Inductive datatypes in HOL — lessons learned in Formal-Logic Engineering

Stefan Berghofer and Markus Wenzel

Technische Universität München
Institut für Informatik, Arcisstraße 21, 80290 München, Germany
http://www.in.tum.de/~berghofe/
http://www.in.tum.de/~wenzelm/

Abstract. Isabelle/HOL has recently acquired new versions of definitional packages for inductive datatypes and primitive recursive functions. In contrast to its predecessors and most other implementations, Isabelle/HOL datatypes may be mutually and indirect recursive, even infinitely branching. We also support inverted datatype definitions for characterizing existing types as being inductive ones later. All our constructions are fully definitional according to established HOL tradition. Stepping back from the logical details, we also see this work as a typical example of what could be called "Formal-Logic Engineering". We observe that building realistic theorem proving environments involves further issues rather than pure logic only.

1 Introduction

Theorem proving systems for higher-order logics, such as HOL [5], Coq [4], PVS [15], and Isabelle [18], have reached a reasonable level of maturity to support non-trivial applications. As an arbitrary example, consider Isabelle/Bali [14], which is an extensive formalization of substantial parts of the Java type system and operational semantics undertaken in Isabelle/HOL.

Nevertheless, the current state-of-the-art is not the final word on theorem proving technology. Experience from sizable projects such as Isabelle/Bali shows that there are quite a lot of requirements that are only partially met by existing systems. Focusing on the actual core system only, and ignoring further issues such as user interfaces for theorem provers, there are several layers of concepts of varying logical status to be considered. This includes purely syntactic tools (parser, pretty printer, macros), type checking and type inference, basic deductive tools such as (higher-order) unification or matching, proof procedures (both simple and automatic ones), and search utilities — just to name a few.

Seen from a wider perspective, the actual underlying logic (set theory, type theory etc.) becomes only one element of a much larger picture. Consequently, making a theorem proving system a "success" involves more than being good in the pure logic rating. There is a big difference of being able to express certain concepts *in principle* in some given logic vs. offering our customers scalable mechanisms for *actually doing* it in the system.

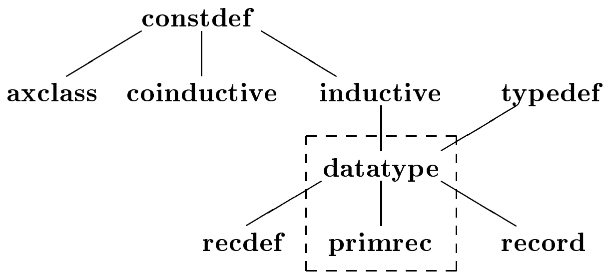
Advanced *definitional mechanisms* are a particularly important aspect of any realistic formal-logic environment. While working in the pure logic would be sufficient in principle, actual applications demand not only extensive libraries of derived concepts, but also general mechanisms for introducing certain kinds of mathematical objects. A typical example of the latter would be inductive sets and types, together with recursive function definitions.

According to folklore, theorem proving is similar to programming, but slightly more difficult. Apparently, the same holds for the corresponding development tools, with an even more severe gap of sophistication, though. For example, consider the present standard in interactive theorem proving technology related to that of incremental compilers for languages such as ML or Haskell. Apparently, our theorem provers are still much more primitive and inaccessible to a wider audience than advanced programming language compilers. In particular, definitional mechanisms, which are in fact resembling a "theory compiler" quite closely, are often much less advanced than our users would expect.

An obvious way to amend for this, we argue, would be to transfer general concepts and methodologies from the established disciplines of Software and Systems Engineering to that of theorem proving systems, eventually resulting in what could be called *Formal-Logic Engineering*.

Getting back to firm grounds, and the main focus of this paper, we discuss the new versions of advanced definitional mechanisms that Isabelle/HOL has acquired recently: **inductive** or **coinductive** definitions of sets (via the usual Knaster-Tarski construction, cf. [17]), inductive datatypes, and primitive recursive functions. Our primary efforts went into the **datatype** and **primrec** mechanisms [2], achieving a considerably more powerful system than had been available before. In particular, datatypes may now involve *mutual* and *indirect recursion*, and *arbitrary branching* over existing types.[1] Furthermore, datatype definitions may now be *inverted* in the sense that existing types (such as natural numbers) may be characterized later on as being inductive, too.

The new packages have been designed for cooperation with further subsystems of Isabelle/HOL already in mind: **recdef** for general well-founded functions [21, 22], and **record** for single-inheritance extensible records [13]. Unquestionably, more such applications will emerge in the future. The hierarchy of current Isabelle/HOL definitional packages is illustrated below. Note that **constdef** and **typedef** refer to HOL primitives [5], and **axclass** to axiomatic type classes [24].



[1] Arbitrary (infinite) branching is not yet supported in Isabelle98-1.

The basic mode of operation of any "advanced" definitional package such as **datatype** is as follows: given a simple description of the desired result theory by the user, the system automatically generates a sizable amount of characteristic theorems and derived notions underneath. There are different approaches, stemming from different logical traditions, of how this is achieved exactly. These approaches can be roughly characterized as follows.

Axiomatic The resulting properties are generated syntactically only, and introduced into the theory as *axioms* (e.g. [16]).

Inherent The underlying *logic is extended* in order to support the desired objects in a very direct way (e.g. [20]).

Definitional Taking an existing logic for granted, the new objects are represented in terms of existing concepts, and the desired properties are *derived from the definitions* within the system (e.g. [2]).

Any of these approaches have well-known advantages and disadvantages. For example, the definitional way is certainly a very hard one, demanding quite a lot of special purpose theorem proving work of the package implementation. On the other hand, it is possible to achieve a very high quality of the resulting system — both in the purely logical sense meaning that no "wrong" axioms are asserted and in a wider sense of theorem proving system technology in general.

The rest of this paper is structured as follows. Section 2 presents some examples illustrating the user-level view of Isabelle/HOL's new **datatype** and **primrec** packages. Section 3 briefly reviews formal-logic preliminaries relevant for our work: HOL basics, simple definitions, inductive sets. Section 4 describes in detail the class of admissible **datatype** specifications, observing fundamental limitations of classical set theory. Section 5 recounts techniques for constructing mutually and indirectly recursive, infinitely branching datatypes in HOL, including principles for induction and recursion. Section 6 discusses some issues of integrating the purely-logical achievements into a scalable working environment.

2 Examples

As our first toy example, we will formalize some aspects of a very simple functional programming language, consisting of arithmetic and boolean expressions formalized as types α aexp and α bexp (parameter α is for program variables).

$$
\begin{array}{llcl}
\textbf{datatype} & \alpha\ \mathsf{aexp} & = & \mathsf{If}\ (\alpha\ \mathsf{bexp})\ (\alpha\ \mathsf{aexp})\ (\alpha\ \mathsf{aexp}) \\
 & & \mid & \mathsf{Sum}\ (\alpha\ \mathsf{aexp})\ (\alpha\ \mathsf{aexp}) \\
 & & \mid & \mathsf{Var}\ \alpha \\
 & & \mid & \mathsf{Num}\ \mathsf{nat} \\
\textbf{and} & \alpha\ \mathsf{bexp} & = & \mathsf{Less}\ (\alpha\ \mathsf{aexp})\ (\alpha\ \mathsf{aexp}) \\
 & & \mid & \mathsf{And}\ (\alpha\ \mathsf{bexp})\ (\alpha\ \mathsf{bexp})
\end{array}
$$

This specification emits quite a lot of material into the current theory context, first of all injective functions $\mathsf{Sum} :: \alpha\ \mathsf{aexp} \to \alpha\ \mathsf{aexp} \to \alpha\ \mathsf{aexp}$ etc. for any of the datatype constructors. Each valid expression of our programming language is denoted by a well-typed constructor-term. Functions on inductive types are typically defined by primitive recursion. We now define evaluation functions for arithmetic and boolean expressions, depending on an environment $e :: \alpha \to \mathsf{nat}$.

$$\begin{array}{l}
\textbf{consts} \\
\quad \mathsf{evala} :: (\alpha \to \mathsf{nat}) \to \alpha\ \mathsf{aexp} \to \mathsf{nat} \\
\quad \mathsf{evalb} :: (\alpha \to \mathsf{nat}) \to \alpha\ \mathsf{bexp} \to \mathsf{bool} \\
\textbf{primrec} \\
\quad \mathsf{evala}\ e\ (\mathsf{If}\ b\ a_1\ a_2) = (\mathsf{if}\ \mathsf{evalb}\ e\ b\ \mathsf{then}\ \mathsf{evala}\ e\ a_1\ \mathsf{else}\ \mathsf{evala}\ e\ a_2) \\
\quad \mathsf{evala}\ e\ (\mathsf{Sum}\ a_1\ a_2) = \mathsf{evala}\ e\ a_1 + \mathsf{evala}\ e\ a_2 \\
\quad \mathsf{evala}\ e\ (\mathsf{Var}\ v) = e\ v \\
\quad \mathsf{evala}\ e\ (\mathsf{Num}\ n) = n \\
\quad \mathsf{evalb}\ e\ (\mathsf{Less}\ a_1\ a_2) = (\mathsf{evala}\ e\ a_1 < \mathsf{evala}\ e\ a_2) \\
\quad \mathsf{evalb}\ e\ (\mathsf{And}\ b_1\ b_2) = (\mathsf{evalb}\ e\ b_1 \wedge \mathsf{evalb}\ e\ b_2)
\end{array}$$

Similarly, we may define substitution functions for expressions. The mapping $s :: \alpha \to \alpha\ \mathsf{aexp}$ given as a parameter is lifted canonically on aexp and bexp.

$$\begin{array}{l}
\textbf{consts} \\
\quad \mathsf{substa} :: (\alpha \to \alpha\ \mathsf{aexp}) \to \alpha\ \mathsf{aexp} \to \alpha\ \mathsf{aexp} \\
\quad \mathsf{substb} :: (\alpha \to \alpha\ \mathsf{aexp}) \to \alpha\ \mathsf{bexp} \to \alpha\ \mathsf{bexp} \\
\textbf{primrec} \\
\quad \mathsf{substa}\ s\ (\mathsf{If}\ b\ a_1\ a_2) = \mathsf{If}\ (\mathsf{substb}\ s\ b)\ (\mathsf{substa}\ s\ a_1)\ (\mathsf{substa}\ s\ a_2) \\
\quad \mathsf{substa}\ s\ (\mathsf{Sum}\ a_1\ a_2) = \mathsf{Sum}\ (\mathsf{substa}\ s\ a_1)\ (\mathsf{substa}\ s\ a_2) \\
\quad \mathsf{substa}\ s\ (\mathsf{Var}\ v) = s\ v \\
\quad \mathsf{substa}\ s\ (\mathsf{Num}\ n) = \mathsf{Num}\ n \\
\quad \mathsf{substb}\ s\ (\mathsf{Less}\ a_1\ a_2) = \mathsf{Less}\ (\mathsf{substa}\ s\ a_1)\ (\mathsf{substa}\ s\ a_2) \\
\quad \mathsf{substb}\ s\ (\mathsf{And}\ b_1\ b_2) = \mathsf{And}\ (\mathsf{substb}\ s\ b_1)\ (\mathsf{substb}\ s\ b_2)
\end{array}$$

The relationship between substitution and evaluation can be expressed by:

$$\begin{array}{l}
\textbf{lemma} \\
\quad \mathsf{evala}\ e\ (\mathsf{substa}\ s\ a) = \mathsf{evala}\ (\lambda x.\ \mathsf{evala}\ e\ (s\ x))\ a\ \wedge \\
\quad \mathsf{evalb}\ e\ (\mathsf{substb}\ s\ b) = \mathsf{evalb}\ (\lambda x.\ \mathsf{evala}\ e\ (s\ x))\ b
\end{array}$$

We can prove this theorem by straightforward reasoning involving *mutual* structural induction on a and b, which is expressed by the following rule:

$$\frac{\begin{array}{lcl}
\forall b\ a_1\ a_2.\ Q\ b \wedge P\ a_1 \wedge P\ a_2 & \Longrightarrow & P\ (\mathsf{If}\ b\ a_1\ a_2) \\
 & \cdots & \\
\forall a_1\ a_2.\ P\ a_1 \wedge P\ a_2 & \Longrightarrow & Q\ (\mathsf{Less}\ a_1\ a_2) \\
 & \cdots &
\end{array}}{P\ a \wedge Q\ b}$$

As a slightly more advanced example we now consider the type $(\alpha, \beta, \gamma)\mathsf{tree}$, which is made arbitrarily branching by nesting an appropriate function type.

$$\textbf{datatype}\quad (\alpha, \beta, \gamma)\mathsf{tree} \;=\; \mathsf{Atom}\ \alpha \;\mid\; \mathsf{Branch}\ \beta\ (\gamma \to (\alpha, \beta, \gamma)\mathsf{tree})$$

Here α stands for leaf values, β for branch values, γ for subtree indexes. It is important to note that γ may be any type, including an infinite one such as nat; it need not even be a datatype. The induction rule for $(\alpha, \beta, \gamma)\mathsf{tree}$ is

$$\frac{\forall a.\ P\ (\mathsf{Atom}\ a) \quad \forall b\ f.\ (\forall x.\ P\ (f\ x)) \Longrightarrow P\ (\mathsf{Branch}\ b\ f)}{P\ t}$$

Note how we may assume that the predicate P holds for all values of f, all subtrees, in order to show P ($\mathsf{Branch}\ b\ f$). Using this induction rule, Isabelle/HOL automatically proves the existence of combinator tree-rec for primitive recursion:

$$\mathsf{tree\text{-}rec} :: (\alpha \to \delta) \to (\beta \to (\gamma \to (\alpha,\beta,\gamma)\mathsf{tree}) \to (\gamma \to \delta) \to \delta) \to (\alpha,\beta,\gamma)\mathsf{tree} \to \delta$$
$$\mathsf{tree\text{-}rec}\ f_1\ f_2\ (\mathsf{Atom}\ a) = f_1\ a$$
$$\mathsf{tree\text{-}rec}\ f_1\ f_2\ (\mathsf{Branch}\ b\ f) = f_2\ b\ f\ ((\mathsf{tree\text{-}rec}\ f_1\ f_2) \circ f)$$

In the case of Branch, the function $\mathsf{tree\text{-}rec}\ f_1\ f_2$ is recursively applied to all function values of f, i.e. to all subtrees. As an example primitive recursive function on type tree, consider the function member c which checks whether a tree contains some Atom c. It could be expressed as $\mathsf{tree\text{-}rec}\ (\lambda a.\ a = c)\ (\lambda b\ f\ f'.\ \exists x.\ f'\ x)$. Isabelle/HOL's **primrec** package provides a more accessible interface:

primrec
$\mathsf{member}\ c\ (\mathsf{Atom}\ a) = (a = c)$
$\mathsf{member}\ c\ (\mathsf{Branch}\ b\ f) = (\exists x.\ \mathsf{member}\ c\ (f\ x))$

3 Formal-Logic Preliminaries

3.1 The Logic of Choice?

This question is a rather subtle one. Actually, when it comes to real applications within a large system developed over several years, there is not much choice left about the underlying logic. Changing the very foundations of your world may be a very bad idea, if one cares for the existing base of libraries and applications.

HOL [5], stemming from Church's "Simple Theory of Types" [3] has proven a robust base over the years. Even if simplistic in some respects, HOL proved capable of many sophisticated constructions, sometimes even *because* of seeming weaknesses. For example, due to simple types HOL admits interesting concepts such as intra-logical overloading [24] or object-oriented features [13]. Our constructions for inductive types only require plain simply-typed set theory, though.

3.2 Isabelle/HOL — Simply-Typed Set Theory

The syntax of HOL is that of simply-typed λ-calculus. *Types* are either variables α, or applications $(\tau_1, \ldots, \tau_n)t$, including function types $\tau_1 \to \tau_2$ (right associative infix). *Terms* are either typed constants c_τ or variables x_τ, applications $(t\ u)$ or abstractions $\lambda x.t$. Terms have to be well-typed according to standard rules. *Theories* consist of a signature of types and constants, and axioms. Any theory induces a set of derivable theorems $\vdash \varphi$, depending on a fixed set of deduction rules that state several "obvious" facts of classical set theory. Starting from a minimalistic basis theory, all further concepts are developed *definitionally.*

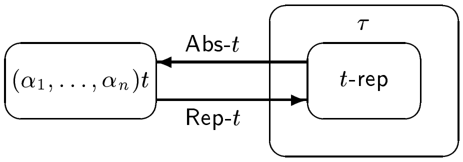
Isabelle/HOL provides many standard notions of classical set-theory. Sets are of type α set; infix f “ A refers to the image, vimage f A to the reverse image of f on A; inv f inverts a function; lfp F and gfp F are the least and greatest fixpoints of F on the powerset lattice. The sum type $\alpha + \beta$ has constructors Inl and Inr. Most other operations use standard mathematical notation.

3.3 Simple Definitions

The HOL methodology dictates that only *definitional* theory extension mechanisms may be used. HOL provides two primitive mechanisms: *constant definitions* and *type definitions* [5], further definitional packages are built on top.

- **Constant definition** We may add a new constant c to the signature and introduce an axiom of the form $\vdash c\ v_1\ \dots\ v_n \equiv t$, provided that c does not occur in t, $TV(t) \subseteq TV(c)$ and $FV(t) \subseteq \{v_1, \dots, v_n\}$.
- **Type definition** Let $t\text{-}\mathsf{rep}$ be a term of type τ set describing a non-empty set, i.e. $\vdash u \in t\text{-}\mathsf{rep}$ for some u. Moreover, require $TV(t\text{-}\mathsf{rep}) \subseteq \{\alpha_1, \dots, \alpha_n\}$. We may then add the type $(\alpha_1, \dots, \alpha_n)t$ and the following constants

$$
\begin{array}{lcl}
\mathsf{Abs}\text{-}t & :: & \tau \to (\alpha_1, \dots, \alpha_n)t \\
\mathsf{Rep}\text{-}t & :: & (\alpha_1, \dots, \alpha_n)t \to \tau
\end{array}
$$



to the signature and introduce the axioms

$$
\begin{array}{ll}
\vdash \mathsf{Abs}\text{-}t\ (\mathsf{Rep}\text{-}t\ x) = x & (\textit{Rep-t-inverse}) \\
\vdash y \in t\text{-}\mathsf{rep} \Longrightarrow \mathsf{Rep}\text{-}t\ (\mathsf{Abs}\text{-}t\ y) = y & (\textit{Abs-t-inverse}) \\
\vdash \mathsf{Rep}\text{-}t\ x \in t\text{-}\mathsf{rep} & (\textit{Rep-t})
\end{array}
$$

Type definitions are a slightly peculiar feature of HOL. The idea is to represent new types by subsets of already existing ones. The axioms above state that there is a bijection (isomorphism) between the set $t\text{-}\mathsf{rep}$ and the new type $(\alpha_1, \dots, \alpha_n)t$. This is justified by the standard set-theoretic semantics of HOL [5].

3.4 Inductive definitions

An **inductive** [17] definition specifies the *least* set closed under certain *introduction rules* — generally, there are many such closed sets. Essentially, an inductively defined set is the least fixpoint $\mathsf{lfp}\ F$ of a certain monotone function F, where $\mathsf{lfp}\ F = \bigcap\{x \mid F\ x \subseteq x\}$. The *Knaster-Tarski* theorem states that $\mathsf{lfp}\ F$ is indeed a fixpoint and that it is the least one, i.e. $F\ (\mathsf{lfp}\ F) = \mathsf{lfp}\ F$ and

$$
\frac{F\ P \subseteq P}{\mathsf{lfp}\ F \subseteq P} \qquad \frac{F\ (\mathsf{lfp}\ F \cap P) \subseteq P}{\mathsf{lfp}\ F \subseteq P}
$$

where P is the set of all elements satisfying a certain predicate. Both rules embody an induction principle for the set $\mathsf{lfp}\ F$. The second (stronger) rule is easily derived from the first one, because F is monotone. See [17] for more detailes on how to determine a suitable function F from a given set of introduction rules. When defining several *mutually inductive sets* $S_1, \dots, S_n$, one first builds the sum T of these and then extracts sets S_i from T with the help of the inverse image operator vimage, i.e. $S_i = \mathsf{vimage}\ \mathsf{in}_i\ T$, where in_i is a suitable injection.

4 Datatype Specifications

4.1 General Form

A general **datatype** specification in Isabelle/HOL is of the following form:

$$
\begin{array}{llcl}
\textbf{datatype} & (\alpha_1,\ldots,\alpha_h)t_1 & = & C_1^1\ \tau_{1,1}^1\ \ldots\ \tau_{1,m_1^1}^1\ \mid\ \ldots\ \mid\ C_{k_1}^1\ \tau_{k_1,1}^1\ \ldots\ \tau_{k_1,m_{k_1}^1}^1 \\
 & & \ldots & \\
\textbf{and} & (\alpha_1,\ldots,\alpha_h)t_n & = & C_1^n\ \tau_{1,1}^n\ \ldots\ \tau_{1,m_1^n}^n\ \mid\ \ldots\ \mid\ C_{k_n}^n\ \tau_{k_n,1}^n\ \ldots\ \tau_{k_n,m_{k_n}^n}^n
\end{array}
$$

where α_i are type variables, constructors C_i^j are distinct, and $\tau_{i,i'}^j$ are admissible types containing at most the type variables $\alpha_1,\ldots,\alpha_h$. Some type $\tau_{i,i'}^j$ occurring in such a specification is *admissible* iff $\{(\alpha_1,\ldots,\alpha_h)t_1,\ldots,(\alpha_1,\ldots,\alpha_h)t_n\} \Vdash \tau_{i,i'}^j$ where $\Vdash$ is inductively defined by the following rules:

- **non-recursive occurrences:** $\quad \Gamma \Vdash \tau$
  where τ is non-recursive, i.e. τ does not contain any of the newly defined type constructors $t_1,\ldots,t_n$
- **recursive occurrences:** $\quad \{\tau\} \cup \Gamma \Vdash \tau$
- **nested recursion involving function types:**
  $$\frac{\Gamma \Vdash \tau}{\Gamma \Vdash \sigma \to \tau} \qquad \text{where } \sigma \text{ is non-recursive}$$
- **nested recursion involving existing datatypes:**
  $$\frac{\begin{array}{lcl}\{(\tau_1',\ldots,\tau_{\tilde{h}}')\tilde{t}_1,\ldots,(\tau_1',\ldots,\tau_{\tilde{h}}')\tilde{t}_{\tilde{n}}\} \cup \Gamma & \Vdash & \tilde{\tau}_{1,1}^1\,[\tau_1'/\beta_1,\ldots,\tau_{\tilde{h}}'/\beta_{\tilde{h}}] \\ & \ldots & \\ \{(\tau_1',\ldots,\tau_{\tilde{h}}')\tilde{t}_1,\ldots,(\tau_1',\ldots,\tau_{\tilde{h}}')\tilde{t}_{\tilde{n}}\} \cup \Gamma & \Vdash & \tilde{\tau}_{\tilde{k}_{\tilde{n}},\tilde{m}_{\tilde{k}_{\tilde{n}}}^{\tilde{n}}}^{\tilde{n}}\,[\tau_1'/\beta_1,\ldots,\tau_{\tilde{h}}'/\beta_{\tilde{h}}]\end{array}}{\Gamma \Vdash (\tau_1',\ldots,\tau_{\tilde{h}}')\tilde{t}_{j'}}$$

  where $\tilde{t}_{j'}$ is the type constructor of an existing datatype specified by

  $$
  \begin{array}{llcl}
  \textbf{datatype} & (\beta_1,\ldots,\beta_{\tilde{h}})\tilde{t}_1 & = & D_1^1\ \tilde{\tau}_{1,1}^1\ \ldots\ \tilde{\tau}_{1,\tilde{m}_1^1}^1\ \mid\ldots\mid\ D_{\tilde{k}_1}^1\ \tilde{\tau}_{\tilde{k}_1,1}^1\ \ldots\ \tilde{\tau}_{\tilde{k}_1,\tilde{m}_{\tilde{k}_1}^1}^1 \\
   & & \ldots & \\
  \textbf{and} & (\beta_1,\ldots,\beta_{\tilde{h}})\tilde{t}_{\tilde{n}} & = & D_1^{\tilde{n}}\ \tilde{\tau}_{1,1}^{\tilde{n}}\ \ldots\ \tilde{\tau}_{1,\tilde{m}_1^{\tilde{n}}}^{\tilde{n}}\ \mid\ldots\mid\ D_{\tilde{k}_{\tilde{n}}}^{\tilde{n}}\ \tilde{\tau}_{\tilde{k}_{\tilde{n}},1}^{\tilde{n}}\ \ldots\ \tilde{\tau}_{\tilde{k}_{\tilde{n}},\tilde{m}_{\tilde{k}_{\tilde{n}}}^{\tilde{n}}}^{\tilde{n}}
  \end{array}
  $$

It is important to note that the admissibility relation $\Vdash$ is not defined within HOL, but as an extra-logical concept. Before attempting to construct a datatype, an ML function of the **datatype** package checks if the user input respects the rules described above. The point of this check is *not* to ensure correctness of the construction, but to provide high-level error messages.

Non-emptiness HOL does not admit empty types. Each of the new datatypes $(\alpha_1,\ldots,\alpha_h)t_j$ with $1 \le j \le n$ is guaranteed to be non-empty iff it has a constructor C_i^j with the following property: for all argument types $\tau_{i,i'}^j$ of the form $(\alpha_1,\ldots,\alpha_h)t_{j'}$ the datatype $(\alpha_1,\ldots,\alpha_h)t_{j'}$ is non-empty.

If there are no nested occurrences of the newly defined datatypes, obviously at least one of the newly defined datatypes $(\alpha_1,\ldots,\alpha_h)t_j$ must have a constructor

C_i^j without recursive arguments, a *base case*, to ensure that the new types are non-empty. If there are nested occurrences, a datatype can even be non-empty without having a base case itself. For example, with α list being a non-empty datatype, **datatype** t = C (t list) is non-empty as well.

Just like $\Vdash$ described above, non-emptiness of datatypes is checked by an ML function before invoking the actual HOL **typedef** primitive, which would never accept empty types in the first place, but report a low-level error.

4.2 Limitations of Set-theoretic Datatypes

Constructing datatypes in set-theory has some well-known limitations wrt. nesting of the *full* function space. This is reflected in the definition of admissible types given above. The last two cases of $\Vdash$ relate to *nested* (or *indirect*) occurrences of some of the newly defined types $(\alpha_1, \ldots, \alpha_h)t_{j'}$ in a type expression of the form $(\ldots, \ldots\ (\alpha_1, \ldots, \alpha_h)t_{j'}\ \ \ldots, \ldots)t'$, where t' may either be the type constructor of an already existing datatype or the type constructor $\rightarrow$ for the full function space. In the latter case, none of the newly defined types may occur in the first argument of the type constructor $\rightarrow$, i.e. all occurrences must be *strictly positive*. If we were to drop this restriction, the datatype could not be constructed (cf. [6]). Recall that in classical set-theory

- there is *no* injection of type $(\mathsf{t} \rightarrow \beta) \hookrightarrow \mathsf{t}$ according to *Cantor's theorem*, if β has more than one element;
- there *is* an injection $\mathsf{in}_1 :: (\mathsf{t} \rightarrow \beta) \hookrightarrow ((\alpha \rightarrow \mathsf{t}) \rightarrow \beta)$, because there is an injection $(\lambda c\ x.\ c) :: \mathsf{t} \hookrightarrow (\alpha \rightarrow \mathsf{t})$;
- there *is* an injection $\mathsf{in}_2 :: (\mathsf{t} \rightarrow \alpha) \hookrightarrow ((\mathsf{t} \rightarrow \alpha) \rightarrow \beta)$, if β has more than one element, since $(\lambda x\ y.\ x = y) :: (\mathsf{t} \rightarrow \alpha) \hookrightarrow ((\mathsf{t} \rightarrow \alpha) \rightarrow \mathsf{bool})$ is an injection and there is an injection $\mathsf{bool} \hookrightarrow \beta$.

Thus datatypes with any constructors of the following form

$$\textbf{datatype}\ \mathsf{t}\ =\ \mathsf{C}\ (\mathsf{t} \rightarrow \mathsf{bool})\ \mid\ \mathsf{D}\ ((\mathsf{bool} \rightarrow \mathsf{t}) \rightarrow \mathsf{bool})\ \mid\ \mathsf{E}\ ((\mathsf{t} \rightarrow \mathsf{bool}) \rightarrow \mathsf{bool})$$

cannot be constructed, because we would have injections C, $\mathsf{D} \circ \mathsf{in}_1$ and $\mathsf{E} \circ \mathsf{in}_2$ of type $(\mathsf{t} \rightarrow \mathsf{bool}) \hookrightarrow \mathsf{t}$, in contradiction to Cantor's theorem. In particular, inductive types in set-theory do *not* admit only weakly positive occurrences of nested function spaces. Moreover, nesting via datatypes exposes another subtle point when instantiating even *non-recursive* occurrences of function types: while **datatype** $(\alpha, \beta)\mathsf{t}\ =\ \mathsf{C}\ (\alpha \rightarrow \mathsf{bool})\ \mid\ \mathsf{D}\ (\beta\ \mathsf{list})$ is legal, the specification of **datatype** $\gamma\ \mathsf{u}\ =\ \mathsf{E}\ ((\gamma\ \mathsf{u}, \gamma)\mathsf{t})\ \mid\ \mathsf{F}$ is not, because it would yield the injection $\mathsf{E} \circ \mathsf{C} :: (\gamma\ \mathsf{u} \rightarrow \mathsf{bool}) \hookrightarrow \gamma\ \mathsf{u}$; **datatype** $\gamma\ \mathsf{u}\ =\ \mathsf{E}\ ((\gamma, \gamma\ \mathsf{u})\mathsf{t})\ \mid\ \mathsf{F}$ is again legal.

Recall that our notion of admissible datatype specifications is an extra-logical one — reflecting the way nesting is handled in the construction (see §5.4). In contrast, [23] *internalizes* nested datatypes into the logic, with the unexpected effect that even non-recursive function spaces have to be excluded.

The choice of internalizing vs. externalizing occurs very often when designing logical systems. In fact, an important aspect of formal-logic engineering is to get the overall arrangement of concepts at *different layers* done right. The notions of *deep* vs. *shallow embedding* can be seen as a special case of this principle.

5 Constructing Datatypes in HOL

We now discuss the construction of the class of inductive types given in §4.1. According to §3.3, new types are defined in HOL "semantically" by exhibiting a suitable representing set. Starting with a universe that is closed wrt. certain injective operations, we cut out the representing sets of datatypes inductively using *Knaster-Tarski* (cf. [17]). Thus having obtained free inductive types, we construct several derived concepts, in particular primitive recursion.

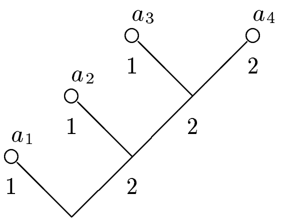
5.1 Universes for Representing Recursive Types

We describe the type $(\alpha, \beta)\mathsf{dtree}$ of trees, which is a variant of the universe formalized by Paulson [19], extended to support arbitrary branching. Type dtree provides the following operations:

$\mathsf{Leaf} :: \alpha \to (\alpha, \beta)\mathsf{dtree}$	embedding non-recursive occurrences of types
$\mathsf{In0}, \mathsf{In1} :: (\alpha, \beta)\mathsf{dtree} \to (\alpha, \beta)\mathsf{dtree}$	modeling distinct constructors
$\mathsf{Pair} :: (\alpha, \beta)\mathsf{dtree} \to (\alpha, \beta)\mathsf{dtree} \to (\alpha, \beta)\mathsf{dtree}$	modeling constructors with multiple arguments
$\mathsf{Lim} :: (\beta \to (\alpha, \beta)\mathsf{dtree}) \to (\alpha, \beta)\mathsf{dtree}$	embedding function types (infinitary products)

All operations are injective, e.g. $\mathsf{Pair}\ t_1\ t_2 = \mathsf{Pair}\ t'_1\ t'_2 \Longleftrightarrow t_1 = t'_1 \wedge t_2 = t'_2$ and $\mathsf{Lim}\ f = \mathsf{Lim}\ f' \Longleftrightarrow f = f'$. Furthermore, $\mathsf{In0}\ t \neq \mathsf{In1}\ t'$ for any t and t'.

Modeling Trees in HOL Set-theory A tree essentially is a set of *nodes*. Each node has a *value* and can be accessed via a unique *path*. A path can be modeled by a function that, given a certain *depth* index of type nat, returns a branching *label* (e.g. also nat). The figure below shows a finite tree and its representation.



$$T = \{(f_1, a_1), (f_2, a_2), (f_3, a_3), (f_4, a_4)\}$$

where

$$\begin{aligned} f_1 &= (1, 0, 0, \ldots) \\ f_2 &= (2, 1, 0, 0, \ldots) \\ f_3 &= (2, 2, 1, 0, 0, \ldots) \\ f_4 &= (2, 2, 2, 0, 0, \ldots) \end{aligned}$$

Here, a branching label of 0 indicates end-of-path. In the sequel, we will allow a node to have either a value of type bool or any type α. As branching labels, we will admit elements of type nat or any type β. Hence we define type abbreviations

$$\begin{aligned} (\alpha, \beta)\mathsf{node} &= (\mathsf{nat} \to (\beta + \mathsf{nat})) \times (\alpha + \mathsf{bool}) \\ (\alpha, \beta)\mathsf{dtree} &= ((\alpha, \beta)\mathsf{node})\mathsf{set} \end{aligned}$$

where the first component of a node represents the path and the second component represents its value. We can now define operations

$$
\begin{array}{lcl}
\mathsf{push} & :: & (\beta + \mathsf{nat}) \to (\alpha,\beta)\mathsf{node} \to (\alpha,\beta)\mathsf{node} \\
\mathsf{push}\ x\ n & \equiv & (\lambda i.\ (\mathsf{case}\ i\ \mathsf{of}\ 0 \Rightarrow x \mid \mathsf{Suc}\ j \Rightarrow \mathsf{fst}\ n\ j),\ \mathsf{snd}\ n) \\
\mathsf{Pair} & :: & (\alpha,\beta)\mathsf{dtree} \to (\alpha,\beta)\mathsf{dtree} \to (\alpha,\beta)\mathsf{dtree} \\
\mathsf{Pair}\ t_1\ t_2 & \equiv & (\mathsf{push}\ (\mathsf{Inr}\ 1)\ \text{``}\ t_1) \cup (\mathsf{push}\ (\mathsf{Inr}\ 2)\ \text{``}\ t_2)
\end{array}
$$

The function push adds a new head element to the path of a node, i.e.

$$\mathsf{push}\ x\ ((y_0, y_1, \ldots), a) = ((x, y_0, y_1, \ldots), a)$$

The function Pair joins two trees t_1 and t_2 by adding the distinct elements 1 and 2 to the paths of all nodes in t_1 and t_2, respectively, and then forming the union of the resulting sets of nodes. Furthermore, we define functions Leaf and Tag for constructing atomic trees of depth 0:

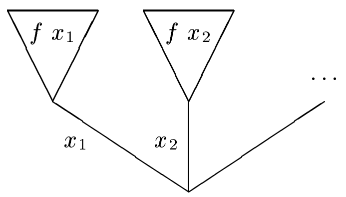
$$
\begin{array}{lcl}
\mathsf{Leaf} & :: & \alpha \to (\alpha,\beta)\mathsf{dtree} \\
\mathsf{Leaf}\ a & \equiv & \{(\lambda x.\ \mathsf{Inr}\ 0, \mathsf{Inl}\ a)\}
\end{array}
\qquad
\begin{array}{lcl}
\mathsf{Tag} & :: & \mathsf{bool} \to (\alpha,\beta)\mathsf{dtree} \\
\mathsf{Tag}\ b & \equiv & \{(\lambda x.\ \mathsf{Inr}\ 0, \mathsf{Inr}\ b)\}
\end{array}
$$

Basic set-theoretic reasoning shows that Pair, Leaf and Tag are indeed injective. We also define In0 and In1 which turn out to be injective and distinct.

$$
\begin{array}{lcl}
\mathsf{In0} & :: & (\alpha,\beta)\mathsf{dtree} \to (\alpha,\beta)\mathsf{dtree} \\
\mathsf{In0}\ t & \equiv & \mathsf{Pair}\ (\mathsf{Tag}\ \mathsf{false})\ t
\end{array}
\qquad
\begin{array}{lcl}
\mathsf{In1} & :: & (\alpha,\beta)\mathsf{dtree} \to (\alpha,\beta)\mathsf{dtree} \\
\mathsf{In1}\ t & \equiv & \mathsf{Pair}\ (\mathsf{Tag}\ \mathsf{true})\ t
\end{array}
$$

Functions (i.e. infinitary products) are embedded via Lim as follows:

$$
\begin{array}{lcl}
\mathsf{Lim} & :: & (\beta \to (\alpha,\beta)\mathsf{dtree}) \to (\alpha,\beta)\mathsf{dtree} \\
\mathsf{Lim}\ f & \equiv & \bigcup\{z \mid \exists x.\ z = \mathsf{push}\ (\mathsf{Inl}\ x)\ \text{``}\ (f\ x)\}
\end{array}
$$



That is, for all x the prefix x is added to the path of all nodes in $f\ x$, and the union of the resulting sets is formed.

Note that some elements of (α,β)dtree, such as trees with nodes of infinite depth, do not represent proper elements of datatypes. However, these junk elements are excluded when inductively defining the representing set of a datatype.

5.2 Constructing an Example Datatype

As a simple example, we will now describe the construction of the type α list, specified by **datatype** α list $=$ Nil $\mid$ Cons α (α list).

The Representing Set The datatype α list will be represented by the set list-rep :: $((\alpha, \mathsf{unit})\mathsf{dtree})\mathsf{set}$. Since α is the only type occurring non-recursively in the specification of list, the first argument of dtree is just α. If more types would occur non-recursively, the first argument would be the sum of these types. Since there is no nested recursion involving function types, the second argument of dtree is just the dummy type unit. We define $\mathsf{list\text{-}rep}$ inductively:

$$\frac{}{\mathsf{Nil\text{-}rep} \in \mathsf{list\text{-}rep}} \qquad \frac{ys \in \mathsf{list\text{-}rep}}{\mathsf{Cons\text{-}rep}\ y\ ys \in \mathsf{list\text{-}rep}} \qquad \begin{array}{lcl} \mathsf{Nil\text{-}rep} & \equiv & \mathsf{In0\ dummy} \\ \mathsf{Cons\text{-}rep}\ y\ ys & \equiv & \mathsf{In1}\ (\mathsf{Pair}\ (\mathsf{Leaf}\ y)\ ys) \end{array}$$

Constructors Invoking the type definition mechanism described in §3.3 introduces the abstraction and representation functions

$$\begin{array}{lcl} \mathsf{Abs\text{-}list} & :: & (\alpha, \mathsf{unit})\mathsf{dtree} \to \alpha\ \mathsf{list} \\ \mathsf{Rep\text{-}list} & :: & \alpha\ \mathsf{list} \to (\alpha, \mathsf{unit})\mathsf{dtree} \end{array}$$

as well as the axioms *Rep-list-inverse*, *Abs-list-inverse* and *Rep-list*. Using these functions, we can now define the constructors Nil and Cons:

$$\begin{array}{lcl} \mathsf{Nil} & \equiv & \mathsf{Abs\text{-}list\ Nil\text{-}rep} \\ \mathsf{Cons}\ x\ xs & \equiv & \mathsf{Abs\text{-}list}\ (\mathsf{Cons\text{-}rep}\ x\ (\mathsf{Rep\text{-}list}\ xs)) \end{array}$$

Freeness We can now prove that Nil and Cons are distinct and that Cons is injective, i.e. $\mathsf{Nil} \neq \mathsf{Cons}\ x\ xs$ and $\mathsf{Cons}\ x\ xs = \mathsf{Cons}\ x'\ xs' \Longleftrightarrow x = x' \wedge xs = xs'$. Because of the isomorphism between $\alpha\ \mathsf{list}$ and $\mathsf{list\text{-}rep}$, the former easily follows from the fact that $\mathsf{In0}$ and $\mathsf{In1}$ are distinct, while the latter is a consequence of the injectivity of $\mathsf{In0}$, $\mathsf{In1}$ and Pair.

Structural Induction For $\alpha\ \mathsf{list}$ an induction rule of the form

$$\frac{P\ \mathsf{Nil} \quad \forall x\ xs.\ P\ xs \Longrightarrow P\ (\mathsf{Cons}\ x\ xs)}{P\ xs}\ (\textit{list-ind})$$

can be proved using the induction rule

$$\frac{Q\ \mathsf{Nil\text{-}rep} \quad \forall y\ ys.\ Q\ ys \wedge ys \in \mathsf{list\text{-}rep} \Longrightarrow Q\ (\mathsf{Cons\text{-}rep}\ y\ ys)}{ys \in \mathsf{list\text{-}rep} \Longrightarrow Q\ ys}\ (\textit{list-rep-ind})$$

for the representing set $\mathsf{list\text{-}rep}$ derived by the inductive definition package from the rules described in §3.4. To prove *list-ind*, we show that $P\ xs$ can be deduced from the assumptions $P\ \mathsf{Nil}$ and $\forall x\ xs.\ P\ xs \Longrightarrow P\ (\mathsf{Cons}\ x\ xs)$ by the derivation

$$\cfrac{\cfrac{\ldots \quad \cfrac{\cfrac{\ldots \Longrightarrow P\ (\mathsf{Cons}\ y\ (\mathsf{Abs\text{-}list}\ ys))}{\ldots \Longrightarrow P\ (\mathsf{Abs\text{-}list}\ (\mathsf{Cons\text{-}rep}\ y\ (\mathsf{Rep\text{-}list}\ (\mathsf{Abs\text{-}list}\ ys))))}}{\forall y\ ys.\ P\ (\mathsf{Abs\text{-}list}\ ys) \wedge ys \in \mathsf{list\text{-}rep} \Longrightarrow P\ (\mathsf{Abs\text{-}list}\ (\mathsf{Cons\text{-}rep}\ y\ ys))}}{\mathsf{Rep\text{-}list}\ xs \in \mathsf{list\text{-}rep} \Longrightarrow P\ (\mathsf{Abs\text{-}list}\ (\mathsf{Rep\text{-}list}\ xs))}}{P\ xs}$$

Starting with the goal $P\ xs$, we first use the axioms *Rep-list-inverse* and *Rep-list*, introducing the local assumption $\mathsf{Rep\text{-}list}\ xs \in \mathsf{list\text{-}rep}$ and unfolding xs to $\mathsf{Abs\text{-}list}\ (\mathsf{Rep\text{-}list}\ xs)$. Now *list-rep-ind* can be applied, where Q and ys are instantiated with $P \circ \mathsf{Abs\text{-}list}$ and $\mathsf{Rep\text{-}list}\ xs$, respectively. This yields two new subgoals, one for the $\mathsf{Nil\text{-}rep}$ case and one for the $\mathsf{Cons\text{-}rep}$ case. We will only consider the $\mathsf{Cons\text{-}rep}$ case here: using axiom *Abs-list-inverse* together with the local assumption $ys \in \mathsf{list\text{-}rep}$, we unfold ys to $\mathsf{Rep\text{-}list}\ (\mathsf{Abs\text{-}list}\ ys)$. Applying the definition of Cons, we fold $\mathsf{Abs\text{-}list}\ (\mathsf{Cons\text{-}rep}\ y\ (\mathsf{Rep\text{-}list}\ (\mathsf{Abs\text{-}list}\ ys)))$ to

$\mathsf{Cons}\ y\ (\mathsf{Abs\text{-}list}\ ys)$. Obviously, $P\ (\mathsf{Cons}\ y\ (\mathsf{Abs\text{-}list}\ ys))$ follows from the local assumption $P\ (\mathsf{Abs\text{-}list}\ ys)$ using the assumption $\forall x\ xs.\ P\ xs \Longrightarrow P\ (\mathsf{Cons}\ x\ xs)$.

In principle, inductive types are already fully determined by freeness and structural induction. Applications demand additional derived concepts, of course, such as case analysis, size functions, and primitive recursion.

Primitive Recursion A function on lists is *primitive recursive* iff it can be expressed by a suitable instantiation of the recursion combinator

$$\begin{array}{l}
\mathsf{list\text{-}rec} :: \beta \to (\alpha \to \alpha\ \mathsf{list} \to \beta \to \beta) \to \alpha\ \mathsf{list} \to \beta \\
\mathsf{list\text{-}rec}\ f_1\ f_2\ \mathsf{Nil} = f_1 \\
\mathsf{list\text{-}rec}\ f_1\ f_2\ (\mathsf{Cons}\ x\ xs) = f_2\ x\ xs\ (\mathsf{list\text{-}rec}\ f_1\ f_2\ xs)
\end{array}$$

As has been pointed out in [8], a rather elegant way of constructing the function list-rec is to build up its graph list-rel by an inductive definition and then define list-rec in terms of list-rel using Hilbert's choice operator ε:

$$\frac{}{(\mathsf{Nil},\ f_1) \in \mathsf{list\text{-}rel}\ f_1\ f_2} \qquad \frac{(xs,\ y) \in \mathsf{list\text{-}rel}\ f_1\ f_2}{(\mathsf{Cons}\ x\ xs,\ f_2\ x\ xs\ y) \in \mathsf{list\text{-}rel}\ f_1\ f_2}$$

$$\mathsf{list\text{-}rec}\ f_1\ f_2\ xs \ \equiv\ \varepsilon y.\ (xs,\ y) \in \mathsf{list\text{-}rel}\ f_1\ f_2$$

To derive the characteristic equations for list-rec given above, we show that list-rel does indeed represent a total function, i.e. for every list xs there is a unique y such that $(xs,\ y) \in \mathsf{list\text{-}rel}\ f_1\ f_2$. The proof is by structural induction on xs.

5.3 Mutual Recursion

Mutually recursive datatypes, such as α aexp and α bexp introduced in §2 are treated quite similarly as above. Their representing sets aexp-rep and bexp-rep of type $((\alpha, \mathsf{unit})\mathsf{dtree})\mathsf{set}$ as well as the graphs aexp-rel and bexp-rel of the primitive recursion combinators are defined by *mutual* induction. For example, the rules for constructor Less are:

$$\frac{b_1 \in \mathsf{aexp\text{-}rep} \quad b_2 \in \mathsf{aexp\text{-}rep}}{\mathsf{In0}\ (\mathsf{Pair}\ b_1\ b_2) \in \mathsf{bexp\text{-}rep}} \qquad \frac{(x_1,\ y_1) \in \mathsf{aexp\text{-}rel}\ f_1 \ldots f_6 \quad (x_2,\ y_2) \in \mathsf{aexp\text{-}rel}\ f_1 \ldots f_6}{(\mathsf{Less}\ x_1\ x_2,\ f_5\ x_1\ x_2\ y_1\ y_2) \in \mathsf{bexp\text{-}rel}\ f_1 \ldots f_6}$$

5.4 Nested Recursion

Datatype $(\alpha, \beta)\mathsf{term}$ is a typical example for nested (or indirect) recursion:

$$\textbf{datatype}\ \ (\alpha, \beta)\mathsf{term} \ = \ \mathsf{Var}\ \alpha \ \mid \ \mathsf{App}\ \beta\ (((\alpha, \beta)\mathsf{term})\mathsf{list})$$

As pointed out in [6, 7], datatype specifications with *nested* recursion can conceptually be unfolded to equivalent *mutual* datatype specifications without nesting. We also follow this extra-logical approach, avoiding the complications of internalized nesting [23]. Unfolding the above specification would yield:

$$\begin{array}{lll}
\textbf{datatype} & (\alpha, \beta)\mathsf{term} & = \ \mathsf{Var}\ \alpha \ \mid \ \mathsf{App}\ \beta\ ((\alpha, \beta)\mathsf{term\text{-}list}) \\
\textbf{and} & (\alpha, \beta)\mathsf{term\text{-}list} & = \ \mathsf{Nil}' \ \mid \ \mathsf{Cons}'\ ((\alpha, \beta)\mathsf{term})\ ((\alpha, \beta)\mathsf{term\text{-}list})
\end{array}$$

However, it would be a bad idea to actually introduce the type $(\alpha, \beta)\mathsf{term\text{-}list}$ and the constructors Nil' and Cons', because this would prevent us from reusing common list lemmas in proofs about terms. Instead, we will prove that the representing set of $(\alpha, \beta)\mathsf{term\text{-}list}$ is isomorphic to the type $((\alpha, \beta)\mathsf{term})\mathsf{list}$.

The Representing Set We inductively define the sets $\mathsf{term\text{-}rep}$ and $\mathsf{term\text{-}list\text{-}rep}$ of type $((\alpha + \beta, \mathsf{unit})\mathsf{dtree})\mathsf{set}$ by the rules

$$\frac{}{\mathsf{In0}\ (\mathsf{Leaf}\ (\mathsf{Inl}\ a)) \in \mathsf{term\text{-}rep}} \qquad \frac{ts \in \mathsf{term\text{-}list\text{-}rep}}{\mathsf{In1}\ (\mathsf{Pair}\ (\mathsf{Leaf}\ (\mathsf{Inr}\ b))\ ts) \in \mathsf{term\text{-}rep}}$$

$$\frac{}{\mathsf{In0}\ \mathsf{dummy} \in \mathsf{term\text{-}list\text{-}rep}} \qquad \frac{t \in \mathsf{term\text{-}rep} \quad ts \in \mathsf{term\text{-}list\text{-}rep}}{\mathsf{In1}\ (\mathsf{Pair}\ t\ ts) \in \mathsf{term\text{-}list\text{-}rep}}$$

Since there are two types occurring non-recursively in the datatype specification, namely α and β, the first argument of dtree becomes $\alpha + \beta$.

Defining a Representation Function Invoking the type definition mechanism for term introduces the functions

$$\begin{array}{lll} \mathsf{Abs\text{-}term} & :: & (\alpha + \beta, \mathsf{unit})\mathsf{dtree} \to (\alpha, \beta)\mathsf{term} \\ \mathsf{Rep\text{-}term} & :: & (\alpha, \beta)\mathsf{term} \to (\alpha + \beta, \mathsf{unit})\mathsf{dtree} \end{array}$$

for abstracting elements of $\mathsf{term\text{-}rep}$ and for obtaining the representation of elements of $(\alpha, \beta)\mathsf{term}$. To get the representation of a list of terms we now define

$$\mathsf{Rep\text{-}term\text{-}list} :: ((\alpha, \beta)\mathsf{term})\mathsf{list} \to (\alpha + \beta, \mathsf{unit})\mathsf{dtree}$$
$$\mathsf{Rep\text{-}term\text{-}list}\ \mathsf{Nil} = \mathsf{In0}\ \mathsf{dummy}$$
$$\mathsf{Rep\text{-}term\text{-}list}\ (\mathsf{Cons}\ t\ ts) = \mathsf{In1}\ (\mathsf{Pair}\ (\mathsf{Rep\text{-}term}\ t)\ (\mathsf{Rep\text{-}term\text{-}list}\ ts))$$

Determining the representation of Nil is trivial. To get the representation of $\mathsf{Cons}\ t\ ts$, we need the representations of t and ts. The former can be obtained using the function $\mathsf{Rep\text{-}term}$ introduced above, while the latter is obtained by a recursive call of $\mathsf{Rep\text{-}term\text{-}list}$. Obviously, $\mathsf{Rep\text{-}term\text{-}list}$ is primitive recursive and can therefore be defined using the combinator $\mathsf{list\text{-}rec}$:

$$\begin{array}{lll} \mathsf{Rep\text{-}term\text{-}list} & \equiv & \mathsf{list\text{-}rec}\ (\mathsf{In0}\ \mathsf{dummy})\ (\lambda t\ ts\ y.\ \mathsf{In1}\ (\mathsf{Pair}\ (\mathsf{Rep\text{-}term}\ t)\ y)) \\ \mathsf{Abs\text{-}term\text{-}list} & \equiv & \mathsf{inv}\ \mathsf{Rep\text{-}term\text{-}list} \end{array}$$

It is a key observation that $\mathsf{Abs\text{-}term\text{-}list}$ and $\mathsf{Rep\text{-}term\text{-}list}$ have the properties

$$\mathsf{Abs\text{-}term\text{-}list}\ (\mathsf{Rep\text{-}term\text{-}list}\ xs) = xs$$
$$ys \in \mathsf{term\text{-}list\text{-}rep} \Longrightarrow \mathsf{Rep\text{-}term\text{-}list}\ (\mathsf{Abs\text{-}term\text{-}list}\ ys) = ys$$
$$\mathsf{Rep\text{-}term\text{-}list}\ xs \in \mathsf{term\text{-}list\text{-}rep}$$

i.e. $((\alpha, \beta)\mathsf{term})\mathsf{list}$ and $\mathsf{term\text{-}list\text{-}rep}$ are isomorphic, which can be proved by structural induction on list and by induction on $\mathsf{rep\text{-}term\text{-}list}$. Looking at the HOL type definition mechanism once again (§3.3), we notice that these properties have exactly the same form as the axioms which are introduced for actual newly defined types. Therefore, all of the following proofs are the same as in the case of mutual recursion without nesting, which simplifies matters considerably.

Constructors Finally, we can define the constructors for term:

$$\begin{array}{lll} \mathsf{Var}\ a & \equiv & \mathsf{Abs\text{-}term}\ (\mathsf{In0}\ (\mathsf{Leaf}\ (\mathsf{Inl}\ a))) \\ \mathsf{App}\ b\ ts & \equiv & \mathsf{Abs\text{-}term}\ (\mathsf{In1}\ (\mathsf{Pair}\ (\mathsf{Leaf}\ (\mathsf{Inr}\ b))\ (\mathsf{Rep\text{-}term\text{-}list}\ ts))) \end{array}$$

5.5 Infinitely Branching Types

We show how to construct infinitely branching types such as $(\alpha, \beta, \gamma)\mathsf{tree}$, cf. §2.

The Representing Set $\mathsf{tree\text{-}rep}$ will be of type $((\alpha + \beta, \gamma)\mathsf{dtree})\mathsf{set}$. Since the two types α and β occur non-recursively in the specification, the first argument of dtree is the sum $\alpha + \beta$ of these types. The only branching type, i.e. a type occurring on the left of some $\rightarrow$, is γ. Therefore, γ is the second argument of dtree. We define $\mathsf{tree\text{-}rep}$ inductively by the rules

$$\frac{}{\mathsf{In0}\ (\mathsf{Leaf}\ (\mathsf{Inl}\ a)) \in \mathsf{tree\text{-}rep}} \qquad \frac{g \in \mathsf{Funs}\ \mathsf{tree\text{-}rep}}{\mathsf{In1}\ (\mathsf{Pair}\ (\mathsf{Leaf}\ (\mathsf{Inr}\ b))\ (\mathsf{Lim}\ g)) \in \mathsf{tree\text{-}rep}}$$

where the premise $g \in \mathsf{Funs}\ \mathsf{tree\text{-}rep}$ means that all function values of g represent trees. The *monotone* function Funs is defined by

$$\begin{array}{lcl} \mathsf{Funs} & :: & \beta\ \mathsf{set} \rightarrow (\alpha \rightarrow \beta)\mathsf{set} \\ \mathsf{Funs}\ S & \equiv & \{g \mid \mathsf{range}\ g \subseteq S\} \end{array}$$

Constructors We define the constructors of tree by

$$\begin{array}{lcl} \mathsf{Atom}\ a & \equiv & \mathsf{Abs\text{-}tree}\ (\mathsf{In0}\ (\mathsf{Leaf}\ (\mathsf{Inl}\ a))) \\ \mathsf{Branch}\ b\ f & \equiv & \mathsf{Abs\text{-}tree}\ (\mathsf{In1}\ (\mathsf{Pair}\ (\mathsf{Leaf}\ (\mathsf{Inr}\ b))\ (\mathsf{Lim}\ (\mathsf{Rep\text{-}tree} \circ f)))) \end{array}$$

The definition of Atom is straightforward. To form a Branch from element b and subtrees denoted by f, we first determine the representation of the subtrees by composing f with $\mathsf{Rep\text{-}tree}$ and then represent the resulting function using Lim.

Structural Induction The induction rule for type tree shown in §2 can be derived from the corresponding induction rule for the representing set $\mathsf{tree\text{-}rep}$ by instantiating Q and u with $P \circ \mathsf{Abs\text{-}tree}$ and $\mathsf{Rep\text{-}tree}$, respectively:

$$\frac{\begin{array}{l} \forall a.\ Q\ (\mathsf{In0}\ (\mathsf{Leaf}\ (\mathsf{Inl}\ a))) \\ \forall b\ g.\ g \in \mathsf{Funs}\ (\mathsf{tree\text{-}rep} \cap \{x \mid Q\ x\}) \Longrightarrow Q\ (\mathsf{In1}\ (\mathsf{Pair}\ (\mathsf{Leaf}\ (\mathsf{Inr}\ b))\ (\mathsf{Lim}\ g))) \end{array}}{u \in \mathsf{tree\text{-}rep} \Longrightarrow Q\ u}$$

The unfold/fold proof technique already seen in §5.2 can also be extended to functions: if $g \in \mathsf{Funs}\ \mathsf{tree\text{-}rep}$, then $g = \mathsf{Rep\text{-}tree} \circ (\mathsf{Abs\text{-}tree} \circ g)$.

Primitive Recursion Again, we define the $\mathsf{tree\text{-}rec}$ combinator given in §2 by constructing its graph $\mathsf{tree\text{-}rel}$ inductively:

$$\frac{}{(\mathsf{Atom}\ a,\ f_1\ a) \in \mathsf{tree\text{-}rel}\ f_1\ f_2} \qquad \frac{f' \in \mathsf{compose}\ f\ (\mathsf{tree\text{-}rel}\ f_1\ f_2)}{(\mathsf{Branch}\ b\ f,\ f_2\ b\ f\ f') \in \mathsf{tree\text{-}rel}\ f_1\ f_2}$$

The *monotone* function $\mathsf{compose}$ used in the second rule is defined by

$$\begin{array}{lcl} \mathsf{compose} & :: & (\alpha \rightarrow \beta) \rightarrow (\beta \times \gamma)\mathsf{set} \rightarrow (\alpha \rightarrow \gamma)\mathsf{set} \\ \mathsf{compose}\ f\ R & \equiv & \{f' \mid \forall x.\ (f\ x,\ f'\ x) \in R\} \end{array}$$

The set $\mathsf{compose}\ f\ R$ consists of all functions that can be obtained by composing the function f with the relation R. Since R may not necessarily represent a total function, $\mathsf{compose}\ f\ R$ can also be empty or contain more than one function.

However, if for every x there is a unique y such that $(f\ x,\ y) \in \mathsf{tree\text{-}rel}\ f_1\ f_2$, then there is a unique f' with $f' \in \mathsf{compose}\ f\ (\mathsf{tree\text{-}rel}\ f_1\ f_2)$. This is the key property used to prove that $\mathsf{tree\text{-}rel}\ f_1\ f_2$ represents a total function.

Even More Complex Function Types The construction described above can be made slightly more general. Assume we want to define the datatype t, whose **datatype** specification contains function types $\sigma_1^i \to \cdots \to \sigma_{m_i}^i \to \mathsf{t}$, where $1 \leq i \leq n$. The representing set $\mathsf{t\text{-}rep}$ then has the type

$$((\ldots,\ (\sigma_1^1 \times \cdots \times \sigma_{m_1}^1) + \cdots + (\sigma_1^n \times \cdots \times \sigma_{m_n}^n))\mathsf{dtree})\mathsf{set}$$

The representation of a function $f_i\ ::\ \sigma_1^i \to \cdots \to \sigma_{m_i}^i \to \mathsf{t}$ is

$$\begin{array}{l}\mathsf{Lim}\ (\mathsf{sum\text{-}case} \\ \quad \underbrace{\mathsf{dummy}\ \ldots\ \mathsf{dummy}}_{i-1\text{ times}}\ (\mathsf{Rep\text{-}t} \circ ((\underbrace{\mathsf{uncurry} \circ \cdots \circ \mathsf{uncurry}}_{m_i-1\text{ times}})\ f_i))\ \underbrace{\mathsf{dummy}\ \ldots\ \mathsf{dummy}}_{n-i\text{ times}})\end{array}$$

where

$$\begin{array}{lll}\mathsf{uncurry} & :: & (\alpha \to \beta \to \gamma) \to \alpha \times \beta \to \gamma \\ \mathsf{sum\text{-}case} & :: & (\alpha_1 \to \beta) \to \cdots \to (\alpha_n \to \beta) \to \alpha_1 + \cdots + \alpha_n \to \beta\end{array}$$

are injective, i.e. $\mathsf{uncurry}\ f = \mathsf{uncurry}\ g$ iff $f = g$, and $\mathsf{sum\text{-}case}\ f_1\ \ldots\ f_n = \mathsf{sum\text{-}case}\ g_1\ \ldots\ g_n$ iff $f_1 = g_1 \wedge \ldots \wedge f_n = g_n$.

6 Building a Working Environment

6.1 Inverting Datatype Definitions

The hierarchy of definitional packages, as illustrated in §1, and the dependency of auxiliary theories used for the construction are often in conflict. For example, we have used basic types such as nat, $\alpha + \beta$, $\alpha \times \beta$ for the universe underlying datatypes. Any of these could be characterized as inductive types, but had to be built "manually", because **datatype** had not yet been available at that stage.

Of course, we would like to keep the Isabelle/HOL standard library free of any such accidental arrangements due to *bootstrap problems* of the HOL logic. Note that with most types being actual datatypes — offering the standard repertoire of derived concepts such as induction, recursion, pattern matching by cases etc. — the resulting system would become conceptually much simpler, with less special cases. Furthermore, proper datatypes may again partake in indirect recursion. Users certainly expect to be able to nest types via $\alpha + \beta$ and $\alpha \times \beta$.

We propose *inverted datatype definitions* as an answer to the above problem. Given a type together with freeness and induction theorems, the **rep-datatype** mechanism figures out the set of constructors and does all the rest of standard datatype constructions automatically. Thus we avoid cycles in theory/package dependencies in a clean way. Note that the same mechanism is used internally when unwinding indirect recursion (reconsider term-list vs. term list in §5.4).

From a purely logical point of view one would probably approach bootstrap issues differently. For example, [8] provides a very careful development of the theory underlying the datatype construction, reducing the requirements to a bare minimum, even avoiding natural numbers. Interestingly the actual implementation does not fully follow this scheme, but *does* use natural numbers.

6.2 Cooperation of Definitional Packages

As depicted in §1, some Isabelle/HOL packages are built on top of each other. For example, **record** [13] constructs extensible records by defining a separate (non-recursive) datatype for any record field. Other packages such as **recdef** [21, 22] refer to certain information about datatypes which are involved in well-founded recursion (e.g. size functions). We see that some provisions have to be made in order to support *cooperation* of definitional packages properly.

In particular, there should be means to store auxiliary information in theories. Then packages such as **datatype** would associate sufficient source level information with any type, such as the set of constructors, induction rule, and primrec combinator. Thus we get a more robust and scalable system than by trying to weed through the primitive logical declarations emitted by the package. Isabelle98-1 does already support an appropriate "theory data" concept.[2]

With extra-logical information associated with logical objects, we may also offer users a more uniform view to certain general principles. For example, "proof by induction" or "case analysis" may be applied to some x, with the actual tactic figured out internally. Also note that by deriving definitional mechanisms from others, such as **record** from **datatype**, these operations are *inherited*. Thus case analysis etc. on record fields would become the same as on plain datatypes.

7 Conclusion and Related Work

We have discussed Isabelle/HOL's new definitional packages for inductive types and (primitive) recursive functions (see also [2]) at two different levels of concept.

At the logical level, we have reviewed a set-theoretic construction of mutual, nested, arbitrarily branching types together with primitive recursion combinators. Starting with a schematic universe of trees — similar to [19], but extended to support infinitely branching — we have cut out representing sets for inductive types using the usual Knaster-Tarski fixed-point approach [17, 8].

Stepping back from the pure logic a bit, we have also discussed further issues we considered important to achieve a scalable and robust working environment. While this certainly does not yet constitute a systematic discipline of "Formal-Logic Engineering", we argue that it is an important line to follow in order to provide theorem proving systems that are "successful" at a larger scale. With a slightly different focus, [1] discusses approaches to "Proof Engineering".

The importance of advanced definitional mechanisms for applications has already been observed many years ago. Melham [12] pioneers a HOL datatype

[2] Interestingly, while admitting arbitrary ML values to be stored, this mechanism can be made *type-safe* within ML (see also [11]).

package (without nesting), extended later by Gunter [6, 7] to more general branching. Paulson [17] establishes Knaster-Tarski as the primary principle underlying (co)inductive types; the implementation in Isabelle/ZF set-theory also supports infinite branching. Völker [23] proposes a version of datatypes for Isabelle/HOL with nested recursion *internalized* into the logic, resulting in some unexpected restrictions of *non-recursive* occurrences of function spaces. Harrison [8] undertakes a very careful logical development of mutual datatypes based on cardinality reasoning, aiming to reduce the auxiliary theory requirements to a minimum. The implementation (HOL Light) has recently acquired nesting, too.

Our Isabelle/HOL packages for **datatype** and **primrec** have been carefully designed to support a superset of functionality, both with respect to the purely logical virtues and as its integration into a scalable system. This is intended not as the end of the story, but the beginning of the next stage.

Codatatypes would follow from Knaster-Tarski by duality quite naturally (e.g. [17]), as long as simple cases are considered. Nesting codatatypes, or even mixing datatypes and codatatypes in a useful way is very difficult. While [10] proposes a way of doing this, it is unclear how the informal categorical reasoning is to be transferred into the formal set-theory of HOL (or even ZF).

Non-freely generated types would indeed be very useful if made available for nesting. Typical applications refer to some type that contains a finitary environment of itself. Currently this is usually approximated by nesting $(\alpha \times \beta)$ list.

Actual *combination of definitional* packages would be another important step towards more sophisticated standards, as are established in functional language compiler technology, for example. While we have already achieved decent cooperation of packages that are built on top of each other, there is still a significant gap towards arbitrary combination (mutual and nested use) of separate packages. In current Haskell compilers, for example, any module (read "theory") may consist of arbitrary declarations of classes, types, functions etc. all of which may be referred to mutually recursive. Obviously, theorem prover technology will still need some time to reach that level, taking into account that "compilation" means actual theorem proving work to be provided by the definitional packages.

References

[1] H. P. Barendregt. The quest for correctness. In *Images of SMC Research*, pages 39–58. Stichting Mathematisch Centrum, Amsterdam, 1996.

[2] S. Berghofer. Definitorische Konstruktion induktiver Datentypen in Isabelle/HOL (in German). Master's thesis, Technische Universität München, 1998.

[3] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, pages 56–68, 1940.

[4] C. Cornes, J. Courant, J.-C. Filliâtre, G. Huet, P. Manoury, and C. Muñoz. *The Coq Proof Assistant User's Guide, version 6.1.* INRIA-Rocquencourt et CNRS-ENS Lyon, 1996.

[5] M. J. C. Gordon and T. F. Melham (editors). *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[6] E. L. Gunter. Why we can't have SML style `datatype` declarations in HOL. In L. J. M. Claesen and M. J. C. Gordon, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume A-20 of *IFIP Transactions*, pages 561–568. North-Holland Press, 1992.

[7] E. L. Gunter. A broader class of trees for recursive type definitions for HOL. In J. Joyce and C. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *LNCS*, pages 141–154. Springer, 1994.

[8] J. Harrison. Inductive definitions: automation and application. In P. J. Windley, T. Schubert, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications: Proceedings of the 8th International Workshop*, volume 971 of *LNCS*, pages 200–213, Aspen Grove, Utah, 1995. Springer.

[9] J. Harrison. HOL done right. Unpublished draft, 1996.

[10] U. Hensel and B. Jacobs. Proof principles for datatypes with iterated recursion. In E. Moggi and G. Rosolini, editors, *Category Theory and Computer Science*, volume 1290 of *LNCS*, pages 220–241. Springer, 1997.

[11] F. Kammüller and M. Wenzel. Locales — a sectioning concept for Isabelle. Technical Report 449, University of Cambridge, Computer Laboratory, October 1998.

[12] T. F. Melham. Automating recursive type definitions in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer, 1989.

[13] W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics*, volume 1479 of *LNCS*. Springer, 1998.

[14] T. Nipkow and D. von Oheimb. Java$_{light}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*. ACM Press, New York, 1998.

[15] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *LNCS*. Springer, 1996.

[16] S. Owre and N. Shankar. Abstract datatypes in PVS. Technical Report CSL-93-9R, Computer Science Laboratory, SRI International, 1993.

[17] L. C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, volume 814 of *LNAI*, pages 148–161, Nancy, France, 1994. Springer.

[18] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

[19] L. C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Journal of Logic and Computation*, 7(2):175–204, 1997.

[20] F. Pfenning and C. Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *Proceedings of Mathematical Foundations of Programming Semantics*, volume 442 of *LNCS*. Springer, 1990.

[21] K. Slind. Function definition in higher order logic. In J. Wright, J. Grundy, and J. Harrison, editors, *9th International Conference on Theorem Proving in Higher Order Logics, TPHOLs'96*, volume 1125 of *LNCS*. Springer, 1996.

[22] K. Slind. Derivation and use of induction schemes in higher-order logic. In *10th International Conference on Theorem Proving in Higher Order Logics, TPHOLs'97*, volume 1275 of *LNCS*. Springer, 1997.

[23] N. Völker. On the representation of datatypes in Isabelle/HOL. In L. C. Paulson, editor, *First Isabelle Users Workshop*, 1995.

[24] M. Wenzel. Type classes and overloading in higher-order logic. In *10th International Conference on Theorem Proving in Higher Order Logics, TPHOLs'97*, volume 1275 of *LNCS*. Springer, 1997.