.

[12] Franz Regensburger. HOLCF: Higher Order Logic of Computable Functions. In E.T. Schubert, P.J. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 293–307. Springer-Verlag, 1995.

[13] Franz Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF*. PhD thesis, Technische Universität München, 1994.

[14] Konrad Slind. Function definition in higher order logic. In J. von Wright, J. Grundy, and J. Harrison, editors, *Proc. 9th Int. Conf. Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 381–398. Springer-Verlag, 1996.

[15] Lennart Augustsson. Compiling pattern matching. In J.P. Jouannnaud, editor, *Conference on Functional Programming Languages and Computer Architecture (LNCS 201)*, pages 368–381, Nancy, France, 1985.

[16] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[17] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[18] Philip Wadler. The essence of functional programming. In *Proc. 19th Annual Symposium on Principles of Programming Languages*. ACM Press, 1992. Invited talk.

[19] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. In *Proc. 16th Int. Conf. Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lecture Notes in Computer Science*, pages 180–192. Springer-Verlag, 1996.

[20] D. Scott and C. Gunter. Semantic Domains and Denotational Semantics. In *Handbook of Theoretical Computer Science*, chapter 12, pages 633 – 674. Elsevier Science Publisher, 1990.

[21] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The Design of Distributed Systems: An Introduction to Focus — Revised Version. Technical Report TUM-I9202-2, Technische Universität München, Fakultät für Informatik, 80290 München, Germany, 1993.

[22] Lawrence C. Paulson. *Logic and Computation*. Cambridge University Press, 1987.

[23] Ching-Tsun Chou and Doron Peled. Formal verification of a partial-order reduction technique for model checking. In T. Margaria and B. Steffen, editors, *Proc. 2nd Workshop Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[24] D. v. Oheimb. Datentypspezifikationen in Higher-Order LCF. Master's thesis, Computer Science Department, Technical University Munich, 1995.

[25] Olaf Müller and Tobias Nipkow. Traces of I/O-automata in Isabelle/HOLCF. In *Proc. 7th Int. Joint Conf. on Theory and Practice of Software Development (TAPSOFT'97)*, volume 1214 of *Lecture Notes in Computer Science*, pages 580–595. Springer-Verlag, 1997.

[26] Marco Devillers, David Griffioen, and Olaf Müller. Possibly infinite sequences in theorem provers: A comparative study. In Elsa Gunter, editor, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOL'97)*, volume 1275

of *Lecture Notes in Computer Science*, pages 89–104. Springer-Verlag, 1997.

[27] Lawrence C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *J. Logic and Computation*, 7:175 – 204, 1997.

[28] U. Hensel and B. Jacobs. Proof principles for datatypes with iterated recursion. Technical Report CSI-R9703, Computing Science Institute, University of Nijmegen, 1997.

[29] Solomom Feferman. Computation on abstract data types. The extensional approach, with an application to streams. *Annals of Pure and Applied Logic*, 81:75–113, 1996.

[30] Greg Nelson. Combining satisfiability procedures by equality-sharing. *Contemporary Mathematics*, 29(1):25–43, 1984.

[31] S. Agerholm and J. Frost. An Isabelle-based theorem prover for VDM-SL. In Elsa Gunter, editor, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOL'97)*, volume 1275 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1997.

[32] J.H. Cheng H. Barringer and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[33] Jan Kuper. An axiomatic theory for partial functions. *Information and Computation*, 107(1):104–150, November 1993.

[34] W. Farmer, M. Kerber, and M. Kohlhase, editors. *Proc. Workshop on Mechanization of Partial Functions*, New Brunswick, NJ, USA, July 1996. Held in conjunction with CADE-13, Lecture Notes in Computer Science 1104.

[35] Simon Finn, Mike Fourman, and John Longley. Partial functions in a total setting. *J. Automated Reasoning*, 18(1):85–104, 1997.

[36] William Farmer, Joshua Guttman, and Javier Thayer. IMPS: an interactive mathematical proof system. In Mark Stickel, editor, *10th International Conference on Automated Deduction (CADE'90)*, volume 449 of *Lecture Notes in Computer Science*, pages 653–654. Springer-Verlag, 1990.

[37] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994.

[38] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, and C. Parent et al. The Coq proof assistant user's guide version 5.8. Technical Report 154, INRIA, May 1993.

[39] Larry Paulson. Set theory for verification: II. induction and recursion. *J. Automated Reasoning*, 15:167–215, 1995.

[40] Larry Paulson. Set theory for verification: I. from foundations to functions. *J. Automated Reasoning*, 11:353–389, 1993.

[41] S. Kromodimoeljo, M. Saaltink, D. Craigen, B. Pase, and I. Meisels. A tutorial on EVES using s-verdi. Technical report, Odyssey Research Associates Canada, December 1995.

[42] M.J.C. Gordon. Merging HOL with set theory: preliminary experiments. Technical Report 353, University of Cambridge Computer Laboratory, 1994.

[43] S. Agerholm. Formalising a model of the λ-calculus in HOL-ST. Technical Report 354, University of Cambridge Computer Laboratory, 1994.