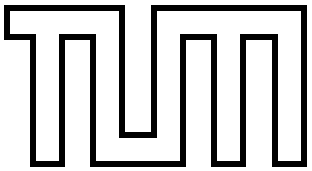


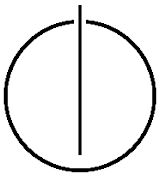
FAKULTÄT FÜR INFORMATIK

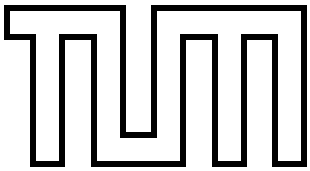
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit in Informatik

# Instantiating deeply embedded many-sorted theories into HOL types in Isabelle

Andreas Schropp





FAKULTÄT FÜR INFORMATIK

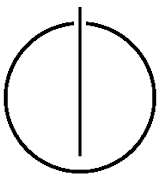
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit in Informatik

# Instantiating deeply embedded many-sorted theories into HOL types in Isabelle

# Instantiierung tief eingebetteter mehrsortiger Theorien mit HOL Typen in Isabelle

| | |
|---|---|
| Supervisor: | Prof. Tobias Nipkow, Ph.D. |
| Advisor: | Dr. Andrei Popescu |
| Submission Date: | March 14, 2012 |



I assure the single handed composition of this diploma thesis only supported by declared resources.

Munich, March 14, 2012 Andreas Schropp

# Acknowledgments

I would like to thank my advisor Andrei Popescu for coming up with the meta-theory of non-freely generated datatypes and the paradigm of package construction in HOL. It nicely plays to both of our interests. He has been a constant source of wisdom and encouragement. His comments improved draft versions of this thesis.

Lively reflections on reflection in theorem proving with Armin Heller made me realize the unrealized potential in these methods early on.

Discussions with Alex Krauss started my quest for soft-types in Isabelle/ZF, which led me to the animation of algorithmic rule systems. Tobias Nipkow introduced me to rewriting and his theorem proving group. That experience formed my symbolic understanding of computer science.

# Abstract

The development of tools for theorem proving still needs in-depth knowledge of the programming interface of the theorem prover. A large class of packages can be understood as HOL theories parameterized on many-sorted signatures, so we try to improve the situation by providing a paradigm for formulating such meta-theories and infrastructure for animating them. We illustrate how a certain class of algorithmic rule systems and rules describing the generation of theory content can be executed. Transfer of theory content under setoid isomorphisms is a requirement in the realization of our paradigm, so we employ the infrastructure in the design of this tool. We apply the paradigm, infrastructure and transfer under setoid isomorphisms in the construction of a package for non-freely generated datatypes.

# Contents

# 1 Introduction

Tool construction in theorem provers has not improved much since the introduction of ML and relies firmly on an elite of expert developers. Tool construction done by users, without in-depth knowledge of the internals of the theorem prover, is almost unheard of.

Packages are a class of tools that are in widespread use. They encapsulate a principled construction mechanism that produces theory content from a specification by the user. Packages shape the way a theorem prover is used in a very direct way: the constructions they implement are regarded as new standardized primitives that do not have to be simulated by a user. Packages for the definition of inductive predicates [22], datatypes [4] and functions [15] realize these as derived concepts in Isabelle, yet they are at the heart of computer science and many provers regard them as foundational principles (e.g., Coq [25]).

This makes us interested in how to describe the general construction mechanism behind a package, without using the programming interface of a theorem prover. The informal meta-theories underlying many packages provide the necessary intuition: interpreting syntactic signatures and stating meta-theorems about that interpretation.

Some logics feature the concept of a universe which is closed under all (small) constructions that are possible in the logic. Packages can then be interpreted as follows:

- Inputs are interpretations of signatures over the universe.
- The construction exhibits certain elements of the universe with properties that are based on the signature.

Universes are not available to us in Isabelle/HOL, but this does not stop us from making use of that insight. In most cases only finitely many and syntactically apparent types are relevant for the construction inside a package, so we will just embed those types into a sum type and use this sum type instead of a proper universe.

We can now formulate packages as meta-theories parameterized on signatures and their interpretation in a pseudo-universe. They become ordinary Isabelle locales [2]. Animating these locales as packages is our main concern and we provide generic infrastructure to execute theorems that represent algorithmic rule systems or generate of theory content.

## 1.1 Contributions

We make the following contributions:

- A paradigm to regard a large class of HOL packages as meta-theories parameterized on a many-sorted signature.

- Infrastructure to animate these meta-theories as packages. We provide an execution strategy for certain algorithmic rule systems and rules that generate theory content in small steps.
- We describe an algorithmic rule system that realizes the transfer of theory content under setoid isomorphisms, as required by our paradigm. In particular we show how currying of functions on finite products can be implemented as a custom higher-order transformation.
- We apply the paradigm, infrastructure and transfer under setoid isomorphisms in the construction of a package for non-freely generated datatypes. We sketch its meta-theory and animation as a package.

## 1.2 Outline

This thesis is structured as follows:

**Chapter 2** provides a short introduction to Isabelle and its object logic Isabelle/HOL. We also introduce some terminology.

**Chapter 3** introduces our paradigm of regarding packages as meta-theories parameterized on a many-sorted signature.

**Chapter 4** shows how certain algorithmic rule systems are defined with "meta recursion clauses" and executed in Isabelle.

**Chapter 5** introduces the concept of a forward rule to describe the generation of theory content in well-defined small steps.

**Chapter 6** explains the necessity of transfer of theory content under setoid isomorphisms and how this is realized.

**Chapter 7** briefly outlines the meta-theory of our application of non-freely generated datatypes and sketches its animation to become a package.

# 2 Introduction to Isabelle

## 2.1 Isabelle/Pure

Isabelle [23] is a generic theorem prover based on the logical framework Isabelle/Pure, which is a minimal higher-order logic with schematic polymorphism and extensional equality. Isabelle/Pure features intuitionistic implication $\Longrightarrow$ and universal quantification $\bigwedge$. Implications $P_1 \Longrightarrow \ldots \Longrightarrow P_n \Longrightarrow P$ associate to the right and are written as

$$\frac{P_1 \qquad \ldots \qquad P_n}{P}$$

when they are proven theorems. The typing judgement is written as $t :: \tau$. Type variables are written as $'id$.

Isabelle features unification variables as a primitive term construct, written as $?x$. They correspond to schematically quantified variables, in particular in implicational rules and proof states. Their prime use is easy instantiation by unification in proof search, especially while applying Isabelle's basic reasoning method higher-order resolution [23]. While we will not use resolution, we sometimes use unification variables as placeholders for which a term has yet to be found. Terms without unification variables are called *ground*. In rules we drop the question mark for reasons of readability. The unification variables are then all identifiers of the rule that are not constants or bounds.

Isabelle/Pure features the following foundational theory extension mechanisms:

**Constant definitions** allow the introduction of a constant with an axiom expressing the equality of the constant to a closed term in the empty context. By careful context-management and lambda-lifting this notion of constant definition is simulated in local theory contexts.

Note very well that constant unfolding is an explicit rewriting with the definition axiom, which is in sharp contrast to $\delta$-reductions in popular type-theory based theorem provers such as Coq [25]. This allows the use of constant definitions as an **intensional abstraction mechanism**. We will make prominent use of that later.

**Storage of named theorems,** also called "noting", corresponds to the definition of proof constants in the theory.

## 2.2 Isabelle/ML

Isabelle is implemented in Standard ML and provides a programming framework referred to as Isabelle/ML. In particular it includes primitives for the management of variable and assumption contexts, tactics based on higher-order resolution [23] and a higher-order

rewriting engine, called the simplifier [19]. Isabelle follows the LCF approach [8]: derivable theorems are represented as an abstract ML datatype and are created by axioms, trusted inference rules and their composition.

## 2.3 Isabelle/HOL

We concentrate on the object logic Isabelle/HOL [20] for the rest of this thesis. Isabelle/HOL is directly based on the minimal HOL provided by Isabelle/Pure, adding further axioms to provide a type $bool$ of classical truth values, which is embedded into intuitionistic propositions. In addition to the foundational theory extensional mechanisms Isabelle/HOL contains the following:

**Type definitions** allow the introduction of a new type constructor, whose application to type arguments is in bijection to a nonempty *representing set* (over the type arguments) in the empty context. The basic theory development for such a new type is carried out by "lifting" terms and theorems using the representing set to the new type, by composing with the bijection and its inverse.

**Type class definitions and instances** Primitives for introducing type classes and their instances. We will not make use of them, so refer the interested reader elsewhere [9]. Actually these primitives for type classes may be counted as part of Isabelle/Pure, but we list them here because without the ability to introduce new types they are of limited use.

## 2.4 Terminology and Notation

*Packages* are principled construction mechanisms for types and terms that satisfy certain theorems. *Definitional packages* define the types and terms by applying foundational theory extension mechanisms and go on to prove the theorems about them, instead of just postulating the types, terms and theorems after checking the input for well-formedness. Most packages of Isabelle/HOL are definitional packages. Examples include packages for the definition of inductive predicates [22], datatypes [4] and functions [15].

We refer to theorems, constant definitions, type definitions and their localized versions in a context of fixed variables and assumptions as *theory content*. In particular we like to regard the bijection arising from a type definition as an isomorphism from theory content about the representing set to theory content about the new type.

We use $\overrightarrow{t_m}$ as the meta-syntactic abbreviation for the sequence $t_1 \ldots t_m$. Substitution of $t_2$ for $x$ in $t_1$ is written as $t_1[x := t_2]$.

## 2.5 Basic Concepts

HOL features a polymorphic $undefined$ constant to allow explicit under specification in terms. It inhabits every type and fulfills those properties that hold for an arbitrary variable of that type, e.g., $undefined = undefined$.

HOL sets are predicates $A :: 'a \Rightarrow bool$, where we write the application as membership $x \in A$. We will often use the HOL set function space

$$A \to B := \{f \mid \forall x \in A.\ fx \in B\},$$

calling $A$ the *domain* and $B$ the *codomain*. We also need abstractions forming elements of this function space, which are just normal abstractions annotated with the domain to ease the synthesis of function spaces:

$$(\lambda\, x \in A.\ t) := (\lambda\, x.\ t).$$

Datatypes are types generated by a number of constructors. This allows induction and recursion over them by treating each constructor.

Lists in HOL are the datatype with the usual polymorphic constructors $Cons$ and $Nil$. Common functions on lists include $map$, forming a new list by applying a function on each element of the old list, and $set$, converting lists to sets.

We will use sum types, which correspond to a datatype $'a + 'b$ with constructors $Inl :: 'a \Rightarrow 'a + 'b$, $Inr :: 'b \Rightarrow 'a + 'b$.

# 3 Definitional Packages as Meta-Theories in HOL

The packages we are interested in take as input a finite number of HOL types and terms satisfying certain compatibility conditions, and produce as output other HOL types and terms, together with theorems about them. But how can we formulate meta-theories with corresponding input/output behaviour in HOL, if they have to abstract over an arbitrary number of HOL types and terms?

## 3.1 Embedding Relevant HOL Types into Pseudo-Universes

The concept of a type-theoretic *universe*, i.e., a type of (small) types, would allow an easy parameterization on arbitrary types and terms. But HOL lacks such a concept. Weaker forms of this concept, such as first-class quantification over types and type constructors are a familiar limitation of HOL, which has led to proposals of extensions [12].

By analysing the use cases of such a universe of types in constructions relevant for definitional packages, one comes to realize that the full power of universes is often not needed. Instead, the construction can be carried out using a sum type, into which the types relevant for the construction are embedded. We call such a sum type a *pseudo-universe*.

Functions between the types relevant for a construction are similarly mapped to functions between subsets of the pseudo-universe. It is now possible to regard the relevant types and functions as an interpretation of a many-sorted signature over the pseudo-universe, with sorts and operations of the signature as the syntactic concepts for HOL types and functions respectively. This allows the application of meta-theories that are parameterized on certain many-sorted signatures, to HOL types and functions. Likewise the output of such a meta-theory consists of nonempty subsets of the construction universe, which are mapped to HOL types, functions on the construction universe, which are mapped to HOL constants, and theorems about them which are transferred accordingly.

Our use of pseudo-universes in the application is actually a bit more refined: we only use them to collect input types. Constructions then use separate construction types that are parameterized on the pseudo-universe for the input types. This could be recast as the principled use that we outlined above, by including the construction types in the pseudo-universe, but that does not seem to have a practical value. The principled view of pseudo-universes still provides nice insights into many HOL packages and how they can be regarded as meta-theories.

## 3.2 Example

We show a simple example of a meta-theory which is parameterized on a finite number of constants of arbitrary types.

The signature of the meta-theory has to contain the following:

**Type variables** A pseudo-universe $'uni$ to embed the types of the constants, a type of syntactic sorts $'sort$ and a type of constant symbols $'csym$.

**Term variables** Constants are assigned sorts by a function $const\_sort :: 'csym \Rightarrow 'sort$. Sorts are a syntactic representation of the subsets of the pseudo-universe corresponding to the types of constants. The function $model :: 'sort \Rightarrow 'uni\ set$ realizes this relationship. Constant symbols are interpreted by the function $const :: 'csym \Rightarrow 'uni$, as the embedding of constants into the pseudo-universe.

**Assumptions** We assume that the interpretations of constant symbols lie in the subsets of the pseudo-universe corresponding to the sorting of constants.

So we arrive at the following locale:

$$
\begin{array}{l}
\textbf{locale} \\
\quad \textbf{fixes} \quad 'uni,\ 'sort,\ 'csym \\
\qquad\qquad const\_sort\ ::\ 'csym \Rightarrow 'sort \\
\qquad\qquad model\ ::\ 'sort \Rightarrow 'uni\ set \\
\qquad\qquad const\ ::\ 'csym \Rightarrow 'uni \\
\quad \textbf{assumes} \quad const\ c \in model(const\_sort\ c) \\
\textbf{begin} \\
\qquad \ldots \\
\textbf{end}
\end{array}
$$

If we want to apply this meta-theory to constants $c_1 :: \tau_1, c_2 :: \tau_2$, we form the following instantiation

$$
\begin{array}{rcl}
'uni & := & \tau_1 + \tau_2 \\
\textbf{datatype}\ 'sort & := & Sort1 \mid Sort2 \\
\textbf{datatype}\ 'csym & := & Const1 \mid Const2 \\
\\
const\_sort\ Const1 & := & Sort1 \\
const\_sort\ Const2 & := & Sort2 \\
\\
const\ Const1 & := & Inl\ c_1 \\
const\ Const2 & := & Inr\ c_2 \\
\\
model\ Sort1 & := & \{Inl\ x \mid x :: \tau_1\} \\
model\ Sort2 & := & \{Inr\ x \mid x :: \tau_2\}
\end{array}
$$

and discharge the conditions

$$const\ Const1 \in model\ (const\_sort\ Const1),$$
$$const\ Const2 \in model\ (const\_sort\ Const2).$$

The transformation of a meta-theorem parameterized on this signature to a theorem about concrete constants then needs to use the embeddings of the types into the pseudo-universe. We will consider their inverses here:

$$\begin{aligned} f_1 : model\ Sort1 &\simeq_\triangleright \tau_1 \\ Inl\ x &\mapsto x \\ f_2 : model\ Sort2 &\simeq_\triangleright \tau_2 \\ Inr\ x &\mapsto x \end{aligned}$$

Let us see how the transformation of the trivial meta-theorem

$$const\ c = const\ c$$

plays out under our previous instantiation.

First we vary the constant symbol $c$ and arrive at the theorems

$$const\ Const1 = const\ Const1$$
$$const\ Const2 = const\ Const2.$$

We then use the correspondence

$$f_1\ (const\ Const1) = c_1$$
$$f_2\ (const\ Const2) = c_2$$

to transform our variations of the meta-theorem into

$$c_1 = c_1$$
$$c_2 = c_2.$$

We will delve into the details of such transformations in section 6.5.

## 3.3 Applicability

We do not know the precise conditions under which this paradigm is applicable to HOL package construction. The main challenges are:

- Finding notions of syntactic signature and their interpretation, which capture the use cases of a package.
- Identifying the types relevant for a construction.
- Finding criteria which characterize valid signatures and their interpretations. Discharging them with the infrastructure is much easier if they are of a syntactic nature.

- Formulating a meta-theory which provides the outputs as constructions and theorems based on the signature.

Meta-theories become much easier if they are based on first-order terms, so applying this paradigm successfully will often be based on restricting the use of higher-order concepts in inputs.

Let us give an example of a package where application of this paradigm would be very hard or impossible. The informal meta-theory of a new HOL package [26] for compositional (co)datatypes is based on the notion of a rich type constructor. If we want to apply our paradigm, we face the challenge that the meta-theory naturally makes use of a proper universe of types. Perhaps the application of our paradigm is possible, if we can regard the types relevant for an invocation as bounded type constructor iterations over a finite collection of base types. If we can syntactically determine the base types and a bound on the number of type constructor iterations that a given invocation of the package needs, then it might be possible to form the collection of relevant types and embed them into a pseudo-universe. Even if this is possible, it would be much harder than our envisioned use here. The meta-theory would have to reason about types formed by bounded type constructor iteration and boundedness assumptions have to be discharged automatically when animating the meta-theorems. In logics which feature proper universes these problems do not occur and an adaptation of our paradigm might be very beneficial. The signatures will be more complicated though.

# 4 Algorithmic Rule Systems as Meta Recursions

*Algorithmic rule systems* are collections of rules about relational judgements, which act on (higher-order) terms as data and have distinguished input/output positions, where inputs determine outputs. They are a frequent companion in the theorem proving literature and practice. Prominent examples include type checking, type inference, reflection and evaluation. This makes algorithmic rule systems interesting as an alternative to ML programming. The desire to make use of them is not new and has been explored to some degree with tactics and the Lambda Prolog programming language.

We focus on algorithmic rule systems that work on ground Isabelle terms and whose rules can be interpreted as Isabelle theorems. Judgements are modeled as defined constants of propositional type, together with an annotation specifying their *mode*, i.e., declaring a fixed designation of input/output argument positions. We are not only interested in the derivation of output terms for given input terms, but also in the derivation of their semantic relation, which is expressed in the proposition that the judgement applied to input and output terms holds. The rules are Isabelle theorems w.r.t. this semantics of judgements and they respect the moding, i.e., if a judgement is applied to terms we have to ensure that the input terms will be ground at execution time. The current implementation does not try to guarantee termination or coverage of certain subclasses of terms.

This idea of algorithmic rule systems corresponds to finding an **intensional and algorithmic presentation for a semantic concept**, with rules about a judgement. Intensional in the sense that the rules work on the syntactic structure of terms, without analysing their semantics, in particular without unfolding judgement definitions. The definition of the judgement plays the part of the semantic concept. The rules play the algorithmic part, i.e., they correspond to recursion clauses, albeit using judgement applications $J\ \overrightarrow{t_n}\ \overrightarrow{t'_m}$ instead of equalities $f\ \overrightarrow{t_n} = \overrightarrow{t'_m}$. Because these recursion clauses are in general not expressible as equations on function symbols in mainstream logics, we call these algorithmic rule systems *meta recursions*. Note very well that the definitions of judgements do not have to be inductive definitions, and the rules do not have to be of a form which would allow an inductive definition. Also the rules do not have to be a complete characterization of the semantic concept represented by the judgement.

## 4.1 Notation and Informal Execution Semantics

We assume that the input arguments of a judgement precede the output arguments, so that judgement application always have the form $J\ \overrightarrow{t_{i,n_i}}\ \overrightarrow{t_{o,n_o}}$ with input arguments $\overrightarrow{t_{i,n_i}}$ and output arguments $\overrightarrow{t_{o,n_o}}$. Because we will later introduce another kind of rule for the generation of theory content, and to emphasize that meta recursion clauses are marked

theorems that are executed from conclusions to premises, we write meta recursion clauses as

$$\uparrow \frac{P_1 \quad \ldots \quad P_m}{J\ \overrightarrow{t_{i,n_i}}\ \overrightarrow{t_{o,n_o}}}.$$

The premises $P_m$ are of the form $\bigwedge \overrightarrow{x_k}.\ \overrightarrow{A_l} \implies J'\ \overrightarrow{t'_{i,n'_i}}\ \overrightarrow{t'_{o,n'_o}}$.

We now give a simplified informal execution semantics for meta recursion rules. When solving a judgement $J$ with inputs $t_{i,n_i}$, we select a meta recursion clause, where $J$ is the judgement in the conclusion and the $t_{i,n_i}$ match against the input arguments in the conclusion of the clause. Then we solve the premises recursively from left to right in the resulting instantiation, where output patterns in the premises can further refine the instantiation. Premises of the form $\bigwedge \overrightarrow{x_k}.\ \overrightarrow{A_l} \implies J'\ \overrightarrow{t'_{i,n'_i}}\ \overrightarrow{t'_{o,n'_o}}$ recursively solve $J'$ on instantiated inputs $\overrightarrow{t'_{i,n'_i}}$ in an extension of the current context by fresh variables $\overrightarrow{x_k}$ and instantiated local meta recursion clauses $\overrightarrow{A_l}$. The found outputs are matched against the $\overrightarrow{t'_{o,n'_o}}$. After subderivations for all premises have been established, we compose them with the meta recursion clause in final instantiation to arrive at a theorem stating the instantiated conclusion of the meta recursion clause. The outputs are the output arguments of this judgement application, together with the theorem.

## 4.2 Example

Let us illustrate these ideas with an example. We might be interested in finding a "canonical" [1] set of which a given term is an element, calling this set the *soft-type* of the term. Because we cannot express "canonicity" easily in Isabelle, we reduce the semantic concept to just membership and regard "canonicity" as an implicit property of the studied rule system. Given the classical nature of Isabelle/HOL, a set containing a given term can be found in a myriad of ways, and certain choices, such as $t \in \{t\}$ are always possible. To achieve "canonicity" we rather take the insight from popular type systems and simulate the way they would synthesize such a set, i.e., by a deterministic intensional analysis of the term, descending into its syntactic structure while managing contexts. We now give a simple example, simulating the way the simply typed lambda calculus synthesizes types for terms, but modelling the concept of types using HOL's sets.

The judgement of interest is defined as $t \rhd A := t \in A$, with the first position as input and the second position as output. We will use HOL's set function space and abstractions annotated with their domain sets.

The rules are easy theorems about our judgement:

$$\uparrow \frac{t_1 \rhd (A \to B) \qquad t_2 \rhd A}{(t_1\ t_2) \rhd B} \qquad \uparrow \frac{\bigwedge x.\ x \rhd A \implies t\ x \rhd B}{(\lambda\, x \in A.\ t\ x) \rhd (A \to B)}$$

Note that all identifiers in these rules, except for the bound variable $x$, are actually unification variables where we have dropped the question mark. While illustrating the animation

[1] e.g., predictable by a user and with good compositional properties

of these rules we will use unification variables to make the synthesis of information more clear.

The animation of the rule for application works as follows: given an application $e_1\ e_2$, we match $e_1\ e_2$ against $?t_1\ ?t_2$, resulting in $?t_1 := e_1,\ ?t_2 := e2$. Next we solve the judgement $e_1 \rhd ?A \to ?B$ recursively, i.e., synthesize the soft-type of $e_1$ and match it against $?A \to ?B$, resulting in ground instantiations of $?A$ and $?B$, e.g., $?A := A,\ ?B := B$, together with a theorem stating $e_1 \rhd A \to B$. Then we synthesize the soft-type of $t_2$ and match it against $A$, resulting in a theorem $t_2 \rhd A$. Finally we compose the theorems for the premises with the rule and produce B as the soft-type of $e_1\ e_2$, together with the theorem stating $e_1\ e_2 \rhd B$.

The animation of the rule for abstractions works as follows: given an abstraction $(\lambda\, x \in A.\ e)$, where $x$ may occur in $e$ and we assume that bound variables are distinctly named, we match $(\lambda\, y.\ e[x := y])$ against $?t$. Now we extend the current context with a fresh variable $z$, an assumption and local rule $z \rhd A$ and solve the judgement $e[x := z] \rhd ?B$ in this new context. This results in the synthesis of a soft-type $B$ for $e[x := z]$ and a theorem stating $e[x := z] \rhd B$. We transport this theorem to the old context, by discharging the assumption $z \rhd A$ and quantifying over $z$. This results in the theorem $\bigwedge z.\ z \rhd A \implies e[x := z] \rhd B$, which we compose with the rule for abstractions. We arrive at the theorem $(\lambda\, x \in A.\ e) \rhd A \to B$, so $A \to B$ is the synthesized soft-type of $(\lambda\, x \in A.\ e)$.

## 4.3 Technicalities

We use a specialized combination of higher-order pattern matching modulo $\beta$ and first-order matching on non-patterns. Why not modulo $\beta\eta$ as usual? We want to regard abstractions as an explicit term construct, which is not invented on-the-fly, to prevent non-termination of rules such as

$$\uparrow \frac{\bigwedge x.\quad t\ x\ \mathsf{rewritesto}\ t'\ x}{(\lambda\, x.\ t\ x)\ \mathsf{rewritesto}\ (\lambda\, x.\ t'\ x)}.$$

Case distinctions with negation-as-failure are realized with a local backtracking primitive and a customizable prioritization of overlapping meta recursion clauses.

## 4.4 Related Work

Our execution model for meta recursion clauses corresponds to a determinization of Lambda Prolog [18], with specialized pattern matching replacing unification, explicitly moded judgements with a semantics instead of rigid atoms and local backtracking instead of global backtracking with a cut primitive.

Twelf [24] is a proof system in which recursion on HOAS is presented in a form very similar to our meta recursions. In particular recursions implementing proof transformations can be checked for totality to allow their interpretation as inductions [10]. We do not extend Isabelle in that direction and rather animate meta recursions only on ground terms, each time anew. It is interesting to note that using the semantics of the judgement resulting from a successful meta recursion corresponds in Twelf to the use of a theorem that is proven by abstractly mapping derivations of the rule system to this semantics. Twelf's

primary concern is the HOAS-based formalization of calculi, while our aim is to enhance reasoning in Isabelle by animating intensional analysis techniques in the form of algorithmic rule systems.

Lukas Bulwahn's predicate compiler [7] compiles inductive definitions into functional equations, making them executable via code generation. Most of the algorithmic rule systems we care about cannot constitute inductive definitions, because the terms they act on do not form a HOL type, but rather form subclasses of the universe of HOL types closed under the treated term construction principles. We want to use them to enhance reasoning, i.e., animate the composition of rules to derive interesting theorems, not just execute them syntactically. This animation takes places on higher-order data, namely Isabelle terms, and requires context-management for bound variables and local rules. Also note that we are fundamentally interested in algorithmic rules systems about deterministic judgements with a fixed moding, while he cares about nondeterministic predicates. Our implementation does not compile meta recursions, but this becomes interesting with their large scale application.

Isabelle features backward resolution [23] which uses higher-order unification. Why don't we just use a depth-first search on goals based on this notion of resolution? The meta recursion clauses we are interested in often feature non-patterns, which can easily result in non-termination of such a search strategy. For example the conclusion of the rule for soft-typing applications

$$\uparrow \frac{t_1 \rhd (A \to B) \qquad t_2 \rhd A}{(t_1\ t_2) \rhd B}$$

unifies with any goal of the form $t \rhd C$, e.g., with $t_1 := (\lambda x.\ x), t_2 := t, B := C$. Furthermore our specialized pattern matching, rule prioritization and local backtracking make execution fully deterministic, which allows for predictability of results, optimizations and better error messages. We can provide generic "foo expected but this is bar" error messages, comparable to errors from type inference engines of various programming languages, instead of just reporting failure of tactics.

Isabelle also features a simplifier [19] which can often solve type checking related goals, if they are ground, by rewriting them to $True$. But the simplifier can synthesize information only with rewriting or the help of unsafe solvers. For example it can solve the goal

$$f \in A \to B \implies x \in A \implies f\ x \in B,$$

by relying on an unsafe solver which resolves the subgoal $f \in ?A \to B$ with the premise $f \in A \to B$, resulting in $?A := A$. This does not work when the assumptions are instead registered as simplification rules. The architecture of Isabelle's simplifier has been a major inspiration for meta recursion. Furthermore note that the simplifier implements a bottom-up execution of small-step rewriting rules, while algorithmic rule systems are executed in a big-step top-down fashion.

# 5 Generation of Theory Content with Forward Rules

We want to declaratively specify the generation of theory content. This task becomes easier if we do not look at large monolithic theory extension mechanisms, but at small steps achieving this result. Forward rules are the abstract representation of these small steps. They wait for input information, process it with algorithmic rule systems, issue foundational theory extension mechanism as side effects, and generate output information.

## 5.1 Informal Semantics of Forward Rules

We realize that the inputs and outputs of forward rules have to get a semantics in Isabelle, in order to discharge assumptions while generating theory content. These pieces of information are modelled as *facts*, i.e., ground judgement applications and are propagated between rules that generate theory content. Because we want to generate new theory content and adapt algorithmic rule systems to it, we realize that the rules doing this may need to generate meta recursion clauses. We now introduce the format of the rules we use to generate theory content: *forward rules* are marked implicational theorems (possibly containing schematic variables),

$$\downarrow \frac{H_1 \wedge \ldots \wedge H_n \qquad P_1 \qquad \ldots \qquad P_n}{F_1 \wedge \ldots \wedge F_n \wedge R_1 \wedge \ldots \wedge R_n}$$

with patterns $H_1, \ldots, H_n$ matching input facts conjunctively combined in the head premise $H_1 \wedge \ldots \wedge H_n$, meta recursion subcalls and foundational theory extension mechanisms as the other premises $P_1, \ldots, P_n$, and generated facts $F_1, \ldots, F_n$ and meta recursion clauses $R_1, \ldots R_n$ conjunctively combined in the conclusion. A simplified informal semantics for such a forward rule is as follows: if a tuple of facts matches the tuple of heads $(H_1, \ldots, H_n)$, we solve the other premises $P_1, \ldots, P_n$ in the resulting instantiation and in this order (with matches on output positions of judgements in the $P_i$ refining the instantiation), then emit facts $F_1, \ldots, F_n$ and meta recursion clauses $R_1, \ldots, R_n$.

Theory extension mechanisms have a monotonic nature w.r.t. theory content, so we choose a *global accumulative pool* as the propagation mechanism for facts:

- facts declared by a user or facts issued by another tool form the initial pool of facts
- we begin applying forward rules by matching facts from the pool against the patterns in the head premise of the forward rule. When the application has been successful, we insert the facts in the conclusion (of the applied forward rules) into the pool, making them available for further forward rule applications.

- duplicate facts in the pool are absorbed
- forward rules are only applied once to the same input facts
- we iterate the application of forward rules on the pool of facts until no new facts are generated (by any forward rule)

We take caution that the sequences of forward rule applications that may be chosen in execution are equivalent. This corresponds to a confluence-like property. For forward rules which only combine facts, this follows from the set-like nature of the pool. If forward rules additionally employ foundational theory transformations, e.g., definitions, this follows because their well-formed orderings — those that respect the data dependencies of types and terms — cannot be distinguished. For forward rules that use meta recursion subcalls, we have to be more careful: meta recursion clauses generated by forward rules may influence the execution of meta recursion subcalls in other forward rules. We have to restrict the orders in which forward rules may execute, to respect the following principle: a forward rule executes only if all judgements which are potentially used in derivations of meta recursion subcalls are not modified later on. To ensure this we construct a dependency graph of forward rules and judgements, by analysing judgement dependencies and judgement modifications in forward rules and meta recursion clauses. This dependency graph is checked for the absence of strongly-connected components which make execution sequences respecting this principle impossible. Then we saturate the strongly-connected components in topological order.

## 5.2 Basic Theory Extension Mechanisms

We provide support for the following basic theory extension mechanisms:

**Foundational theory extensions** Constant and type definitions, storage of named theorems.

**Derived theory extensions** Enumerative data types (i.e., sums of empty products) and functions on them.

## 5.3 Execution is Explicit and Localized

Execution of forward rules is triggered explicitly, thereby communicating that all initial facts necessary for the execution should now be available. Because facts are ground terms, it is often necessary to localize execution of forward rules into a context with fixed variables. To this end we employ Isabelle locales [2] as a sectioning mechanism with implicit export of results, generalizing them over the locally fixed variables. This localized execution of forward rule is also beneficial in their design: they can pretend that the current construction is the only one of its kind, so facts do not have to contain markers signifying to which construction they belong.

## 5.4 Example of Forward-Chaining

We start with a basic example illustrating the forward-chaining nature of forward rules. Our application will be concerned with establishing inhabitedness of the sets of first-order terms of a certain sort, given by a predicate $inhab$ on sorts $s$.

The sorted first-order terms are formed by well-sorted operation applications, i.e., if $c : \overrightarrow{s_n} \to s$ is a sorted operation in the signature and we are given first-order terms $\overrightarrow{t_n}$ of sort $\overrightarrow{s_n}$, then the application $c\ \overrightarrow{t_n}$ is of sort $s$. Note that vectors of sorts are represented in Isabelle as lists of sorts, written $ss$.

If sorts $\overrightarrow{s_n}$ are inhabited and a sorted operation $c : \overrightarrow{s_n} \to s$ is available, we can conclude that the sort $s$ is inhabited, namely by the application of $c$ to the inhabitants of $\overrightarrow{s_n}$. This is an algorithmic scheme in the meta-variable $n$, so if we want to express it with forward rules about a judgement in Isabelle, we have to find a step-wise formulation. This suggests reflecting the partially-discharged implications stating "if these sorts are inhabited, then this sort is inhabited". We introduce a judgement for the satisfaction of inhabitedness implications

$$inhabimplies\ ss\ s\ :=\ (\forall s_2 \in set\ ss.\ inhab\ s_2) \longrightarrow inhab\ s,$$

where we use the $set$ function to extensionally convert the list of sorts to a set of sorts. We employ a forward rule to further eliminate implications:

$$\downarrow \frac{inhab\ s \quad \wedge \quad inhabimplies\ (Cons\ s\ ss)\ s_2}{inhabimplies\ ss\ s_2}.$$

This states: if sort $s$ is inhabited and the inhabitedness of $s$ and $ss$ implies inhabitedness of $s_2$, then inhabitedness of $ss$ implies inhabitedness of $s_2$. We further need a rule to extract results from fully-discharged inhabitedness implications

$$\downarrow \frac{inhabimplies\ Nil\ s}{inhab\ s}$$

and a rule to introduce inhabitedness implications from sortings of operations

$$\downarrow \frac{c : ss \to s}{inhabimplies\ ss\ s}.$$

These rules are now executed as follows (but note that we do not care about the exact order of steps):

- the initial pool consists of facts representing the sortings of operations
- these sortings generate corresponding inhabitedness implication facts
- fully-discharged inhabitedness implication facts become inhabitedness facts
- inhabitedness facts together with inhabitedness implication facts with matching first implication, generate inhabitedness implication facts without the first implication

- we always reach a fixpoint because there are only finitely many sorts and operation sortings, and inhabitedness implication facts become smaller in the process.
- the result, i.e., theorems stating inhabitedness of the inhabited sorts, can be extracted from the pool by considering the facts of the form $inhab\ s$.

## 5.5 Example of Theory Content Generation

Let us now take a look at a forward rule, which realizes the lifting of a constant under an isomorphism between sets. In our case an isomorphism is not just a single bijection, but a marker $\phi$, which selects a subset of bijection forming meta recursion clauses. But the details are not important for now and section 6.2 will shed more light on this. We use the mixfix syntax

$$\phi:\ A \simeq_{\triangleright} A' \text{ via } f \quad := \quad A' \text{ is the image of set } A \text{ under the bijection } f$$

for the judgement stating the transfer of sets under isomorphisms, with inputs $\phi, A$ and outputs $A', f$. And we use the mixfix syntax

$$\phi:\ t \in A \mapsto t' \in A' \text{ via } f \quad := \\ t' \text{ is the image of } t \text{ under the bijection } f \text{ between } A \text{ and } A'$$

for the judgement stating the transfer of elements under isomorphisms, with inputs $\phi, t$ and outputs $A, t', A', f$. Since we want to define a new constant, we use the corresponding foundational theory extension mechanism, which is triggered by judgement

$$\text{define } c' \text{ as } t \quad := \quad c' = t$$

with input $t$ and output $c'$. To trigger the forward rule, we use a logically vacuous judgement collecting the relevant input data:

$$liftalong\ c\ \phi\ :=\ True$$

The presence of a fact with this judgement can be interpreted as a subroutine call to forward rules with a matching head. The outputs will be meta recursion clauses, so we rely on the ordering of forward rule applications w.r.t. their dependencies to communicate the outputs implicitly. Concretely this means that forward rules employing transfers under isomorphisms are executed after forward rules that generate meta recursion clauses that might be needed in those transfers.

Our example forward rule is the following:

$$\downarrow \frac{liftalong\ c\ \phi \qquad c \rhd A \qquad \phi:\ A \simeq_{\triangleright} A' \text{ via } f \qquad \text{define } c' \text{ as } f\ c}{c' \rhd A' \qquad \wedge \qquad \phi:\ c \in A \mapsto c' \in A' \text{ via } f}$$

This rule waits for facts of the form $liftalong\ \ c\ \ \phi$, then employs meta-recursion to synthesize the soft-type of $c$, namely $A$, and forms the isomorphic image $A'$ of $A$ under the isomorphism $\phi$ via the concrete bijection $f$. Then it defines a new constant $c'$ as $fc$. Extensionally speaking, i.e., unfolding the definition of judgements $\rhd, \_ :\ \_ \simeq_{\triangleright} \_ \text{ via } \_, \text{define}$,

we have at this point established that $c \in A$, $f$ is a bijection between $A$ and $A'$, $c' = f\ c$ and will now go on using this and the easy consequence $c' \rhd A'$. Then the forward rule generates meta-recursion clauses without premises, stating that the synthesized soft-type of $c'$ is $A'$ and that the transfer of $c$ under $\phi$ is $c'$, via the concrete bijection $f$ between $A$ and $A'$.

You might ask yourself why this forward rule generates meta recursion clauses and not just facts. The point here is that we want to state elementary pieces of algorithmics, which will be used by larger algorithmic rule systems, and not just pieces of information that can be extracted from a pool. E.g. if we later on invoke a meta recursion for the synthesis of the soft-type of $g\ c'$, we will need the clause $c' \rhd A'$ to synthesize the soft-type of $c'$. With all of that being said, the abuse of notation using facts as meta recursion clauses without premises is just too tempting, so our implementation supports this.

## 5.6 Related Work

Isabelle's Locales [2] propagate theory content under substitutions along paths of "development graphs". Because the animation of the meta-theory of our application requires transformations under elaborate algorithmic rule systems, we use locales mainly as a sectioning mechanism.

Logic translations (e.g., [21, 16]) and the AWE extension [6] transform theory content by reinterpreting proofs. The focus here is on a uniform transformation that works for all theory content that can be expressed in the source logic or signature. This means that the results are often unnatural encodings. Some translations try to play tricks to mitigate this, such as easy replacements of theory content while replaying the proofs. Transforming produced theory content under isomorphisms is a more general and principled mechanism to achieve this and we will discuss it in section 6.5. In principle we could animate reinterpretations of proofs as algorithmic rule systems, and use forward rules to resolve the dependencies between proofs, terms and types. But without a compiled implementation of meta recursion this will not result in practical tools.

# 6 Transfer of Theory Content under Setoid Isomorphisms

## 6.1 Motivation

Let us first discuss why we are interested in the transfer of theory content under isomorphisms and why transfer between isomorphic HOL types is insufficient for our purpose.

HOL types and sets are in a very close relationship: new type constructors can only be introduced as bijective images of nonempty subsets of existing types, with the typedef primitive. The basic theory development for such a new types is carried out by "lifting" terms and theorems using the representing set to the new type, by composing with the bijection and its inverse. If we want to introduce new types we have to make use of this relationship, so naturally we have to consider the transfer between isomorphic sets. Also we have to form pseudo-universes, which is a form of transfer under isomorphisms, namely embedding existing HOL types into a sum type and mapping functions between them to functions on the sum type.

Unfortunately the natural HOL set function space is not well-behaved under bijections: because of the underspecification outside the domain $A$, a strong extensionality principle

$$(\forall x \in A.\ f\ x = g\ x) \implies f = g$$

is not available in this setting. But this would be needed to prove that the construction $h \mapsto g \circ h \circ f^{-1}$ is a bijection between function spaces $A \to B$, $A' \to B'$, if $f$ is a bijection from $A$ to $A'$ and $g$ a bijection from $B$ to $B'$.

The (implicit) partial set function space additionally demands that its functions are equal to $\mathit{undefined}$ outside of their domain:

$$A \rightharpoonup_{\mathsf{impl}} B := \{f \mid \forall x \in A.\ f\ x \in B \quad \wedge \quad \forall x \notin A.\ f\ x = \mathit{undefined}\}$$

This function space satisfies a strong extensionality principle, but is not closed under composition:

$$f \in A \rightharpoonup_{\mathsf{impl}} B\ \wedge\ g \in B \rightharpoonup_{\mathsf{impl}} C\ \not\longrightarrow\ g \circ f \in A \rightharpoonup_{\mathsf{impl}} C$$

because if $x \notin A$ we have $g(f\ x) = g(\mathit{undefined})$, which can not be shown to be equal to $\mathit{undefined}$ (we do not know whether $\mathit{undefined} \notin B$). But we need closure under composition to form bijections between function spaces. Repairing this by considering only bijections which map $\mathit{undefined}$ to $\mathit{undefined}$ does not work either: this needs the condition $\forall x.\ f\ x = \mathit{undefined} \longrightarrow x = \mathit{undefined}$, from which we can derive that $\mathit{UNIV} - A \subseteq \{\mathit{undefined}\}$, which undermines the purpose of using sets instead of types.

The (explicit) partial set function space

$$A \rightharpoonup_{\mathsf{expl}} B := \{f \mid \forall x \in A.\ \exists y \in B.\ f\ x = \mathit{Some}\ y \quad \wedge \quad \forall x \notin A.\ f\ x = \mathit{None}\}$$

is well-behaved under bijections, but requires using a non-standard application, which is underspecified outside of the domain of the applied function. We do not want to impose the usage of this function space on meta-theories, so instead we generalize the concept of bijections between sets by using equivalence relations instead of equalities. Because the usual meta-theorems do not make use of equalities between functions, we are then free to use a custom equivalence relation on the function space, which features the strong extensionality principle.

## 6.2 Basic Concepts

A *setoid* $(A, \sim_A)$ is a set $A$ together with an equivalence relation $\sim_A$ over it, which we use on elements of the set instead of the foundational equality of the logic. For ease of presentation we go with the usual mathematical practice of letting the underlying set denote the mathematical structure, i.e., we write $A$ for the setoid $(A, \sim_A)$.

The concept of a *generalized bijection* between setoids results when replacing equalities with equivalence relations in the usual definition of bijections between sets:

$$\begin{aligned} f \text{ generalized bijection between } A, B \quad := \\ (\forall x_1, x_2 \in A.\ x_1 \sim_A x_2 \ \leftrightarrow\ f\ x_1\ \sim_B f\ x_2) \\ \wedge\ (\forall y \in B.\ \exists x \in A.\ f\ x \sim_B y) \end{aligned}$$

Note that the direction $\rightarrow$ of the biimplication corresponds to well-definedness of $f$ as a setoid functions, the direction $\leftarrow$ corresponds to generalized injectivity and the rest of the formula corresponds to generalized surjectivity. If we would quotient $A$ and $B$ under their equivalence relations and lift $f$ to equivalence classes, the result would be a bijection in the usual sense.

The interesting parts of the theory of bijections between sets carry over mutatis mutandis to generalized bijections between setoids. Note that defining generalized inverses of generalized bijections naturally requires choice (instead of just a definite descriptor which selects a uniquely characterized element), because of the consequent use of equivalence relations instead of equalities. We write a generalized inverse of a generalized bijection $f$ in the usual way as $f^{-1}$.

A *setoid isomorphism* is a family of generalized bijections between setoids, such that relevant theory content using the source setoids maps to corresponding theory content using the target setoids.

**Example 6.1** *As an example from everyday mathematics, consider the fields* $\mathbb{R}$, $\{z \in \mathbb{C} \mid \Im\ z = 0\}$ *as setoids with equality as the equivalence relation. They are isomorphic via* $f : x \mapsto x + 0 \cdot i$*, which means that their constants correspond to each other under* $f$*:* $f(x + x_2) = f\ x + f\ x_2$*,* $f(-x) = -(f\ x)$*,* $f(x \cdot x_2) = f\ x \cdot f\ x_2$*,* $f(x^{-1}) = (fx)^{-1}$*,* $f\ 0 = 0$*,* $f\ 1 = 1$*. It is now intuitively obvious to mathematicians that they can identify the two fields. Formally speaking this is the case because we can transfer theory content about fields between them. In order to animate the transfer of theory content we have to extend* $f$ *to act on propositions, and where necessary even to act on higher-order concepts, e.g., subsets of the fields.*

In our case a setoid isomorphism is implemented as a set of meta recursion clauses, so we represent it in Isabelle as a logically-vacuous marker $\phi$, which occurs in the applications of the relevant judgements, in order to select these clauses.

## 6.3 A Library of Generic Isomorphisms and their Animation

The library for transfer between isomorphic setoids is structured as follows:

- It provides generic isomorphisms of general interest, in particular on propositions and setoid function spaces. Those isomorphisms act as functors, i.e., they preserve the setoid formation constructs.
- These general isomorphisms can be selectively overridden by ad-hoc isomorphisms. This works by registering new meta recursion clauses, which match on certain markers. In particular the base cases for atomic setoids (e.g., variables) are provided in this way by the caller.

We provide generic isomorphisms for the propositions as a setoid and setoid function space.

We want the isomorphism on propositions to be the identity, so that input and output propositions are equivalent and we can conclude the output proposition if the input proposition is true. This means that the transformation of propositional terms corresponds to rewriting.

Isomorphisms on setoid function spaces have to be compatible with the operations of this pathological theory: application and abstraction. I.e., we want to transform applications to applications and abstractions to abstractions.

Let us take make things more formal now.

We consider propositions as a setoid $\mathbb{B} := (\{\mathit{True}, \mathit{False}\}, (=))$. The *setoid function space* $A \rightsquigarrow B$ between setoids $A, B$ is defined as:

$$(\{f \in A \to B \mid \forall x_1, x_2 \in A.\ x_1 \sim_A x_2 \longrightarrow f\ x_1 \sim_B f\ x_2\}, \sim_{A \rightsquigarrow B})$$
$$\text{where} \quad f \sim_{A \rightsquigarrow B} g\ :=\ \forall x \in A.\ f\ x \sim_B g\ x$$

That is we consider the functions that are well-defined w.r.t. the equivalences of domain and codomain, and compare them pointwise on their domain with the codomain equivalence.

First we need a judgement and meta recursion clauses to establish that we are indeed dealing with setoids:

$$\mathit{setoid}\ A := A \text{ is a setoid} \qquad (A \text{ is an input argument})$$
$$\uparrow \frac{}{\mathit{setoid}\ \mathbb{B}} \qquad \uparrow \frac{\mathit{setoid}\ A \qquad \mathit{setoid}\ B}{\mathit{setoid}\ A \rightsquigarrow B}$$

Keep in mind that these meta recursion clauses can be extended to deal with new setoid formation constructs, in particular we can declare variables as setoids.

## 6.4 Composing Generalized Bijections to Form Transforms away from Setoids

We introduce the judgement which composes generalized bijections of the isomorphism to form the generalized bijection needed to transfer away from a concrete setoid.

$$\phi : \ A \simeq_{\triangleright} A' \ \mathsf{via} \ f \ := f \text{ is a generalized bijection between } A \text{ and } A' \\ \wedge \ \mathit{setoid}\ A \ \wedge \ \mathit{setoid}\ A'$$

The inputs of this judgement are $\phi, A$, the outputs are $A', f$. Note that the marker $\phi$ is indeed logically vacuous. Now let us look at the rules composing generalized bijections. We need a concept to compose the generalized bijections from the domain and codomain of a setoid function space, to form a generalized bijection from the setoid function space to another setoid function space. If $f$ is a generalized bijection from $A$ to $A'$ and $g$ is a generalized bijection from $B$ to $B'$ then we define a certain generalized bijection from $A \rightsquigarrow B$ to $A' \rightsquigarrow B'$:

$$f \gg g \ := \ (\lambda\, h \in A \rightsquigarrow B.\ g \circ h \circ f^{-1}).$$

We use $\gg$ as an infix operator that associates to the right, so that $f_1 \gg f_2 \gg f_3$ means $f_1 \gg (f_2 \gg f_3)$. The characteristic property of $\gg$ and our reason for choosing it is compatibility with application:

$$((f \gg g)\ t_1)\ (f\ t_2) \sim_{B'} g\ (t_1\ t_2)$$

We can now state the meta recursion clauses that compose generalized bijections:

$$\uparrow \frac{\phi : \ A \simeq_{\triangleright} A' \ \mathsf{via} \ f \qquad \phi : \ B \simeq_{\triangleright} B' \ \mathsf{via} \ g}{\phi : \ (A \rightsquigarrow B) \simeq_{\triangleright} (A' \rightsquigarrow B') \ \mathsf{via} \ f \gg g} \qquad \uparrow \frac{}{\phi : \ \mathbb{B} \simeq_{\triangleright} \mathbb{B} \ \mathsf{via} \ \mathit{id}}$$

## 6.5 Transforming Terms in Setoids under Isomorphisms

Let us now look at the transfer of terms under isomorphisms. We use abstractions and quantors annotated with the setoid of their domain, written as $\lambda\, x \in A.\ t$, $\forall x \in A.\ P$ respectively. In order to show that abstractions inhabit the setoid function space, we have show that their body is invariant under exchanges of the bound variable modulo the equivalence relation of the domain. To this end we introduce a judgement

$$t_1 \ \sim \ t_2 \ \triangleright \ A \ := \ t_1 \sim_A t_2 \ \wedge \ t_1 \in A \ \wedge \ t_2 \in A \ \wedge \ \mathit{setoid}\ A$$

with inputs $t_1, t_2$ and output $A$, which we will use to establish this invariance with the rules

$$\uparrow \frac{f_1 \sim f_2 \rhd A \rightsquigarrow B \qquad \mathit{setoid}\ B \qquad a_1 \sim a_2 \rhd A}{(f_1\ a_1) \sim (f_2\ a_2) \rhd B}$$

$$\uparrow \frac{\begin{array}{c}\bigwedge x.\ x \sim x \rhd A \implies t_1\ x \sim t_2\ x \rhd B \\ \bigwedge x_1, x_2.\ x_1 \sim x_1 \rhd A \implies x_2 \sim x_2 \rhd A \implies x_1 \sim x_2 \rhd A \implies t_1\ x_1 \sim t_1\ x_2 \rhd B \\ \bigwedge x_1, x_2.\ x_1 \sim x_1 \rhd A \implies x_2 \sim x_2 \rhd A \implies x_1 \sim x_2 \rhd A \implies t_2\ x_1 \sim t_2\ x_2 \rhd B\end{array}}{\lambda x \in A.\ t_1\ x \sim \lambda x \in A.\ t_2\ x \rhd A \rightsquigarrow B}$$

$$\uparrow \frac{(\lambda x \in A.\ P_1\ x) \sim (\lambda x \in A.\ P_2\ x) \rhd A \rightsquigarrow \mathbb{B}}{(\forall x \in A.\ P_1\ x) \sim (\forall x \in A.\ P_2\ x) \rhd \mathbb{B}}$$

$$\uparrow \frac{}{(\sim_A) \sim (\sim_A) \rhd A \rightsquigarrow A \rightsquigarrow \mathbb{B}} \qquad \uparrow \frac{}{(\longrightarrow) \sim (\longrightarrow) \rhd \mathbb{B} \rightsquigarrow \mathbb{B} \rightsquigarrow \mathbb{B}}.$$

In the rule for abstractions we have to introduce reflexive invariance assumptions for $x_1, x_2$ even when varying them as $x_1 \sim x_2$. This is because recursive calls, e.g., while deriving the invariance premise $t_1\ x_1 \sim t_1\ x_2 \rhd B$, might need to vary a different bound variable, in order to soft-type an abstraction in $t_1\ x_1$ or $t_1\ x_2$. When deriving goals $t_1 \sim t_2 \rhd B$ we generally expect the context to contain reflexive invariance assumptions $x \sim x \rhd A$ for any free variable $x$ of $t_1$ or $t_2$.

Rules for other logical connectives are similar. Invariance derivations are exponential in the number of abstractions and quantifications. As an optimization, the implementation also features a rule that is specialized to abstractions whose domain is a set. In that case checking invariance of the body under exchanges of the bound variable is not necessary.

Now we introduce the judgement for the transformation of term under an isomorphism:

$$\phi :\ t \in A \mapsto t' \in A' \ \mathsf{via}\ f \quad := \quad f\ t \sim_{A'} t' \ \wedge\ t \in A \ \wedge\ t' \in A' \ \wedge\ \phi :\ A \simeq_{\rhd} A' \ \mathsf{via}\ f$$

The inputs of this judgement are $\phi, t$, the outputs are $t', A', f$. First the rules treating the primitive operations on setoid function space:

$$\uparrow \frac{\begin{array}{c}\phi :\ t_1 \in (A \rightsquigarrow B) \mapsto t_1' \in (A' \rightsquigarrow B') \ \mathsf{via}\ f \gg g \\ \phi :\ B \simeq_{\rhd} B' \ \mathsf{via}\ g \qquad \phi :\ t_2 \in A \mapsto t_2' \in A' \ \mathsf{via}\ f\end{array}}{\phi :\ (t_1\ t_2) \in B \mapsto (t_1'\ t_2') \in B' \ \mathsf{via}\ g}$$

$$\uparrow \frac{\begin{array}{c}\phi :\ A \simeq_{\rhd} A' \ \mathsf{via}\ f \\ \bigwedge x_1, x_2.\ x_1 \sim x_1 \rhd A \implies x_2 \sim x_2 \rhd A \implies x_1 \sim x_2 \rhd A \implies t\ x_1 \sim t\ x_2 \rhd B \\ \bigwedge x, x'.\ \phi :\ x \in A \mapsto x' \in A' \ \mathsf{via}\ f \implies x \sim x \rhd A \implies \phi :\ t\ x \in B \mapsto t'\ x' \in B' \ \mathsf{via}\ g\end{array}}{\phi :\ (\lambda x \in A.\ t\ x) \in (A \rightsquigarrow B) \mapsto (\lambda x' \in A'.\ t'\ x') \in (A' \rightsquigarrow B') \ \mathsf{via}\ f \gg g}$$

In the rule for applications, we first synthesize the transformation $t_1'$ of the operator $t_1$ and a generalized bijection realizing this transformation, which has to be of the form $f \gg g$. Then we go on to check that $g$ is in fact the generalized bijection transforming $B$ to $B'$. Lastly we transform the argument $t_2$ to an argument $t_2'$ and match the synthesized generalized bijection against $f$. The application $t_1'\ t_2'$ results via transformation under the generalized bijection $g$, because of the characteristic property of $f \gg g$.

In the rule for abstractions, we first synthesize the transformation $A'$ of the domain $A$ under the synthesized generalized bijection $f$. Next we show the invariance of the body $t$ under exchanges of the bound variable modulo the equivalence relation of $A$. This also synthesizes the setoid $B$ of the body. Then we transform the body $t\ x$ under the assumption that the bound variable $x$ is transformed to $x'$ via $f$, and abstract $x'$ out of the result, to synthesize the term context $t'$. The result is the abstraction $(\lambda\, x' \in A'.\ t'\ x')$ which was transformed under $f \gg g$.

In the rule for applications we crucially rely on the assumption that the generalized bijection inferred to act on the domain of the operator $t_1$ is the same as the generalized bijection inferred to act on $t_2$. We named both of these $f$, so they are matched against each other in the execution. It is a global invariant on the meta recursion clauses for any isomorphism that the inferred bijections (and also the inferred setoids) for combined subterms are compatible. At the moment this is not checked for custom meta recursion clauses of the relevant judgements. This property cannot be included in the semantics of the judgement, because it is fundamentally intensional. It can be shown informally with an easy induction on derivations of the rule system. Automating this informal concept would be very interesting. It is noteworthy that the proof system Twelf gives inductions like these the status of a reasoning primitive [10].

Now the rules for some logical connectives just provide the corresponding setoids and shift the work to the transformations of applications and abstractions.

$$\uparrow \frac{\phi :\ (\lambda\, x \in A.\ P\ x) \in (A \rightsquigarrow \mathbb{B}) \mapsto (\lambda\, x' \in A'.\ P'\ x') \in (A' \rightsquigarrow \mathbb{B}) \text{ via } f \gg id \qquad \phi :\ A \simeq_{\triangleright} A' \text{ via } f}{\phi :\ (\forall x \in A.\ P\ x) \in \mathbb{B} \mapsto (\forall x' \in A'.\ P'\ x') \in \mathbb{B} \text{ via } id}$$

$$\uparrow \frac{\phi :\ A \simeq_{\triangleright} A' \text{ via } f}{\phi :\ (\sim_A) \in (A \rightsquigarrow A \rightsquigarrow \mathbb{B}) \mapsto (\sim_{A'}) \in (A' \rightsquigarrow A' \rightsquigarrow \mathbb{B}) \text{ via } f \gg f \gg id}$$

$$\uparrow \frac{}{\phi :\ (\longrightarrow) \in (\mathbb{B} \rightsquigarrow \mathbb{B} \rightsquigarrow \mathbb{B}) \mapsto (\longrightarrow) \in (\mathbb{B} \rightsquigarrow \mathbb{B} \rightsquigarrow \mathbb{B}) \text{ via } id \gg id \gg id}$$

Note that the rule for equivalence relations makes essential use of generalized bijectivity. Rules for other logical connectives are similar.

## 6.6 Transforming Theory Content Under Setoid Isomorphisms

We realize the transformation of theory content with customizable forward rules that employ the transformations of setoids and terms under isomorphisms.

Constant definitions can be transformed (or "lifted") in the way we have seen in the example of section 5.5, defining the new constant $c'$ as image of the old constant $c$ under the corresponding generalized bijection $f$ and registering the new constant as the transformation result of the old constant:

$$\downarrow \frac{\ldots \qquad c \rhd A \qquad \phi :\ A \simeq_{\triangleright} A' \text{ via } f \qquad \text{define } c' \text{ as } f\ c}{c' \rhd A' \qquad \wedge \qquad \phi :\ c \in A \mapsto c' \in A' \text{ via } f}$$

Variations are possible of course, in particular we might want to use a $c'$ which already exists and is equivalent to $f\ c$.

Usually we do not transform a typedef'd type, but rather use a typedef to introduce a new type $\tau$ for a certain representing set $A$ and then emit the abstraction function as a bijection which maps the representing set to the new type. Corresponding forward rules thus look like

$$\downarrow \frac{\ldots \qquad \mathsf{typedef}\ \tau\ :=\ A\ \mathsf{via}\ Abs}{\phi :\ A \simeq_{\triangleright} \tau\ \mathsf{via}\ Abs},$$

where we use the typedef judgement to trigger the foundational theory extension mechanism of the same name, synthesizing the new type $\tau$ and the abstraction function $Abs$. The semantics of the typedef $\tau\ :=\ A$ via $Abs$ judgement states bijectivity of the abstraction function $Abs$ between $A$ and $\tau$, so a forward rule like the one above is trivial. In the conclusion we write $\tau$ instead of the setoid based on the set $UNIV :: \tau\ set$ containing every inhabitant of that type (with equality as the equivalence relation).

Theorems are transformed by transforming their proposition $P$ under the identity bijection, so the resulting proposition $P'$ is equally true. Forward rules realizing this take forms such as:

$$\downarrow \frac{\ldots \qquad \phi :\ P \in \mathbb{B} \mapsto P' \in \mathbb{B}\ \mathsf{via}\ id}{\mathsf{note}\ P'},$$

We generate a fact with the special judgement note (semantically the identity on propositions), which triggers the storage of $P'$ as a theorem.

## 6.7 Currying Functions on Finite Products

We want to make meta-theories as simple as possible. In particular we only want to deal with fully applied functions, i.e., we want to enable meta-theories to stay essentially first-order. Functions of arbitrary arities are thus represented as functions on arbitrary finite products. This makes the transformation to functions we want to see in the result non-trivial: we need to map functions $f \in A_1 \times \ldots \times A_n \to B$ on arbitrary finite products into a curried form $f' \in A_1 \to \ldots \to A_n \to B$.

First of all we need to represent finite products $A_1 \times \ldots \times A_n$, where the sets of the components $A_i$ lie in a pseudo-universe. The sequence of sets $A_1, \ldots, A_n$ is represented as a HOL list and tuples in the product $A_1 \times \ldots \times A_n$ as lists of length $n$. So we arrive at the following notion of product:

$$\begin{aligned} &product\ Nil\ :=\ \{\ Nil\ \} \\ &product\ (Cons\ A\ As)\ :=\ \{\ Cons\ a\ as \mid a \in A\ \wedge\ as \in product\ As\ \} \end{aligned}$$

Our application only uses products of sets and setoid function spaces over them, but not products of general setoids. So we will only treat these cases. We write products $product\ As$ and component sets $A_i$ that occur in them for the setoids $(product\ As, =)$ and $(A_i, =)$. We write sets $B, C$ for the general setoids $(B, \sim_B), (C, \sim_C)$.

The implementation makes use of annotated versions of the tuples made of $Cons$ and $Nil$, to make the setoids they lie in easily synthesizable. Because of this do not show the reflexive invariance rules for tuples made of $Cons$ and $Nil$, but of course $Cons \in A \rightsquigarrow product\ As \rightsquigarrow product\ (Cons\ A\ As)$, for any $A, As$, and $Nil \in product\ Nil$.

We now state the bijection forming rules.

$$\uparrow \frac{\phi : \ B \simeq_{\triangleright} B' \text{ via } g}{\phi_{curry} : \ (product \ Nil \rightsquigarrow B) \simeq_{\triangleright} B' \text{ via } prod\_nil\_iso \ g}$$

$$\uparrow \frac{\phi : \ A \simeq_{\triangleright} A' \text{ via } f \qquad \phi_{curry} : \ (product \ As \rightsquigarrow B) \simeq_{\triangleright} C' \text{ via } g}{\phi_{curry} : \ (product \ (Cons \ A \ As) \rightsquigarrow B) \simeq_{\triangleright} (A' \rightsquigarrow C') \text{ via } prod\_cons\_iso \ f \ g}$$

Note that they are selected by using the special marker construction $\phi_{curry}$ which signifies the currying isomorphism based on the existing isomorphism $\phi$. The first rules treats the base case of empty products and just postprocesses the transformation of the codomain $B$. The second rules takes the product $product \ (Cons \ A \ As)$ in the domain apart, transforming $A$ and $product \ As \rightsquigarrow B$ independently and later composing the results.

They use the following basic generalized bijections for composing the transformation steps:

$$\begin{aligned} prod\_nil\_iso \ f \ &:= \ (\lambda \, h. \ f \ (h \ Nil)) \\ prod\_cons\_iso \ f \ g \ &:= \ (\lambda \, h, x. \ g \ (\lambda \, as. \ h \ (Cons \ (f^{-1} \ x) \ as))) \end{aligned}$$

The term formation construct that we want to treat specially now is the application of uncurried functions to tuples. These terms take the form $t \ [t_1, \ldots, t_n]$ and we transform them stepwise to $t' \ t'_1 \ \ldots \ t'_n$, by recursion on $n$ and starting at the head argument $t_1$.

For the base case $n = 0$ we know that $t$ is a function on the empty product, so it is transformed via a generalized bijection of the form $prod\_nil\_iso \ f$. We make simple use of $f \ (t \ Nil) = prod\_nil\_iso \ f \ t$.

$$\uparrow \frac{\phi_{curry} : \ t \in (product \ Nil \rightsquigarrow B) \mapsto t' \in B' \text{ via } prod\_nil\_iso \ f}{\phi_{curry} : \ t \ Nil \in B \mapsto t' \in B' \text{ via } f}$$

In the case of $n > 0$ we first transform the operator $t$ and the first component $t_1$ of the tuple in the usual way to $t'$ and $t'_1$. Then we use the generalized bijection $h$, which is responsible for currying functions such as $(\lambda \, as. \ t \ (Cons \ t_1 \ as))$. We use its generalized inverse $h^{-1}$ on $t' \ t'_1$ to form a function with the same behaviour as $(\lambda \, as. \ t \ (Cons \ t_1 \ as))$ and where the transformation result $t' \ t'_1$ is already fixed as a local rule. Under this local rule we transform $h^{-1} \ (t' \ t'_1) \ ts$ to the end result $t'$. This corresponds to seeing $h^{-1} \ (t' \ t'_1)$ as the new operator (replacing $t$) in the subcall. Termination holds because the arguments $ts$ of this operator are smaller than the arguments $Cons \ t_1 \ ts$ of the previous operator $t$.

$$\uparrow \frac{\begin{array}{c} \phi_{curry} : \ t \in (product \ (Cons \ A \ As) \rightsquigarrow B) \mapsto t' \in (A' \rightsquigarrow C') \text{ via } prod\_cons\_iso \ f \ h \\ \phi_{curry} : \ t_1 \in A \mapsto t'_1 \in A' \text{ via } f \qquad \phi_{curry} : \ (product \ As \rightsquigarrow B) \simeq_{\triangleright} C' \text{ via } h \\ \phi_{curry} : \ (h^{-1} \ (t' \ t'_1)) \in (product \ As \rightsquigarrow B) \mapsto (t' \ t'_1) \in C' \text{ via } h \\ \Longrightarrow \ \phi_{curry} : \ (h^{-1} \ (t' \ t'_1) \ ts) \in B \mapsto t' \in B' \text{ via } g \end{array}}{\phi_{curry} : \ (t \ (Cons \ t_1 \ ts)) \in B \mapsto t' \in B' \text{ via } g}$$

Note that the term transformation used $\phi_{curry}$ throughout. This is because our generic rule for the transformation of abstractions does not know about the additional structure in

the isomorphism $\phi_{curry}$ and just inserts assumptions $\phi_{curry}: \ x \in A \mapsto x' \in A'$ via $f$. So we need a rule to embed the term transformation of the underlying base isomorphism $\phi$ into the term transformation of $\phi_{curry}$.

$$\uparrow \frac{\phi: \ t \in A \mapsto t' \in A' \text{ via } f}{\phi_{curry}: \ t \in A \mapsto t' \in A' \text{ via } f}$$

## 6.8 Related Work

The primary use of quotient packages [14, 11] available in various HOL implementations is the transfer of theory content under the canonical setoid isomorphism $x \mapsto [x]_{\sim_A} : (A, \sim_A) \simeq_\triangleright (A/\sim_A, =)$, which embeds elements into their equivalence class. Isabelle/HOL's quotient package [14] can also be used to transfer results under setoid isomorphisms, if the codomain is a type and not a general setoid. This restriction makes the application of the tool on arbitrary generalized inverses impossible, so it remains a one-way transformation. Further limitations are that the resulting terms are not synthesized, lifting is based on constants, not on general terms, and cannot be adapted to produce existing terms or types. Customized higher-order transformations, such as currying of functions on finite products, do not seem plausible given the architecture of the quotient package. None of these limitations arise with our approach and we make essential use of some of these features. Now we start a more technical comparison. In our case lifting is not part of the algorithmic rule system for transfer under setoid isomorphisms, but rather implemented as custom forward rules that emit meta recursion clauses realizing the transfer of terms. The construction of aggregate equivalence relations happens as we compose setoids in the corresponding positions of our judgements. "Injecting" a theorem is unnecessary in our case because we do not rely on rewriting, but rather use a custom top-down one-pass transformation with specialized context management. "Regularization" and "cleaning" are not handled by the component for transfer under setoid isomorphisms, but rather are separate steps. "Cleaning" is just an easy rewriting pass in our case. "Regularization" was unnecessary in our application because we can just state the meta-theorems we want to transfer in a "regularized" way. It is interesting to note the difference in implementation complexity: the quotient package is implemented in roughly 2300 lines of ML, while our basic rule system for transfer under setoid isomorphisms consists of just a few rules and is highly customizable.

The purely structural nature of the meta-theoretic proofs in Peter Andrews treatment of isomorphisms between HOL models [1], made me internalize and later generalize them.

Setoids and functions between them are frequently used concepts in type theories (see e.g., [3]). But we are not aware of the use of generalized bijections between setoids to transfer theory content.

Voevodksy, Coquand et. al. [27] set out to define a dependent type theory with a generalized notion of isomorphism as the foundational equality. In this type theory every proposition is invariant under isomorphisms, while we have to check that the proposition contains only invariant term constructions. The invariance under isomorphisms is given by a substitution principle, whose application should be related to our notion of transfer of theory content. The precise relationship is not known to the author.

# 7 Application: Non-Freely Generated Datatypes

We tested our paradigm and infrastructure in the construction of a package for non-freely generated datatypes. Non-freely generated datatypes are interesting in that they allow the direct realization of many abstract datatypes [28] as constructors with equational axioms. E.g. the abstract datatype $'a\ fset$ of finite sets can be specified by declaring constructors

$$\begin{aligned} &\mathit{Empty} &&:: {'a}\ \mathit{fset} \\ &\mathit{Single} &&:: {'a} \Rightarrow {'a}\ \mathit{fset} \\ &\mathit{Union} &&:: {'a}\ \mathit{fset} \Rightarrow {'a}\ \mathit{fset} \Rightarrow {'a}\ \mathit{fset}, \end{aligned}$$

together with axioms stating that $\mathit{Union}$ is associative, commutative, idempotent and $\mathit{Empty}$ is a neutral element for $\mathit{Union}$. The algebraic specification [28] community is also concerned with relating abstract datatypes of related specifications. We do not provide features in that direction.

We treat mutually-recursive non-nested datatypes, which are quotiented under equational consequences of Horn theories. In our case these consist of implicational formulas with schematic variables over a many-sorted signature of operation and relation symbols, also featuring so called parameter conditions. Because the construction is parametric in them, we call existing types *parameters* in this context. *Parameter conditions* are relations over existing types with a fixed interpretation and are not characterized by Horn clauses.

Sorts play the role of mutually-recursive datatypes, operations play the role of constructors. Relations and parameter conditions are used as the premises of the conditional equations of the Horn theory. Relations are defined inductively, by Horn clauses, while parameter conditions have a fixed interpretation and act only on parameters.

## 7.1 Technical Challenges

Apart from the general animation needed for meta-theories in our paradigm, this application presents further technical challenges:

- Input should deal with curried operations, but we want to formulate the meta-theory as a first-order theory. So a currying transformation is necessary. Also we have to consider both directions of this isomorphism: uncurrying for importing theory content and currying for exporting it. We only the realize currying transformation as an algorithmic rule system and implement uncurrying by applying the inverse of the generalized bijection that results when transforming the expected result under the currying isomorphism.

- We have to relate the abstract notion of satisfaction of a Horn theory to the provability of nice HOL formulas. In particular this requires the elimination of quantifications over interpretations, in favor of variable quantification.
- The animation of the iteration principle on these datatypes needs to be separated from the basic construction of the datatype, in order to allow the construction of several different iterations at different places in a theory.
- This animation of the iteration principle has to transfer the semantic interpretations of operations (which represent the iteration steps) and relations to act on a pseudo-universe.
- This animation of the iteration principle needs the proven interpretation of the Horn theory as input, showing that the iteration is invariant under the quotienting of its domain.

## 7.2 Sketch of the Meta Theory

The meta-theory that we animate to become this package is parameterized on the following signature:

- Type variables $'sort$ for sorts, $'psort$ for parameter sorts, $'opsym$ for operation symbols, $'relsym$ for relation symbols, and $'paramuni$ for the parameter pseudo-universe.
- Term variables for functions realizing the sortings of operation and relation symbols. The codomain sorting of operations is given by $codom\_sort :: 'opsym \Rightarrow 'sort$, the internal argument arity of operations by $op\_ar :: 'opsym \Rightarrow 'sort\ list$, the parameter argument arity of operations by $op\_param\_ar :: 'opsym \Rightarrow 'psort\ list$, the internal argument arity of relations by $rel\_ar :: 'relsym \Rightarrow 'sort\ list$, the parameter argument arity of relations by $rel\_param\_ar :: 'relsym \Rightarrow 'psort\ list$.
- A term variable $param\_int :: 'psort \Rightarrow 'paramuni\ set$ realizing the interpretation of parameter sorts as subsets of the parameter pseudo-universe.
- A set of acceptable sortings of parameter conditions.
- A set of Horn clauses $hcls$, representing the Horn theory.
- An assumption stating the well-sortedness of equations and applications of relation symbols and parameter conditions in the Horn clauses.
- An assumption stating the non-emptiness $param\_int\ ps \neq \emptyset$ of the interpretations of parameter sorts $ps$.
- An assumption stating the *abstract inhabitedness* of sorts by sorted operation applications, which is the concept we were animating in section 5.4, disregarding parameter arguments, whose existence is guaranteed by the last assumption.

We now sketch the notions and meta-theorems needed in the animation. *Horn clauses* are implicational formula containing schematic sorted variables. We distinguish between *parameter variables* having parameter sorts $ps$ and *term variables* having sorts $s$. *Horn clause atoms* consist of well-sorted applications of *relation symbols* $rel$ to parameter variables and terms, parameter conditions applied to parameter variables and sorted equations between terms. *Terms* are formed by applying sorted operations symbols $op$ to parameters and terms of the respective argument sorts.

Terms are interpreted by replacing operation symbols and variables with their interpretations. *Satisfaction of interpreted Horn clauses* $sat\ \mathcal{I}\ hcl$ is defined in the usual way: for all interpretations of sorted parameter variables $\mathcal{I}_{\mathsf{pvar}}$ in the interpretations of parameter sorts $param\_int$ and all interpretations $\mathcal{I}_{\mathsf{var}}$ of sorted term variables in the interpretations $\mathcal{I}$ of sorts, the implication is satisfied. *Satisfaction of interpreted Horn clause atoms* $sat\_atom\ \mathcal{I}\ \mathcal{I}_{\mathsf{pvar}}\ \mathcal{I}_{\mathsf{var}}\ A$ is defined as follows: relation applications are satisfied if the interpretation of the relation is true on the interpreted arguments, parameter condition applications are satisfied if the fixed interpretation of the parameter condition holds for the interpreted arguments and equalities are satisfied if the interpreted terms are equal. We will be slightly informal in our use of $sat\_atom\ \mathcal{I}\ \mathcal{I}_{\mathsf{pvar}}\ \mathcal{I}_{\mathsf{var}}$ and $sat\ \mathcal{I}$, in particular writing just $\mathcal{I}$ for the necessary interpretations of sorts, operators, relations.

The constructed datatypes are modeled as subsets of a quotient term model. The *term model* consists of well-sorted ground terms. *Ground terms* are a new datatype which consists of operation symbols applied to parameter interpretations and ground terms. We impose a sort discipline on ground terms in the usual way, in particular we demand that parameter interpretation arguments lie in the interpretation of the corresponding parameter sort.

We inductively define the *equational and relational consequences of the Horn theory* on ground terms. This uses satisfiablity of Horn atoms interpreted on ground terms to discharge premises of Horn clauses. Equational consequences are completed to form a congruence of course.

Now we form the *quotient term model* by quotienting ground terms under equational consequences of the Horn theory. The notions of application of operation and relation symbols to arguments, are lifted to equivalence classes of the quotient term model. We write them as $\mathcal{I}_{\mathsf{term}}\ op$, $\mathcal{I}_{\mathsf{term}}\ rel$. The sorting of ground terms is also lifted to equivalence classes of the quotient term model. The set of ground term equivalence classes of a sort $s$ are referred to as $\mathcal{I}_{\mathsf{term}}\ s$. For each operation symbol $op$ we can prove

$$\begin{aligned}\mathcal{I}_{\mathsf{term}}\ op \in\ & product\ (map\ param\_int\ (op\_param\_ar\ op)) \\ & \to product\ (map\ \mathcal{I}_{\mathsf{term}}\ (op\_ar\ op)) \to\ \mathcal{I}_{\mathsf{term}}\ (codom\_sort\ op).\end{aligned}$$

The abstract inhabitedness of a sort implies the inhabitedness of ground terms of this sort, by using the assumption that interpretations $param\_int\ ps$ of parameter sorts $ps$ are always nonempty. This further implies inhabitedness of the sets $\mathcal{I}_{\mathsf{term}}\ s$. We use these sets to model the mutually-recursive datatypes.

We define an *iteration construct* on ground terms, which replaces occurring operation symbols $op$ by an interpretation $\mathcal{I}\ op$. This iteration construct is lifted to equivalence classes of ground terms and we refer to it as $iter\ \mathcal{I}$. For each sort $s$ we have $iter\ \mathcal{I} \in \mathcal{I}_{\mathsf{term}}\ s \to \mathcal{I}\ s$. Applying this iteration construct is only sensible if its interpretation of

operation symbols, together with an interpretation of sorts and relations satisfies the Horn clauses and if the interpretation of operation symbols is compatible with the sorting of the operation symbol.

The following meta-theoretic results are derived:

- Satisfaction of the Horn clauses interpreted on the quotient term model.
- An *induction principle* for a family of predicates indexed by sorts: if the satisfaction of a family of predicates is closed under application of sorted operations to quotiented ground terms, we conclude that each of the predicates holds for any quotiented ground term of corresponding sort.
- The *iteration principle* is an equation specifying the behaviour of the iteration construct on the quotient term model: operation applications are replaced by an interpretation. Assumptions of this equations are the compatibility of the operation interpretation with the sorting interpretation and the satisfaction of the Horn clauses under this interpretation.
- Relational consequences (lifted to the quotient term model) also hold if relations are interpreted and arguments are iterated on. This is true under the assumption of compatibility of the operation interpretation with the sorting interpretation and the satisfaction of the Horn clauses under this interpretation.

## 7.3 Input Language and Input Processing

As an experiment we even eliminated the use of ML for input processing. To this end we introduce a datatype of input terms, featuring operations, relations, parameter conditions, applications, and the necessary logical operators. We use meta recursion to establish a typing and kinding discipline. In particular it is necessary to distinguish between *parameter types*, i.e., types that already exists and the construction is parametric in them, and *internal types* that are constructed.

The input to the package is now given as the statement of facts that are logically vacuous:

- kinding declarations of type constructors
- semantic interpretations of the occurring parameter types
- type declarations of operations, relations and parameter conditions, which also distinguish these three syntactic entities
- semantic interpretations of parameter conditions, agreeing with the semantics of the type declaration of the parameter condition
- implicational formulas with outermost quantification to declare the Horn theory

Since we do not allow nested type constructors, we introduce a datatype of input kinds.

$$K \quad ::= \quad *^{param} \to K \quad | \quad *^{int} \quad | \quad *^{param}$$

This ensures that type constructors may only take parameter types as arguments, by restricting the kinds that may be used in type constructor declarations.

The datatype of input types contains type constructors $\kappa$, (partial) type constructor applications $T_1\ T_2$, function spaces $T_1 \to T_2$, type variables $\alpha$ and truth values $o$.

$$T \quad ::= \quad \kappa \quad | \quad T_1\ T_2 \quad | \quad T_1 \to T_2 \quad | \quad \alpha \quad | \quad o$$

The kinding judgement is logically vacuous and we give the following meta recursion clauses for it, with an implicit base case for kind declarations of type constructors.

$$\uparrow \frac{T_1 : *^{param} \to K \qquad T_2 : *^{param}}{T_1\ T_2 : K} \qquad \uparrow \frac{}{\alpha : *^{param}}$$

Note that we treat only type constructor applications and type variables. i.e., we do not allow the kinding of function spaces or $o$. We treat them outside the kinding system in type declaration constructs for operations, relations and parameter conditions, to cater to the first-order nature of the meta-theory. For reasons of simplicity, parameter type arguments have to precede internal type arguments in type declarations. These are the allowed type declarations for signature elements, where $T_n : *^{param},\ T'_m : *^{int},\ T'' : *^{int}$:

- Types declared for operations have the form $\overrightarrow{T_n} \to \overrightarrow{T'_m} \to T''$.
- Types declared for relations have the form $\overrightarrow{T_n} \to \overrightarrow{T'_m} \to o$.
- Types declared for parameter conditions have the form $\overrightarrow{T_n} \to o$.

We now introduce a datatype of input terms, enhanced with logical constructs for implications, equalities and quantification over types, but without general abstractions.

$$t \quad ::= \quad x \quad | \quad c \quad | \quad t_1\ t_2 \quad | \quad \forall x : T.\ t \quad | \quad t_1 \longrightarrow t_2 \quad | \quad t_1 = t_2$$

Constants $c$ are further syntactically stratified into operations, relations and parameter conditions. The typing rules are the usual ones, except that we only allow equations over internal types. Horn formulas are well-typed input terms of propositional type $o$ of the form $\forall \overrightarrow{x_n : T_n}.\ \overrightarrow{t_m} \longrightarrow t$, where $t_m, t$ do not contain quantifiers or implications.

In a step separate from the basic construction, an iteration over a non-free datatype can be specified. The input for this step consists of logically vacuous facts for the following:

- Semantic interpretations for the occurring internal types, which serve as the codomains of the iterator functions. We call these the *target types*.
- Semantic interpretations for the iteration step of each operation, which agree with the semantic interpretation of the declared operation types. We just call these the *interpretations of the operations*.
- Similarly, we need *semantic interpretations of the relations*, which agree with the semantic interpretation of the declared relation types.
- Declarations of the proven semantic interpretation of the clauses in the Horn theory.

## 7.4 Sketch of the Basic Meta-Theory Animation up to the Construction of the Datatypes

We now provide a rough sketch how the meta-theory is animated, leading up to the construction of datatypes, but not including the animation of meta-theorems.

- By kind synthesis of the subtypes in the operation type declarations we determine which input types are internal or parameter types.
- We check that parameter input types have a semantic interpretation declaration.
- We construct an algorithmic bijection between internal input types and sorts, which are the constructors of a new datatype.
- We construct an algorithmic bijection between parameter input types and parameter sorts, which are the constructors of a new datatype.
- We construct an algorithmic bijection between operations in the input and operation symbols, which are constructors of a new datatype.
- We construct an algorithmic bijection between relations in the input and relation symbols, which are constructors of a new datatype.
- We construct functions representing the sorting of operations and relations in the form necessary for the signature.
- We check inhabitedness of the sorts by forward chaining of sort inhabitedness implications resulting from the sorting of operations. This is the example of section 5.4.
- We embed the semantic interpretations of parameter types into a parameter pseudo-universe and register bijections out of those subsets of the parameter pseudo-universe to the semantic interpretations of parameter types. Non-emptiness of these subsets follows from non-emptiness of the types they correspond to. The bijections taken together and closed under the rules for forming generalized bijections are called the *parameter reification isomorphism*. We use inverses of generalized bijections resulting from this isomorphism, to transform the interpretations of parameter conditions, as given by the user, to relations over the parameter pseudo-universe.
- We typecheck the Horn formulas given by the user, and process them into the internal format used by the meta theory, transforming occurring parameter conditions to relations that act on the parameter pseudo-universe. Then we show well-sortedness of these Horn formulas.
- We instantiate the locale and discharge its assumptions by looking up established facts: sorts are inhabited and the subsets of the parameter universe corresponding to interpretations of parameter types are nonempty.

- We define types corresponding to the mutually-recursive datatypes, with typedefs using the subsets $\mathcal{I}_{\mathsf{term}}\ s$ of the quotient term model as representing sets. We register bijections from the representing sets to the newly defined types. We refer to the isomorphism that results in combination with the parameter reification isomorphism, as the *reification isomorphism*.
- We define the constructors for and relations on the datatypes, by transforming the corresponding functions $\mathcal{I}_{\mathsf{term}}\ op$, $\mathcal{I}_{\mathsf{term}}\ rel$ under the reification isomorphism.

## 7.5 Animating the Meta-Theorems

We state the meta theorems and discuss the steps necessary for their animation to produce the output theorems of this package.

### 7.5.1 Satisfaction of the Horn clauses in the Quotient Term Model

Satisfaction of the Horn clauses in the quotient term model is a meta theorem of the form

$$\forall t \in hcls.\ sat\ \mathcal{I}_{\mathsf{term}}\ t.$$

First we regard this meta-theorem as instantiated to every horn clause $\overrightarrow{t_n} \longrightarrow t \in hcls$ in separation. For every such instantiation we have (with implicit assumptions on $\mathcal{I}_{\mathsf{pvar}}, \mathcal{I}_{\mathsf{var}}$ for readability)

$$\forall \mathcal{I}_{\mathsf{pvar}}, \mathcal{I}_{\mathsf{var}}.\ \overrightarrow{sat\_atom\ \ \mathcal{I}_{\mathsf{term}}\ \mathcal{I}_{\mathsf{pvar}}\ \mathcal{I}_{\mathsf{var}}\ t_n} \longrightarrow\ sat\_atom\ \mathcal{I}_{\mathsf{term}}\ \mathcal{I}_{\mathsf{pvar}}\ \mathcal{I}_{\mathsf{var}}\ t$$

and transform this to a usable property.

First we unroll the quantifications over the (parameter) variable interpretations $\mathcal{I}_{\mathsf{var}}, \mathcal{I}_{\mathsf{pvar}}$ to quantifications over the occurring (parameter) variables $\overrightarrow{v_n}$ in $\overrightarrow{t_n} \longrightarrow t$:

- We replace the variable interpretation $\mathcal{I}_{\mathsf{var}}$ by a variable interpretation $\mathcal{I}_{\mathsf{var}} \oplus \overrightarrow{v_n \mapsto x_n}$, where all variables $v_i$ are updated to a universally quantified element $x_i$ of their respective domain.
- We evaluate the definition of satisfiablity, down to the use of the variable interpretation, and evaluate the applications $(\mathcal{I}_{\mathsf{var}} \oplus \overrightarrow{v_n \mapsto x_n})\ v_i$ to the respective elements $x_i$. This gets rid of the use of the variable interpretation.

After this we transfer the resulting property under the reification isomorphism and replace HOL with Pure connectives.

### 7.5.2 Induction Principle

To state the induction principle we need the concept that a predicate with two arguments holds over two lists for the two inputs:

$$\begin{aligned} holdsAll2\ P\ Nil\ Nil\ &:=\ True \\ holdsAll2\ P\ (Cons\ x\ xs)\ (Cons\ y\ ys)\ &:=\ P\ x\ y\ \wedge\ holdsAll2\ P\ xs\ ys \end{aligned}$$

The induction principle is the following meta-theorem:

$$\begin{array}{l}\forall s, P.\ \forall x \in \mathcal{I}_{\mathsf{term}}\ s.\\ \quad (\forall op.\ \forall pxs \in product\ (map\ param\_int\ (op\_param\_ar\ op)).\\ \quad\ \forall xs \in product\ (map\ \mathcal{I}_{\mathsf{term}}\ (op\_ar\ op)).\\ \quad\quad holdsAll2\ P\ (op\_ar\ op)\ xs\ \longrightarrow\ P\ (codom\_sort\ op)\ (\mathcal{I}_{\mathsf{term}}\ op\ pxs\ xs))\\ \longrightarrow\ P\ s\ x\end{array}$$

We transform it as follows:

- We unroll the quantification over sorts $s$ into a conjunction. This is an easy rewriting step that is available for each enumerative datatype of $n$ constructors $\overrightarrow{C_n}$:

$$(\forall\ x.\ P\ x) = P\ C_1\ \wedge\ \ldots \wedge\ P\ C_n$$

- We unroll the quantification over sort-dependent predicates $P$ into quantifications over predicates, one for each sort. This works by rewriting bounded quantifications over sort-dependent predicates.
- We unroll quantifications over operation symbols $op$ into conjunctions.
- We evaluate the arities of operation symbols and applications of $map$ over them.
- We unroll the quantifications over tuples $pxs, xs$ in products into quantifications over elements, with the tuple variables replaced by tuples containing the elements. This is achieved by rewriting with

$$\begin{array}{r}(\forall xs \in product\ Nil.\ P\ xs) = P\ Nil\\ (\forall xs \in product\ (Cons\ A\ As).\ P\ xs) = (\forall x \in A.\ \forall xs \in product\ As.\ P\ (Cons\ x\ xs)).\end{array}$$

- We transfer of the resulting proposition under the reification isomorphism.
- We replace HOL with Pure connectives and rewrite with

$$(P_1 \wedge P_2 \longrightarrow P) = (P_1 \longrightarrow P_2 \longrightarrow P).$$

### 7.5.3 Iteration Principle

The iteration principle is the meta-theorem (with implicit assumptions about $\mathcal{I}$ for readability):

$$\begin{array}{l}\forall \mathcal{I}, op.\ \forall pxs \in product\ (map\ param\_int\ (op\_param\_ar\ op)).\\ \quad \forall xs \in product\ (map\ \mathcal{I}_{\mathsf{term}}\ (op\_ar\ op)).\\ \quad\quad iter\ \mathcal{I}\ (\mathcal{I}_{\mathsf{term}}\ op\ pxs\ xs) = \mathcal{I}\ op\ pxs\ (map\ (iter\ \mathcal{I})\ xs)\end{array}$$

To animate this we first have to construct the interpretation of sorts, operation and relation symbols, just referred to as $\mathcal{I}$ here. These actually carry assumptions: the interpretation of operations has to be compatible with the sorting of the operations and the interpretation of sorts.

To construct the interpretation of sorts we construct a pseudo-universe embedding the target types given by the user. Each sort is interpreted as the subset of the pseudo-universe corresponding to its target type. We combine the bijections from these subsets of the pseudo-universe to the target types, with the parameter reification isomorphism. We define the interpretations of operations as the "unlifting" of their interpretation in the target types, which the user has given. This is achieved by applying inverses of generalized bijections that are constructed with this isomorphism. Interpretations of relations are constructed in the same way.

The compatibility of the interpretation of operation symbols follows by the typing of the isomorphism and the type of the interpretation the user has given. The restriction of the generic iterator $iter$ to the parts $\mathcal{I}_{\mathsf{term}}\ s$ of the quotient term model corresponding to each datatype, is transformed under the isomorphism into a family of iterators, one for each of the mutually-recursive datatypes. Then we have to show the satisfaction of the Horn clauses under this interpretation $\mathcal{I}$, which is realized by processing the satisfaction of each Horn clause like in the step for satisfaction of Horn clauses in the quotient term model, and then looking up corresponding facts which the user should have proven.

Now the assumptions of the iteration principle can be discharged, and we transform it as follows:

- We unroll the quantification over operation symbols into a conjunction.
- We evaluate the arities of the operation symbols and applications of $map$ over them.
- We unroll the quantifications over tuples $pxs, xs$ into quantifications over elements, replacing the tuple variables by concrete tuples of these elements. We evaluate the application of $map$ over tuples.
- We transfer the resulting proposition under the constructed isomorphism, in particular mapping the generic iterator on quotient terms to its versions on the datatypes.
- We replace HOL with Pure connectives.

Animation of the meta-theoretic result that relational consequences of the Horn theory are preserved under iteration works the same, just the conclusion to be transformed is different.

## 7.6 Related Work

Horn clauses [13] are an important concept in algebraic specification and logic programming. Our meta-theory of non-free datatypes is based on the existence of initial term models, which is a characteristic property [17] of universal Horn theories and our reason for choosing them.

Isabelle features a package [4] for the construction of nested free datatypes. This takes place in the usual manner by animating the construction principle with ML code. In our case this approach seems harder: the resulting inductive definitions characterizing the equational consequences of a Horn theory are not as uniform. The function package [15] is well integrated with the datatype package and allows the easy construction of many

terminating functions. We only offer non-nested non-free datatypes with an iteration principle. Our meta-theory and animation naturally make use of free datatypes to represent the terms, ground terms, Horn clauses and syntactic signatures.

We follow the initial algebra approach [28] to the algebraic specification of abstract data types: the term algebra is quotiented under equational consequences of a Horn theory and serves as the initial model of the specification. More precisely, we select it as the representant of the isomorphism class of initial $\Sigma$-algebras that satisfy the Horn theory. The unique morphisms to other $\Sigma$-algebras that satisfy the Horn theory, are constructed with the iteration principle. We provide no structuring concepts [28] in specifications. In our case we do not have to assign a separate semantics to the data types: they are their semantics, as constructed by the meta-theory. Parameter conditions in specifications make the connection to fixed semantic concepts in existing types. As far as we know, this definitional approach to non-free datatypes is new. Furthermore the meta-theory is also formalized, making it more open to experimentation. Isabelle is a mature theorem prover, so many tools are available for reasoning about the satisfaction of specifications. Also these tools do not have to be trusted, because all inferences are checked by the LCF kernel.

# 8 Conclusion

We have seen how a large class of packages can be recast as meta-theories parameterized on signatures that are interpreted over a pseudo-universe. Instantiating and postprocessing these meta-theories corresponds to the application of packages. We have described generic infrastructure to specify such postprocessing steps in general.

The use of certain algorithmic rule systems for processing theory content has been successful in our application. In particular the transfer of theory content under isomorphisms is a problem for which have sketched a very flexible and lightweight solution. Using meta recursion instead of traditional tactics provides a more robust setting for the deterministic derivation of facts. The generic error messages and traces of rule applications made debugging much easier than when using traditional techniques.

Forward rules seem to be a very fitting primitive to describe the stepwise generation of theory content, largely because they make the collection of information with data dependencies comfortable. Inspecting the current pool of facts, as a form of abstract heap image, has proven to be a vital and high-level debugging tool.

While it is not clear yet whether the approach helps interested users in building packages, it has allowed the author to build a package for non-freely generated datatypes in comparatively short amount of time (roughly 2 months of conception and 1 month of implementation). Realizing the necessary quotienting under equational consequences of a Horn theory seems like a tough problem to solve with the traditional programming techniques, because elaborate inductive definitions and proofs about them are necessary. We on the other hand can formulate clean meta-theorems, which merely have to be postprocessed in a largely standard way.

We want to note here that traditionally the meta-theories underlying packages are of an informal nature. They rely on the dynamic checking of inferences in the LCF kernel to catch errors before they lead to wrong consequences. This does not eradicate errors, but shifts them into the realm of inconveniences. Making the meta-theories precise and equipping them with an algorithmic reading will certainly contribute to the reliability and extensibility of a theorem prover.

## 8.1 Future Work

We mention future work and connections to ongoing work.

The groundness and dependency analyses are essentially first-order. To satisfy their demands we had to introduce notable amounts of code duplication in the form of forward rules. Of course we would like to explore higher-order versions of these analyses, which promise to enable code reuse with higher-order programming techniques.

The input language of the constructed datatype package and its performance are suboptimal. These aspects can be greatly improved with a little more work. Specialized error

messages are almost non-existent. Relying on the generic error messages provided by the animation machinery is better than no feedback mechanism at all, but users will have trouble identifying their relevance. Using primitives for explaining errors in meta recursion clauses and forward rules might prove to be an alternative to the usual ML code.

Establishing intensional properties about meta recursion derivations for a judgement is interesting, in particular for properties guaranteeing global invariants that are not expressible in the semantics of the judgement. Twelf-style inductions [10] will probably be a good source of inspiration here.

Stronger logics, such as extensions of ZFC with large cardinal axioms, feature proper universes that are closed under any construction. Pseudo-universes and the syntactic characterization of the objects relevant for a construction are not necessary. This allows the easy abstraction over objects that correspond to types and functions between them, and could lead to the comparatively easy formulation of meta-theories for packages [26] that employ notions from category theory in their constructions.

The infrastructure for animating algorithmic rule systems was designed by the author to make experimentation with soft-types easy and the formulation of generic soft-type systems possible. This effort is ongoing and regularly suggests the generalization of intensional analysis concepts. E.g. reinterpreting the meta-theoretic proof of internalized parametricity in PTS [5] as an algorithmic rule system for checking parametricity of soft-typed terms.

# Bibliography


[1] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Computer Science and Applied Mathematics. Academic Press, Inc., 1986.

[2] Clemens Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In *Mathematical Knowledge Management (MKM 2006), LNAI 4108*, pages 31–43. Springer-Verlag, 2006.

[3] Gilles Barthe, Venanzio Capretta, and Olivier Pons. Setoids in type theory, 2000.

[4] Stefan Berghofer and Markus Wenzel. Inductive datatypes in HOL - lessons learned in formal-logic engineering. In *Theorem Proving in Higher Order Logics: TPHOLs 1999, LNCS 1690*. Springer, 1999.

[5] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Parametricity and dependent types. In Paul Hudak and Stephanie Weirich, editors, *ICFP*, pages 345–356. ACM, 2010.

[6] Maksym Bortin, Einar Broch Johnsen, and Christoph Lüth. Structured formal development in Isabelle. *Nordic Journal of Computing*, 13:1– 20, 2006.

[7] Lukas Bulwahn. Code generation from inductive predicates in Isabelle/HOL. Master's thesis, Technische Universität München, 2009.

[8] Mike Gordon. From LCF to HOL: a short history. In *Proof, Language, and Interaction*, pages 169–185. MIT Press, 2000.

[9] Florian Haftmann and Makarius Wenzel. Constructive type classes in Isabelle. In Thorsten Altenkirch and Connor McBride, editors, *Types for Proofs and Programs (TYPES 2006)*, volume 4502 of *LNCS*. Springer, 2007.

[10] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *J. Funct. Program.*, 17(4-5):613–673, 2007.

[11] Peter V. Homeier. A design structure for higher order quotients. In Joe Hurd and Thomas F. Melham, editors, *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 130–146. Springer, 2005.

[12] Peter V. Homeier. The HOL-Omega logic. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *LNCS*, pages 244–259. Springer, 2009.

[13] Alfred Horn. On Sentences Which are True of Direct Unions of Algebras. *The Journal of Symbolic Logic*, 16(1):14–21, 1951.

[14] Cezary Kaliszyk and Christian Urban. Quotients revisited for Isabelle/HOL. In *the Proc. of the 26th ACM Symposium On Applied Computing*, 2011.

[15] Alexander Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic*. PhD thesis, Technische Universität München, 2009.

[16] Alexander Krauss and Andreas Schropp. A mechanized translation from higher-order logic to set theory. In Matt Kaufmann and Lawrence Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *LNCS*, pages 323–338. Springer, 2010.

[17] Johann A. Makowsky. Why horn formulas matter in computer science: Initial structures and generic examples. *J. Comput. Syst. Sci.*, 34(2/3):266–292, 1987.

[18] Gopalan Nadathur and Dale Miller. An overview of Lambda-Prolog. In *ICLP/SLP*, pages 810–827, 1988.

[19] Tobias Nipkow. Constructive rewriting. *Computer Journal*, 34:34–41, 1991.

[20] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[21] Steven Obua and Sebastian Skalberg. Importing HOL into Isabelle/HOL. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning (IJCAR 2006)*, volume 4130 of *LNAI*, pages 298–302. Springer, 2006.

[22] L. C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In *In Proceedings of the 12th International Conference on Automated Deduction*, pages 148–161. Springer, 1994.

[23] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[24] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206. Springer-Verlag LNAI, 1999.

[25] The Coq Development Team. The Coq Proof Assistant — reference manual version 8.3, 2010.

[26] Dmitriy Traytel, Andrei Popescu, and Jasmin Christian Blanchette. Foundational, compositional (co)datatypes for higher-order logic — category theory applied to theorem proving. Submitted, 2012.

[27] Vladimir Voevodsky, Andreij Bauer, Thomas Streicher, Michael Shulman, Richard Garner, Thierry Coquand, Nicola Gamino, Peter Aczel, Michael A. Warren, Pieter Hofstra, and Benno van den Berg. Mini-workshop: The homotopy interpretation of constructive type theory, 2011.

[28] Martin Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chapter 13. Elsevier Science Publishers B.V., 1990.