Fakultät für Informatik
Lehrstuhl für Logik und Verifikation

Types, Abstraction and Parametric Polymorphism in Higher-Order Logic

Ondřej Kunčar

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

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Die Dissertation wurde am 09.12.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 05.03.2016 angenommen.

ABSTRACT

Thinking and working abstractly has been one of the most fruitful instruments of modern society. This thesis aims to promote types as a powerful tool for abstraction in the interactive theorem prover Isabelle/HOL by making them definitional and easy to use.

In order to keep our prover trustworthy, we are allowed to define new types on the logical level by only one elementary rule.

Our first contribution is to have made this rule consistent. This is a prerequisite to use types as a tool for abstraction. Isabelle extends its higher-order logic (HOL) with Haskell-like type classes, which require overloaded constants. It was an open question under which criteria overloaded constant definitions and type definitions can be combined together while still guaranteeing consistency. We have defined such criteria (which are even decidable), developed novel semantics, and provided a semantic explanation of the correct interplay of these features.

Our second contribution is an automation for building libraries of abstract types (subtypes or quotients). This is a tedious task if using only the elementary rule. Therefore we developed two new tools—Transfer and Lifting. A novel aspect of Transfer, which transfers propositions across related types, is that the transferring can be organized and largely automated using Reynolds's relational parametricity. Lifting, which automatically lifts operations across partial quotients, is the first quotient tool that supports the whole type universe of HOL.

We also studied how to make abstraction more concrete (data refinement) and how to make types more widely usable (local type definitions in proofs).

About 3500 uses of Transfer and Lifting in Isabelle's theories confirm that Isabelle's users started reasoning more abstractly thanks to our work.

ACKNOWLEDGMENTS

First, I would like to thank Tobias Nipkow for giving me the opportunity to work in his group, for creating an open and relaxed working environment, and the freedom that allowed me to work on my ideas without pressure.

I was delighted to hear that the father of the next 700 theorem provers, Larry Paulson, accepted to referee my thesis.

It was a pleasure to meet and work with all these people that were or still are part of the Isabelle group in Munich: Jasmin Blanchette, Sascha Böhme, Julian Brunner, Lukas Bulwahn, Martin Desharnais, Manuel Eberl, Holger Gast, Florian Haftmann, Johannes Hölzl, Brian Huffman, Lars Hupel, Fabian Immler, Cezary Kaliszyk, Gerwin Klein, Alexander Krauss, Peter Lammich, Silke Müller, Eleni Nikolaou-Weiß, Lars Noschinski, Helma Piller, Andrei Popescu, Chris Poskitt, Dmitriy Traytel, Thomas Türk, Christian Urban, Makarius Wenzel, and Simon Wimmer.

I would like to express my gratitude to my family and my friends (especially Silke Müller) for their unfailing and selfless support.

Jasmin Blanchette, Manuel Eberl, Johannes Hölzl, Fabian Immler, Tobias Nipkow and Dmitriy Traytel read parts of the draft of my thesis and provided many helpful comments, for which I thank them.

A special appreciation goes to two people that I worked closely with: Brian Huffman and Andrei Popescu. I started working with Brian on Transfer and Lifting tools soon after my move to Munich and thanks to him I learned how to stop worrying and tame Isabelle. After Brian left, I adapted to the Romanian time zone and worked with Andrei on the foundations of Isabelle/HOL. They both inspired me by their ability to get out of a shell of technocratic engineering and see the forest for the trees.

Dmitriy Traytel, my officemate for the longest part of my Ph.D., deserves a special thanks for being a great listener when I kept rambling on about philosophical problems, local_theories and a small kernel. He always provided first feedback on my ideas and answered my questions. Although none of our individual discussions was worth publishing in *Revue de métaphysique et de morale*, they meant a lot to me as a whole.

Cezary Kaliszyk and Christian Urban reimplemented Peter Hommier's Quotient package in Isabelle. Alexander Krauss came up with the idea to use it to automate Florian Haftmann's approach to executing subtypes.

Andreas Lochbihler and René Thiemann were among the first users of Transfer and Lifting packages and provided helpful feedback.

My research was financed by the Deutsche Forschungsgemeinschaft from the grants NI 491/10-2, NI 491/13-2 and NI 491/13-3.

CONTENTS

*Science is interesting, and if you don't agree you can f*** off.*

— Richard Dawkins (2006)[1]

1 INTRODUCTION

This thesis describes work on making types a powerful definitional tool for abstraction in the interactive theorem prover Isabelle/HOL.

1.1 motivation

When somebody utters the term *types*, the association that most probably comes to mind is types in programming languages as a tool to rule out certain (malformed) programs. For example, if you try to assign 5 (a value of type integer) to a variable of type string, the compiler must emit an error, or alternatively an implicit conversion must be used. In either case, we have to know the corresponding types. B. Pierce [80] gives this as a first example of what types are good for: detecting errors. Other applications that he mentions are abstraction, documentation and efficiency.

Types (or type systems) in general is a broad field of study, which, besides computer science, comprises fields such as philosophy, logic and mathematics. In this thesis, I concentrate on types in the context of theorem proving, which can be seen as a boundary field of logic, mathematics and computer science.

What are types good for when we use Isabelle/HOL? There is a slight shift from the world of programming languages. I argue that detecting errors is not the main usage of types in higher-order logic. We could remove all the custom types from our formalizations and use only the built-in function and infinite type, and the correctness of our results would still be guaranteed by the inference kernel. We use types as a tool for *abstraction* instead. When I create the type of rational numbers in Isabelle/HOL, I do not do it because I am afraid that mischief could happen without this type. I do it because I want to carve out the very concept of rational numbers from the chaos of all elements of my logic and encode the precondition that the denominator should be nonzero into the type.

We have to ensure that our terms are type correct, i.e., the usage of the introduced abstractions is sound. In this sense, types still help to rule out

1 Richard Dawkins was quoting a former editor of New Scientist Magazine, who is as yet unidentified.

flawed inputs, but the existence of type checking is just a consequence of the primary motivation (abstraction) and is not the reason for having the concept of types in the first place.

Another (but tightly connected) way to view an abstraction is to see it as a tool for hiding implementation details. Without the type of rational numbers, I would have to work with the pair of integer numbers (a, b) such that if $a * d = b * c$ for some other pair (c, d), then I would think of (a, b) and (c, d) as elements representing the same number.

There exist many ways of achieving abstraction via types. I will concentrate on two most prominent ones in my thesis: *subtypes* and *quotients*. Let me briefly introduce them:

- subtypes – it is a concept for creating types with invariants (or specializing/restricting already existing types). Examples include types such as lists with nonrepetitive elements, binary trees with a restricted shape or functions that are constant almost everywhere. The invariant is encoded in the type and therefore we do not have to carry it around as an explicit assumption in our theorems. For example, removing the first occurrence of a specified element from a list gives us a list without this element only under the assumption that the list does not contain repetitive elements. Encoding this assumption into a type as an invariant and working with lists of this type gives us a theorem without the extra assumption.

- quotients – they allow us to replace a comparison for equivalence on the original type by a comparison for equality on the quotient type. Equality reasoning is well understood and usually better supported than reasoning modulo equivalence relations.

We will encounter two additional cases of abstract types in my thesis: partial quotients and type copies. Partial quotients are a combination of subtypes and quotients, i.e., we will define equivalence classes only for a strict subset of the original elements. Type copies are merely a tool to create a new name for an already existing type, which is useful in a situation when we want to treat the type differently in different contexts. A typical example is the type of partial functions, which we, depending on the context, can treat as an ordinary function or, for example, an abstraction of a hash table.

Interactive theorem proving systems based on higher-order logic à la Gordon [24] offer only one primitive way to definitionally introduce a new type (or rather a type constructor). Using Figure 1 on the facing page, I will describe this mechanism informally for the moment: let us consider a type σ and a set of its elements $S : \sigma\ \mathsf{set}$[2] such that S is nonempty (which

2 Most often we specify S as $\{x : \sigma \mid inv\ x\}$ by a unary predicate $inv : \sigma \to \mathsf{bool}$ representing an invariant on σ.

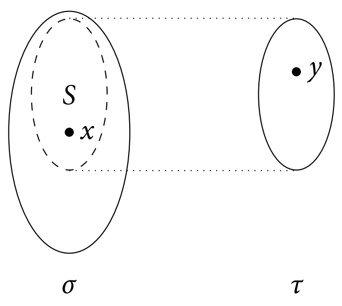


Figure 1: Creating a subtype

is witnessed by x in the picture). The mechanism declares a new type τ and defines τ as S. Thus τ is a restriction of σ and morally, all elements of type τ could be seen as elements of σ.

The complication is that the type system of Isabelle/HOL does not contain any notion that would capture the fact that τ is a specialization (a subtype) of σ. The types τ and σ are logically independent entities, whose parts just happen to be isomorphic; more precisely, τ is isomorphic to S. Practically, that means if we have $y : \tau$, we cannot obtain $y : \sigma$ and similarly if $x : \sigma$ and moreover $x \in S$, we cannot obtain $x : \tau$.

Therefore, if I define, for example, a type of lists with nonrepetitive elements as a restriction of the type list—let us call the new type dlist—we cannot automatically obtain functions operating on dlist as a restriction of the functions on list. For example, head : α list $\rightarrow$ α cannot be used for values of type α dlist. A similar restriction applies to theorems about lists, which we cannot directly use for lists with nonrepetitive elements since they talk about a different type.

How do we then proceed if we want to use a new abstract type? I mentioned that we define τ as S. Let me state this more precisely: in fact, we postulate that there exist two functions Abs : $\sigma \rightarrow \tau$ and Rep : $\tau \rightarrow \sigma$ such that Abs and Rep are isomorphisms between τ and S; see Figure 2 on the next page. This means that if we want to transform values from τ to σ and backwards, we have to *explicitly* use the morphisms Abs and Rep. For example, dhead x = head(Rep x) is a definition of the head function for dlist in terms of head for list. The need to use the morphisms explicitly leads to a tedious work, and in a higher-order setting, it can be difficult to even find the right combination of the morphisms.

Going back to the beginning of the chapter, where we talked about how types are an important technique for achieving abstraction in HOL, we realize that the explicit casting through the morphisms is a major hindrance for the usage of this technique. It was the main objective of mine to create new definitional commands and proof methods that would allow us to lift the limitation. I developed such commands and methods together with collaborators. In order to present the essence of

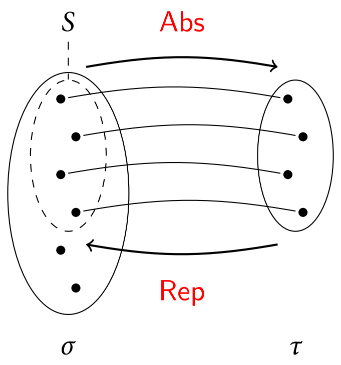


Figure 2: A subtype and its morphisms

the complication, I showed you only the case where we deal with subtypes, but we focused on a uniform solution for other ways of abstraction as well, especially for quotients. We grouped our new definitional commands and proof methods into two tools—Transfer and Lifting.

If we look at the title of this thesis, "Types, Abstraction and Parametric Polymorphism in Higher-Order Logic", we can see that besides types and abstraction it also mentions *parametric polymorphism*. It is a crucial part of our solution, which I will demonstrate informally now. We already saw that theorems about lists cannot be used for dlist directly. Let us consider a theorem $\forall x : \alpha\ \mathsf{list}.\ x \neq [\,] \longrightarrow (\mathsf{head}\ x)\ \#\ (\mathsf{tail}\ x) = x$, which says that destructing a nonempty list $[a_1, \ldots, a_n]$ into its first element a_1 (by head) and the rest of it $[a_2, \ldots, a_n]$ (by tail) and prepending a_1 back to $[a_2, \ldots, a_n]$ by the cons operator # is the identity. Using that theorem about list, we wish to obtain an analogous theorem for dlist: $\forall x : \alpha\ \mathsf{dlist}.\ x \neq [\,]_{\mathsf{d}} \longrightarrow (\mathsf{dhead}\ x)\ \#_{\mathsf{d}}\ (\mathsf{dtail}\ x) = x$. In order to do so, we have to know not only that the list operations are related to dlist operations, such as head is related to dhead or $[\,]$ to $[\,]_{\mathsf{d}}$, but also that universal quantification $\forall$ and equality = behave nicely, i.e., they behave in essentially the same way for list and dlist. This is where a notion of restricted parametricity for $\forall$ and = comes into play.

Besides the direct use case—building an abstraction—there are two other nonobvious use cases that I describe in this thesis.

The Isabelle/HOL users like to define specifications and prove their correctness only later to obtain an executable version of the specification possessing the same correctness guarantees. In principle, any theorem with = at the top can be used as an executable equation but not a conditional theorem of the form $inv\ x \longrightarrow c = t(x)$. I already mentioned that encoding an invariant into a type removes an assumption about this invariant from the theorem statement, which is exactly what we would do here. In fact, this use case was the first motivation that triggered my study of types in Isabelle/HOL.

The other use case comes from the observation that type-based theorems are easier to prove whereas set-based theorems are more general and easier to apply. Let me explain the statement on this example: Consider the following HOL statements, where we explicitly indicate the top quantification over types:[3]

1. $\forall\alpha.\ \exists xs : \alpha\ \mathsf{list}.\ P\ xs$
2. $\forall\beta.\ \forall A : \beta\ \mathsf{set}.\ A \neq \varnothing \longrightarrow (\exists xs \in \mathsf{lists}\ A.\ P\ xs)$

We call 1. a type-based theorem and 2. a set-based theorem. Type-based theorems are easier to prove than the set-based ones because types are more rigid than sets and therefore enjoy better automation. Set-based theorems are more flexible because users often define their structures (measure functions, algebras, etc.) only on a particular subset of their types, not necessarily on the whole type. From the set-theoretic semantics point of view, the two statements are equivalent.

Ideally, the users would develop their theories in a type-based fashion, and then export the main theorems as set-based statements. Unfortunately, this operation is not supported by the current system—let us assume 1. and try to prove 2: we would fix $A : \beta\ \mathsf{set}$ and assume that A is nonempty and define a new type α corresponding to A. Now we are in a standard subtype situation and we could complete the proof by Transfer and Lifting. Unfortunately, the current system does not allow us to define a type (depending on local data—A) inside of a proof. My main goal was to develop an extension of the logic (in the form of a new rule) that would allow us to simulate a local typedef with some restrictions[4] that would still permit our use case of exporting set-based statements.

While working on this extension, I discovered that typedef in Isabelle/HOL is not a definitional extension. What does it mean? As I already mentioned, typedef is the only way to introduce new types definitionally (i.e., not axiomatically) in the family of HOL-based provers. A. Pitts [81] proved that the traditional typedef is a consistency preserving and conservative theory extension. But Isabelle/HOL diverges from the standard HOL-based systems and brings its own touch to HOL: *ad hoc overloaded constant definitions*. Briefly, overloading allows us to introduce independent definitions for different type instances of the given (declared) constant; e.g., let us declare a new constant $c : \alpha$ and define its meaning for nat and bool separately: $c : \mathsf{nat} \equiv 42$ and $c : \mathsf{bool} \equiv \mathsf{True}$. This feature allows us to implement Haskell-like type classes in Isabelle/HOL.

Unfortunately, the combination of ad hoc overloading and typedef causes problems—I could prove False in Isabelle/HOL. The proof exploits

3 Recall that the Isabelle/HOL constant $\mathsf{lists} : \alpha\ \mathsf{set} \to \alpha\ \mathsf{list}\ \mathsf{set}$ takes a set A and returns the set of lists whose elements are in A.

4 There are no dependent types in Isabelle/HOL and we do not want to step outside this restriction.

the fact that we are allowed to create a cycle in definitions: first we declare a constant c : bool, then we define a type σ in terms of the constant, and then we overload c : bool in terms of σ. Thus we have a cycle c : bool $\rightsquigarrow$ $\sigma \rightsquigarrow c$: bool. With a little bit of inventiveness, we can define c and σ such that $c = \neg c$.

This is not an acceptable situation for most Isabelle users. The users expect that when they define new constants and types, they cannot introduce inconsistency into the system. This is in accordance with the best traditions of the LCF methodology [25]. Putting the definitional mechanism of Isabelle onto firm (foundational) ground became a major goal of my thesis. If we want to promote types as good tools for abstraction in HOL, our types must be defined only in a consistent manner.

1.2 contributions

This section describes contributions of this thesis. The listed contributions were achieved by me and my collaborators, and the particular credit is given separately in each chapter.

1.2.1 Consistent Foundation for Isabelle/HOL

The first contribution of this thesis is a development of a consistent foundation for Isabelle/HOL. This contribution comprises the following parts:

- A custom semantics for polymorphic HOL with ad hoc overloading, which is a variation of the HOL standard model specifically to cater for overloading.

- A new well-formedness criterion for definitions: they must not overlap and the substitutive closure of the definition-dependency relation, which now factors in dependencies between types and constants as well, must terminate.

- A proof that our notion of well-formedness implies the existence of a model with respect to our custom semantics and thus ensures relative consistency of Isabelle/HOL.

- An additional restriction (a certain notion of composability) under which the well-formedness check is decidable. Termination of the definition-dependency relation is not, in general, decidable, but under composability it is equivalent to acyclicity, which is a decidable problem.

- A quadratic algorithm (in the size of the dependency relation) that decides the well-formedness and composability criterion and a proof of its completeness and correctness.

1.2.2 Automation for Building Abstract Types

The second contribution is the development of tools that streamline the process of building abstract types in Isabelle/HOL. We created two tools called Transfer and Lifting.

The approach crucially depends on additional structure on type constructors and builds an infrastructure for tracking it: for example, we need a notion of a map function, a relator and a predicator associated to a type constructor and how these functions are connected, or distributivity and monotonicity of the relator.

As a part of Transfer, we developed proof tactics that can derive theorems about the abstract types from the theorems about the original type and the other way around. The main contribution of Transfer is that transferring of properties across related types can be organized and largely automated using Reynolds's relational parametricity.

As a part of Lifting, we developed a command that is responsible for defining a new constant on the abstract type in terms of a corresponding operation on the original type. The main contribution of Lifting is that lifting of operations to abstract types is largely independent of the construction of these types— whether it is a subtype or a quotient type, for example. Other contributions are:

- We support type copies, subtypes, quotients and partial quotients. There is no limitation on the structure of the type: higher-order types are supported as well as arbitrary nesting of abstract types.

- Automation: the correctness proof obligation that the original operation respects the abstraction is proved automatically in many cases by our reflexivity prover. For type copies, the automation is guaranteed to succeed. For subtypes, the obligation is rewritten from a general statement into the form using only invariants, which is simpler to prove.

- Integration with the Transfer tool: a transfer rule relating the new constant with the original one is automatically proved; if the user can prove that the original operation is parametric, the tool proves a fully parametrized transfer rule.

- Integration with the code generator: code equations for the new constant are proved if they lie in the supported fragment of the code generator. Moreover, we developed a procedure that by introducing additional types and transforming the equations, enables us to execute broader set of functions than it is directly supported by the code generator.

1.2.3 Use Cases of Transfer and Lifting

The third contribution is two inventive deployments of types (and Transfer and Lifting) in addition to the direct use cases—building abstraction.

Firstly, this thesis demonstrates how the problem of performing data refinement for code generation via a data structure that possesses an invariant, can be solved by creating an auxiliary subtype. Moreover, as already mentioned, it is sometimes necessary to distinguish between, for example, the type of partial functions $\alpha \to \beta$ option as a type of an ordinary function and an abstraction of a table that should be refined to a red-black tree. We propose to address this issue by introducing a new type constructor for $\alpha \to \beta$ option (a type copy). Transfer and Lifting provide a crucial automation for both problems here.

Secondly, we proposed a technique to automatically prove set-based theorems from type-based ones. On this account, we formulated an extension of Isabelle/HOL's logic in the form of a new rule, proved its soundness and provided a prototypical implementation of it.

1.3 publications

Most of the contributions included here have been published in conference and workshop papers. This thesis is based on the following ones:

- F. Haftmann, A. Krauss, O. Kunčar, and T. Nipkow. Data Refinement in Isabelle/HOL. in S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *ITP 2013*. Vol. 7998, in LNCS, pp. 100–115. Springer, 2013

- B. Huffman and O. Kunčar. Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. in G. Gonthier and M. Norrish, editors, *CPP 2013*. Vol. 8307, in LNCS, pp. 131–146. Springer, 2013

- O. Kunčar and A. Popescu. From Types to Sets in Isabelle/HOL. in, *Isabelle Workshop 2014*, 2014. URL: http://www21.in.tum.de/~kuncar/documents/kuncar-popescu-itp2014.pdf

- O. Kunčar. Correctness of Isabelle's Cyclicity Checker: Implementability of Overloading in Proof Assistants. In X. Leroy and A. Tiu, editors, *CPP 2015*, pp. 85–94. ACM, 2015

- O. Kunčar and A. Popescu. A Consistent Foundation for Isabelle/HOL. in C. Urban and X. Zhang, editors, *ITP 2015*. Vol. 9236, in LNCS, pp. 234–252. Springer, 2015

1.4 structure of this thesis

Although the history of achieving the results in this thesis went chronologically from problems involved in data refinement to creating general tools for abstraction in HOL and continued with a detour into dangerous waters of models and cyclic definitions, I present the results in a more traditional theory–tools–use-cases manner:

- Chapter 2 introduces syntax, inference rules and definitional principles of Isabelle/HOL.
- Chapter 3 shows how to make the definitional principles consistent and provides semantics of Isabelle/HOL.
- Chapter 4 describes the Transfer tool and its connection to relational parametricity.
- Chapter 5 presents the Lifting tool, its integration with the code generator and partial quotients representing abstract types uniformly.
- Chapter 6 shows how to use abstract types and Transfer and Lifting for data refinement for code generation.
- Chapter 7 formulates an extension of Isabelle/HOL's logic to automatically prove set-based theorems from type-based ones.
- Chapter 8 summarizes our work and proposes future work.

Related work is listed in each chapter separately.

My God—it's full of stars!

— Arthur C. Clarke (1968)

Okay, but it seems to me there's a meta-something in there.

— Gottfried Barrow (2012)

2 ISABELLE/HOL

Isabelle/HOL [72] is an interactive theorem prover that implements classical higher-order logic based on Church's simple type theory [17] with Hilbert choice, the axiom of infinity, rank-1 polymorphism, axiomatic type classes, ad hoc overloaded constant definitions and type definitions.

As in many PhD theses and papers concerned with Isabelle, it is not my primary goal to provide an accurate description of all the minutiae of the system here. I will follow a general rule of thumb and focus on the aspects that matter for my thesis, gloss over unimportant ones and abstract over technicalities.

2.1 isabelle simplified

Since it is my goal to provide a model for the logic of Isabelle/HOL in the next chapter, this chapter has to prepare the ground for this. Therefore the description must be more precise in many regards and focused on the fundamental level of the system than the usual Isabelle/HOL's introductory text. In order to keep mine and reader's sanity, I will compensate for the increased complexity by leaving out the metalogic of Isabelle and axiomatic type classes. Let me briefly explain what these two features of Isabelle are and argue why omitting them is admissible.

2.1.1 The Metalogic

Isabelle was designed by Larry Paulson [79] to be a generic interactive theorem prover, in which the user could define its own logics of interest. This can be done by using the metalogic of Isabelle, called Isabelle/Pure, which serves here as a natural deduction framework that can embed other object logics, for example HOL or Zermelo–Fraenkel set theory.

Let me give you a taste of the metalogic by utilizing a simple example: we take one of the basic rules of HOL, modus ponens. In another representative of the family of HOL-based provers, HOL4, which implements the HOL logic directly in comparison to Isabelle, modus ponens is a function in the inference kernel. On the other hand in Isabelle/HOL, this rule is stated as a formula and introduced as an axiom when one defines HOL. The metalogic has its own set of operators representing universal quantification, implication and equality: $\bigwedge$, $\Longrightarrow$ and $\equiv$. Then the formula expressing modus ponens reads $\bigwedge P\,Q.\ (P \longrightarrow Q) \Longrightarrow P \Longrightarrow Q$, where $\longrightarrow$ is the HOL implication.

The mechanism of specifying and embedding Isabelle/HOL in the Isabelle/Pure is merely a technical detail. In the following development, we will abstract over this detail and take Isabelle/HOL as a standalone logic without inspecting how this logic was bootstrapped and describe it in the usual Hilbert-system style of rules. Furthermore, we will use HOL formulas exclusively even at places where Isabelle experts would expect formulas with the metalogical operators. We cannot cause any harm by this abstraction in terms of what can or cannot be proved since Isabelle/Pure is an intuitionistic fragment of higher-order logic and Isabelle/HOL provides a strictly stronger logic. Thus, any reasoning carried out in Isabelle/Pure can be also internalized in our abstraction.

2.1.2 Type Classes

Type classes were introduced in Haskell by Wadler and Blott [94]—they allow programmers to write functions that operate generically on types endowed with operations. For example, assuming a type α which is a semigroup (i.e., comes with a binary associative operation $+$), one can write a program that computes the sum of all the elements in a nonempty α-list. Then the program can be run on any concrete type T that replaces α if T provides this binary operation $+$. Prover-powered type classes in Isabelle were introduced by Nipkow and Snelting [74] —they additionally feature verifiability of the type-class conditions upon instantiation: a type T is accepted as a member of the semigroup class only if associativity can be proved for its $+$ operation.

Isabelle's type classes rely on arbitrary ad hoc overloading and axiomatic type classes, two primitives of Isabelle's logic, as follows: to introduce the semigroup class, the system declares a global constant $+ : \alpha \to \alpha \to \alpha$ and defines an axiomatic type class that consists of the associativity predicate; then different instance types T are registered after defining the corresponding overloaded operation $+ : T \to T \to T$ and verifying the condition.

Overloading is most useful in conjunction with axiomatic type classes, but it really is an orthogonal feature, which is used also independently of type classes in Isabelle. Furthermore, it is overloading that has been causing consistency problems in Isabelle/HOL, not the axiomatic type classes. For these two reasons, we study overloading separately from the axiomatic type classes. Moreover, Wenzel [99] showed how to compile out axiomatic type classes by interpreting them as predicates on types in Isabelle/Pure and therefore, as he argues, this mechanism turns out to be just an addition to user convenience, without really changing expressiveness of the logic. This is why we will not deal with them in this text anymore.

2.2 syntax

In what follows, by "countable" I mean "either finite or countably infinite." Throughout the development, we fix the following:

- A countably infinite set TVar, of *type variables*, ranged over by α, β, γ.
- A countably infinite set Var, of *(term) variables*, ranged over by a, A, P, p, q, x, y, z.
- A countable set K of symbols, ranged over by k, called *type constructors*, containing four special symbols: "bool", "ind", "set" and "$\rightarrow$" (aimed at representing the type of booleans, an infinite type, a set type and the function type constructor, respectively).

We also fix a function $\mathsf{arOf} : K \rightarrow \mathbb{N}$ associating an arity to each type constructor, such that $\mathsf{arOf}(\mathsf{bool}) = \mathsf{arOf}(\mathsf{ind}) = 0$, $\mathsf{arOf}(\mathsf{set}) = 1$ and $\mathsf{arOf}(\rightarrow) = 2$. We define the set Type, ranged over by σ, τ, of *types*, inductively as follows:

- $\mathsf{TVar} \subseteq \mathsf{Type}$
- $(\sigma_1, \ldots, \sigma_n)\, k \in \mathsf{Type}$ if $\sigma_1, \ldots, \sigma_n \in \mathsf{Type}$ and $k \in K$ such that $\mathsf{arOf}(k) = n$

We use postfix notation for the application of an n-ary type constructor k to the types $\sigma_1, \ldots, \sigma_n$. If $n = 1$, instead of $(\sigma)k$ we write $\sigma\ k$ (e.g., σ list).

A *typed variable* is a pair of a term variable x and a type σ, written x_σ. Given $T \subseteq \mathsf{Type}$, we write Var_T for the set of typed variables x_σ with $\sigma \in T$. Finally, we fix the following:

- A countable set Const, ranged over by c, of symbols called *constants*, containing seven special symbols: "$\longrightarrow$", "$=$", "ε", "$\in$", "Collect", "zero", "suc" (aimed at representing logical implication, equality, Hilbert choice of some element from a type, set membership, set comprehension, zero and successor, respectively)
- A function $\mathsf{tpOf} : \mathsf{Const} \rightarrow \mathsf{Type}$ associating a type to every constant, such that:

$$\begin{aligned}
&\mathsf{tpOf}(\longrightarrow) = \mathsf{bool} \rightarrow \mathsf{bool} \rightarrow \mathsf{bool}\\
&\mathsf{tpOf}(=) = \alpha \rightarrow \alpha \rightarrow \mathsf{bool}\\
&\mathsf{tpOf}(\varepsilon) = (\alpha \rightarrow \mathsf{bool}) \rightarrow \alpha
\end{aligned}$$

$$\begin{aligned}
&\mathsf{tpOf}(\in) = \alpha \to \alpha\ \mathsf{set} \to \mathsf{bool}\\
&\mathsf{tpOf}(\mathsf{Collect}) = (\alpha \to \mathsf{bool}) \to \alpha\ \mathsf{set}\\
&\mathsf{tpOf}(\mathsf{zero}) = \mathsf{ind}\\
&\mathsf{tpOf}(\mathsf{suc}) = \mathsf{ind} \to \mathsf{ind}
\end{aligned}$$

We define the type variables of a type, $\mathsf{TV} : \mathsf{Type} \to \mathcal{P}(\mathsf{TVar})$, as expected: $\mathsf{TV}(\alpha) = \{\alpha\}$, $\mathsf{TV}((\sigma_1, \ldots, \sigma_n)\ k) = \bigcup_{1 \le i \le n} \mathsf{TV}(\sigma_i)$.

A *type substitution* is a function $\rho : \mathsf{TVar} \to \mathsf{Type}$; we let TSubst denote the set of type substitutions. Each $\rho \in \mathsf{TSubst}$ extends to a homonymous function $\rho : \mathsf{Type} \to \mathsf{Type}$ by defining $\rho((\sigma_1, \ldots, \sigma_n)\ k) = (\rho(\sigma_1), \ldots, \rho(\sigma_n))\ k$. We write $\tau[\overline{\sigma}/\overline{\alpha}]$ for $\rho(\tau)$ where ρ is a type substitution that sends each α_i to σ_i and every $\beta \neq \alpha_i$ to β.

We say that σ is an *instance* of τ *via a substitution* of $\rho \in \mathsf{TSubst}$, written $\sigma \le_\rho \tau$, if $\rho(\tau) = \sigma$. We say that σ is an *instance* of τ, written $\sigma \le \tau$, if there exists $\rho \in \mathsf{TSubst}$ such that $\sigma \le_\rho \tau$.

Two types τ_1 and τ_2 are called *orthogonal*, written $\tau_1 \ \#\ \tau_2$, if they have no common instance; i.e., for all τ it holds that $\tau \not\le \tau_1$ or $\tau \not\le \tau_2$.

Given $c \in \mathsf{Const}$ and $\sigma \in \mathsf{Type}$ such that $\sigma \le \mathsf{tpOf}(c)$, we call the pair (c, σ), written c_σ, an *instance of* c. A *constant instance* is therefore any such pair c_σ. We let CInst be the set of all constant instances, and we extend the notions of being an instance ($\le$) and being orthogonal (#) from types to constant instances, as follows:

$$c_\tau \le d_\sigma \text{ iff } c = d \text{ and } \tau \le \sigma \qquad\qquad c_\tau \ \#\ d_\sigma \text{ iff } c \neq d \text{ or } \tau \ \#\ \sigma$$

We also extend tpOf to constant instances by $\mathsf{tpOf}(c_\sigma) = \sigma$.

The tuple $(K, \mathsf{arOf}, \mathsf{Const}, \mathsf{tpOf})$, which will be fixed in what follows, is called a *signature*. This signature's *pre-terms* are defined by the grammar:

$$t = x_\sigma \mid c_\sigma \mid t_1\, t_2 \mid \lambda x_\sigma.\, t$$

Thus, a pre-term is either a typed variable, or a constant instance, or an application, or an abstraction. As usual, we identify pre-terms modulo alpha equivalence.

The typing relation for pre-terms $t : \sigma$ is defined inductively in the expected way:

$$\frac{}{x_\sigma : \sigma} \qquad \frac{}{c_\sigma : \sigma} \qquad \frac{t_1 : \sigma \to \tau \qquad t_2 : \sigma}{t_1\, t_2 : \tau} \qquad \frac{t : \tau}{\lambda x_\sigma.\, t : \sigma \to \tau}$$

A *term* is a well-typed pre-term, and Term, ranged over by f, g, S, s and t, denotes the set of terms. Given $t \in \mathsf{Term}$, we write $\mathsf{tpOf}(t)$ for its (uniquely determined) type and $\mathsf{FV}(t)$ for the set of its free (term) variables. We call t *closed* if $\mathsf{FV}(t) = \varnothing$. We let $\mathsf{TV}(t)$ denote the set of type variables occurring in t.

We can apply a type substitution ρ to a term t, written $\rho(t)$, by applying ρ to all the type variables occurring in t with the proviso that if two distinct bound variables become identified, we replace the term by an alpha-equivalent term where the variables stay distinct.

We say that a function $\delta : \mathsf{Var}_{\mathsf{Type}} \to \mathsf{Term}$ is a *term substitution* if $t : \sigma$ whenever $\delta(x_\sigma) = t$. We write $\delta(t)$ for an application of a term substitution δ to a term t and define it as a simultaneous replacement of each free variable x_σ in t by $\delta(x_\sigma)$ with the usual renaming of bounded variables if they get captured.[1] If $s_i : \sigma_i$, we write $t[\overline{s}/\overline{x_\sigma}]$ for $\delta(t)$ where δ is a term substitution sending each $x_{i\sigma_i}$ to s_i and every $y_\tau \neq x_{i\sigma_i}$ to y_τ.

A *formula* is a term of type bool. We let Fmla, ranged over by φ and χ, denote the set of formulas. When writing concrete terms or formulas, we take the following conventions:

- We omit redundantly indicating the types of the variables, e.g., we shall write $\lambda x_\sigma.\ x$ instead of $\lambda x_\sigma.\ x_\sigma$.
- We omit redundantly indicating the types of the variables and constants in terms if they can be inferred by typing rules, e.g., we shall write $\lambda x.\ (y_{\sigma\to\tau}\ x)$ instead of $\lambda x_\sigma.\ (y_{\sigma\to\tau}\ x)$ or $\varepsilon(\lambda x_\sigma.\ P\ x)$ instead of $\varepsilon_{(\sigma\to\mathsf{bool})\to\sigma}(\lambda x_\sigma.\ P_{\sigma\to\mathsf{bool}}\ x)$.
- We write $\lambda x_\sigma\ y_\tau.\ t$ instead of $\lambda x_\sigma.\ \lambda y_\tau.\ t$
- We apply the constants $\longrightarrow$, $=$ and $\in$ in an infix manner, e.g., we shall write $t_\sigma = s$ instead of $=\ t_\sigma\ s$. We use ε as a binder, i.e., we shall write $\varepsilon x_\sigma.\ t$ instead of $\varepsilon\ (\lambda x_\sigma.\ t)$. Finally, we shall write $\{x_\sigma \mid t\}$ instead of Collect $(\lambda x_\sigma.\ t)$.
- We may write $\longleftrightarrow$ for $=_{\mathsf{bool}\to\mathsf{bool}\to\mathsf{bool}}$.

2.3 deduction system

The formula connectives and quantifiers are defined in the standard way, starting from the implication and equality primitives:

$$\begin{array}{lcl}
\mathsf{True} & = & (\lambda x_{\mathsf{bool}}.\ x) = (\lambda x_{\mathsf{bool}}.\ x) \\
\mathsf{All} & = & \lambda p_{\alpha\to\mathsf{bool}}.\ (p = (\lambda x.\ \mathsf{True})) \\
\mathsf{Ex} & = & \lambda p_{\alpha\to\mathsf{bool}}.\ \mathsf{All}\ (\lambda q.\ (\mathsf{All}\ (\lambda x.\ p\ x \longrightarrow q)) \longrightarrow q) \\
\mathsf{False} & = & \mathsf{All}\ (\lambda p_{\mathsf{bool}}.\ p) \\
\mathsf{not} & = & \lambda p.\ p \longrightarrow \mathsf{False} \\
\mathsf{and} & = & \lambda p\ q.\ \mathsf{All}\ (\lambda r.\ (p \longrightarrow (q \longrightarrow r)) \longrightarrow r) \\
\mathsf{or} & = & \lambda p\ q.\ \mathsf{All}\ (\lambda r.\ (p \longrightarrow r) \longrightarrow ((q \longrightarrow r) \longrightarrow r))
\end{array}$$

1 Isabelle uses de Bruijn indices to represent bound variables, therefore no renaming is needed. But this is just an implementation detail.

It is easy to see that the above terms are closed and well typed as follows:

- $\mathsf{tpOf}(\mathsf{True}) = \mathsf{tpOf}(\mathsf{False}) = \mathsf{bool}$
- $\mathsf{tpOf}(\mathsf{not}) = \mathsf{bool} \to \mathsf{bool}$
- $\mathsf{tpOf}(\mathsf{and}) = \mathsf{tpOf}(\mathsf{or}) = \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$
- $\mathsf{tpOf}(\mathsf{All}) = \mathsf{tpOf}(\mathsf{Ex}) = (\alpha \to \mathsf{bool}) \to \mathsf{bool}$

As customary, we shall write:

- $\forall x_\alpha.\ t$ instead of $\mathsf{All}\ (\lambda x_\alpha.\ t)$
- $\exists x_\alpha.\ t$ instead of $\mathsf{Ex}\ (\lambda x_\alpha.\ t)$
- $\neg\ \varphi$ instead of not φ
- $\varphi \wedge \chi$ instead of and $\varphi\ \chi$
- $\varphi \vee \chi$ instead of or $\varphi\ \chi$

We consider the following formulas:

refl	=	$x_\alpha = x$
subst	=	$x_\alpha = y \longrightarrow P\ x \longrightarrow P\ y$
iff	=	$(p \longrightarrow q) \longrightarrow (q \longrightarrow p) \longrightarrow (p = q)$
True_or_False	=	$(b = \mathsf{True}) \vee (b = \mathsf{False})$
some_intro	=	$p_{\alpha\to\mathsf{bool}}\ x \longrightarrow p\ (\varepsilon\ p)$
mem_Collect_eq	=	$a \in \{x_\sigma \mid P\ x\} \longleftrightarrow P\ a$
Collect_mem_eq	=	$\{x_\sigma \mid x \in A\} = A$
suc_inj	=	$\mathsf{suc}\ x = \mathsf{suc}\ y \longrightarrow x = y$
suc_not_zero	=	$\neg\ (\mathsf{suc}\ x = \mathsf{zero})$

We let Ax denote the set of the above formulas, which we call *axioms*. The formulas refl and subst axiomatize equality and iff ensures that equality on the bool type behaves as a logical equivalence. The formula True_or_False makes the logic classical, some_intro axiomatizes Hilbert choice, and mem_Collect_eq and Collect_mem_eq postulate the set type is isomorphic to the type of unary predicates ($\alpha \to \mathsf{bool}$). Finally, suc_inj and suc_not_zero ensure that ind is an infinite type.

A *theory D* is a set of formulas and a *context* Γ is a finite set of formulas. The notation $\alpha \notin S$ (or $x \notin S$) means that the variable α (or x) is not free in any of the formulas in the set S. We define *deduction* as a ternary relation $\vdash$ between theories, contexts and formulas by the following set of *deduction rules*:

$$\frac{}{D, \{\} \vdash \varphi}\ [\varphi \in \mathsf{Ax} \cup D]\ (\textsc{Fact}) \qquad \frac{}{D, \Gamma \vdash (\lambda x_\sigma.\ t)\ s = t[s/x_\sigma]}\ (\textsc{Beta})$$

$$\frac{}{D, \{\varphi\} \vdash \varphi}\ (\textsc{Assum}) \qquad \frac{D, \Gamma_1 \vdash \varphi \longrightarrow \chi \qquad D, \Gamma_2 \vdash \varphi}{D, \Gamma_1 \cup \Gamma_2 \vdash \chi}\ (\text{MP})$$

$$\frac{D, \Gamma \vdash \chi}{D, \Gamma \smallsetminus \{\varphi\} \vdash \varphi \longrightarrow \chi}\ (\textsc{ImpI}) \qquad \frac{D, \Gamma \vdash f\, x_\sigma = g\, x_\sigma}{D, \Gamma \vdash f = g}\ [x_\sigma \notin \Gamma]\ (\textsc{Ext})$$

$$\frac{D, \Gamma \vdash \varphi}{D, \Gamma \vdash (\varphi[\overline{\sigma}/\overline{\alpha}])[\overline{t}/\overline{x_\tau}]}\ [\overline{\alpha} \notin \Gamma; \overline{x_\tau} \notin \Gamma; t_i : \tau_i]\ (\textsc{Inst})$$

Isabelle also contains a powerful higher-order resolution rule as another primitive. We will not deal with the rule here since it can be simulated by aforementioned rules. A motivated reader can find more about this rule and other references on this topic in Wenzel's thesis [98].

If Γ is empty, we will simply write $D \vdash \varphi$. If a theory D is fixed (notice that it does not change in the deduction rules), we will write $\Gamma \vdash \varphi$.

A theory D is called *consistent* if there exists φ such that $D \nvdash \varphi$ (or equivalently if $D \nvdash \mathsf{False}$).

2.4 definitional principles

Isabelle/HOL provides two primitives for making definitions: an overloaded constant definition primitive and a type definition primitive. Following the LCF methodology [25], we require that all the other definitional principles of Isabelle/HOL have to be defined in terms of these two.

2.4.1 Overloaded Constant Definition

In order to use overloading, the user has to first declare a constant. This is done in Isabelle as follows:

$$\texttt{consts}\ c : \sigma$$

With the proviso that c is a fresh constant symbol, the command extends the signature by a new uninterpreted constant c and sets $\mathsf{tpOf}(c) = \sigma$.

Overloading allows us to define the meaning of a declared constant for different type instances separately. The respective command is `defs`:

$$\texttt{defs}\ c : \tau = t$$

As usual, there are some restrictions under which this definition is accepted and the theorem $c_\tau = t$ is introduced. Let me first state conditions that the system can check without looking at other definitions—we call them local conditions: it must hold that c is a declared constant with type

σ, τ is an instance of σ, t is a closed term of type τ, and finally all type variables in t are present in τ. Let me highlight that we do not forbid that c occurs in t.

Before we proceed to the global conditions, examples of overloading could be helpful at this stage. Let us overload the symbol o, aimed to represent a neutral element, for natural numbers, products and functions:

Example 2.1.

$$\begin{aligned}
&\texttt{consts}\ \mathsf{o} : \alpha\\
&\texttt{defs}\ \mathsf{o}_{\mathsf{nat}} = \mathsf{zero}\\
&\texttt{defs}\ \mathsf{o}_{\alpha\times\beta} = (\mathsf{o}_\alpha, \mathsf{o}_\beta)\\
&\texttt{defs}\ \mathsf{o}_{\alpha\to\beta} = \lambda x_\alpha.\, \mathsf{o}_\beta
\end{aligned}$$

Notice that the definition of o for products contains o on the right-hand side and that it is not fully specified: we do not know how o is defined for α and β. This depends on what α and β are.

This way, one can get quickly into trouble. Without further constraints, nothing prevents us from conducting the following definitions:

Example 2.2.

$$\begin{aligned}
&\texttt{defs}\ \mathsf{o}_{\mathsf{nat}} = \mathsf{zero}\\
&\texttt{defs}\ \mathsf{o}_{\alpha\times\mathsf{nat}} = (\mathsf{o}_\alpha, \mathsf{one})\\
&\texttt{defs}\ \mathsf{o}_{\mathsf{nat}\times\alpha} = (\mathsf{one}, \mathsf{o}_\alpha)
\end{aligned}$$

This leads to an inconsistency: $(\mathsf{zero}, \mathsf{one}) = \mathsf{o}_{\mathsf{nat}\times\mathsf{nat}} = (\mathsf{one}, \mathsf{zero})$.

Another example causing a trouble is cycles in definitions:

Example 2.3.

$$\begin{aligned}
&\texttt{defs}\ \mathsf{o}_{\mathsf{nat}} = \ldots \mathsf{o}_{\mathsf{int}} \ldots\\
&\texttt{defs}\ \mathsf{o}_{\mathsf{int}} = \ldots \mathsf{o}_{\mathsf{nat}} \ldots
\end{aligned}$$

In general, we can misuse a cycle to create the following inconsistency: just define $c_{\mathsf{bool}} = \neg c_{\mathsf{bool}}$.

And finally let us consider the following example:

Example 2.4.

$$\texttt{defs}\ \mathsf{o}_{\alpha\times\beta} = \ldots \mathsf{o}_{(\alpha\times\beta)\times\beta} \ldots.$$

This example leads to an infinite descent if we try to figure out the meaning of the constant by unfolding. Definitions are traditionally justified by the notion that they can be eliminated by *unfolding* them, yielding a unique, finite (even if huge) term that contains no defined constant. Although it may be harmless, infinite descent breaks that argument.

In the systems where overloading is not supported, none of the harmful Examples 2.2 to 2.4 is accepted. They are disallowed by (1) an additional local condition that c cannot occur on the right side of its definition and (2) not allowing to separate declaration and definition of c (that is to say, we force a global condition (the only one) that when c is being defined, it must be a fresh symbol). Then one can straightforwardly use the unfolding argument to justify correctness of such definitions.

Isabelle has been using the following global conditions to make overloading a safe theory extension:

1. The definitions cannot overlap—their left-hand sides are not unifiable after renaming. This prevents Example 2.2 because $\alpha \times \mathsf{nat}$ and $\mathsf{nat} \times \alpha$ are unifiable.

2. If we treat the definitional equations as rewriting rules (oriented from left to right), this rewriting system has to terminate. This prevents Examples 2.3 and 2.4.

The two conditions guarantee that we can again use the unfolding argument and produce a unique, finite term that contains no defined constant. But does this guarantee correctness? I wrote a whole chapter (Chapter 3) to answer this question.

2.4.2 Type Definition

The type definition principle (or typedef) allows us to define a new type that is isomorphic to a nonempty subset of an already existing type. This is achieved by the `typedef` command

$$\texttt{typedef}\ \overline{\alpha}\,k = S \quad \langle \mathit{proof} \rangle$$

where S has type σ set and $\overline{\alpha}$ is a sequence of distinct type variables. If S is a closed term, k a fresh type constructor and all type variable in S are among variables in $\overline{\alpha}$, the command opens a proof environment and the user is required to prove that S is a nonempty set, i.e., $\exists x_\sigma.\ x \in S$.

If the the proof obligation could be discharged, the command extends the signature by a new type constructor k of arity $|\overline{\alpha}|$ and declares two constants $\mathsf{Abs}_{\sigma \to \tau}$ and $\mathsf{Rep}_{\tau \to \sigma}$ where τ denotes the new type $\overline{\alpha}\ k$. Finally, the command postulates that Abs and Rep are isomorphisms between S and τ, i.e., the following axioms are added to the system:

$$\forall x_\tau.\ \mathsf{Rep}\ x \in S$$
$$\forall x_\tau.\ \mathsf{Abs}\ (\mathsf{Rep}\ x) = x$$
$$\forall x_\sigma.\ x \in S \longrightarrow \mathsf{Rep}\ (\mathsf{Abs}\ x) = x$$

The interplay between σ, τ and the morphisms Abs and Rep can be seen in Figure 2 on page 4.

2.4.3 Derived Definitional Principles

Life of the working formalizer would be hard if they had to use only the two definitional primitives. Isabelle/HOL contains a whole range of derived definitional commands.

(co)inductive datatypes [12]. The command `datatype` and its cousin `codatatype` define types similar to (co)datatypes in functional programming languages by specifying their constructors. For example, we can define natural numbers, (finite) lists and streams:

$$\begin{aligned}&\texttt{datatype}\ \ \mathsf{nat} = \mathsf{Zero} \mid \mathsf{Suc\ nat}\\&\texttt{datatype}\ \ \alpha\ \mathsf{list} = \mathsf{Nil} \mid \mathsf{Cons}\ \alpha\ (\alpha\ \mathsf{list})\\&\texttt{codatatype}\ \ \alpha\ \mathsf{stream} = \mathsf{Stream}\ \alpha\ (\alpha\ \mathsf{stream})\end{aligned}$$

We use the following syntactic sugar for lists: $[]$ is Nil, $x \# xs$ is Cons $x\ xs$ and $[x_1, \ldots, x_n]$ is $x_1 \# \ldots \# x_n \# []$.

Nested and mutually recursive (co)datatypes are supported as well.

quotients [48]. Provided R is a binary relation of type $\sigma \to \sigma \to \mathsf{bool}$, all type variables in σ are present in $\overline{\alpha}$, a sequence of distinct type variables, and k is a fresh type constructor, the following command

$$\texttt{quotient\_type}\ \ \overline{\alpha}\ k = \sigma\ /\ R \quad \langle \mathit{proof} \rangle$$

opens a proof environment and asks the user to prove that R is an equivalence relation on σ. Only then the command defines a new type constructor k of arity $|\overline{\alpha}|$ and introduces Abs and Rep with the following properties (τ again denotes the new type $\overline{\alpha}\ k$):

$$\begin{aligned}&\forall x_\tau.\ \mathsf{Abs}\ (\mathsf{Rep}\ x) = x\\&\forall x_\sigma\ y_\sigma.\ R\ x\ y \longleftrightarrow (\mathsf{Abs}\ x = \mathsf{Abs}\ y)\end{aligned}$$

In other words, Abs is a left inverse for Rep and two elements are in the same equivalence class of R if and only if they are mapped to the same representative on τ—this is nothing else than to say that τ is a quotient of σ under R. Internally, Rep x is defined by Hilbert choice, which choses some element from x's corresponding equivalence class.

We define the type of integer numbers as follows

$$\texttt{quotient\_type}\ \ \mathsf{int} = \mathsf{nat} \times \mathsf{nat}\ /\ \mathsf{intrel} \quad \langle \mathit{proof} \rangle \tag{1}$$

where intrel $(x, y)\ (u, v) \longleftrightarrow x + v = u + y$. Thus we interpreted the representation value (u, v) as the integer number $u - v$.

The quotient_type can also construct partial quotients when the user provides a partial equivalence relation (and proves this fact). The newly defined type is then described by the following properties:

$$\forall x_\tau.\ R\ (\mathsf{Rep}\ x)\ (\mathsf{Rep}\ x)$$
$$\forall x_\tau.\ \mathsf{Abs}\ (\mathsf{Rep}\ x) = x$$
$$\forall x_\sigma\ y_\sigma.\ R\ x\ y \longleftrightarrow (R\ x\ x \wedge R\ y\ y \wedge \mathsf{Abs}\ x = \mathsf{Abs}\ y)$$

Thus, the reflexive part of R implicitly specifies for which elements of the representation type σ the quotient is constructed. As an example, we define the type of rational numbers as a partial quotient of integer pairs

$$\texttt{quotient\_type}\ \ \mathsf{rat} = \mathsf{int} \times \mathsf{int}\ /\ \mathsf{partial} : \mathsf{ratrel} \quad \langle \mathit{proof} \rangle \qquad (2)$$

where ratrel $r\ s = (\mathsf{snd}\ r \neq 0 \wedge \mathsf{snd}\ s \neq 0 \wedge \mathsf{fst}\ r * \mathsf{snd}\ s = \mathsf{fst}\ s * \mathsf{snd}\ r)$. The quotient is partial as we do not allow the denominator to be zero.

simple definitions [73, §2.7.2]. Simple definitions are the most basic tool to introduce a new constant. The user just specifies that c equals a term:

$$\texttt{definition}\ c : \sigma\ \texttt{where}\ c\ \overline{x} = t$$

Under the condition that c is not in t, the definition is simply translated to definitional primitives as follows:

$$\texttt{consts}\ \ c : \sigma$$
$$\texttt{defs}\ \ c_\sigma = \lambda \overline{x}.\ t$$

The translation gives us the usual correctness proviso: c is fresh, $\lambda \overline{x}.\ t$ is a closed term of type σ and all its type variables are in σ. The definition command is usually preferred to the consts and defs primitives.

(co)recursive functions [12, 51]. The primrec command defines primitive recursive functions on datatypes; e.g., the reverse functions for lists, where @ is the concatenation operator for lists:

$$\texttt{primrec}\ \ \mathsf{reverse} : \alpha\ \mathsf{list} \to \alpha\ \mathsf{list}\ \ \texttt{where}$$
$$\mathsf{reverse}\ [\,] = [\,]$$
$$\mathsf{reverse}\ (x\ \#\ xs) = \mathsf{reverse}\ xs\ @\ [x]$$

The main correctness condition is that each recursive call on the right side must strip one constructor from the argument. Then the recursion is well founded and the command can reduce the list of defining equations to a single equation via a corresponding datatype recursion combinator.

If the primrec condition is too strict, the user can resort to the fun command, which supports a larger class of terminating recursive functions with pattern matching by constructing explicit function graphs.

The primcorec is dual to primrec in the sense that each equation has to produce one constructor immediately before each co-recursive call.

other principles. There are other derived definitional principles in Isabelle/HOL; let me name inductive definitions, which allow us to specify sets and predicates by their introduction rules [76], and truly extensible records as a generalization of tuples [68].

2.5 tactics, methods and attributes

If we view Isabelle from the user's perspective, we cannot expect that the users prove new theorems by using directly the deduction system from Section 2.3. One of the often-used possibilities is to use more complex proof procedures called tactics. A *tactic* is a procedure that transforms proof states, which can have multiple goals. Tactic's complexity can range from single rules to sophisticated proof-search algorithms. One of the styles of proving theorems in Isabelle is the goal-oriented approach—first state what you want to prove as a proof state with one goal and then reason backwards by tactics until no goals are left.

In the proof text, we set a proof state by the command `lemma`, e.g.,

$$\texttt{lemma}\ 1_{\mathsf{nat}} = 1, \tag{3}$$

and we transform proof goals by *methods*, which, roughly speaking, we can view as tactics that are available from the proof text. The proof state (3) can be transformed into an empty state by applying the method simp:

apply simp

The method simp is Isabelle's simplifier, which knows about reflexivity. In this thesis, we developed a new method called transfer, which transforms a goal into an equivalent goal along a pair of related types.

One of the distinctive features of Isabelle is schematic proof states. A *schematic proof state* is a proof state that contains term (or type) schematic variables. Schematic variables are variables that are not fixed during proving and therefore tactics are allowed to substitute terms (or types) for them. Thus the statement we want to prove is effectively not known/fixed before we carry out the proof and we can synthesize the statement. We distinguish schematic variables from regular variables by prefixing their names by "?", for example $?a$. We did not introduce schematic variables in Section 2.2 because logically they are equivalent with nonschematic variables (one can transform the former to the latter and vice verse). They are different only operationally. The method transfer establishes a schematic proof state internally to synthesize the new equivalent goal.

Besides obtaining new theorems by reasoning backwards, we can also reason forwards. One way to do that is to use attributes. An *attribute* (used as a theorem transformer) is a procedure that transforms a theorem into another theorem. For example, if we have a theorem called less−add−one, we can apply the attribute simplified to it as follows:

$$\text{less−add−one}[\text{simplified}]$$

The attribute transforms the given theorem into its normal form (if there is one). In this thesis, we developed the attribute transferred, which transforms the given theorem into an equivalent theorem along a pair of related types but in the opposite direction than the method transfer.

2.6 sets and binary relations

Sets and binary relations are one of the most basic concepts in mathematics. In my thesis, I represent binary relations by the type $\alpha \to \beta \to \text{bool}$. I will present how some basic constants on those types are defined in Isabelle/HOL. Let us start with the set operations:

$$\begin{aligned}
\varnothing &: \alpha\ \text{set} \\
\text{UNIV} &: \alpha\ \text{set} \\
\text{Ball} &: \alpha\ \text{set} \to (\alpha \to \text{bool}) \to \text{bool} \\
\cup &: \alpha\ \text{set} \to \alpha\ \text{set} \to \alpha\ \text{set} \\
\cap &: \alpha\ \text{set} \to \alpha\ \text{set} \to \alpha\ \text{set} \\
- &: \alpha\ \text{set} \to \alpha\ \text{set} \to \alpha\ \text{set} \\
\text{insert} &: \alpha \to \alpha\ \text{set} \to \alpha\ \text{set}
\end{aligned}$$

These constants are supposed to represent the empty set, the universal set, i.e., the set containing all elements of its element type, bounded quantification, the (set) union, the (set) intersection, the set difference and the addition of an element into a set. Their definitions are straightforward:

$$\begin{aligned}
\varnothing &= \{x \mid \text{False}\} \\
\text{UNIV} &= \{x \mid \text{True}\} \\
\text{Ball}\ A\ P &= \forall x.\ x \in A \longrightarrow P\ x \\
A \cup B &= \{x \mid x \in A \lor x \in B\} \\
A \cap B &= \{x \mid x \in A \land x \in B\} \\
A - B &= \{x \mid x \in A \land x \notin B\} \\
\text{insert}\ a\ A &= \{x \mid x = a\} \cup A
\end{aligned}$$

As customary, we shall write $\forall x \in A.\ t$ instead of Ball $A\ (\lambda x.\ t)$ and $\{a_1, \ldots, a_n\}$ instead of insert a_1 $(\ldots(\text{insert}\ a_n\ \varnothing)\ldots)$.

The binary relation operations that we use in the thesis are as follows:

$$
\begin{aligned}
.^{-1} &: (\alpha \rightarrow \beta \rightarrow \mathsf{bool}) \rightarrow \beta \rightarrow \alpha \rightarrow \mathsf{bool} \\
\circ\circ &: (\alpha \rightarrow \beta \rightarrow \mathsf{bool}) \rightarrow (\beta \rightarrow \gamma \rightarrow \mathsf{bool}) \rightarrow \alpha \rightarrow \gamma \rightarrow \mathsf{bool} \\
\sqsubseteq &: (\alpha \rightarrow \beta \rightarrow \mathsf{bool}) \rightarrow (\alpha \rightarrow \beta \rightarrow \mathsf{bool}) \rightarrow \mathsf{bool} \\
\mathsf{Domp} &: (\alpha \rightarrow \beta \rightarrow \mathsf{bool}) \rightarrow \alpha \rightarrow \mathsf{bool}
\end{aligned}
$$

These constants represent the relational converse and composition, the subrelation comparison and the domain operator. Their characterizations:

$$
\begin{aligned}
R^{-1}\ x\ y &\longleftrightarrow R\ y\ x \\
(R \circ\circ S)\ x\ z &\longleftrightarrow \exists y.\ R\ x\ y \wedge S\ y\ z \\
R \sqsubseteq S &\longleftrightarrow \forall x\ y.\ R\ x\ y \longrightarrow S\ x\ y \\
\mathsf{Domp}\ R\ x &\longleftrightarrow \exists y.\ \ R\ x\ y
\end{aligned}
$$

Look! A trickle of water running through some dirt! I'd say our afternoon just got booked solid!
— Bill Watterson, *Calvin and Hobbes*

3 HIGHER-ORDER LOGIC WITH AD HOC OVERLOADING CONSISTENTLY

In this chapter, I will provide semantics of Isabelle/HOL in terms of set theory with the axiom of choice. This is not meant to be a pretentious exercise to cover 40 pages with formulas but a practically motivated deed. The recently discovered consistency issues arising from the combination of overloading and typedef in Isabelle/HOL called us to arms to develop the first (semantic) explanation of the correct interplay of these features.

First, I present the discovered inconsistency (Section 3.1) and previous attempts to settle this problem (Section 3.2). Then we will formally define the consistency problem by defining a notion of a definitional theory (Section 3.3). We will define a definitional dependency relation, which gives a rise to well-formed definitional theories, and show that well-formedness implies consistency by means of our novel, ground, fragment-localized semantics (Section 3.4). On the practical side, I will present an algorithm (together with its soundness and completeness proof) that decides if the given theory is well formed (Section 3.5). I will conclude the chapter with some assessing remarks (Section 3.6).

This chapter is based on joint work with Andrei Popescu [53, 54].

3.1 hol with ad hoc overloading inconsistently

Polymorphic HOL, more precisely, Classic Higher-Order Logic with Infinity, Hilbert Choice and Rank-1 Polymorphism, endowed with a mechanism for constant and type definitions, was proposed in the nineties as a logic for interactive theorem provers by Mike Gordon, who also implemented the seminal HOL theorem prover [24]. This system has produced many successors and emulators known under the umbrella term "HOL-based provers" (e.g., HOL4 [89], HOL Light [32], ProofPower [6] and HOL Zero [2]), launching a very successful paradigm in interactive theorem proving.

A main strength of HOL-based provers is a sweet spot in expressiveness versus complexity: on the one hand HOL is sufficient for most mainstream mathematics and computer science applications, but on the other, it is a well-understood logic. In particular, the consistency of HOL has a standard semantic argument, comprehensible to any science graduate: one interprets its types as sets, in particular the function types as sets of functions, and the terms as elements of these sets, in a natural way; the rules of the logic are easily seen to hold in this model. The definitional mechanism has two flavors:

- New constants c are introduced by equations $c \equiv t$, where t is a closed term not containing c
- New types τ are introduced by typedef equations $\tau \equiv t$, where $t : \sigma \to \mathsf{bool}$ is a predicate on an existing type σ (not containing τ anywhere in the types of its subterms)

Again, this mechanism is manifestly consistent by an immediate semantic argument [81]; alternatively, its consistency can be established by regarding definitions as mere abbreviations (which here are acyclic by construction).

As we already saw, Isabelle/HOL adds its personal touch to the aforementioned sweet spot: it extends polymorphic HOL with a mechanism for (ad hoc) overloading. As an example, consider the following Nominal-style [92] definitions, where prm is the type of finite-support bijections on an infinite type atom, and where we write apply $\pi\ a$ for the application of a bijection π to an atom a and π^{-1} for the inverse of π. The intended behavior of the constant $\bullet : \mathsf{prm} \to \alpha \to \alpha$ (we use it as an infix operator) is the application of a permutation to all atoms contained in an element of a type α:

Example 3.1.

$$\begin{array}{lll}
\texttt{consts} & \bullet : \mathsf{prm} \to \alpha \to \alpha & \\
\texttt{defs} & \bullet_{\mathsf{prm}\to\mathsf{atom}\to\mathsf{atom}} & = \lambda \pi\ a.\ \mathsf{apply}\ \pi\ a \\
\texttt{defs} & \bullet_{\mathsf{prm}\to\mathsf{nat}\to\mathsf{nat}} & = \lambda \pi\ n.\ n \\
\texttt{defs} & \bullet_{\mathsf{prm}\to\alpha\ \mathsf{list}\to\alpha\ \mathsf{list}} & = \lambda \pi\ xs.\ \mathsf{map}\ (\lambda x.\ \pi \bullet x)\ xs \\
\texttt{defs} & \bullet_{\mathsf{prm}\to(\alpha\to\beta)\to\alpha\to\beta} & = \lambda \pi\ f\ x.\ \pi \bullet f\ (\pi^{-1} \bullet x)
\end{array}$$

For atoms, $\bullet$ applies the permutation; for numbers (which don't have atoms), $\bullet$ is the identity function; for α list and $\alpha \to \beta$, the instance of $\bullet$ is defined in terms of the instance for the components α and β. All these definitions fulfill the conditions that we stated in Section 2.4.1 on page 19: they are nonoverlapping and their type-based recursion is terminating, hence Isabelle is fine with them.

Of course, one may not be able to specify all the relevant instances immediately after declaring a constant such as $\bullet$. At a later point, a user may define their own atom-container type, such as

$$\texttt{datatype}\ \ \mathsf{myTree} = \mathsf{A}\ \mathsf{atom} \mid \mathsf{LNode}\ \mathsf{atom}\ \mathsf{list} \mid \mathsf{FNode}\ \mathsf{nat} \to \mathsf{atom}$$

and instantiate $\bullet$ for this type. (In fact, the Nominal tool automates instantiations for user-requested datatypes, including terms with bindings.)

To support such delayed instantiations, which are also crucial for the implementation of type classes, Isabelle/HOL allows intermixing definitions of instances of an overloaded constant with definitions of other constants and types. Unfortunately, the improper management of the intermixture leads to inconsistency: Isabelle/HOL accepts the following definitions[1]

Example 3.2.

$$\begin{aligned}&\texttt{consts}\ \mathsf{c} : \alpha\\ &\texttt{typedef}\ \ \tau = \{\mathsf{True}, \mathsf{c}_{\mathsf{bool}}\}\ \ \mathsf{by\ blast}\\ &\texttt{defs}\ \mathsf{c}_{\mathsf{bool}} = \neg\,(\forall x_\sigma\, y.\, x = y)\end{aligned}$$

which immediately yield a proof of False:

$\texttt{lemma}\ \ \mathsf{L} : (\forall x_\sigma\, y.\, x = y) \longleftrightarrow \mathsf{c}$

$\quad\texttt{using}\ \mathsf{Rep}_\tau\ \mathsf{Rep}_\tau\text{\_}\mathsf{inject}\ \mathsf{Abs}_\tau\text{\_}\mathsf{inject}\ \mathsf{by}\ (\mathsf{cases}\ \mathsf{c}_{\mathsf{bool}})\ \mathsf{force+}$

$\texttt{theorem}\ \ \mathsf{False}$

$\quad\texttt{using}\ \mathsf{L}\ \texttt{unfolding}\ \mathsf{c\_bool\_def}\ \mathsf{by\ auto}$

The inconsistency argument takes advantage of the circularity $\tau \rightsquigarrow \mathsf{c}_{\mathsf{bool}} \rightsquigarrow \tau$ in the dependencies introduced by the definitions: one first defines τ to contain only one element just in case $\mathsf{c}_{\mathsf{bool}}$ is True, and then defines $\mathsf{c}_{\mathsf{bool}}$ to be True just in case τ contains more than one element.

Before we start settling the consistency problem of the overloading mechanism, we should ask ourselves three motivational/design decision questions.

Do we need overloading? Yes. We do because it would not be possible to implement usable type classes without it. Substantial developments such as the Nominal [41, 92] and HOLCF [66] tools and Isabelle's mathematical analysis library [38] rely heavily on type classes. One of Isabelle's power users writes [60]: "Thanks to type classes and refinement during code generation, our light-weight framework is flexible, extensible, and easy to use."

Do we need types depending on overloaded constants? This is apparently the offending feature in Example 3.2. But this feature allows us to define many concepts in a natural way. Let me mention one as an example: the type of all red-black trees (α, β) rbt, which is defined by a restriction on all binary trees. The definition requires that rbt is a search tree, which means that the definition depends on a linear order on α, i.e., on a definition of the overloaded constant $\leq_{\alpha\to\alpha\to\mathsf{bool}}$:

$$\texttt{typedef}\ \ (\alpha, \beta)\ \mathsf{rbt} = \{t_{(\alpha\times\beta)\ \mathsf{binary\_tree}} \mid \ldots \leq_{\alpha\to\alpha\to\mathsf{bool}} \ldots\}$$

1 This example works in Isabelle2014 and Isabelle2015. My correction patch [1] based on the results of our papers [53, 54] was accepted by the Isabelle headquarters for the next official release, Isabelle2016.

There are many types like this one in Isabelle/HOL and therefore the answer to the question is yes.

Could not we make overloading a derived definitional mechanism? If so, this would mean that any conceptual or implementation issue could not lead to an inconsistency. One could take an example from another proof assistant, Coq. Overloading in Coq—implemented by Sozeau and Oury [87] in the context of first-class type classes—uses the dictionary construction, i.e., during processing a theory, a type-class operation call is replaced by a projection from a certain dependent record, which represents a type class and whose concrete instance is found by a special tactic for an instance search. But we cannot use this approach because the dictionary construction is not expressible in Isabelle/HOL. If we used it, the type rbt would have to be parametrized by the linear order; that is to say, it would depend on a *term* parameter *le*:

$$\texttt{typedef}\ \ (\alpha, \beta, le_{\alpha \to \alpha \to \mathsf{bool}})\ \mathsf{rbt} = \{t_{(\alpha \times \beta)\ \mathsf{binary\_tree}} \mid \ldots le \ldots\}$$

But since dependent types are not supported in HOL, we cannot use this approach. To my best knowledge, there is no other technique how to make overloading a derived definitional mechanism in HOL and therefore the answer to our question is no.

3.2 related work

overloading. I have already mentioned that overloading was introduced by Nipkow and Snelting [74] in Isabelle/HOL and by Sozeau and Oury [87] in Coq.

Mizar provides overloading for functions, types and other entities of the system (see a description by Grabowski et al. [26]). Moreover, there are two types of overloading: ad hoc and parameter overloading. The whole mechanism of retrieving the meaning of an overloaded symbol is involved but it holds that after the theory is processed, each overloaded symbol has been resolved to a unique logical symbol.

Concerning other proof assistants, to the best of my knowledge, there exists no notion of overloading in ACL2, HOL4, HOL Light or PVS.

previous consistency attempts. The settling of the consistency of the mechanism of ad hoc overloading has been previously attempted by Wenzel [99] and Obua [75]. In 1997, Wenzel defined a notion of a safe theory extension and showed that overloading conforms to this notion. But he did not consider type definitions and worked with a simplified version of the system where all overloadings for a constant c are provided at once. Years later, when Obua took over the problem, he found that the overloadings were almost completely unchecked—the following trivial inconsistency was accepted by Isabelle2005:

Example 3.3.

$$
\begin{aligned}
&\texttt{consts}\ \mathsf{c} : \alpha \to \mathsf{bool} \\
&\texttt{defs}\ \mathsf{c}_{\alpha\ \mathsf{list} \times \alpha \to \mathsf{bool}} = \lambda x.\, \mathsf{c}(\mathsf{snd}\ x \ \#\ \mathsf{fst}\ x) \\
&\texttt{defs}\ \mathsf{c}_{\alpha\ \mathsf{list} \to \mathsf{bool}} = \lambda x.\, \neg\, \mathsf{c}(\mathsf{tail}\ x, \mathsf{head}\ x) \\
&\texttt{lemma}\ \mathsf{c}\,[x] = \neg\, \mathsf{c}([\,], x) = \neg\, \mathsf{c}[x]
\end{aligned}
$$

Obua noticed that the rewrite system produced by the definitions has to terminate to avoid inconsistency, and implemented a private extension based on a termination checker. He did consider intermixing overloaded constant definitions and type definitions but his syntactic proof sketch failed to consider inconsistency through type definitions.

Triggered by Obua's observations, Wenzel implemented a simpler and more structural solution based on work of Haftmann, Obua and Urban: fewer overloadings are accepted in order to make the consistency/termination problem decidable (which Obua's original problem is not). Wenzel's solution has been part of the kernel since Isabelle2007 without any important changes—parts of this solution (which still does not consider dependencies through types) are described by Haftmann and Wenzel [30].

In 2014, I discovered that dependencies through types are not covered (Example 3.2 on page 27), and Andrei Popescu discovered an unrelated issue in the termination checker that caused an inconsistency even without exploiting types. I amended the latter issue by presenting a modified version of the termination checker and proving its correctness [53] and later worked with Andrei on the proof that termination of the definition dependency relation through types guarantees consistency [54].

inconsistency club. Inconsistency problems arise quite frequently with provers that step outside the safety of a simple and well-understood logic kernel. The various proofs of False in the early PVS system [84] are folklore. Coq's [9] current stable version[2] is inconsistent in the presence of Propositional Extensionality; this problem stood undetected by the Coq users and developers for 17 years; interestingly, just like the Isabelle/HOL problem under scrutiny, it is due to an error in the termination checker [21]. Agda [15] suffers from similar problems [64]. The recent Dafny prover [58] proposes an innovative combination of recursion and corecursion whose initial version turned out to be inconsistent [13].

Of course, such "dangerous" experiments are often motivated by better support for the users' formal developments. As I already mentioned, the Isabelle/HOL type class experiment was practically successful.

2 Namely, Coq 8.4pl6; the inconsistency is fixed in Coq 8.5 beta1.

consistency club. Members of this select club try to avoid inconsistencies by impressive efforts of proving soundness of logics and provers by means of interactive theorem provers themselves. Harisson's pioneering work [34] uses HOL Light to give semantic proofs of soundness of the HOL logic without definitional mechanisms, in two flavors: either after removing the infinity axiom from the object HOL logic, or after adding a "universe" axiom to HOL Light; a proof that the OCaml implementation of the core of HOL Light correctly implements this logic is also included.

Kumar et al. [52] formalize in HOL4 the semantics and the soundness proof of HOL, with its definitional principles included; from this formalization, they extract a verified implementation of a HOL theorem prover in CakeML, an ML-like language featuring a verified compiler. None of the above verified systems factor in ad hoc overloading, the starting point of our work.

Krauss and Schropp [50] implemented an automated translation of theories from Isabelle/HOL to Isabelle/ZF—Zermelo–Fraenkel set theory with the axiom of choice. They translate recorded proof terms (the translated proofs are rechecked) and in principle follow the standard semantics approach [81]. Overloaded constants are compiled out by the dictionary construction but their implementation does not support types depending on overloaded constants.

Outside the HOL-based prover family, there are formalizations of Milawa [67], Nuprl [3] and fragments of Coq [7, 8].

3.3 the consistency problem

In this section, we will move on from destruction (inconsistency) to restoration: we will define what it means to be consistent and what should be consistent. On this account, we will come up with a notion of a definitional theory, which is for us just a set of certain formulas, and observe a useful trick that allows us to separate definitions from proving.

3.3.1 Built-In and Non-Built-In Types and Constants

The distinction between built-in and non-built-in types and constants will be important, since we will employ a slightly nonstandard semantics only for the latter.

A *built-in type* is any type of the form bool, ind, σ set, or $\sigma \to \tau$ for $\sigma, \tau \in$ Type. We let Type$^\bullet$ denote the set of *non-built-in types*, i.e., types that are *not* built-in. Note that we look only at the topmost type constructor to decide if the given type is built-in or non-built-in. Therefore, a non-built-in type can have a built-in type as a subexpression, and vice versa; e.g., if list is a type constructor, then bool list and $(\alpha \to \beta)$ list are non-built-in types, whereas $\alpha \to \beta$ list is a built-in type.

Given a type σ, we define $\mathsf{types}^\bullet(\sigma)$, the *set of non-built-in types* of σ, as follows:

$$\begin{aligned}
&\mathsf{types}^\bullet(\mathsf{bool}) = \varnothing\\
&\mathsf{types}^\bullet(\mathsf{ind}) = \varnothing\\
&\mathsf{types}^\bullet(\sigma\ \mathsf{set}) = \mathsf{types}^\bullet(\sigma)\\
&\mathsf{types}^\bullet(\sigma_1 \to \sigma_2) = \mathsf{types}^\bullet(\sigma_1)\ \cup\ \mathsf{types}^\bullet(\sigma_2)\\
&\mathsf{types}^\bullet(\alpha) = \varnothing\\
&\mathsf{types}^\bullet(\overline{\sigma}\ k) = \{\overline{\sigma}\ k\}, \quad \text{if } k \neq \mathsf{bool}, \mathsf{ind}, \mathsf{set}, \to
\end{aligned}$$

Thus, $\mathsf{types}^\bullet(\sigma)$ is the smallest set of non-built-in types that can produce σ by repeated application of the built-in type constructors. For example, if the type constructors prm (0-ary) and list (unary) are in the signature and if σ is $(\mathsf{bool} \to \alpha\ \mathsf{list}) \to \mathsf{prm} \to (\mathsf{bool} \to \mathsf{ind})\ \mathsf{list}$, then $\mathsf{types}^\bullet(\sigma)$ has three elements: α list, prm and $(\mathsf{bool} \to \mathsf{ind})$ list.

A built-in constant is a constant of the form $\to$, $=$, $\in$, Collect, ε, zero or suc. We let $\mathsf{CInst}^\bullet$ be the set of constant instances that are *not* instances of built-in constants.

In our semantics (Section 3.4.4 on page 38), we will stick to the standard interpretation of built-in items, whereas for non-built-in items we will allow an interpretation looser than customary. The standardness of the bool, ind, set and function-type interpretation will allow us to always automatically extend the interpretation of a set of non-built-in types to the interpretation of its built-in closure.

Given a term t, we let $\mathsf{consts}^\bullet(t) \subseteq \mathsf{CInst}^\bullet$ be the set of all non-built-in constant instances occurring in t and $\mathsf{types}^\bullet(t) \subseteq \mathsf{Type}^\bullet$ be the set of all non-built-in types that compose the types of non-built-in constants and (free or bound) variables occurring in t. Note that the $\mathsf{types}^\bullet$ operator is overloaded for types and terms.

$$\begin{aligned}
&\mathsf{types}^\bullet(x_\sigma) = \mathsf{types}^\bullet(\sigma)\\
&\mathsf{types}^\bullet(c_\sigma) = \mathsf{types}^\bullet(\sigma)\\
&\mathsf{types}^\bullet(t_1\ t_2) = \mathsf{types}^\bullet(t_1) \cup \mathsf{types}^\bullet(t_2)\\
&\mathsf{types}^\bullet(\lambda x_\sigma.\ t) = \mathsf{types}^\bullet(\sigma) \cup \mathsf{types}^\bullet(t)\\
&\mathsf{consts}^\bullet(x_\sigma) = \varnothing\\
&\mathsf{consts}^\bullet(c_\sigma) = \begin{cases} \{c_\sigma\} & \text{if } c_\sigma \in \mathsf{CInst}^\bullet\\ \varnothing & \text{otherwise}\end{cases}\\
&\mathsf{consts}^\bullet(t_1\ t_2) = \mathsf{consts}^\bullet(t_1) \cup \mathsf{consts}^\bullet(t_2)\\
&\mathsf{consts}^\bullet(\lambda x_\sigma.\ t) = \mathsf{consts}^\bullet(t)
\end{aligned}$$

Note that the $\mathsf{consts}^\bullet$ and $\mathsf{types}^\bullet$ operators commute with ground type substitutions (and similarly with type substitutions, of course):

Lemma 3.3.1. (1) $\mathsf{consts}^\bullet(\theta(t)) = \{c_{\theta(\sigma)} \mid c_\sigma \in \mathsf{consts}^\bullet(t)\}$
(2) $\mathsf{types}^\bullet(\theta(t)) = \{\theta(\sigma) \mid \sigma \in \mathsf{types}^\bullet(t)\}$

3.3.2 Definitional Theories

We are interested in the consistency of theories arising from constant-instance and type definitions, which we call definitional theories.

Given $c_\sigma \in \mathsf{CInst}^\bullet$ and a closed term $t : \sigma$, we let $c_\sigma \equiv t$ denote the formula $c_\sigma = t$. We call $c_\sigma \equiv t$ a *constant-instance definition* provided $\mathsf{TV}(t) \subseteq \mathsf{TV}(c_\sigma)$ (i.e., $\mathsf{TV}(t) \subseteq \mathsf{TV}(\sigma)$).

Given the types $\tau \in \mathsf{Type}^\bullet$ and $\sigma \in \mathsf{Type}$ and the closed term t whose type is σ set, we let $\tau \equiv t$ denote the formula

$$\begin{aligned}
&(\exists x_\sigma.\, x \in t) \longrightarrow \\
&\quad \exists rep_{\tau\to\sigma}.\, \exists abs_{\sigma\to\tau}. \\
&\qquad (\forall x_\tau.\, rep\ x \in t) \wedge \\
&\qquad (\forall x_\tau.\, abs\ (rep\ x) = x) \wedge \\
&\qquad (\forall y_\sigma.\, y \in t \longrightarrow rep\ (abs\ y) = y).
\end{aligned}$$

We call $\tau \equiv t$ a *type definition*, provided $\mathsf{TV}(t) \subseteq \mathsf{TV}(\tau)$ (which also implies $\mathsf{TV}(\sigma) \subseteq \mathsf{TV}(\tau)$).

Note that we defined $\tau \equiv t$ *not* to mean:

(*): *The type τ is isomorphic, via abs and rep, to t, the subset of σ.*

as customary in most HOL-based systems, but rather to mean:

If t is a nonempty subset of σ, then (*) holds

Moreover, note that we do *not* require τ to have the form $(\alpha_1, \ldots, \alpha_n)k$, as is currently required in Isabelle/HOL and the other HOL provers, but, more generally, allow any $\tau \in \mathsf{Type}^\bullet$.[3] This enables an interesting feature: ad hoc overloading for type definitions. For example, given a unary type constructor tree, we can have totally different definitions for nat tree, bool tree and α list tree.

In general, a *definition* will have the form $u \equiv t$, where u is either a constant instance or a type and t is a term (subject to the specific constraints of constant-instance and type definitions). Given a definition $u \equiv t$, we call u and t the left-hand and right-hand sides of the definition. In what follows, we are interested only in theories that consist solely of definitions. We call such a theory D a *definitional theory*.

3 To ensure consistency, we will also require that τ has no common instance with the left-hand side of any other type definition.

3.3.3 The Consistency Problem

In this section, we will precisely define the consistency problem for Isabelle/HOL. Let us recall that a (definitional) theory D is consistent if $D \nvdash \mathsf{False}$.

Let us first inspect what a development in Isabelle/HOL looks like. Every step in the development falls in one of the following three categories:

1. declaring constants and types
2. defining constant instances and types
3. stating and proving theorems using the deduction rules of polymorphic HOL

Consequently, at any point in the development, one has:

1. a signature $(K, \mathsf{arOf} : K \to \mathbb{N}, \mathsf{Const}, \mathsf{tpOf} : \mathsf{Const} \to \mathsf{Type})$
2. a definitional theory D
3. other proved theorems

In our abstract formulation of Isabelle/HOL's logic, we do not represent explicitly point 3, namely the stored theorems that are not produced as a result of definitions, i.e., are not in D. The reason is that, in Isabelle/HOL, the definitional theorems in D are not influenced by the other theorems.

Note that this is not the case of the other HOL provers, due to the type definitions: there, $\tau \equiv t$, with $\mathsf{tpOf}(t) = \sigma\ \mathsf{set}$, is introduced in the unconditional form (*), and only after the user has proved that t is a nonempty subset (i.e., that $\exists x_\sigma.\ x \in t$ holds).[4] Of course, Isabelle/HOL's behavior converges with standard HOL behavior since the user is also required to prove nonemptiness, after which (*) is inferred by the system—however, this last inference step is normal deduction, having nothing to do with the definition itself. This very useful trick, due to Wenzel, cleanly separates definitions from proofs.

In summary, we only need to guarantee the consistency of D:

> **The Consistency Problem:** Find a sufficient criterion for a definitional theory D to be consistent (while allowing flexible ad hoc overloaded constant definitions).

4 In other HOL provers, sets are usually represented by unary predicates (i.e., terms of type $\sigma \to \mathsf{bool}$). Therefore, typedef takes a term t of type $\sigma \to \mathsf{bool}$ and the user has to show that t gives a nonempty set by proving $\exists x_\sigma.\ t\ x$. Since sets are defined axiomatically to be isomorphic to unary predicates in Isabelle/HOL, we can happily ignore this discrepancy.

3.4 the solution to the consistency problem

Assume for a moment that we have a proper dependency relation between defined items, where the defined items can be types or constant instances. Recall that the closure of this relation under type substitutions needs to terminate, otherwise inconsistency arises immediately, as shown in Example 3.3 on page 29. We also already saw that it is clear that the left-hand sides of the definitions need to be orthogonal to prevent examples such as Example 2.2 on page 18.

It turns out that these necessary criteria are also *sufficient* for consistency. This was also believed by Wenzel and Obua; what they were missing was a proper dependency relation and a transparent argument for its consistency, which is what we provide next.

3.4.1 Definitional Dependency Relation

Given any binary relation R on $\mathsf{Type}^\bullet \cup \mathsf{CInst}^\bullet$, we write R^+ for its transitive closure, R^* for its reflexive-transitive closure and $R^\downarrow$ for its (type-)substitutive closure, defined as follows: $p\ R^\downarrow\ q$ iff there exist p', q' and a type substitution ρ such that $p = \rho(p')$, $q = \rho(q')$ and $p'\ R\ q'$. We say that a relation R is *terminating* if there exists no sequence $(p_i)_{i\in\mathbb{N}}$ such that $p_i\ R\ p_{i+1}$ for all i.

Let us fix a definitional theory D. We say D is *orthogonal* if for all distinct definitions $u \equiv t$ and $u' \equiv t'$ in D, one of the following cases holds:

- either one of $\{u, u'\}$ is a type and the other is constant instance
- or both u and u' are types and are orthogonal ($u \ \#\ u'$)
- or both u and u' are constant instances and are orthogonal ($u \ \#\ u'$)

We define the binary relation $\rightsquigarrow$ on $\mathsf{Type}^\bullet \cup \mathsf{CInst}^\bullet$ by setting $u \rightsquigarrow v$ iff one of the following holds:

1. there exists a (constant-instance or type) definition in D of the form $u \equiv t$ such that $v \in \mathsf{consts}^\bullet(t) \ \cup\ \mathsf{types}^\bullet(t)$
2. $u = c_{\mathsf{tpOf}(c)}$ and $v \in \mathsf{types}^\bullet(\mathsf{tpOf}(c))$ for some $c \in \mathsf{Const}^\bullet$

We call $\rightsquigarrow$ the *dependency relation* (associated to D).

Thus, when defining an item u by means of t (as in $u \equiv t$), we naturally record that u depends on the constants and types appearing in t (clause 1); moreover, any constant c should depend on its type (clause 2). But notice the bullets! We only record dependencies on the non-built-in items, since intuitively the built-in items have a predetermined semantics which cannot be redefined or overloaded, and hence by themselves cannot introduce inconsistencies. Moreover, we do not dig for dependencies under any non-built-in type constructor—this can be seen from the definition of the types$^\bullet$ operator on types which yields a singleton whenever it meets a non-built-in type constructor; the rationale for this is that a non-built-in type constructor has an "opaque" semantics which does not expose the components (as does the function type constructor). These intuitions will be made precise by our semantics in Section 3.4.4.

Consider the following example, where the definition of α k is omitted:

Example 3.4.

$$\begin{array}{l}
\texttt{consts}\ \mathsf{c} : \alpha \\
\texttt{consts}\ \mathsf{d} : \alpha \\
\texttt{typedef}\ \alpha\,\mathsf{k} = \ldots \\
\texttt{defs}\ \mathsf{c}_{\mathsf{ind\ k \to bool}} = \mathsf{d}_{\mathsf{bool\ k\ k \to ind\ k \to bool}}\ \mathsf{d}_{\mathsf{bool\ k\ k}}
\end{array}$$

We record that the constant $\mathsf{c}_{\mathsf{ind\ k \to bool}}$ depends on the non-built-in constants $\mathsf{d}_{\mathsf{bool\ k\ k \to ind\ k \to bool}}$ and $\mathsf{d}_{\mathsf{bool\ k\ k}}$, and on the non-built-in types bool k k and ind k. We do *not* record any dependency on the built-in types bool k k → ind k → bool, ind k → bool or bool. Also, we do *not* record any dependency on bool k, which can only be reached by digging under k in bool k k.

3.4.2 The Consistency Theorem

We can now state our main result. We call a definitional theory D *well formed* if it is orthogonal and the substitutive closure of its dependency relation, $\rightsquigarrow^{\downarrow}$, is terminating.

Note that a well-formed definitional theory is allowed to contain definitions of two different (but orthogonal) instances of the same constant—this ad hoc overloading facility is a distinguishing feature of Isabelle/HOL among the HOL provers.

Theorem 1. If D is well formed, then D is consistent.

Previous attempts to prove consistency employed syntactic methods [75, 99]. Instead, we will give a semantic proof:

1. We define a new semantics of Polymorphic HOL that is suitable for overloading and in which False is not a valid formula (Section 3.4.4).

2. We prove that models of our semantics are preserved by Isabelle's deduction rules—soundness (Section 3.4.5).

3. We prove that D has a model according to our semantics (Section 3.4.6).

Then 1-3 immediately imply consistency.

3.4.3 Inadequacy of the Standard Semantics of Polymorphic HOL

But why define a new semantics? Recall that our goal is to make sense of definitions as in Example 3.1 on page 26. In the standard (Pitts) semantics [81], one chooses a universe collection of sets $\mathcal{U}$ closed under suitable set operations (function space, an infinite set, etc.) and interprets:

1. the built-in type constructors and constants as their standard counterparts in $\mathcal{U}$:
   - $[\mathsf{bool}]$ and $[\mathsf{ind}]$ are some chosen two-element set and infinite set in $\mathcal{U}$
   - $[\rightarrow] : \mathcal{U} \rightarrow \mathcal{U} \rightarrow \mathcal{U}$ takes two sets $A_1, A_2 \in \mathcal{U}$ to the set of functions $A_1 \rightarrow A_2$
   - $[\mathsf{set}] : \mathcal{U} \rightarrow \mathcal{U}$ maps a set $A \in \mathcal{U}$ to a power set $\mathcal{P}(A)$.
   - $[\mathsf{True}]$ and $[\mathsf{False}]$ are the two distinct elements of $[\mathsf{bool}]$, etc.
2. the non-built-in type constructors and constants similarly (we interpret prm and $\bullet$ introduced in Example 3.1 on page 26):
   - a defined type prm or type constructor list as an element $[\mathsf{prm}] \in \mathcal{U}$ or operator $[\mathsf{list}] : \mathcal{U} \rightarrow \mathcal{U}$, produced according to their typedef definition
   - a polymorphic constant such as $\bullet : \mathsf{prm} \rightarrow \alpha \rightarrow \alpha$ as a family $[\bullet] \in \prod_{A \in \mathcal{U}}([\mathsf{prm}] \rightarrow A \rightarrow A)$

In standard polymorphic HOL, $\bullet$ would be either completely unspecified, or completely defined in terms of previously existing constants—this has a faithful semantic counterpart in $\mathcal{U}$. But now how to represent the overloaded definitions of $\bullet$ from Example 3.1? In $\mathcal{U}$, they would become:

$$[\bullet]_{[\mathsf{atom}]}\ \pi\ a = [\mathsf{apply}]\ \pi\ a$$

$$[\bullet]_{[\mathsf{nat}]}\ \pi\ n = n$$

$$[\bullet]_{[\mathsf{list}](A)}\ \pi\ xs = [\mathsf{map}]_A\ ([\bullet]_A\ \pi)\ xs$$

$$[\bullet]_{A \rightarrow B}\ \pi\ f\ x = [\bullet]_B\ \pi\ (f\ ([\bullet]_A\ ([\mathsf{inv}]\ \pi)\ x))$$

There are two problems with these semantic definitions. First, given $B \in \mathcal{U}$, the value of $[\bullet]_B$ varies depending on whether B has the form $[\mathsf{atom}]$, or

$[\mathsf{nat}]$, $[\mathsf{list}](A)$ or $A \to B$ for some $A, B \in \mathcal{U}$; hence the interpretations of the type constructors need to be nonoverlapping—this is not guaranteed by the assumptions about $\mathcal{U}$, so we would need to perform some low-level set-theoretic tricks to achieve the desired property. Second, even though the definitions are syntactically terminating, their semantic counterparts may not be: unless we again delve into low-level tricks in set theory (based on the axiom of foundation), it is not guaranteed that decomposing a set A_0 as $[\mathsf{list}](A_1)$, then A_1 as $[\mathsf{list}](A_2)$, and so on (as prescribed by the third equation for $[\bullet]$) is a terminating process.

Even worse, termination is in general a global property, possibly involving both constants and type constructors, as shown in the following example where c and k are mutually defined (so that a copy of $\mathsf{e}_{\mathsf{bool}\ \mathsf{k}^n}$ is in $\mathsf{bool}\ \mathsf{k}^{n+1}$ iff n is even):

Example 3.5.

$$\begin{array}{lll}
\texttt{consts}\ \mathsf{c} : \alpha \to \mathsf{bool} & \mathsf{d} : \alpha & \mathsf{e} : \alpha \\
\texttt{typedef}\ \alpha\ \mathsf{k} & = \{\mathsf{d}_\alpha\} \cup \{\mathsf{e}_\alpha \mid \mathsf{c}\ \mathsf{d}_\alpha\} & \\
\texttt{defs}\ \mathsf{c}_{\alpha\ \mathsf{k} \to \mathsf{bool}} & = \lambda x_{\alpha\ \mathsf{k}}.\ \neg\ (\mathsf{c}\ \mathsf{d}_\alpha) & \\
\texttt{defs}\ \mathsf{c}_{\mathsf{bool} \to \mathsf{bool}} & = \lambda x.\ \mathsf{True} &
\end{array}$$

The above would require a set-theoretic setting where such fixpoint equations have solutions; this is, in principle, possible, provided we tag the semantic equations with enough syntactic annotations to guide the fixpoint construction. However, such a construction seems excessive given the original intuitive justification: the definitions are "OK" because they do not overlap and they terminate. On the other hand, a purely syntactic (proof-theoretic) argument also seems difficult due to the mixture of constant definitions and (conditional) type definitions.

Therefore, we decide to go for a natural syntactic-semantic blend, which avoids stunt performance in set theory: we do *not* semantically interpret the polymorphic types, but only the ground types—a type σ is called *ground* if $\mathsf{TV}(\sigma) = \varnothing$ and we let GType be the set of ground types. We think of polymorphic types as "macros" for families of the ground types. For example, $\alpha \to \alpha\ \mathsf{list}$ represents the family $(\tau \to \tau\ \mathsf{list})_{\tau \in \mathsf{GType}}$. Consequently, we think of the meaning of $\alpha \to \alpha\ \mathsf{list}$ not as $\prod_{A \in \mathcal{U}}(A \to [\mathsf{list}](A))$, but rather as $\prod_{\tau \in \mathsf{GType}}([\tau] \to [\tau\ \mathsf{list}])$. Moreover, a polymorphic formula φ of type, say, $(\alpha \to \alpha\ \mathsf{list}) \to \mathsf{bool}$, is considered true if and only if all its ground instances of types $(\tau \to \tau\ \mathsf{list}) \to \mathsf{bool}$ are true.

Another (small) departure from standard HOL semantics is motivated by our goal to construct a model for a well-formed definitional theory. Whereas in standard semantics, one first interprets all type constructors and constants and only afterwards extends the interpretation to terms, here we need to interpret some of the terms eagerly *before* some of the types and constants. Namely, given a definition $u \equiv t$, we interpret t before we interpret u (according to t). This requires a straightforward refinement of the notion of semantic interpretation: to interpret a term, we only need the interpretations for a sufficient fragment of the signature containing all the items appearing in t.

3.4.4 Ground, Fragment-Localized Semantics

Let me start with definitions regarding the ground part of the semantics. Recall that a type σ is called ground if $\mathsf{TV}(\sigma) = \varnothing$ and GType is the set of ground types. We let $\mathsf{GType}^\bullet = \mathsf{GType} \cap \mathsf{Type}^\bullet$ denote the set of ground non-built-in types. Clearly $\mathsf{GType}^\bullet \subset \mathsf{Type}$. We let GCInst be the set of constant instances whose type is ground and $\mathsf{GCInst}^\bullet = \mathsf{GCInst} \cap \mathsf{CInst}^\bullet$ be its subset of ground non-built-in instances. As a general notation rule: the prefix "G" indicates ground items, whereas the superscript $\bullet$ indicates non-built-in items, where an item can be either a type or a constant instance.

A ground type substitution is a function $\theta : \mathsf{TVar} \to \mathsf{GType}$, which again extends to a homonymous function $\theta : \mathsf{Type} \to \mathsf{GType}$.

A term t is called *ground* if $\mathsf{TV}(t) = \varnothing$. Thus, closedness refers to the absence of free (term) variables in a term, whereas groundness refers to the absence of type variables in a type or a term. Note that, for a term, being ground is a stronger condition than having a ground type: $(\lambda x_\alpha.\ c_{\mathsf{bool}})\ x_\alpha$ has the ground type bool, but is not ground.

Recall that we can apply a type substitution ρ to a term t, written $\rho(t)$, by applying ρ to all the type variables occurring in t; we use the same notation for ground type substitutions θ; note that $\theta(t)$ is always a ground term.

Given $T \subseteq \mathsf{Type}$, we define $\mathsf{Cl}(T) \subseteq \mathsf{Type}$, the *built-in closure* of T, inductively as follows:

$$
\begin{array}{ll}
T \subseteq \mathsf{Cl}(T) & \\
\{\mathsf{bool}, \mathsf{ind}\} \subseteq \mathsf{Cl}(T) & \\
\sigma\ \mathsf{set} \in \mathsf{Cl}(T) & \text{if } \sigma \in \mathsf{Cl}(T) \\
\sigma \to \tau \in \mathsf{Cl}(T) & \text{if } \sigma \in \mathsf{Cl}(T) \text{ and } \tau \in \mathsf{Cl}(T)
\end{array}
$$

This means that $\mathsf{Cl}(T)$ is the smallest set of types built from T by repeatedly applying built-in type constructors.

A *(signature) fragment* is a pair (T, C) with $T \subseteq \mathsf{GType}^\bullet$ and $C \subseteq \mathsf{GCInst}^\bullet$ such that $\sigma \in \mathsf{Cl}(T)$ for all $c_\sigma \in C$.

Let $F = (T, C)$ be a fragment. We write:

- Type^F, for the set of types generated by this fragment, namely $\mathsf{Cl}(T)$

- Term^F, for the set of terms that fall within this fragment, namely $\{t \in \mathsf{Term} \mid \mathsf{types}^\bullet(t) \subseteq T \wedge \mathsf{consts}^\bullet(t) \subseteq C\}$
- Fmla^F, for $\mathsf{Fmla} \cap \mathsf{Term}^F$

Lemma 3.4.1. The following hold:

(1) $\mathsf{Type}^F \subseteq \mathsf{GType}$

(2) $\mathsf{Term}^F \subseteq \mathsf{GTerm}$

(3) If $t \in \mathsf{Term}^F$, then $\mathsf{tpOf}(t) \in \mathsf{Type}^F$

(4) If $t \in \mathsf{Term}^F$, then $\mathsf{FV}(t) \subseteq \mathsf{Term}^F$

(5) If $t \in \mathsf{Term}^F$, then each subterm of t is also in Term^F

(6) If $t_1, t_2 \in \mathsf{Term}^F$ and $x_\sigma \in \mathsf{Var}_{\mathsf{Type}^F}$, then $t_1[t_2/x_\sigma] \in \mathsf{Term}^F$

The above straightforward lemma shows that fragments F include only ground items (points (1) and (2)) and are autonomous entities: the type of a term from F is also in F (3), and similarly for the free (term) variables (4), subterms (5) and substituted terms (6). This autonomy allows us to define semantic interpretations for fragments.

For the rest of this section, we fix the following:

- a singleton set $\{*\}$
- a two-element set $\{\mathsf{true}, \mathsf{false}\}$
- a global choice function, choice, that assigns to each nonempty set A an element $\mathsf{choice}(a) \in A$

Let $F = (T, C)$ be a fragment. An *F-interpretation* is a pair $\mathcal{I} = (([\tau])_{\tau \in T}, ([c_\tau])_{c_\tau \in C})$ such that:

1. $([\tau])_{\tau \in T}$ is a family such that $[\tau]$ is a nonempty set for all $\tau \in T$. We extend this to a family $([\tau])_{\tau \in \mathsf{Cl}(T)}$ by interpreting the built-in type constructors as expected:

   $[\mathsf{bool}] = \{\mathsf{true}, \mathsf{false}\}$

   $[\mathsf{ind}] = \mathbb{N}$ (the set of natural numbers)[5]

   $[\sigma\ \mathsf{set}] = \mathcal{P}([\sigma])$ (the set of all subsets of $[\sigma]$).

   $[\sigma \to \tau] = [\sigma] \to [\tau]$ (the set of functions from $[\sigma]$ to $[\tau]$)

2. $([c_\tau])_{c_\tau \in C}$ is a family such that $[c_\tau] \in [\tau]$ for all $c_\tau \in C$

5 Any infinite (not necessarily countable) set would do here; we only choose $\mathbb{N}$ for simplicity.

(Note that, in condition 2 above, $[\tau]$ refers to the extension described at point 1.)

Let GBI^F be the set of ground built-in constant instances c_τ with $\tau \in \mathsf{Type}^F$. We extend the family $([c_\tau])_{c_\tau \in C}$ to a family $([c_\tau])_{c_\tau \in C \cup \mathsf{GBI}^F}$, by interpreting the built-in constants as expected:

- $[\longrightarrow_{\mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}}]$ as the logical implication on $\{\mathsf{true}, \mathsf{false}\}$
- $[=_{\tau \to \tau \to \mathsf{bool}}]$ as the equality predicate in $[\tau] \to [\tau] \to \{\mathsf{true}, \mathsf{false}\}$
- $[\varepsilon_{(\tau \to \mathsf{bool}) \to \tau}]$ as the following function, where, for each $f : [\tau] \to \{\mathsf{true}, \mathsf{false}\}$, we let $A_f = \{a \in [\tau] \mid f(a) = \mathsf{true}\}$:
  $$[\varepsilon_{(\tau \to \mathsf{bool}) \to \tau}](f) = \begin{cases} \mathsf{choice}(A_f) & \text{if } A_f \text{ is nonempty} \\ \mathsf{choice}([\tau]) & \text{otherwise} \end{cases}$$
- $[\in_{\tau \to \sigma\ \mathsf{set} \to \mathsf{bool}}]$ as the set membership
- $[\mathsf{Collect}_{(\tau \to \mathsf{bool}) \to \sigma\ \mathsf{set}}](P)$ as $\{a \in [\tau] \mid P\ a\}$ for every function $P : [\tau] \to \{\mathsf{true}, \mathsf{false}\}$
- $[\mathsf{zero}_{\mathsf{ind}}]$ as 0 and $[\mathsf{suc}_{\mathsf{ind} \to \mathsf{ind}}]$ as the successor function for $\mathbb{N}$

To summarize, given an interpretation $\mathcal{I}$, which is a pair of families

$$(([\tau])_{\tau \in T}, ([c_\tau])_{c_\tau \in C}),$$

we can always obtain an extended pair of families

$$(([\tau])_{\tau \in \mathsf{Cl}(T)}, ([c_\tau])_{c_\tau \in C \cup \mathsf{GBI}^F}).$$

Now we are ready to interpret the terms in Term^F according to $\mathcal{I}$. A valuation $\xi : \mathsf{Var}_{\mathsf{Type}^F} \to \mathsf{Set}$ is called *$\mathcal{I}$-compatible* if $\xi(x_\sigma) \in [\sigma]^{\mathcal{I}}$ for each $x_\sigma \in \mathsf{Var}_{\mathsf{GType}}$. We write $\mathsf{Comp}^{\mathcal{I}}$ for the set of compatible valuations. For each $t \in \mathsf{Term}^F$, we define a function

$$[t] : \mathsf{Comp}^{\mathcal{I}} \to [\mathsf{tpOf}(t)]$$

recursively over terms as expected:

$$[x_\sigma](\xi) = \xi(x_\sigma)$$
$$[c_\sigma](\xi) = [c_\sigma]$$
$$[t_1\ t_2](\xi) = [t_1](\xi)\ ([t_2](\xi))$$
$$[\lambda x_\sigma.\ t](\xi) = \bigwedge_{a \in [\sigma]} [t](\xi[x_\sigma \leftarrow a])$$

The term $\bigwedge_{a \in [\sigma]}([t](\xi[x_\sigma \leftarrow a]))$ is the function sending each $a \in [\sigma]$ to $[t](\xi[x_\sigma \leftarrow a])$, where $\xi[x_\sigma \leftarrow a]$ is ξ updated with a at x_σ. Note that the recursive definition of $[t]$ is correct thanks to Lemma 3.4.1.(5). The above concepts are parametrized by a fragment F and an F-interpretation $\mathcal{I}$. If $\mathcal{I}$ or F are not clear from the context, we may write, e.g., $[t]^{\mathcal{I}}$ or $[t]^{F,\mathcal{I}}$.

Lemma 3.4.2. For each $t \in \mathsf{Term}^F$, the interpretation function $[t]$ is a function that only depends on the restriction of its inputs to $\mathsf{FV}(t)$, which means for all $\xi, \xi' \in \mathsf{Comp}^{\mathcal{I}}$ if $\xi(x) = \xi'(x)$ for all $x \in \mathsf{FV}(t)$, then $[t](\xi) = [t](\xi')$.

Therefore, if t is a closed term, then $[t]$ does not truly depend on ξ and hence we can simplify the definition of $[.]$ and assume $[t] : [\mathsf{tpOf}(t)]$. We will use the fragment interpretations to interpret only definitions, which are closed terms.

Note that the pairs $(F, \mathcal{I})$ are naturally ordered: Given fragments $F_1 = (T_1, C_1)$ and $F_2 = (T_2, C_2)$, F_1-interpretation $\mathcal{I}_1$ and F_2-interpretation $\mathcal{I}_2$, we define $(F_1, \mathcal{I}_1) \leq (F_2, \mathcal{I}_2)$ to mean $T_1 \subseteq T_2$, $C_1 \subseteq C_2$ and $[u]^{\mathcal{I}_1} = [u]^{\mathcal{I}_2}$ for all $u \in T_1 \cup C_1$.

Lemma 3.4.3. If $(F_1, \mathcal{I}_1) \leq (F_2, \mathcal{I}_2)$, then the following hold:

(1) $\mathsf{Type}^{F_1} \subseteq \mathsf{Type}^{F_2}$

(2) $\mathsf{Term}^{F_1} \subseteq \mathsf{Term}^{F_2}$

(3) $[\tau]^{F_1,\mathcal{I}_1} = [\tau]^{F_2,\mathcal{I}_2}$ for all $\tau \in \mathsf{Type}^{F_1}$

(4) $[t]^{F_1,\mathcal{I}_1} = [t]^{F_2,\mathcal{I}_2}$ for all closed $t \in \mathsf{Term}^{F_1}$

The *total fragment* $\top = (\mathsf{GType}^\bullet, \mathsf{GCInst}^\bullet)$ is the top element in this order. Note that $\mathsf{Type}^\top = \mathsf{GType}$ and $\mathsf{Term}^\top = \mathsf{GTerm}$.

So far, the notion of $\mathcal{I}$ interpreting φ was only defined for (closed) formulas φ that belong to Term^F, in particular, that are ground formulas. In the following, we will extend this to polymorphic formulas (in a context) by quantifying universally over all ground type substitutions. We only care about such an extension for the total fragment: Given a $\top$-interpretation $\mathcal{I}$, a context Γ, a polymorphic formula φ, a ground type substitution θ, and a valuation $\xi \in \mathsf{Comp}^{\mathcal{I}}$, we say that $\mathcal{I}$ *satisfies* φ *under the valuations* θ *and* ξ, written $\mathcal{I} \models_{\theta,\xi} \varphi$, if $[\theta(\varphi)]^{\mathcal{I}}(\xi) = \mathsf{true}$. We extend the notion to a set of formulas Γ as expected: $\mathcal{I} \models_{\theta,\xi} \Gamma$ if $\mathcal{I} \models_{\theta,\xi} \varphi$ for every $\varphi \in \Gamma$. Finally, we define $\mathcal{I} \models_{\theta,\xi} (\Gamma, \varphi)$ to mean $\mathcal{I} \models_{\theta,\xi} \Gamma$ implies $\mathcal{I} \models_{\theta,\xi} \varphi$.

We say $\mathcal{I}$ is a *model* of (Γ, φ), written $\mathcal{I} \models (\Gamma, \varphi)$, if $\mathcal{I} \models_{\theta,\xi} (\Gamma, \varphi)$ for all ground type substitutions θ and valuations $\xi \in \mathsf{Comp}^{\mathcal{I}}$. If $\Gamma = []$, we simply write $\mathcal{I} \models \varphi$. Furthermore, this extends to a set of (polymorphic) formulas D: $\mathcal{I} \models D$ is defined as $\mathcal{I} \models \varphi$ for all $\varphi \in D$. Finally, we define $\mathcal{I} \models (D, \Gamma, \varphi)$ to mean $\mathcal{I} \models D$ implies $\mathcal{I} \models (\Gamma, \varphi)$.

3.4.5 Soundness

We say that a deduction rule

$$\frac{D, \Gamma_1 \vdash \varphi_1 \qquad \ldots \qquad D, \Gamma_n \vdash \varphi_n}{D, \Gamma \vdash \varphi}$$

preserves models (or is *sound*) if for every $\top$-interpretation $\mathcal{I}$, it holds that $\mathcal{I} \models (D, \Gamma, \varphi)$ whenever $\mathcal{I} \models (D, \Gamma_i, \varphi_i)$ holds for all $1 \leq i \leq n$.

Theorem 2. The deduction rules of Isabelle/HOL preserve models.

Proof. It is routine to verify that the deduction rules for Isabelle/HOL are sound w.r.t. our ground semantics. □

Corollary 3.4.4. Let D be a definitional theory that has a total-fragment model, i.e., there exists a $\top$-interpretation $\mathcal{I}$ such that $\mathcal{I} \models D$. Then D is consistent.

3.4.6 The Model Construction

The only missing piece from the proof of consistency is the following:

Theorem 3. Assume D is a well-formed definitional theory. Then it has a total-fragment model, i.e., there exists a $\top$-interpretation $\mathcal{I}$ such that $\mathcal{I} \models D$.

Proof. For each $u \in \mathsf{GType}^\bullet \cup \mathsf{GCInst}^\bullet$, we define $[u]$ by well-founded recursion on $\rightsquigarrow^{\downarrow+}$, the transitive closure of $\rightsquigarrow^{\downarrow}$; indeed, the latter is a terminating (well-founded) relation by the well-formedness of D, hence the former is also terminating.

We assume $[v]$ has been defined for all $v \in \mathsf{GType}^\bullet \cup \mathsf{GCInst}^\bullet$ such that $u \rightsquigarrow^{\downarrow+} v$. In order to define $[u]$, we first need some terminology: We say that a definition $w \equiv s$ matches u if there exists a type substitution θ with $u = \theta(w)$. We distinguish the following two cases:

1. There exists no definition in D that matches u. Here we have two subcases:
   - $u \in \mathsf{GType}^\bullet$. Then we define $[u] = \{*\}$.
   - $u \in \mathsf{GCInst}^\bullet$. Say u has the form c_σ. Then $u \rightsquigarrow^{\downarrow} \sigma$, and hence $[\sigma]$ is defined; we define $[u] = \mathsf{choice}([\sigma])$.
2. There exists a definition $w \equiv s$ in D that matches u. Then let θ be such that $u = \theta(w)$, and let $t = \theta(s)$. Let $V_u = \{v \mid u \rightsquigarrow^{\downarrow+} v\}$, $T_u = V_u \cap \mathsf{Type}$ and $C_u = V_u \cap \mathsf{CInst}$. It follows from the definition of $\rightsquigarrow$ that $F_u = (T_u, C_u)$ is a fragment; moreover, from the definition of $\rightsquigarrow$ and Lemma 3.3.1, we obtain that $\mathsf{types}^\bullet(t) \subseteq T_u$

and $\mathsf{consts}^\bullet(t) \subseteq C_u$, which implies $t \in \mathsf{Term}^{F_u}$; hence we can speak of the value $[t]^{F_u,\mathcal{I}_u}$ obtained from the F_u-interpretation $\mathcal{I}_u = (([v])_{v\in T_u}, ([v])_{v\in C_u})$. We have two subcases:

- $u \in \mathsf{GCInst}^\bullet$. Then we define $[u] = [t]^{F_u,\mathcal{I}_u}$.
- $u \in \mathsf{GType}^\bullet$. Then $t : \sigma$ set for some σ and therefore $[t]^{F_u,\mathcal{I}_u} \in \mathcal{P}([\sigma]^{F_u,\mathcal{I}_u})$.[6] We have two subsubcases:
  - $[\exists x_\sigma.\ x \in t] = \mathsf{false}$. Then we define $[u] = \{*\}$.
  - $[\exists x_\sigma.\ x \in t] = \mathsf{true}$. Then we define $[u] = [t]^{F_u,\mathcal{I}_u}$.

Having defined the $\top$-interpretation

$$\mathcal{I} = (([u])_{u\in\mathsf{GType}^\bullet}, ([u])_{u\in\mathsf{GCInst}^\bullet}),$$

it remains to show that $\mathcal{I} \models D$. To this end, let $w \equiv s$ be in D and let θ' be a ground type substitution. We need to show $\mathcal{I} \models_{\theta'} w \equiv s$, i.e., $[\theta'(w) \equiv \theta'(s)]^{\mathcal{I}} = \mathsf{true}$.

Let $u = \theta'(w)$; then u matches $w \equiv s$, and by orthogonality this is the only definition in D that it matches. So the definition of $[u]$ proceeds with case 2 above, using $w \equiv s$—let θ be the ground type substitution considered there. Since $\theta'(w) = \theta(w)$, it follows that θ' and θ coincide on the type variables of w, and hence on the type variables of s (because, in any definition, the type variables of the right-hand side are included in those of the left-hand side); hence $\theta'(s) = \theta(s)$.

That means, we have to prove $[u \equiv v]^{\mathcal{I}} = \mathsf{true}$, where $u = \theta(w)$ and $v = \theta(s)$. The desired fact follows from the definition of $\mathcal{I}$ by a case analysis matching the subcases of the above case 2. (Note that the definition operates with $[t]^{F_u,\mathcal{I}_u}$, whereas we need to prove the fact for $[t]^{\top,\mathcal{I}}$; however, since $(F_u, \mathcal{I}_u) \leq (\top, \mathcal{I})$, by Lemma 3.4.3 the two values coincide and similarly for $[\sigma]^{F_u,\mathcal{I}_u}$ vs. $[\sigma]^{\top,\mathcal{I}}$.) □

3.5 deciding well-formedness

We proved that every well-formed theory is consistent. But we should not forget that we have a real-life proof assistant in our hands—we have to be able to check if a theory is well formed after we extend it by new definitions. We can check that D is definitional and orthogonal by simple polynomial algorithms. On the other hand, Obua [75] showed that a dependency relation generated by overloaded definitions can encode the Post correspondence problem and therefore termination of such a relation is not even a semi-decidable problem.

As I already mentioned, Wenzel proposed and implemented a solution that imposes a restriction on the definitions such that termination becomes decidable while preserving flexibility of overloading.

6 The interpretation $[\sigma]^{F_u,\mathcal{I}_u}$ is well defined since $\sigma \in \mathsf{types}^\bullet(t) \subseteq \mathsf{Type}^{F_u}$.

Back in 2014, there did not exist any convincing, complete proof that the algorithm decides termination of the corresponding rewriting system even though this algorithm should guarantee preservation of consistency of the system. Andrei Popescu triggered this work when he found out that the algorithm is not sound—he could prove False in Isabelle by misusing the discovered issue. Later I found out that the algorithm is not complete either—there are two independent issues that cause the algorithm not to terminate.[7] Based on this, I present a modified version of the algorithm and prove its soundness and completeness.

3.5.1 The Termination Problem

Recall that we have a dependency relation $\rightsquigarrow$ on $\mathsf{Type}^\bullet \cup \mathsf{CInst}^\bullet$ and that a theory D is well formed if D is definitional, orthogonal and $\rightsquigarrow^{\downarrow}$ is terminating. We have to work out how to decide the termination:

> **The Termination Problem:** Find a predicate P on binary relations on $\mathsf{Type}^\bullet \cup \mathsf{CInst}^\bullet$ such that for finite relations $\rightsquigarrow$ the following holds:
>
> - $P(\rightsquigarrow)$ is decidable
> - $P(\rightsquigarrow)$ implies that $\rightsquigarrow^{\downarrow}$ terminates
> - P contains interesting relations $\rightsquigarrow$

It is helpful to view The Termination Problem as an optimization problem; namely, we have to find a solution P between two extremities: P could either admit all terminating inputs (but then, as Obua showed, P cannot be decidable) or be false for all inputs (but then, while trivially decidable, no overloadings would be accepted). In other words, P should cover all of our use cases of overloading in Isabelle/HOL while still decidable. Our attack plan to The Termination Problems is as follows:

- We introduce an additional technical background (Section 3.5.2).
- We define composability (the main component of P) and show that under composability termination of $\rightsquigarrow^{\downarrow}$ is equivalent to acyclity of $\rightsquigarrow$ (Section 3.5.3).
- We present an algorithm that can decide composability and acyclity of $\rightsquigarrow$ and resolves the original problem—decidability of well-formedness (Section 3.5.4).

7 The original algorithm was primarily designed to achieve consistency preservation. Termination was not guaranteed and nontermination was meant as a measure of last resort against introducing inconsistency [31]. My work identifies these instances of nontermination and shows how to eliminate them and still preserve consistency.

- Finally, we look at the three issues of the original algorithm and how the modified one solves them (Section 3.5.5).

3.5.2 Preliminaries

We use the notation $(p_i, q_i)_{i \leq n}$ for sequence $(p_0, q_0), \ldots, (p_n, q_n)$. The image of a function under a set is defined as $f[A] = \{f(x) \mid x \in A\}$. If $f : A \to B$ and $C \subseteq A$, the restriction of f to C is a function $f \restriction_C : C \to B$ defined as $f \restriction_C (x) = f(x)$ for all $x \in C$.

Recall that we have a fixed signature $(K, \mathsf{arOf}, \mathsf{Const}, \mathsf{tpOf})$ and a type substitution is a function $\rho : \mathsf{TVar} \to \mathsf{Type}$ that is almost everywhere the identity. We range over type substitutions by ρ, σ, η.

The *size function* counts the number of type constructors in a type: $\mathsf{size}(\alpha) = 0$ and $\mathsf{size}((\tau_1, \ldots, \tau_n)k) = 1 + \Sigma_{1 \leq i \leq n} \mathsf{size}(\tau_i)$.

For a substitution σ, a *domain* is a (finite) set of variables $\mathsf{dom}(\sigma) = \{\alpha \mid \sigma(\alpha) \neq \alpha\}$, and an image is a set of types $\mathsf{img}(\sigma) = \sigma[\mathsf{dom}(\sigma)]$.

A *renaming* is a substitution σ such that $\mathsf{dom}(\sigma) = \mathsf{img}(\sigma)$ (and therefore each renaming is a bijection).

We assume that we have a function $\mathsf{Ren}(R, C)$ that gives us a renaming σ such that $\mathsf{dom}(\sigma) = R \cup \sigma[R]$ and $\sigma[R] \cap C = \varnothing$ (i.e., σ renames variables in R not to clash with variables in C). If R and C are finite, $\mathsf{Ren}(R, C)$ is always defined since TVar is infinite.

We write that $\rho =_\tau \rho'$ if $\rho(\alpha) = \rho'(\alpha)$ for all $\alpha \in \mathsf{TV}(\tau)$. We say that ρ is *equivalent* to ρ' and write $\rho \approx \rho'$ if there exists a renaming η such that $\rho = \eta \circ \rho'$. We write that $\rho \approx_\tau \rho'$ if there exists a renaming η such that $\rho =_\tau \eta \circ \rho'$.

Lemma 3.5.1. a) $\mathsf{size}((\eta \circ \rho)(\tau)) = \mathsf{size}(\rho(\tau))$ if η is a renaming.
b) $\mathsf{size}(\rho(\tau)) > \mathsf{size}(\tau)$ if $\rho \not\approx_{\tau'} \mathsf{id}$ and $\mathsf{TV}(\tau') \subseteq \mathsf{TV}(\tau)$.

Proof. A substitution that is not equivalent to the identity on τ has to map at least one type variable of τ to a type constructor, whereas a renaming cannot. ☐

Lemma 3.5.2. If η is a renaming, then $\eta[\mathsf{TV}(\tau)] = \mathsf{TV}(\eta(\tau))$.

Proof. Straightforward from definitions. ☐

Recall that two types τ_1 and τ_2 are called orthogonal, written $\tau_1 \# \tau_2$, if they have no common instance, i.e., for all τ it holds that $\tau \not\leq \tau_1$ or $\tau \not\leq \tau_2$. Or equivalently, τ_1 and τ_2 are orthogonal if and only if τ_1 and τ_2 cannot be unified after renaming variables in τ_1 and τ_2 apart.

Two types τ_1 and τ_2 have a nontrivial instance, written $\tau_1 \downarrow \tau_2$, if there exists τ' such that $\tau' \leq \tau_1$, $\tau' \leq \tau_2$ and $\tau_1 \not\leq \tau_2$ and $\tau_2 \not\leq \tau_1$. Notice if $\tau_1 \downarrow \tau_2$ and $\tau' \leq_\rho \tau_1$ and $\tau' \leq_{\rho'} \tau_2$, then $\rho \not\approx \mathsf{id}$ and $\rho' \not\approx \mathsf{id}$.

Lemma 3.5.3. Let $\rho \approx_{\tau_1} \rho'$; then $\rho(\tau_1) \leq \tau_2 \longleftrightarrow \rho'(\tau_1) \leq \tau_2$, $\rho(\tau_1) \geq \tau_2 \longleftrightarrow \rho'(\tau_1) \geq \tau_2$, $\rho(\tau_1) \;\#\; \tau_2 \longleftrightarrow \rho'(\tau_1) \;\#\; \tau_2$ and $\rho(\tau_1) \downarrow \tau_2 \longleftrightarrow \rho'(\tau_1) \downarrow \tau_2$.

Proof. It can be proved by an easy manipulation with substitutions. $\square$

Since the dependency relation $\rightsquigarrow$ relates elements of type $\mathsf{Type}^\bullet \cup \mathsf{CInst}^\bullet$, we will in general work with elements of this type, ranged over by p, q, r and s. We defined all of the aforementioned functions and relations TV, dom, img, Ren, $\leq$, $\approx$, $\#$, $\downarrow$ and size for elements of Type. But we want to apply all of these concepts to $\mathsf{Type}^\bullet \cup \mathsf{CInst}^\bullet$, which means that we want to lift these functions and relations from Type to Type $\cup$ CInst. We do not have to do the lifting separately for each of them because there exists a more elegant way: all of these functions and relations were defined in terms of three concepts: equality $=$, application of a substitution and the size function.[8] Therefore in order to complete the lifting from Type to Type $\cup$ CInst, it suffices if we define $= : \mathsf{Type} \cup \mathsf{CInst} \to \mathsf{Type} \cup \mathsf{CInst} \to \mathsf{bool}$, $\mathsf{App} : (\mathsf{Type} \to \mathsf{Type}) \to \mathsf{Type} \cup \mathsf{CInst} \to \mathsf{Type} \cup \mathsf{CInst}$ and $\mathsf{size} : \mathsf{Type} \cup \mathsf{CInst} \to \mathbb{N}$, which we do as follows:

- Two elements of Type $\cup$ CInst are equal iff they are both constant instances and they are equal or they are both types and they are equal.
- $\mathsf{App}\ \rho\ \tau = \rho(\tau)$ and $\mathsf{App}\ \rho\ c_\tau = c_{\rho(\tau)}$.
- $\mathsf{size}(c_\tau) = \mathsf{size}(\tau)$.

For readability reasons, we still write $\rho(p)$ for $\mathsf{App}\ \rho\ p$.

Recall that the substitutive closure was defined as follows: $p\ R^{\downarrow}\ q$ iff there exist p', q' and a type substitution ρ such that $p = \rho(p')$, $q = \rho(q')$ and $p'\ R\ q'$.

Recall that we say that a relation R is *terminating* if there exists no sequence $(p_i)_{i \in \mathbb{N}}$ such that $p_i\ R\ p_{i+1}$ for all i.

3.5.3 From Termination to Acyclicity

Since $\rightsquigarrow^{\downarrow}$ is the substitutive closure of $\rightsquigarrow$, $\rightsquigarrow^{\downarrow}$ is generally infinite even if $\rightsquigarrow$ is finite. In this section, we show that the problem of termination of infinite $\rightsquigarrow^{\downarrow}$ is equivalent to a finite problem on $\rightsquigarrow$, namely that $\rightsquigarrow$ is acyclic. We will define cyclicity formally later, but informally it means that we can find a finite sequence $r_0 \rightsquigarrow^{\downarrow} r_1 \rightsquigarrow^{\downarrow} \dots \rightsquigarrow^{\downarrow} r_n$ such that $r_n \leq r_0$. If we find a cycle, it is easy to show that $\rightsquigarrow^{\downarrow}$ does not terminate. To prove the other direction is involved.

8 Notice that $\mathsf{TV} : \mathsf{Type} \to \mathcal{P}(\mathsf{TVar})$ can be formally defined as $\mathsf{TV}(\tau) = \{\alpha \mid \exists \beta.\ \rho(\tau) \neq \tau$ where $\rho = \alpha \mapsto \beta\}$.

If $\rightsquigarrow^{\downarrow}$ does not terminate, it means there exists an infinite sequence $(p_i)_{i\in\mathbb{N}}$ such that $p_0 \rightsquigarrow^{\downarrow} p_1 \rightsquigarrow^{\downarrow} p_2 \rightsquigarrow^{\downarrow} \ldots$. We could hope that we could find a cycle as a subsequence of $(p_i)_{i\in\mathbb{N}}$ but the following example shows that this is not always the case: consider $\rightsquigarrow$ defined as $\alpha \rightsquigarrow \alpha\ \mathsf{list}$. Now $\mathsf{nat} \rightsquigarrow^{\downarrow} \mathsf{nat\ list} \rightsquigarrow^{\downarrow} \mathsf{nat\ list\ list} \rightsquigarrow^{\downarrow} \ldots$ is a nonterminating sequence but no subsequence of it is a cycle since all elements of this sequence are incomparable. But another sequence $\alpha \rightsquigarrow^{\downarrow} \alpha\ \mathsf{list} \rightsquigarrow^{\downarrow} \alpha\ \mathsf{list\ list} \rightsquigarrow^{\downarrow} \ldots$ contains many cycles (e.g., $\alpha\ \mathsf{list} \le \alpha$). Our intuition is that if we want to find a cycle, we have to search in sequences of instances that are as general as possible. We formalize this intuition now.

Definition 3.5.4. $\rightsquigarrow$ is *monotone* if for each p, q such that $p \rightsquigarrow q$, we have $\mathsf{TV}(q) \subseteq \mathsf{TV}(p)$.

Lemma 3.5.5. Let us assume that

- $p_i \rightsquigarrow q_i$ for all $i \le n$,
- $\rightsquigarrow$ is monotone.

If $(\rho_i)_{i\le n}$ is a sequence of substitutions such that we have $\rho_i(q_i) = \rho_{i+1}(p_{i+1})$ for all $i < n$, then $\mathsf{TV}(\rho_i(p_i)) \supseteq \mathsf{TV}(\rho_i(q_i)) \supseteq \mathsf{TV}(\rho_j(p_j)) \supseteq \mathsf{TV}(\rho_j(q_j))$ holds for all $0 \le i < j \le n$.

Proof. If $\mathsf{TV}(p) \supseteq \mathsf{TV}(q)$, then $\mathsf{TV}(\rho(p)) \supseteq \mathsf{TV}(\rho(q))$ for any ρ. Therefore it suffices to prove $\mathsf{TV}(\rho_i(q_i)) \supseteq \mathsf{TV}(\rho_j(p_j))$, which can be done by backward induction. □

Monotonicity is a natural notion if we remember the original motivation for $\rightsquigarrow$: given a definition of an overloaded constant $c = t$, it must hold that $\mathsf{TV}(t) \subseteq \mathsf{TV}(c)$. Monotonicity gives $\rightsquigarrow$ more regular structure, which allows us to simplify some definitions, and is crucial in some coming proofs—especially the following consequence of Lemma 3.5.5: if we know that ρ' has some effect on $\rho_n(q_n)$ (i.e., $\mathsf{dom}(\rho') \cap \mathsf{TV}(\rho_n(q_n)) \neq \varnothing$)), we know that ρ' has also some effect on $\rho_0(p_0)$.

Definition 3.5.6. We say that a sequence of substitutions $(\rho_i)_{i\le n}$ is a *solution* to the sequence $(p_i, q_i)_{i\le n}$ if $\rho_i(q_i) = \rho_{i+1}(p_{i+1})$ for all $i < n$.

We say that a solution $(\rho_i)_{i\le n}$ to $(p_i, q_i)_{i\le n}$ is *the most general solution* if for any other solution $(\rho'_i)_{i\le n}$ there exists a sequence of substitutions $(\eta_i)_{i\le n}$ such that $\rho'_i(p_i) = (\eta_i \circ \rho_i)(p_i)$.

Thanks to monotonicity, we can talk only about p_is and omit q_is in the last definition because we will be able to derive from $\rho'_i(p_i) = (\eta_i \circ \rho_i)(p_i)$ also $\rho'_i(q_i) = (\eta_i \circ \rho_i)(q_i)$ provided $\rightsquigarrow$ is monotone and $p_i \rightsquigarrow q_i$.

Lemma 3.5.7. If $(\rho_i)_{i\le n}$ and $(\rho'_i)_{i\le n}$ are both the most general solutions to the sequence $(p_i, q_i)_{i\le n}$, then $\rho_i \approx_{p_i} \rho'_i$ for all $i \le n$.

Proof. Since $(\rho_i)_{i\leq n}$ and $(\rho'_i)_{i\leq n}$ are both the most general solutions, there exist $(\eta_i)_{i\leq n}$ and $(\eta'_i)_{i\leq n}$ such that $\rho_i(p_i) = (\eta_i \circ \rho'_i)(p_i)$ and $\rho'_i(p_i) = (\eta'_i \circ \rho_i)(p_i)$ for all $i \leq n$. Now $\rho_i(p_i) = ((\eta_i \circ \eta'_i) \circ \rho_i)(p_i)$ and $\rho'_i(p_i) = ((\eta'_i \circ \eta_i) \circ \rho'_i)(p_i)$. Thus $\eta_i \circ \eta'_i =_{\rho_i(p_i)} \mathsf{id}$ and $\eta'_i \circ \eta_i =_{\rho'_i(p_i)} \mathsf{id}$ and therefore $\eta_i \restriction_{\mathsf{TV}(\rho'_i(p_i))}$ is a bijection between $\mathsf{TV}(\rho'_i(p_i))$ and $\mathsf{TV}(\rho_i(p_i))$. There surely exists a bijection $\hat{\eta}$ between $\mathsf{TV}(\rho_i(p_i)) \smallsetminus \mathsf{TV}(\rho'_i(p_i))$ and $\mathsf{TV}(\rho'_i(p_i)) \smallsetminus \mathsf{TV}(\rho_i(p_i))$. Then the following function $\eta_i \restriction_{\mathsf{TV}(\rho'_i(p_i))} \circ \hat{\eta}$ is a renaming that witnesses $\rho_i \approx_{p_i} \rho'_i$. □

We defined a notion of a most general solution and proved that most general solutions are unique modulo renaming. This notion formalizes our intuition that we should look for cycles in sequences that are as general instances as possible. Most general solutions define such sequences. Let us get back to our example: let $(p_i, q_i)_{i\leq 1} = (\alpha, \alpha\ \mathsf{list}), (\alpha, \alpha\ \mathsf{list})$. Then $(\rho_i)_{i\leq 1} = \alpha \mapsto \alpha\ \mathsf{list}, \alpha \mapsto \alpha\ \mathsf{list\ list}$ is the most general solution to $(p_i, q_i)_{i\leq 1}$ and yields a sequence $\alpha \rightsquigarrow^{\downarrow} \alpha\ \mathsf{list} \rightsquigarrow^{\downarrow} \alpha\ \mathsf{list\ list}$, which contains cycles. On the other hand, $(\rho'_i)_{i\leq 1} = \alpha \mapsto \mathsf{nat\ list}, \alpha \mapsto \mathsf{nat\ list\ list}$ is only a solution but not the most general one. $(\rho'_i)_{i\leq 1}$ yields this sequence $\mathsf{nat} \rightsquigarrow^{\downarrow} \mathsf{nat\ list} \rightsquigarrow^{\downarrow} \mathsf{nat\ list\ list}$, which does not contain any cycle.

Given a nonterminating sequence $p_0 \rightsquigarrow^{\downarrow} p_1 \rightsquigarrow^{\downarrow} p_2 \rightsquigarrow^{\downarrow} \ldots$, how do we construct a most general solution to its subsequences? And does a most general solution always exist? We will show that under an additional restriction (composability) we can always extend a most general solution to first n elements of the infinite sequence to $n + 1$ elements. This means that we will provide an inductive description of most general solutions. In order to achieve this, we need to first introduce some additional notions.

Definition 3.5.8. We say that sequences $(p_i, q_i)_{i\leq n}$ and $(\rho_i)_{i\leq n}$ form a *path* starting at k under $\rightsquigarrow$ and write $(p_i, q_i, \rho_i)^{\rightsquigarrow}_{k\leq i\leq n}$ if

- $p_i \rightsquigarrow q_i$ for all $k \leq i \leq n$,
- $\rho_k \approx_{p_k} \mathsf{id}$,
- $(\rho_i)_{k\leq i\leq n}$ is a solution to $(p_i, q_i)_{k\leq i\leq n}$.

If $k = 0$, we usually omit this index.

A path is a sequence together with its solution, which is allowed to only rename the first element of the sequence (i.e., not to apply a nontrivial substitution).

Definition 3.5.9. We say that $\rightsquigarrow$ is a *cyclic* relation if there exists a path $(p_i, q_i, \rho_i)^{\rightsquigarrow}_{i\leq n}$ and $\rho_n(q_n) \leq p_0$.

The formal definition of a cycle admits only a strict subset of cycles that we informally introduced at the beginning of this section, namely those that are paths. For example, if $\alpha \rightsquigarrow \alpha$ list, then β, β list is a cycle[9], whereas α list, α list list is not.

Definition 3.5.10. $\rightsquigarrow$ is *composable* if for all p and q such that $p \rightsquigarrow q$ and for each path $(p_i, q_i, \rho_i)^{\rightsquigarrow}_{i \le n}$, it holds that either $\rho_n(q_n) \le p$, or $p \le \rho_n(q_n)$, or $\rho_n(q_n) \ \# \ p$.

Composability is an important restriction on $\rightsquigarrow$. It reduces the search space when we are looking for a most general solution. Later we will prove that each sequence defined by a most general solution has a path as a suffix. Therefore if we already have a most general solution to the n first elements, composability tells us that there exist three cases concerning the extension of this most general solution: in two cases we can still (possibly) extend the sequence ($\rho_n(q_n) \le p$ or $p \le \rho_n(q_n)$) and in one case we cannot ($\rho_n(q_n) \ \# \ p$). We prove in the following lemma that if there exists already some solution, the case $\rho_n(q_n) \ \# \ p$ cannot occur.

Lemma 3.5.11. Let us assume that

- $(\rho_i)_{i \le n+1}$ is a solution to $(p_i, q_i)_{i \le n+1}$,
- $p_i \rightsquigarrow q_i$ for all $i \le n+1$,
- $\rightsquigarrow$ is monotone and composable,
- $(\rho'_i)_{i \le n}$ is the most general solution to $(p_i, q_i)_{i \le n}$,
- there exists $k \le n$ such that $(p_i, q_i, \rho'_i)^{\rightsquigarrow}_{k \le i \le n}$.

Then $\rho'_n(q_n) \le p_{n+1}$ or $p_{n+1} \le \rho'_n(q_n)$.

Proof. From composability it follows that $\rho'_n(q_n) \le p_{n+1}$, or $p_{n+1} \le \rho'_n(q_n)$, or $\rho'_n(q_n) \ \# \ p_{n+1}$. Since $(\rho'_i)_{i \le n}$ is the most general solution, there exists $(\eta_i)_{i \le n}$ such that $\rho_i(p_i) = (\eta_i \circ \rho'_i)(p_i)$ for all $i \le n$ and by monotonicity also $\rho_i(q_i) = (\eta_i \circ \rho'_i)(q_i)$ for all $i \le n$. Therefore we can rewrite $\rho_n(q_n) = \rho_{n+1}(p_{n+1})$ to $\eta_n(\rho'_n(q_n)) = \rho_{n+1}(p_{n+1})$, which means there exists a common instance of $\rho'_n(q_n)$ and p_{n+1} and thus only two cases $\rho'_n(q_n) \le p_{n+1}$ or $p_{n+1} \le \rho'_n(q_n)$ can occur. □

The last lemma shows that the existence of some solution (see this solution as a subsequence of our nonterminating sequence) and composability guarantee that we are left with two cases. The two following lemmas show that the extension is always possible in either of the cases and give us concrete instructions how to do it; i.e., how to extend a most general solution from n to $n+1$ elements.

9 We should write a path formally as $(\beta, \beta \text{ list}, \alpha \mapsto \beta)^{\rightsquigarrow}$ but we simplify our (heavy) notation in an informal description.

Lemma 3.5.12. Let

- $(\rho_i)_{i\le n}$ be the most general solution to $(p_i, q_i)_{i\le n}$,
- $p_i \rightsquigarrow q_i$ for all $i \le n$,
- $\rightsquigarrow$ be monotone,
- $\rho_n(q_n) \le_{\rho'} p_{n+1}$.

Then $(\rho_i)_{i\le n}, \rho'$ is the most general solution to $(p_i, q_i)_{i\le n+1}$.

Proof. The sequence $(\rho_i)_{i\le n}, \rho'$ is a solution to $(p_i, q_i)_{i\le n+1}$. We prove that it is the most general solution. Let $(\rho'_i)_{i\le n+1}$ be a solution to the sequence $(p_i, q_i)_{i\le n+1}$. Then $(\rho'_i)_{i\le n}$ is surely a solution to $(p_i, q_i)_{i\le n}$. Therefore there exists $(\eta_i)_{i\le n}$ such that $\rho'_i(p_i) = (\eta_i \circ \rho_i)(p_i)$ for all $i \le n$ and by monotonicity also $\rho'_i(q_i) = (\eta_i \circ \rho_i)(q_i)$. From this and $\rho'_n(q_n) = \rho'_{n+1}(p_{n+1})$ (since $(\rho'_i)_{i\le n+1}$ is a solution), it follows that $\eta_n(\rho_n(q_n)) = \rho'_{n+1}(p_{n+1})$ and since $\rho_n(q_n) = \rho'(p_{n+1})$, we get finally $\eta_n(\rho'(p_{n+1})) = \rho'_{n+1}(p_{n+1})$. Define $\eta_{n+1} := \eta_n$. □

Lemma 3.5.13. Let

- $(\rho_i)_{i\le n}$ be the most general solution to $(p_i, q_i)_{i\le n}$,
- $p_i \rightsquigarrow q_i$ for all $i \le n$,
- $\rightsquigarrow$ be monotone,
- $\rho_n(q_n) \ge_{\rho'} p_{n+1}$.

There exists a substitution $\hat\rho$ such that the sequence $(\hat\rho \circ \rho_i)_{i\le n}, \mathsf{id}$ is the most general solution to $(p_i, q_i)_{i\le n+1}$ and $\hat\rho =_{\rho_n(q_n)} \rho'$.

Proof. Since the formal proof is technical, we explain some ideas of it first on a little example: Let us have:

$$(p_i, q_i)_{i\le 1} = (\alpha \times \beta, \beta), (\alpha\ \mathsf{list}, \alpha)$$
$$(\rho_i)_{i\le 0} = \mathsf{id}$$

$(\rho_i)_{i\le 0}$ is trivially the most general solution to $(p_i, q_i)_{i\le 0}$. $q_0 \le_{\rho'} p_1$, where $\rho' = \beta \mapsto \alpha\ \mathsf{list}$. Let us define $(\rho^?_i)_{i\le 1}$, a candidate for a most general solution to $(p_i, q_i)_{i\le 1}$, as $(\rho^?_i)_{i\le 1} = \rho' \circ \rho_0, \mathsf{id}$, i.e., we try setting $\hat\rho = \rho'$. $(\rho^?_i)_{i\le 1}$ is a solution to $(p_i, q_i)_{i\le 1}$ because it yields a sequence

$$(\alpha \times \alpha\ \mathsf{list}, \alpha\ \mathsf{list}), (\alpha\ \mathsf{list}, \alpha). \tag{4}$$

But in general it is not a most general solution: Let us take $(\rho'_i)_{i\le 1} = [\alpha \mapsto \mathsf{int}, \beta \mapsto \mathsf{nat\ list}], \alpha \mapsto \mathsf{nat}$, which is a solution to $(p_i, q_i)_{i\le 1}$ because it yields this sequence

$$(\mathsf{int} \times \mathsf{nat\ list}, \mathsf{nat\ list}), (\mathsf{nat\ list}, \mathsf{nat}). \tag{5}$$

If $(\rho^?_i)_{i\leq 1}$ were the most general solution, we should be able to find $(\eta_i)_{i\leq 1}$ such that $\rho'_i(p_i) = (\eta_i \circ \rho^?_i)(p_i)$ for all $i = 0, 1$. But this is not possible since by comparing sequences (4) and (5) we derive $\eta_0(\alpha) = \mathsf{int}$ and $\eta_0(\alpha) = \mathsf{nat}$.

We have to first rename the type variable from p_0 that is not in q_0 such that it does not clash with variables from $\rho'(q_0)$ and q_0: we use a renaming $\sigma = \alpha \mapsto \gamma$. Observe that $(\sigma \circ \rho_i)_{i\leq 0}$ is a solution to $(p_i, q_i)_{i\leq 0}$. We define $\hat{\rho} = \rho' \circ \sigma$ and then a new candidate for the most general solution $\hat{\rho} \circ \rho_0$, id yields

$$(\gamma \times \alpha\ \mathsf{list}, \alpha\ \mathsf{list}), (\alpha\ \mathsf{list}, \alpha).$$

Now we can find $(\eta_i)_{i\leq 1}$: we obtain $\eta(\gamma)$ from comparing $(\sigma \circ \rho_i)_{i\leq 0}$ and $(\rho'_i)_{i\leq 0}$ (the former is a most general solution and the latter a solution) and $\eta(\alpha)$ from comparing $\rho'(q_0) = p_1$ and $\rho'_1(p_1)$ (which trivially yields $\eta(\alpha) = \rho'_1(\alpha)$).

Now we proceed to carry out the formal proof. Assume

$$\mathsf{dom}(\rho') \subseteq \mathsf{TV}(\rho_n(q_n)). \tag{6}$$

If it were not the case, we would use $\rho' \restriction_{\mathsf{TV}(\rho_n(q_n))}$ instead of ρ'.

Let $R := \mathsf{TV}(\rho_0(p_0)) \setminus \mathsf{TV}(\rho_n(q_n))$ and $C := \mathsf{TV}(\rho_n(q_n)) \cup \mathsf{TV}((\rho' \circ \rho_n)(q_n))$. We define $\sigma := \mathsf{Ren}(R, C)$ and obtain the two following properties of σ from the definition of Ren:

$$\mathsf{dom}(\sigma) \cap \mathsf{TV}(\rho_n(q_n)) = \varnothing \tag{7}$$
$$\sigma[R] \cap \mathsf{TV}((\rho' \circ \rho_n)(q_n)) = \varnothing \tag{8}$$

Let $\tilde{\rho}_i := \sigma \circ \rho_i$ for each $i \leq n$. Obviously $(\tilde{\rho}_i)_{i\leq n}$ is a solution to the sequence $(p_i, q_i)_{i\leq n}$. Now we prove that it is the most general solution: let us take another solution $(\rho'_i)_{i\leq n}$; thus there exists $(\eta_i)_{i\leq n}$ such that $\rho'_i(p_i) = (\eta_i \circ \rho_i)(p_i)$. Define $\tilde{\eta}_i := \eta_i \circ \sigma^{-1}$. Then $\rho'(p_i) = (\tilde{\eta}_i \circ \tilde{\rho}_i)(p_i)$.

From (7) it follows that $\tilde{\rho}_n(q_n) \geq_{\rho'} p_{n+1}$. Let $(\overline{\rho}_i)_{i\leq n+1} := (\rho' \circ \tilde{\rho}_i)_{i\leq n}$, id. The sequence $(\overline{\rho}_i)_{i\leq n+1}$ is clearly a solution to $(p_i, q_i)_{i\leq n+1}$. We prove that it is the most general solution. Let $(\rho'_i)_{i\leq n+1}$ be a solution to $(p_i, q_i)_{i\leq n+1}$ then $(\rho'_i)_{i\leq n}$ is clearly a solution to $(p_i, q_i)_{i\leq n}$. Thus $(\eta_i)_{i\leq n}$ exists such that

$$\rho'_i(p_i) = (\eta_i \circ \tilde{\rho}_i)(p_i) \text{ for all } i \leq n. \tag{9}$$

We define $(\overline{\eta})_{i\leq n+1}$ as follows: if $i = n+1$ then $\overline{\eta}_{n+1} := \rho'_{n+1}$, otherwise $(i \leq n)$

$$\overline{\eta}_i(x) := \begin{cases} \rho'_{n+1}(x) & \text{if } x \in \mathsf{TV}((\rho' \circ \rho_n)(q_n)) \\ \eta_i(x) & \text{otherwise.} \end{cases}$$

We prove by backward induction that $\rho'_i(p_i) = (\overline{\eta}_i \circ \overline{\rho}_i)(p_i)$ for all $i \leq n+1$. Base case ($i = n+1$): Since $\overline{\rho}_{n+1} = \mathsf{id}$, we get immediately

$\rho'_{n+1} = \overline{\eta}_{n+1} \circ \overline{\rho}_{n+1}$. Inductive step: $0 \le i < n+1$ and $\rho'_{i+1}(p_{i+1}) = (\overline{\eta}_{i+1} \circ \overline{\rho}_{i+1})(p_{i+1})$. We prove

$$\eta_i(x) = (\overline{\eta}_i \circ \rho')(x) \text{ for all } x \in \mathsf{TV}(\tilde{\rho}_i(p_i)) \tag{10}$$

by the case distinction:

a) $x \in \mathsf{TV}(\tilde{\rho}_i(p_i)) \setminus \mathsf{TV}(\rho_n(q_n))$: therefore $\rho'(x) = x$ by (6) and since $x \in \sigma[R]$ (by Lemmas 3.5.2 and 3.5.5), $\eta_i(x) = \overline{\eta}_i(x)$ follows from (8). Therefore $\eta_i(x) = (\overline{\eta}_i \circ \rho')(x)$.

b) $x \in \mathsf{TV}(\rho_n(q_n))$: $(\eta_i \circ \tilde{\rho}_i)(q_i) = (\overline{\eta}_{i+1} \circ \rho' \circ \tilde{\rho}_i)(q_i)$ holds since

$$\begin{aligned}
(\eta_i \circ \tilde{\rho}_i)(q_i) &= \rho'_i(q_i) && \text{by mon. and (9)}\\
&= \rho'_{i+1}(p_{i+1}) && (\rho'_i)_{i \le n+1} \text{ is sol.}\\
&= (\overline{\eta}_{i+1} \circ \overline{\rho}_{i+1})(p_{i+1}) && \text{by IH}\\
&= (\overline{\eta}_{i+1} \circ \rho' \circ \tilde{\rho}_i)(q_i) && (\overline{\rho_i})_{i \le n+1} \text{ is sol.}
\end{aligned}$$

Thus $\eta_i(y) = (\overline{\eta}_{i+1} \circ \rho')(y)$ for all $y \in \mathsf{TV}(\tilde{\rho}_i(q_i))$. By Lemma 3.5.5 and by (7) $x \in \mathsf{TV}(\tilde{\rho}_i(q_i))$. Therefore $\eta_i(x) = (\overline{\eta}_{i+1} \circ \rho')(x) = (\rho'_{n+1} \circ \rho')(x) = (\overline{\eta}_i \circ \rho')(x)$.

We know that $\rho'_i(p_i) = (\eta_i \circ \tilde{\rho}_i)(p_i)$ and by using (10) we get $\rho'_i(p_i) = (\overline{\eta}_i \circ \rho' \circ \tilde{\rho}_i)(p_i) = (\overline{\eta}_i \circ \overline{\rho}_i)(p_i)$, which concludes the proof that $(\overline{\rho}_i)_{i \le n+1}$ is the most general solution.

Let $\hat{\rho} := \rho' \circ \sigma$ then obviously $(\overline{\rho}_i)_{i \le n+1} = ((\rho' \circ \tilde{\rho}_i)_{i \le n}, \mathsf{id}) = ((\hat{\rho} \circ \rho_i)_{i \le n}, \mathsf{id})$. The equality $\hat{\rho} =_{\rho_n(q_n)} \rho'$ follows from (7). □

Now nothing prevents us from combining the previous results and proving that there is always a most general solution if some solution already exists.

Lemma 3.5.14. Let us assume that

- $(\rho_i)_{i \le n}$ is a solution to $(p_i, q_i)_{i \le n}$,
- $p_i \rightsquigarrow q_i$ for all $i \le n$,
- $\rightsquigarrow$ is monotone and composable,

then there exist a most general solution $(\rho'_i)_{i \le n}$ to $(p_i, q_i)_{i \le n}$ and $k \le n$ such that $(p_i, q_i, \rho'_i)^{\rightsquigarrow}_{k \le i \le n}$.

Proof. By induction on the length of the sequence. Base case $n = 0$: define $\rho'_0 = \mathsf{id}$. Inductive step $n = i + 1$: we assume that $(\rho_j)_{j \le i+1}$ is a solution to $(p_j, q_j)_{j \le i+1}$. Then $(\rho_j)_{j \le i}$ is surely a solution to $(p_j, q_j)_{j \le i}$ and thus by the induction hypothesis we obtain the most general solution $(\rho'_j)_{j \le i}$ to $(p_j, q_j)_{j \le i}$ and $k \le i$ such that $(p_j, q_j, \rho'_j)^{\rightsquigarrow}_{k \le j \le i}$.

By Lemma 3.5.11, only two cases can occur:

a) $\rho'_i(q_i) \leq_{\rho'} p_{i+1}$, then by Lemma 3.5.12 $(\rho'_j)_{j\leq i}, \rho'$ is the most general solution to $(p_j, q_j)_{j\leq i+1}$ and still $\rho'_k \approx_{p_k} \mathsf{id}$.

b) $\rho'_i(q_i) \geq_{\rho'} p_{i+1}$, then by Lemma 3.5.13 there exists $\hat{\rho}$ such that the sequence $\hat{\rho} \circ \rho'_1, \dots, \hat{\rho} \circ \rho'_i, \mathsf{id}$ is the most general solution to $(p_j, q_j)_{j\leq i+1}$. Then $k = i + 1$ and obviously $\rho'_k \approx_{p_k} \mathsf{id}$.

□

We proved even more: a sequence defined by a most general solution has always a path as a suffix. This is an important result because we claimed that we can look for cycles in sequences produced by most general solutions and we did define a cycle such that each cycle is a path. Thus this suffix is a candidate for a cycle. To find a real cycle among these candidates, we extend this suffix potentially ad infinitum in our proof, i.e., we find an infinite sequence where each prefix is a path. To capture this idea we introduce new notions.

Definition 3.5.15. We write $(p_i, q_i)_{i\leq n} \preceq p$ if there exists a most general solution $(\rho_i)_{i\leq n}$ to $(p_i, q_i)_{i\leq n}$ and if $\rho_n(q_n) \leq p$. We define $(p_i, q_i)_{i\leq n} \succeq p$ analogously.

Corollary 3.5.16. Let $\rightsquigarrow$ be composable and monotone, and let us have sequences $(p_i, q_i)_{i\leq n+1}$ and $(\rho_i)_{i\leq n+1}$ such that $(\rho_i)_{i\leq n+1}$ is a solution to $(p_i, q_i)_{i\leq n+1}$ and $p_i \rightsquigarrow q_i$ for all $i \leq n + 1$. Then $(p_i, q_i)_{i\leq n} \preceq p_{i+1}$ or $(p_i, q_i)_{i\leq n} \succeq p_{i+1}$.

Proof. From Lemmas 3.5.11 and 3.5.14. □

Definition 3.5.17. A sequence $(p_0, q_0), (p_1, q_1), \dots$ is called *ascending* if $(p_0, q_0), \dots, (p_{i-1}, q_{i-1}) \preceq p_i$ holds for all $i \geq 1$.

A sequence $(\alpha, \alpha\ \mathsf{list}), (\alpha, \alpha\ \mathsf{list}), \dots$ is an example of an ascending sequence. A most general solution $(\rho_i)_{i\leq n}$ to a prefix of length $n + 1$ of this sequence is defined as $\rho_i = \alpha \mapsto \alpha\ \mathsf{list}^n$ for all $i \leq n$.

Lemma 3.5.18. Let $\rightsquigarrow$ be monotone, $(p_0, q_0), (p_1, q_1), \dots$ be an ascending sequence and $p_i \rightsquigarrow q_i$ for all i. Then for all n it holds that there exists the most general solution $(\rho_i)_{i\leq n}$ to $(p_i, q_i)_{i\leq n}$ such that $(p_i, q_i, \rho_i)^{\rightsquigarrow}_{i\leq n}$.

Proof. We fix n. Let $(\rho_i)_{i\leq n}$ be the most general solution to $(p_i, q_i)_{i\leq n}$. $(\rho_i)_{i\leq n}$ always exists because of $(p_i, q_i)_{i\leq n} \preceq p_{n+1}$.

We prove by backward induction on n that $(\rho_j)_{j\leq i}$ is the most general solution to $(p_j, q_j)_{j\leq i}$ for all $i \leq n$. Base case: trivial. Inductive step ($n = i$, $i > 0$): $(\rho_j)_{j\leq i}$ is the most general solution to $(p_j, q_j)_{j\leq i}$. From $(p_0, q_0), \dots, (p_{i-1}, q_{i-1}) \preceq p_i$ it follows that there exists $(\rho'_j)_{j\leq i-1}$, a most general solution to $(p_0, q_0), \dots, (p_{i-1}, q_{i-1})$ such that $\rho'_{i-1}(q_{i-1}) \leq_{\rho'} p_i$. By Lemma 3.5.12 we know that $(\rho'_j)_{j\leq i-1}, \rho'$ is the most general solution to

$(p_j, q_j)_{j \leq i}$ but then Lemma 3.5.7 gives us $\rho_j \approx_{p_j} \rho'_j$ for all $j \leq i - 1$. Hence $(\rho_j)_{j \leq i-1}$ is the most general solution to $(p_0, q_0), \ldots, (p_{i-1}, q_{i-1})$.

Now we can see that $\rho_0 \approx_{p_0}$ id since ρ_0 is the most general solution to (p_0, q_0). □

An ascending sequence is a formalization of the notion that we mentioned before, i.e., a sequence whose each prefix is a path. Lemma 3.5.14 tells us that in a sequence given by a most general solution, there always exists a suffix that is a path. But this lemma does not say much about the length of such a suffix. Let us inspect the proof of Lemma 3.5.14: there are two cases in which we extend the most general solution. In the first case, we also extend the suffix that is a path. But in the second case ($\rho'_i(q_i) \geq_{\rho'} p_{i+1}$), the path gets reset to a sequence of length one. The following lemma shows that the second case can happen only finitely many times and thus we can always find an ascending sequence.

Lemma 3.5.19 (The Key Technical Lemma)**.** Assume that $\rightsquigarrow$ is composable and monotone, and that $(p_i)_{i \in \mathbb{N}}$, $(q_i)_{i \in \mathbb{N}}$ and $(\rho_i)_{i \in \mathbb{N}}$ are sequences such that $p_i \rightsquigarrow q_i$ and $\rho_i(q_i) = \rho_{i+1}(p_{i+1})$ for all i. Then there exists k such that the sequence $(p_k, q_k), (p_{k+1}, q_{k+1}), \ldots$ is ascending.

Proof. First informally: When we construct a most general solution to a prefix of $(p_i, q_i)_{i \leq \mathbb{N}}$, each extension done by Lemma 3.5.13 means that we apply a substitution $\hat{\rho}$ to the first element of the sequence. This means that the size of the first element increases (because of monotonicity). But the size of the first element cannot increase ad infinitum for this reason: $(\hat{\rho}_1 \circ \ldots \circ \hat{\rho}_k)(p_0) \geq \rho_0(p_0)$ must hold since a prefix of $(\rho_i)_{i \leq \mathbb{N}}$ is a solution to a corresponding prefix of $(p_i, q_i)_{i \leq \mathbb{N}}$ and $\hat{\rho}_1 \circ \ldots \circ \hat{\rho}_k$ is from a most general solution.

We proceed by contradiction. No such k exists; thus, there exists an infinite sequence $(i_j)_{j \in \mathbb{N}}$ such that $i_0 = 0$ and $(p_0, q_0), \ldots, (p_{i_j - 1}, q_{i_j - 1}) \not\leq p_{i_j}$ for all $j > 0$ and $(i_j)_{j \in \mathbb{N}}$ iterates all these cases (maximality). By Corollary 3.5.16 we get that $(p_0, q_0), \ldots, (p_{i_j - 1}, q_{i_j - 1}) \geq p_{i_j}$. Let $(\rho^j_i)_{i \leq j}$ denote the most general solution to $(p_0, q_0), \ldots, (p_j, q_j)$, which always exists by Lemma 3.5.14. Let Q_j denote $\rho^{i_j - 1}_{i_j - 1}(q_{i_j - 1})$. And finally let $\rho'_0 = \mathrm{id}$ and ρ'_j denote the substitution such that $Q_j \geq_{\rho'_j} p_{i_j}$ for all $j > 0$.

Now we prove by induction on j that $\rho^i_0 \approx_{p_0} \hat{\rho}'_j \circ \cdots \circ \hat{\rho}'_0$ for all $i_j \leq i < i_{j+1}$ and for some $\hat{\rho}'_l$ such that $\hat{\rho}'_l \approx_{Q_l} \rho'_l$ for all $l \leq j$. Base case $j = 0$: from the maximality of $(i_j)_{j \in \mathbb{N}}$ it follows that i_1 is the first index i when $(p_0, q_0), \ldots, (p_{i-1}, q_{i-1}) \geq p_i$. Therefore $(p_0, q_0), \ldots, (p_{i-1}, q_{i-1}) \leq p_i$ for all $i < i_1$ and therefore $\rho^i_0 \approx_{p_0} \mathrm{id} = \rho'_0$ for all $i < i_1$ (formally by induction and Lemmas 3.5.7 and 3.5.12). Inductive step $j = l$ and $l > 0$: $\rho^i_0 \approx_{p_0} \hat{\rho}'_{l-1} \circ \cdots \circ \hat{\rho}'_0$ for all $i_{l-1} \leq i < i_l$. Thus $(p_0, q_0), \ldots, (p_{i_l - 1}, q_{i_l - 1}) \geq p_{i_l}$ holds and by Lemmas 3.5.7 and 3.5.13 we obtain $\rho^{i_l}_0 \approx_{p_0} \hat{\rho}'_l \circ \cdots \circ \hat{\rho}'_0$ such that $\hat{\rho}'_l \approx_{Q_l} \rho'_l$. Because $(i_j)_{j \in \mathbb{N}}$ is maximal, $(p_0, q_0), \ldots, (p_{i-1}, q_{i-1}) \preceq p_i$ for all $i_l < i < i_{l+1}$ and $\rho^i_0 \approx_{p_0} \hat{\rho}'_l \circ \cdots \circ \hat{\rho}'_0$ (again formally by induction and Lemmas 3.5.7 and 3.5.12).

We reason for each $j > 0$: $(p_0, q_0), \ldots, (p_{i_j-1}, q_{i_j-1}) \npreceq p_{i_j}$ and also $(p_0, q_0), \ldots, (p_{i_j-1}, q_{i_j-1}) \succeq p_{i_j}$ gives us that $\rho'_j \not\approx_{Q_j}$ id. By Lemma 3.5.5 we get that $\mathsf{TV}(Q_j) \subseteq \mathsf{TV}(\rho_0^{i_j-1}(p_0))$. Using this, the fact that $\hat{\rho}'_j \approx_{Q_j} \rho'_j \not\approx_{Q_j}$ id and Lemma 3.5.1 we obtain $\mathsf{size}(\rho_0^{i_j}(p_0)) > \mathsf{size}(\rho_0^{i_j-1}(p_0))$ and finally again from Lemma 3.5.1 and $\rho_0^{i_j-1} \approx_{p_0} \rho_0^{i_{j-1}}$, it follows that

$$\mathsf{size}(\rho_0^{i_j}(p_0)) > \mathsf{size}(\rho_0^{i_{j-1}}(p_0)). \tag{11}$$

But since each $(\rho_i^{i_j})_{i \le i_j}$ is the most general solution to a prefix of the sequence $(p_0, q_0), \ldots$, there exist η_j for each $(\rho_i^{i_j})_{i \le i_j}$ such that $\rho_0(p_0) = \eta_j(\rho_0^{i_j}(p_0))$. Since $\mathsf{size}(\rho_0(p_0))$ is already fixed and (11) tells us that $\mathsf{size}(\rho_0^{i_j}(p_0))$ is an increasing sequence (in j), there must be j' such that $\mathsf{size}(\rho_0^{i_{j'}}(p_0)) > \mathsf{size}(\rho_0(p_0))$, which prevents the existence of $\eta^{j'}$ and is a contradiction to the fact that $(\rho_i)_{i \le i_{j'}}$ is a solution to the sequence $(p_0, q_0), \ldots, (p_{i_{j'}}, q_{i_{j'}})$. □

Please note that composability is crucial for the last lemma. Consider this example: the dependency relation $\rightsquigarrow$ is defined as $(\alpha \times \mathsf{nat})\ \mathsf{list} \rightsquigarrow (\mathsf{nat} \times \alpha)\ \mathsf{list}$ and we take an infinite sequence $((\alpha \times \mathsf{nat})\ \mathsf{list}, (\mathsf{nat} \times \alpha)\ \mathsf{list}), ((\alpha \times \mathsf{nat})\ \mathsf{list}, (\mathsf{nat} \times \alpha)\ \mathsf{list}), \ldots$. There is a most general solution $(\rho_i)_{i \le n}$ to any prefix of this sequence defined as $\rho_i = \alpha \mapsto \mathsf{nat}$ for all $i \le n$. But there is no ascending sequence since a prefix of an ascending sequence must be a path. However, $\rightsquigarrow$ is not composable since the only path is $((\alpha \times \mathsf{nat})\ \mathsf{list}, (\mathsf{nat} \times \alpha)\ \mathsf{list}, \mathrm{id})^{\rightsquigarrow}$ and $(\alpha \times \mathsf{nat})\ \mathsf{list} \downarrow (\mathsf{nat} \times \alpha)\ \mathsf{list}$.

We can view the last lemma also from a different perspective: if there exists a nonterminating sequence, we can find p and q such that $p \rightsquigarrow q$ and such that there exists a nonterminating sequence starting from p. We have to consider only finitely many such p's and q's since $\rightsquigarrow$ is finite, so there is no need to consider the infinitely many possible instantiations.

An ascending sequence is a key ingredient that allows us to prove the main result of this section because an ascending sequence always gives rise to a cycle.

Lemma 3.5.20. Let us assume that $\rightsquigarrow$ is finite, monotone and composable, then the following statements are equivalent:

1. $\rightsquigarrow^{\downarrow}$ is nonterminating
2. $\rightsquigarrow$ is cyclic

Proof. 2. implies 1. There exists a path $(p_i, q_i, \rho_i)^{\rightsquigarrow}_{i \le n}$. Since $\rightsquigarrow^{\downarrow}$ is the substitutive closure of $\rightsquigarrow$, it must hold that $\rho_i(p_i) \rightsquigarrow^{\downarrow} \rho_i(q_i)$ for all $i \le n$

and $\rho_i(q_i) = \rho_{i+1}(p_{i+1})$ for all $i < n$. Thus $p_0 \rightsquigarrow^{\downarrow} q_0 \rightsquigarrow^{\downarrow} \rho_1(q_1) \rightsquigarrow^{\downarrow} \cdots \rightsquigarrow^{\downarrow} \rho_{n-1}(q_{n-1}) \rightsquigarrow^{\downarrow} \rho_n(q_n)$. Since $\rho_n(q_n) \leq_\rho p_0$, we have $p_0 \rightsquigarrow^{\downarrow+} \rho(p_0)$ and thus we can conclude $p_0 \rightsquigarrow^{\downarrow+} \rho(p_0) \rightsquigarrow^{\downarrow+} \rho(\rho(p_0)) \rightsquigarrow^{\downarrow+} \rho(\rho(\rho(p_0))) \rightsquigarrow^{\downarrow+} \cdots$. Therefore $\rightsquigarrow^{\downarrow}$ is nonterminating.

1. implies 2. If $\rightsquigarrow^{\downarrow}$ does not terminate, there exists a nonterminating sequence. Thus there exist sequences $(p'_i)_{i\in\mathbb{N}}$ and $(q'_i)_{i\in\mathbb{N}}$ such that $p'_i \rightsquigarrow^{\downarrow} q'_i$ and $q'_i = p'_{i+1}$ for all i. The nonterminating sequence is then $p'_0 \rightsquigarrow^{\downarrow} p'_1 \rightsquigarrow^{\downarrow} p'_2 \rightsquigarrow^{\downarrow} \cdots$. From the definition of $\rightsquigarrow^{\downarrow}$ follows that there exist sequences $(p_i)_{i\in\mathbb{N}}$, $(q_i)_{i\in\mathbb{N}}$ and $(\rho_i)_{i\in\mathbb{N}}$ such that $p_i \rightsquigarrow q_i$ and $\rho_i(p_i) = p'_i$ and $\rho_i(q_i) = q'_i$ and since $q'_i = p'_{i+1}$, $\rho_i(q_i) = \rho_{i+1}(p_{i+1})$ holds for all i.

The key step is to find an ascending sequence $(p_k, q_k), (p_{k+1}, q_{k+1}), \ldots$ such that there exists also $k' > k$ and $(p_k, q_k) = (p_{k'}, q_{k'})$: We define $\Theta_k = \{(p_l, q_l).\ l \geq k\}$. Since $\rightsquigarrow$ is a finite relation, Θ_k is a finite set for all k (recall $p_i \rightsquigarrow q_i$) and is never empty. We use Lemma 3.5.19 to obtain k_1 such that the sequence $(p_{k_1}, q_{k_1}), (p_{k_1+1}, q_{k_1+1}), \ldots$ is ascending. If there does not exist a $k'_1 > k_1$ such that $(p_{k_1}, q_{k_1}) = (p_{k'_1}, q_{k'_1})$, we use Lemma 3.5.19 for subsequences $(p_i)_{i>k_1}$, $(q_i)_{i>k_1}$ and obtain the respective k_2. Because $\Theta_{k_1} \supsetneq \Theta_{k_2}$, the whole process must stop after at most $|\Theta_0|$ steps by finding k and $k' > k$ such that $(p_k, q_k), (p_{k+1}, q_{k+1}), \ldots$ is an ascending sequence and $(p_k, q_k) = (p_{k'}, q_{k'})$.

We define sequences $(r_i)_{i\leq n}$ and $(s_i)_{i\leq n}$ as following: $n = k' - k - 1$, $r_i = p_{k+i}$ and $s_i = q_{k+i}$ for all $i \leq n$. Since $(p_k, q_k), (p_{k+1}, q_{k+1}), \ldots$ is an ascending sequence, we get the most general solution $(\rho_i)_{i\leq n}$ to $(r_i, s_i)_{i\leq n}$ such that $(r_i, s_i, \rho_i)^{\rightsquigarrow}_{i\leq n}$ (by Lemma 3.5.18). From $(p_k, q_k) = (p_{k'}, q_{k'})$, it follows that $(r_0, s_0), \ldots, (r_n, s_n) \leq r_0$ and finally we get $\rho_n(s_n) \leq r_0$ by Lemmas 3.5.3 and 3.5.7. This concludes the proof that $\rightsquigarrow$ is cyclic. $\square$

3.5.4 From Acyclicity to a Decision Procedure

Since acyclicity of a finite $\rightsquigarrow$ is a finite problem, there should be a decision procedure for this problem. We introduce this procedure in Algorithm 1 and prove it correct.

Definition 3.5.21. We say that $\rightsquigarrow$ is *orthogonal* if for all p, q, p' and q' such that $(p, q) \neq (p', q')$ it holds that if $p \rightsquigarrow q$ and $p' \rightsquigarrow q'$ then $p \ \# \ p'$.

Orthogonality is another restriction on $\rightsquigarrow$. As we prove in the following lemma, this constraint guarantees that if two paths start from the same value p_0, then these paths are the same modulo equivalent substitutions. We need this property to restrict once more the search space of our algorithm.

Lemma 3.5.22. Let $\rightsquigarrow$ be orthogonal and monotone. If there exist two paths $(p_i, q_i, \rho_i)^{\rightsquigarrow}_{i\leq n}$ and $(p'_i, q'_i, \rho'_i)^{\rightsquigarrow}_{i\leq n}$ such that $p_0 = p'_0$, then $p_i = p'_i$, $q_i = q'_i$ and $\rho_i \approx_{p_i} \rho'_i$ for all $i \leq n$.

Proof. By contradiction: Let $k \leq n$ be the smallest k such that:

- Either $p_k \neq p'_k$: $k > 0$, $\rho_{k-1} \approx_{p_{k-1}} \rho'_{k-1}$, $\rho_{k-1}(q_{k-1}) \leq_{\rho_k} p_k$ and $\rho'_{k-1}(q_{k-1}) \leq_{\rho'_k} p'_k$. But this contradicts $p_k \ \# \ p'_k$ (by Lemma 3.5.3).
- Or $q_k \neq q'_k$: but $p_k = p'_k$, which contradicts $p_k \ \# \ p'_k$.
- Or $\rho_k \not\approx_{p_k} \rho'_k$: a) $k = 0$: contradiction to $\rho_0 \approx_{p_0} \mathsf{id}$ and $\rho'_0 \approx_{p_0} \mathsf{id}$. b) $k > 0$: $\rho_{k-1} \approx_{p_{k-1}} \rho'_{k-1}$, $\rho_{k-1}(q_{k-1}) \leq_{\rho_k} p_k$ and $\rho'_{k-1}(q_{k-1}) \leq_{\rho'_k} p_k$. By Lemma 3.5.3 we get a contradiction to uniqueness of $\leq_\rho$, i.e., if $p \leq_\rho q$ and $p \leq_{\rho'} q$, then $\rho =_q \rho'$.

□

We assume that we have a unification algorithm on types, which we again extend to $\mathcal{U}_\Sigma$. This algorithm is used internally in function HasCommonInst in Algorithm 1. Function HasCommonInst is used to test composability and orthogonality.

Lemma 3.5.23. $\neg$HasCommonInst(p, q) iff $p \ \# \ q$.

Proof. p and q have a common instance iff there exists p' such that $p' \leq_\rho p$ and $p' \leq_{\rho'} q$. Since ρ and ρ' can be different, we have to rename variables in q in the procedure. The unification then decides if there exists a common instance. □

Let us somewhat informally describe the cyclicity decision procedure presented in Algorithm 1. First of all, a syntactic comment: the algorithm contains ghost variables, which help us analyze the algorithm but are not used during the real computation. Commands where the ghost variables are used are always prefixed by character #. We use two ghost variables i and R_i, whose meaning will be explain later.

The core of the computation of the program happens in the main loop, i.e., between lines 48 and 58. We call the loop *reduction phase*. The value of i is an iteration counter of the reduction phase.

The variable *dep* is the only input of the algorithm and does not get changed during the whole computation. This is the relation for which we must check whether it contains a cycle or not. In order to do that, we start by discovering for each p from *dep*[10] a path starting from p and we store the beginning and end of such a path in dep_+. That is to say, if $(p, q) \in dep_+$, this means we have discovered a path from p to q. This path is a candidate for a cycle and therefore after each iteration we check if $q \leq p$ in the function IsAcyclic. Moreover, we store in dep_+ only the longest path from p that we have discovered so far. The function ReduceStep extends all paths from dep_+ by one step if it is possible. If

10 More precisely, for each p such that there exists q and $(p, q) \in dep$.

not, p is marked as final. Thanks to Lemma 3.5.22, there always exists a unique extension of dep_+ modulo renaming. That is to say, when we look for (p', q') on line 24, there is at most one such pair.

We store only the beginning and end of a path starting from p in dep_+ because this is enough information for the algorithm to check if this path comprises a cycle. But for the analysis of the algorithm, we also want to know intermediate steps of such a path and therefore we store these steps in ghost variable $R_i(p)$. The two following lemmas show that $R_i(p)$ gets defined in the ith iteration if p is not final and that $R_i(p)$ is a path starting from p of length $i + 1$ and that the algorithm does not miss any path. If dep_+ does not change after ReduceStep, it means no further cycle candidate exists and we can report that dep is acyclic.

Lemma 3.5.24. Let $p_0 \rightsquigarrow q$. Then p_0 is not final in the nth iteration of Check($\rightsquigarrow$) iff $R_n(p_0)$ gets defined in the nth iteration of Check($\rightsquigarrow$). Moreover, if $R_n(p_0)$ is defined, there exists a path $(p_i, q_i, \rho_i)^{\rightsquigarrow}_{i \le n}$ such that $R_n(p_0) = (p_i, q_i, \rho_i)_{i \le n}$.

Proof. $R_n(p_0)$ can get defined only in the nth iteration of the algorithm, since the ghost variable i is strictly increasing. Each p is not final at the beginning of computation and when it gets final, it stays final for the rest of computation. Then clearly p_0 gets final in the nth iteration iff $R_n(p_0)$ does not get defined in the nth iteration.

We prove that defined $R_n(p_0)$ comprises a path by induction on n. Base case ($n = 0$): $R_0(p_0)$ is defined at the beginning of the algorithm and defines a trivial path. Inductive step ($n = i + 1$): p_0 does not get final in the $(i + 1)$st step of the algorithm and therefore it was not final in the ith step either. We take the sequence $R_i(p_0) = (p_j, q_j, \rho_j)_{j \le i}$ defined in the ith step and we know that $(p_j, q_j, \rho_j)^{\rightsquigarrow}_{j \le i}$. We take q that is considered on line 23 in the $(i + 1)$st step such that $(p_0, q) \in dep_+$. It is clear that $q = \rho_i(q_i)$ and there must ρ_{i+1} and p_{i+1} such that $\rho_i(q_i) \le_{\rho_{i+1}} p_{i+1}$, otherwise p_0 would get final. Then $R_{i+1}(p_0)$ gets defined in the $(i + 1)$st step to $(p_j, q_j, \rho_j)_{j \le i+1}$ and $(p_j, q_j, \rho_j)^{\rightsquigarrow}_{j \le i+1}$. □

In the light of the previous lemma we will write $R_n(p) = (p_i, q_i, \rho_i)^{\rightsquigarrow}_{i \le n}$ from now on.

Lemma 3.5.25. Let $\rightsquigarrow$ be orthogonal and monotone. If there exists a path $(p_i, q_i, \rho_i)^{\rightsquigarrow}_{i \le n}$, then $R_n(p_0)$ gets defined in the nth iteration of Check($\rightsquigarrow$).

Proof. We do the proof by induction on n. Base case ($n = 0$): $R_0(p_0)$ is defined at the beginning of the algorithm. Inductive step $n = i + 1$: Let $(p_j, q_j, \rho_j)^{\rightsquigarrow}_{j \le i+1}$, then $(p_j, q_j, \rho_j)^{\rightsquigarrow}_{j \le i}$ is also a path and we can use the induction hypothesis and obtain by Lemma 3.5.24 the path $R_i(p_0) = (p'_j, q'_j, \rho'_j)^{\rightsquigarrow}_{j \le i}$ defined in the ith step. We take q that is considered on

Algorithm 1 The main algorithm

```
 1: function HasCommonInst($p$, $q$)
 2:     $p', q' \leftarrow$ rename variables in $p$ and $q$ apart
 3:     return $p'$ and $q'$ can be unified
 4: end function
 5:
 6: function IsOrthogonal($dep$)
 7:     return $\forall (p, q), (p', q') \in dep.\ (p, q) = (p', q') \vee \neg$HasCommonInst$(p, p')$
 8: end function
 9:
10: function IsMonotone($dep$)
11:     return $\forall (p, q) \in dep.\ \mathsf{TV}(q) \subseteq \mathsf{TV}(p)$
12: end function
13:
14: function IsAcyclic($dep_+$)
15:     return $\forall (p, q) \in dep_+.\ q \nleq p$
16: end function
17:
18: function IsComposable($q$, $dep$)
19:     return $\forall (p', q') \in dep.\ q \geq p' \vee \neg$HasCommonInst$(q, p')$
20: end function
21:
22: function ReduceStep($dep$, $dep_+$)
23:     for all $(p, q) \in dep_+$ such that final$(p)$ = false do
24:         if can find $(p', q') \in dep$ such that $q \leq_\rho p'$ then
25:             $dep_+ \leftarrow dep_+ \setminus (p, q) \cup (p, \rho(q'))$
26:             $\#R_i(p) \leftarrow R_{i-1}(p), (p', q', \rho)$
27:         else
28:             final$(p) \leftarrow$ true
29:             if $\neg$IsComposable$(q, dep)$ then
30:                 return fail
31:             end if
32:         end if
33:     end for
34:     return $dep_+$
35: end function
36:
37: function Check($dep$)
38:     $\#i = 0$
39:     for all $(p, q) \in dep$. final$(p) \leftarrow$ false
40:     #for all $(p, q) \in dep.\ R_0(p) \leftarrow (p, q, \mathsf{id})$
41:     if $\neg$IsOrthogonal$(dep)$ then
42:         return fail
43:     end if
44:     if $\neg$IsMonotone$(dep)$ then
45:         return fail
46:     end if
47:     $dep_+ \leftarrow dep$
48:     loop
49:         $\#i \leftarrow i + 1$
50:         $dep'_+ \leftarrow$ ReduceStep$(dep, dep_+)$
51:         if $dep_+ = dep'_+$ then
52:             exit loop
53:         end if
54:         $dep_+ \leftarrow dep'_+$
55:         if $\neg$IsAcyclic$(dep_+)$ then
56:             return fail
57:         end if
58:     end loop
59:     return success
60: end function
```

line 23 in the $(i+1)$st step such that $(p_0, q) \in dep_+$. It is clear that $q = \rho'_i(q'_i)$. By Lemma 3.5.22 we get that $p_j = p'_j$, $q_j = q'_j$ and $\rho_j \approx_{p_j} \rho'_j$ for all $j \leq i$ and from the definition of $(p_j, q_j, \rho_j)^{\leadsto}_{j \leq i+1}$ that $\rho_i(q_i) \leq_{\rho_{i+1}} p_{i+1}$. By Lemma 3.5.3 we can finally derive that $\rho'_i(q'_i) \leq p_{i+1}$ and therefore p_0 cannot get final in the $(i+1)$st step and $R_{i+1}(p_0)$ gets defined in this step (by Lemma 3.5.24). □

Algorithm 1 can either return **success** (we write $\textsc{Check}(\leadsto) =$ **success**) or **fail** (we write $\textsc{Check}(\leadsto) =$ **fail**) or not terminate (we write $\textsc{Check}(\leadsto) = \uparrow$).

As we know from Section 3.5.3, we reduce termination to cyclicity only under some assumptions, where the most important one is composability. Our algorithm also checks composability of *dep*, which is done during the reduction phase for two reasons: 1) It is too late to do it after the phase because when the composability does not hold, the reduction phase may fail to terminate. 2) We cannot do it before because composability must be checked for all possible paths and we have to be sure that there exists no infinite path by checking also acyclicity dynamically, which is the goal of the reduction phase.

In order to make our algorithm more efficient, we check composability only for paths that start at some p that is final. The next lemma shows that this suffices. The key observation is that if we can extend a path, composability still holds locally.

Lemma 3.5.26. If $\textsc{Check}(\leadsto) \neq$ **fail**, then $\leadsto$ is monotone, composable, orthogonal and acyclic.

Proof. $\leadsto$ is monotone, since this property is checked directly at the beginning of the algorithm. The same holds for orthogonality by using Lemma 3.5.23.

We prove composability by contradiction: There must exist a path $(p_i, q_i, \rho_i)^{\leadsto}_{i \leq n}$ and p and q such that $p \leadsto q$ and $\rho_n(q_n) \downarrow p$. Then from Lemmas 3.5.24 and 3.5.25 it follows that $R_n(p_0) = (p'_i, q'_i, \rho'_i)^{\leadsto}_{i \leq n}$ was defined in the nth iteration of $\textsc{Check}(\leadsto)$ and by Lemma 3.5.22 that $p_i = p'_i$, $q_i = q'_i$ and $\rho_i \approx_{p_i} \rho'_i$ for all $i \leq n$. Then $\rho'_n(q'_n) \downarrow p$ follows from Lemma 3.5.3. In the $(n+1)$st iteration two cases can occur:

- There do not exist any p' and q' such that $p' \leadsto q'$ and $\rho'_n(q'_n) \leq p'$. Then $\textsc{IsComposable}(\rho'_n(q'_n), \leadsto)$ is executed. This function checks that $\rho'_n(q'_n) \geq p'$ or $\rho'_n(q'_n) \ \# \ p'$ (by Lemma 3.5.23) for all (p', q') such that $p' \leadsto q'$. This is in contradiction to $\rho'_n(q'_n) \downarrow p$.

- There exist p' and q' such that $p' \leadsto q'$ and $\rho'_n(q'_n) \leq p'$. But then for all $p'' \neq p'$ and q'' such that $p'' \leadsto q''$, $\rho'_n(q'_n) \ \# \ p''$ holds (otherwise $\leadsto$ would not be orthogonal). But this is again a contradiction to $\rho'_n(q'_n) \downarrow p$.

We prove acyclicity by contradiction as well: If $\rightsquigarrow$ is cyclic, there exists a path $(p_i, q_i, \rho_i)^{\rightsquigarrow}_{i \leq n}$ such that $\rho_n(q_n) \leq p_0$. By Lemmas 3.5.24 and 3.5.25 we obtain another path $R_n(p_0) = (p'_i, q'_i, \rho'_i)^{\rightsquigarrow}_{i \leq n}$ defined in the nth step of the algorithm and by Lemmas 3.5.3 and 3.5.22 finally $\rho'_n(q'_n) \leq p'_0$. But this means that the cyclicity check on line 55 returns **fail**. □

Let $\mathsf{P}(\rightsquigarrow)$ abbreviate the conjunction of the following properties:

- $\rightsquigarrow$ is monotone,
- $\rightsquigarrow$ is composable,
- $\rightsquigarrow$ is orthogonal,
- $\rightsquigarrow$ is acyclic.

Lemma 3.5.27. If $\rightsquigarrow$ is finite then the following holds:

- Check$(\rightsquigarrow)$ always terminates,
- Check$(\rightsquigarrow)$ = **success** if and only if $\mathsf{P}(\rightsquigarrow)$.

Proof. We start by proving termination by contradiction. The only way how Check$(\rightsquigarrow)$ can fail to terminate is when the program never exits the reduction phase. This happens when the check on line 51 is always false,[11] i.e., when the function ReduceStep always changes dep_+, which means we can always find $(p', q') \in dep$ on line 24 such that $q \leq_\rho p'$. That means there exists an infinite sequence $R_0(p), R_1(p), \ldots$ for a certain p. From this sequence, we can easily construct an infinite sequence $\rho_0(p_0) \rightsquigarrow^{\downarrow} \rho_1(p_1) \rightsquigarrow^{\downarrow} \rho_2(p_2) \rightsquigarrow^{\downarrow} \cdots$. Since Check$(\rightsquigarrow)$ does not return **fail**, $\rightsquigarrow$ is monotone, composable, orthogonal and acyclic by Lemma 3.5.26 and thus we can invoke Lemma 3.5.20 and get that $\rightsquigarrow$ is cyclic, which is a contradiction.

Now we continue by the proof of the equivalence. Left-to-right: By Lemma 3.5.26. Right-to-left: Since we proved that the algorithm terminates, we know that Check$(\rightsquigarrow) \neq$ **fail** implies Check$(\rightsquigarrow)$ = **success** and therefore it suffices to prove that Check$(\rightsquigarrow) \neq$ **fail**. The algorithms can return fail only on lines 42, 45, 30 and 56. We prove by contradiction that the program cannot return **fail** on any of those lines:

- If **fail** is returned on line 42, it means that $\rightsquigarrow$ is not orthogonal (by Lemma 3.5.23).
- If **fail** is returned on line 45, it means that $\rightsquigarrow$ is not monotone.

11 And when the check on line 55 does not detect a cycle.

- If **fail** is returned on line 30 in the nth iteration, there exist p and a path $R_{n-1}(p) = (p_i, q_i, \rho_i)^{\rightsquigarrow}_{n-1}$ by Lemma 3.5.24 such that for all p' and q' such that $p' \rightsquigarrow q'$, we have $\rho_{n-1}(q_{n-1}) \nleq p'$. Moreover, it has to hold that $\neg\textsc{IsComposable}(\rho_{n-1}(q_{n-1}), \rightsquigarrow)$ and therefore there exist p' and q' such that $p' \rightsquigarrow q'$ and $\neg(\rho_{n-1}(q_{n-1}) \ \# \ p')$ and $\rho_{n-1}(q_{n-1}) \ngeq p'$. But this means $\rho_{n-1}(q_{n-1}) \downarrow p'$ and therefore $\rightsquigarrow$ is not composable.

- If **fail** is returned on line 56 in the nth iteration, there exist p_0 that is not final[12] and a path $R_n(p_0) = (p_i, q_i, \rho_i)^{\rightsquigarrow}_n$ (by Lemma 3.5.24) such that $\rho_n(q_n) \leq p_0$. But this means that $\rightsquigarrow$ is cyclic.

□

Now we can formulate a theorem that we found a solution to The Termination Problem.

Theorem 4. There exists a predicate P on binary relations on $\mathsf{Type}^\bullet \cup \mathsf{CInst}^\bullet$ such that for finite relations $\rightsquigarrow$ the following holds:

- $P(\rightsquigarrow)$ is decidable
- $P(\rightsquigarrow)$ implies that $\rightsquigarrow^\downarrow$ terminates
- P contains interesting relations $\rightsquigarrow$

Proof. Lemma 3.5.27 shows that P is decidable by the program Check. Lemma 3.5.20 shows that $\mathsf{P}(\rightsquigarrow)$ implies termination of $\rightsquigarrow^\downarrow$.

Now we proceed to the last question: do the restrictions to relations $\rightsquigarrow$ that are monotone, composable, and orthogonal still allow for suitable expressiveness for overloaded definitions? Monotonicity and orthogonality are such natural conditions that one would expect that any reasonable (overloaded) definitions must fulfill them. We will argue now that composability still admits all main use cases of overloading in Isabelle:

- In the context of type classes, only what Haftmann and Wenzel [30] call *restricted* overloading is allowed: constants can be declared only with a linear polymorphic type, e.g., $c_{\alpha\ \tau}$. Overloaded definitions have this form $c_{(\overline{\alpha}\ k)\ \tau} = \ldots c_{\alpha_i\ \tau} \ldots$, i.e., if c appears on the right hand side, it uses some α_i from $\overline{\alpha}$. Such definitions generate only composable dependency relations.

- Our experience shows that all cases of unrestricted overloading that have been required by users so far also fulfill composability. A classical example would be a basic concept of nth power of composition of f_α (written f^n) defined in Isabelle/HOL. This operation is then overloaded for unary functions ($f_{\alpha\to\alpha}$), binary relations ($f_{\alpha\to\alpha\to\mathsf{bool}}$) and relations as sets ($f_{(\alpha\times\alpha)\ \mathsf{set}}$).

12 Otherwise the cycle would have been detected in the previous iteration.

$\square$

Finally, we can state the main result of this section.

Theorem 5. The property of D of being composable and well formed is decidable.

Proof. The algorithm CHECK checks that D is definitional, orthogonal and composable, and that $\rightsquigarrow^{\downarrow}$ terminates. Thus, CHECK decides whether D is composable and well formed. $\square$

3.5.5 Issues with the Original Algorithm

During inspection of the original cyclicity checker and during the subsequent formalization, we identified three issues:

- Completeness issue: The original algorithm (Isabelle2014) does not always terminate because the cyclicity check (function IsACYCLIC) misses some cycles and therefore the reduction phase might loop. Leaving out details, consider this minimal example:

  $$a_{\alpha\,\mathsf{list}\times\beta} \rightsquigarrow a_{\alpha\,\mathsf{list}\times\alpha}.$$

  The algorithm concludes that $\rightsquigarrow$ must be acyclic since α in $a_{\alpha\,\mathsf{list}\times\alpha}$ is also contained in $a_{\alpha\,\mathsf{list}\times\beta}$. But $\rightsquigarrow$ is obviously cyclic. Instead, we use solely an instance test ($q \not\leq p$) in our modified algorithm. The issue was resolved for the Isabelle2015 release.

- Completeness issue: In the original algorithm (Isabelle2015), composability is checked[13] at the end of the algorithm after the reduction phase is finished. But if the composability does not hold, the reduction phase may fail to terminate, as in the following example:

  $$a_{\mathsf{int}} \rightsquigarrow b_{\mathsf{int}\times\mathsf{nat}},\ b_{\alpha\times\mathsf{nat}} \rightsquigarrow c_{\alpha\times\mathsf{nat}},\ c_{\mathsf{int}\times\alpha} \rightsquigarrow b_{\mathsf{int}\times\alpha}$$

  $\rightsquigarrow$ is not composable because $c_{\alpha\times\mathsf{nat}} \downarrow c_{\mathsf{int}\times\alpha}$. But this is never detected in the original algorithm for this reason: starting from a_{int} the reduction phase does not terminate since no cycle is detected ($c_? \not\leq a_{\mathsf{int}}$ and $b_? \not\leq a_{\mathsf{int}}$).

  We modified the algorithm such that composability is checked during the reduction phase. But the change is subtler than just moving the original test into the reduction phase because then the complexity increases from $\mathcal{O}(|dep|^2)$ to $\mathcal{O}(|dep|^3)$. Therefore we test composability only for p's that are final, i.e., for which the reduction phase terminates. This suffices by Lemma 3.5.26 and preserves quadratic complexity.

13 By $\forall(p, q) \in dep_+.\ \textsc{IsComposable}(q, dep)$.

- Correctness issue: Andrei Popescu found the following soundness issue caused by the cyclicity checker:

  $$\begin{aligned}&\texttt{consts}\ \mathsf{c} : \alpha \to \mathsf{bool}\\ &\texttt{consts}\ \mathsf{d} : (\alpha \times \beta) \to \mathsf{bool}\\ &\texttt{defs}\ \mathsf{c}_{\alpha\to\mathsf{bool}} \quad\quad = \lambda x.\ \mathsf{d}\ \mathsf{any}_{\alpha\times\alpha}\\ &\texttt{defs}\ \mathsf{d}_{(\alpha\times\mathsf{nat})\to\mathsf{bool}} = \lambda x.\ \neg\,(\mathsf{c}\ \mathsf{any}_{\alpha})\end{aligned}$$

  This input is accepted by Isabelle2013-2 and leads to an inconsistency since the following can be proved:

  $$\mathsf{c}\ \mathsf{any}_{\mathsf{nat}} = \neg\,(\mathsf{c}\ \mathsf{any}_{\mathsf{nat}})$$

  The derived dependency relation is as follows:

  $$c_{\alpha\to\mathsf{bool}} \rightsquigarrow d_{(\alpha\times\alpha)\to\mathsf{bool}},\ d_{(\alpha\times\mathsf{nat})\to\mathsf{bool}} \rightsquigarrow c_{\alpha\to\mathsf{bool}}$$

  $\rightsquigarrow$ is not composable since $d_{(\alpha\times\alpha)\to\mathsf{bool}} \downarrow d_{(\alpha\times\mathsf{nat})\to\mathsf{bool}}$. There exists a cycle (substitute nat for α) but this cycle is not detected by the cyclicity check and the reduction phase terminates. The issue is that the composability check (function IsComposable) in the original algorithm was implemented as follows:

  $$\begin{aligned}&\text{type of } q \text{ has the same shape as } \mathsf{tpOf}(q)\ \vee\\ &\forall(p', q') \in \mathit{dep}.\ \neg\textsc{HasCommonInst}(q, p')\end{aligned}$$

  And indeed the type of $d_{(\alpha\times\alpha)\to\mathsf{bool}}$ has the same shape as d's declared type $(\alpha \times \beta) \to \mathsf{bool}$.

  The issue was amended for Isabelle2014 release by changing the condition from having the same shape to

  $$\text{type of } q \text{ is alpha equivalent to } \mathsf{tpOf}(q).$$

  Our work clarifies this issue in two ways:

  - The proof shows that the check from Isabelle2014 is correct because it is strictly stronger than the check that I propose in this thesis: if the type of q is alpha equivalent to $\mathsf{tpOf}(q)$ and q and p' have a common instance, then $q \geq p'$.
  - The current composability check can be generalized to

    $$\forall(p', q') \in \mathit{dep}.\ q \geq p' \vee \neg\textsc{HasCommonInst}(q, p'),$$

    which allows more instances of overloading to be accepted and does not require any other change of the algorithm.

At the time of writing this thesis (Isabelle2015), the second completenesses issue was still unresolved and was still under scrutiny at the Isabelle headquarters. Isabelle theory files illustrating all three issues can be found on my web page [46].

3.6 discussion

After this laborious work, let me add my last thought on this topic. I would like to argue that we should not regard the inconsistency that I found as *just another bug* that had to be solved, but we can learn a more general lesson from this: in my opinion, the main factor why this issue managed to escape from being discovered for so many years is usage of syntactic methods to justify correctness of overloading.

Apparently it is tempting to persuade yourself that we can remove overloaded constants from any term by unfolding their occurrences according to their definitions and thus produce a unique, finite term without these constants. One just easily misses that the term still contains types whose meanings can still depend on the overloaded constants in question.

The semantic approach goes in the other direction—bottom-up. It forces us first to explain the meaning of these types before we move on to an interpretation of terms. This approach is more rigid, thus less error prone, or if you forgive me this analogy, it is strongly "type checked".

I can still remember many afternoons (see the quote from the beginning of the chapter) when Andrei Popescu and I worked together on the interpretation of types and overloaded constants and when our solutions just did not want to start working. We were literally going round in circles.

Write down the definition of a polymorphic function on a piece of paper. Tell me its type, but be careful not to let me see the function's definition. I will tell you a theorem that the function satisfies.

— Philip Wadler (1989)

4 RELATIONAL PARAMETRICITY IMPLEMENTED: TRANSFER

The primary function of the Transfer tool is to transfer theorems from one type to another, by proving equivalences between pairs of related propositions. Let us consider a series of motivational examples:

- Natural numbers are carved out from the axiomatic infinite type ind and integer numbers are defined as a quotient of pairs of natural numbers (see (1) on page 20). Thus, those two types were defined independently and although each has its own implementation of numerals, arithmetic operations, comparisons, etc., we would still expect that they are related. For example, we would expect that we can derive the following two theorems from each other:[1]

$$\forall x_{\mathsf{nat}}.\quad \forall y_{\mathsf{nat}}.\quad x + y = 0 \longrightarrow x = 0 \wedge y = 0 \qquad (12)$$

$$\forall x_{\mathsf{int}} \in \{0..\}.\ \forall y_{\mathsf{int}} \in \{0..\}.\ x + y = 0 \longrightarrow x = 0 \wedge y = 0 \qquad (13)$$

  Another example: we would like to obtain from the following theorem about integers

$$\forall x_{\mathsf{int}}.\ x < x + 1 \qquad (14)$$

  the corresponding theorem about natural numbers

$$\forall n_{\mathsf{nat}}.\ n < n + 1. \qquad (15)$$

- We define the type of finite sets α fset as a restriction of the set type α set. Assuming that we have already defined the fset-version of image and union operators

$$\mathsf{fimage} : (\alpha \to \beta) \to \alpha\ \mathsf{fset} \to \beta\ \mathsf{fset}$$
$$\widetilde{\bigcup} : \alpha\ \mathsf{fset}\ \mathsf{fset} \to \alpha\ \mathsf{fset},$$

  we would like to reduce a proof of the statement

$$\forall S_{\alpha\ \mathsf{fset}}.\ \widetilde{\bigcup}\ \mathsf{fimage}\ (\lambda x_\alpha.\ \{x\})\ S = S \qquad (16)$$

  to a proof of the equivalent statement for α set

$$\forall S_{\alpha\ \mathsf{set}}.\ \mathsf{finite}\ S \longrightarrow \bigcup\ \mathsf{image}\ (\lambda x_\alpha.\ \{x\})\ S = S. \qquad (17)$$

1 The term $\{0..\}$ denotes the set of all integer numbers greater or equal to 0.

- We defined rational numbers as a partial quotient of integer pairs (see (2) on page 21). We want to reduce the proof of the statement

$$1_{\mathsf{rat}} = \frac{2_{\mathsf{int}}}{2_{\mathsf{int}}} \tag{18}$$

to the proof of the statement

$$\mathsf{ratrel}\ (1_{\mathsf{int}}, 1_{\mathsf{int}})\ (2_{\mathsf{int}}, 2_{\mathsf{int}}). \tag{19}$$

Notice that we can naturally relate the pairs of the involved types in each of the examples: obviously, we can relate 0_{int} to 0_{nat}, 1_{int} to 1_{nat} and so on; since the type of finite sets is a subtype of the set type, the relation is a restricted identity here; and finally, every rational number could be related to any representative in the corresponding equivalence class (e.g., 1_{rat} relates to $(1_{\mathsf{int}}, 1_{\mathsf{int}}), (2_{\mathsf{int}}, 2_{\mathsf{int}}), \ldots$). To represent types as relations is the key technique that enables us to perform the above theorem transformations. I will make this observation more formal in Section 4.1.

The process of transferring theorems from one type to another is guided by an extensible collection of *transfer rules*, which establish connections between pairs of related types or constants (e.g., 0_{int} and 0_{nat}, or image and fimage). I will present the transfer algorithm in Section 4.2.

Transferring (16) to (17) poses an extra challenge because of nesting of types—we transfer α fset fset to α set set. For this we need parametrized transfer rules, which I will explain in Section 4.3.

Notice that we transfer = to = in the fset example, but in the example with rational numbers, we want to replace = by the equivalence relation ratrel while transferring (18) to (19). Thus, we need multiple transfer rules for equality for different situations, which yields the notion of transfer rules with side conditions, the topic of Section 4.4.

Observe that (14) and (15) are not equivalent (unlike other examples), but the former is stronger than the latter since we do not require that $x_{\mathsf{int}} \geq 0$. We will explore the possibility to prove an implication instead of an equivalence between the pair of theorems in Section 4.5.

In Section 4.6, I will present the automation for proving transfer rules—the transfer_prover method.

To make transferring work, we need some additional properties of the type constructors involved in the transferring. I will define a class of type constructors possessing these properties in Section 4.7 and show that the class is a very large class—in particular it contains all (co)datatypes.

Section 4.8 briefly introduces the interface of the Transfer tool, i.e., which methods and attributes it offers.

In Section 4.9, we will explore the limitation of the Transfer tool and the corresponding future work. We finally conclude by related work in Section 4.10.

The original idea and initial implementation of the Transfer tool is due to Brian Huffman. I contributed by various improvements and extensions later. This chapter is based on our joint paper [40].

4.1 types as relations

The design of the Transfer tool is based on the idea of viewing types as binary relations. The notions of *relational parametricity* by Reynolds [83], *free theorems* by Wadler [95], and last but not least, *representation independence* by Mitchell [65] were primary sources of inspiration.

I will explain the notions of relational parametricity and representation independence now.

4.1.1 Relational Parametricity

Relational parametricity tells us that different type instances of a parametrically polymorphic function must behave uniformly—that is, they must be related by a binary relation derived from the function's type. For example, the standard filter function on lists satisfies the parametricity property shown below in (21).

$$\mathsf{filter} : (\alpha \to \mathsf{bool}) \to \alpha\ \mathsf{list} \to \alpha\ \mathsf{list} \tag{20}$$

$$\forall A_{\alpha \to \beta \to \mathsf{bool}}.\ ((A \Mapsto =) \Mapsto \mathsf{list\_all2}\ A \Mapsto \mathsf{list\_all2}\ A)\ \mathsf{filter}\ \mathsf{filter} \tag{21}$$

This parametricity property means that if predicates P_1 and P_2 agree on related inputs, i.e., $A\ x_1\ x_2$ implies $P_1\ x_1 = P_2\ x_2$, then filter P_1 and filter P_2 applied to related lists will yield related results.[2] The parametricity property formally captures our intuition that different instances of a polymorphic function must behave essentially in the same way. Thus for example the following behavior of filter is ruled out by its parametricity property: if $\alpha = \mathsf{int}$, filter $P\ xs$ filters out the integer number 42 from xs even if $P\ 42 = \mathsf{True}$.

In general, we derive the statement of the parametricity property for term $t : \sigma$ by generating a corresponding binary relation from its type. First, we assume that there exists a relator for every nonnullary type constructor in σ. Relators lift relations over type constructors: Related data structures have the same shape, with pointwise-related elements, and related functions map related input to related output (see Figure 3 on the next page). The derivation is done purely syntactically. We start with σ and replace every nullary type constructor (e.g., bool or int) by the identity relation =, every nonnullary type constructor by its corresponding relator

2 Wadler-style free theorems are derived by instantiating A with the graph of a function f; in this manner, we can obtain a rule stating essentially that filter commutes with map.

$$\begin{aligned}
&\mathsf{rel\_prod} : (\alpha \to \gamma \to \mathsf{bool}) \to (\beta \to \delta \to \mathsf{bool}) \to \alpha \times \beta \to \gamma \times \delta \to \mathsf{bool}\\
&\Mapsto : (\alpha \to \gamma \to \mathsf{bool}) \to (\beta \to \delta \to \mathsf{bool}) \to (\alpha \to \beta) \to (\gamma \to \delta) \to \mathsf{bool}\\
&\mathsf{rel\_set} : (\alpha \to \beta \to \mathsf{bool}) \to \alpha\ \mathsf{set} \to \beta\ \mathsf{set} \to \mathsf{bool}\\
&\mathsf{list\_all2} : (\alpha \to \beta \to \mathsf{bool}) \to \alpha\ \mathsf{list} \to \beta\ \mathsf{list} \to \mathsf{bool}
\end{aligned}$$

$$\begin{aligned}
(\mathsf{rel\_prod}\ A\ B)\ x\ y &= A\ (\mathsf{fst}\ x)\ (\mathsf{fst}\ y) \wedge B\ (\mathsf{snd}\ x)\ (\mathsf{snd}\ y)\\
(A \Mapsto B)\ f\ g &= \forall x\ y.\ A\ x\ y \longrightarrow B\ (f\ x)\ (g\ y)\\
(\mathsf{rel\_set}\ A)\ X\ Y &= (\forall x \in X.\ \exists y \in Y.\ A\ x\ y) \wedge\\
&\ (\forall y \in Y.\ \exists x \in X.\ A\ x\ y)\\
(\mathsf{list\_all2}\ A)\ xs\ ys &= (\mathsf{length}\ xs = \mathsf{length}\ ys) \wedge\\
&\ (\forall (x, y) \in \mathsf{set}\ (\mathsf{zip}\ xs\ ys).\ \ A\ x\ y)
\end{aligned}$$

Figure 3: Relators for various type constructors

(e.g., $\to$ by $\Mapsto$ or list by list_all2) and finally every type variable α by a term variable $A_{\alpha_1 \to \alpha_2 \to \mathsf{bool}}$. We call any relation that is derived by this process a *parametricity relation*. We derived the relation in (21) from filter's type (20) by the same process.

If the parametricity property holds for some t, we say that t is *parametric*. Not every constant in Isabelle/HOL is parametric. We will explore this in Section 4.4.

4.1.2 Representation Independence

Representation independence is one useful application of relational parametricity. Mitchell [65] used it to reason about data abstraction in functional programming. Imagine we have an interface to an abstract datatype (e.g. queues) with two different implementations. We would hope for any queue-using program to behave identically no matter which queue implementation is used—i.e., that the two queue implementations are *contextually equivalent*. Representation independence implies that this is so, as long as we can find a relation between the two implementation types that is preserved by all the corresponding operations.

The Transfer tool is essentially a working implementation of the idea of representation independence, but in a slightly different setting: Instead of a typical functional programming language, we use higher-order logic; and instead of showing contextual equivalence of programs, we show logical equivalence of propositions.

In order to show that some propositions are related by the logical equivalence relation, we need facts that witness that smaller parts of the propositions (usually constants) are related by some relation as well. We call such facts transfer rules. Formally, any theorem of the form $R\ t\ t'$ where R is a binary relation of the type $\sigma \to \tau \to \mathsf{bool}$ is a *transfer rule*. Usually t and t' are two different constants and σ and τ are of the form $\overline{\alpha}\ \kappa$ for some type constructors κ. Given the transfer rule $R\ t\ t'$, we call t the left-hand side and t' the right-hand side of the rule and we call R a *transfer relation*. Thus every parametricity property is a transfer rule and every parametricity relation is a transfer relation. The converse does not hold. Motivated by this terminology, we also call a parametricity property a *parametricity rule*.

A trivial example of a transfer rule is the reflexivity $t = t$. A more involved example would be the transfer rule

$$(\mathsf{ZN} \Mapsto \mathsf{ZN} \Mapsto \mathsf{ZN})\ (+_{\mathsf{int}\to\mathsf{int}\to\mathsf{int}})\ (+_{\mathsf{nat}\to\mathsf{nat}\to\mathsf{nat}}),$$

which relates the addition on integer numbers with the addition on natural numbers. The relation $\mathsf{ZN} : \mathsf{int} \to \mathsf{nat} \to \mathsf{bool}$, whose definition is not important here, relates corresponding integer and natural numbers.

We can view transfer rules as generalization of parametricity rules. Transfer rules may relate two different constants, and in addition to identity relations and the standard relators, it may also use specific relations between distinct types or type constructors as for example ZN.

4.1.3 Example: int/nat Transfer

We consider the example from the beginning of this chapter and transfer propositions between the integer and natural numbers. Recall that these types were logically defined independently, but we can still think of type int as a concrete representation of the more abstract type nat. To specify the connection between the two types, we define a transfer relation $\mathsf{ZN} : \mathsf{int} \to \mathsf{nat} \to \mathsf{bool}$ as

$$\mathsf{ZN}\ x\ n = (x = \mathsf{int}\ n), \tag{22}$$

where the conversion function $\mathsf{int} : \mathsf{nat} \to \mathsf{int}$ is defined as

$$\begin{aligned} \mathsf{int}\ 0 &= 0 \\ \mathsf{int}\ (\mathsf{Suc}\ n) &= \mathsf{int}\ n + 1. \end{aligned}$$

We can then use ZN to express relationships between constants in the form of transfer rules. Obviously, the integers 0 and 1 corresponds to the natural numbers 0 and 1. The respective addition operators map related arguments to related results. Similarly, order operators (less-than and less-than-or-equal-to) on integers are related to corresponding operators on naturals. Finally, bounded quantification over the nonnegative integers corresponds to universal quantification over type nat.

$$\mathsf{ZN}\ 0_{\mathsf{int}}\ 0_{\mathsf{nat}} \tag{23}$$

$$\mathsf{ZN}\ 1_{\mathsf{int}}\ 1_{\mathsf{nat}} \tag{24}$$

$$(\mathsf{ZN} \Mapsto \mathsf{ZN} \Mapsto \mathsf{ZN})\ +\ + \tag{25}$$

$$(\mathsf{ZN} \Mapsto \mathsf{ZN} \Mapsto =)\ <\ < \tag{26}$$

$$(\mathsf{ZN} \Mapsto \mathsf{ZN} \Mapsto =)\ \le\ \le \tag{27}$$

$$((\mathsf{ZN} \Mapsto =) \Mapsto =)\ (\mathsf{Ball}\ \{0..\})\ \mathsf{All} \tag{28}$$

Those rules allow us to replace 0_{nat} in a term by 0_{int}, + on natural numbers by + on integer numbers, quantification over all natural numbers by bounded quantification and so on. However, this replacement is not simple term replacement, we replace the subterms such that the resulting term after the replacement is provably equivalent to the original one.

Thus the Transfer tool can use the rules above to derive equivalences like the following:

$$(\forall x_{\mathsf{int}} \in \{0..\}.\ x < x + 1) \longleftrightarrow (\forall n_{\mathsf{nat}}.\ n < n + 1) \tag{29}$$

Using the replacement analogy, we can imagine that the subterms on the right-hand side of the equivalence got replaced by the corresponding terms on the left-hand side. In general, Transfer can handle any lambda term constructed from constants for which it has transfer rules. In other words, we can prove that two terms are equivalent if there are corresponding transfer rules enabling the replacement between both sides.

4.2 transfer algorithm

The core functionality of the Transfer tool is to prove equivalence theorems in the style of (29). To derive an equivalence theorem, the Transfer tool uses transfer rules for constants, along with elimination and introduction rules for $\Mapsto$.

$$\frac{(A \Mapsto B)\ f\ g \qquad A\ x\ y}{B\ (f\ x)\ (g\ y)}\ (\Mapsto\text{-Elim})$$

$$\frac{\forall x\ y.\ A\ x\ y \longrightarrow B\ (f\ x)\ (g\ y)}{(A \Mapsto B)\ f\ g}\ (\Mapsto\text{-Intro})$$

Alternatively, these rules can be restated in the form of structural typing rules, similar to those for the simply typed lambda calculus. A typing judgment here involves two terms instead of one, and a binary relation takes the place of a type. The environment Γ collects the local assumptions for bound variables.

$$\frac{A\ x\ y \in \Gamma}{\Gamma \vdash A\ x\ y}\ (\textsc{Var}) \qquad \frac{\Gamma_1 \vdash (A \Mapsto B)\ f\ g \qquad \Gamma_2 \vdash A\ x\ y}{\Gamma_1 \cup \Gamma_2 \vdash B\ (f\ x)\ (g\ y)}\ (\textsc{App})$$

$$\frac{\Gamma,\ A\ x\ y \vdash B\ (f\ x)\ (g\ y)}{\Gamma \vdash (A \Mapsto B)\ (\lambda x.\ f\ x)\ (\lambda y.\ g\ y)}\ (\textsc{Abs})$$

To transfer a theorem requires us to build a derivation tree using these rules, with transfer rules for constants at the leaves of the tree. For the transfer method, we are given only the abstract right-hand side; for the transferred attribute, only the left-hand side. The job of the Transfer tool is to fill in the remainder of the tree—essentially a type inference problem.

Our implementation splits the process into two steps:

1. **Skeleton step.** This step is to determine the overall shape of the derivation tree: the arrangement of App, Abs, and Var nodes, and the pattern of unknown term and relation variables. The step starts by building a skeleton term s of the known term t—a lambda term with the same structure, but with constants replaced by fresh variables. Using Isabelle's standard type inference algorithm, we annotate s with types; the inferred types determine the pattern of relation variables in the derivation tree.

2. **Search step.** This step is to fill in the leaves of the tree using the collection of transfer rules, at the same time instantiating the unknown variables. We set up a schematic proof state with one goal for each leaf of the tree, and then match transfer rules with subgoals. We use backtracking search in case multiple transfer rules match a given left- or right-hand side.

As an example, we will transfer the proposition that the $\leq$ relation on natural numbers is reflexive: $\forall n_{\mathsf{nat}}.\ n \leq n$. This is actually syntax for

$$\mathsf{All}\ (\lambda n_{\mathsf{nat}}.\ \mathsf{le}\ n\ n),$$

so its skeleton has the form $t\ (\lambda x.\ u\ x\ x)$. Type inference yields a most general typing with $t : (\alpha \to \beta) \to \gamma$ and $u : \alpha \to \alpha \to \beta$, where α, β, and γ are fresh type variables. We generate fresh relation variables $?A$, $?B$, and $?C$ corresponding to these, and use them to build an initial derivation tree following the skeleton's structure and inferred types:

$$\dfrac{\vdash ((?A \Mapsto ?B) \Mapsto ?C)\ ?t\ \mathsf{All} \qquad \dfrac{\dfrac{\dfrac{\vdash (?A \Mapsto ?A \Mapsto ?B)\ ?u\ \mathsf{le} \qquad \dfrac{}{?A\ x\ n \vdash ?A\ x\ n}}{?A\ x\ n \vdash (?A \Mapsto ?B)\ (?u\ x)\ (\mathsf{le}\ n)} \qquad \dfrac{}{?A\ x\ n \vdash ?A\ x\ n}}{?A\ x\ n \vdash ?B\ (?u\ x\ x)\ (\mathsf{le}\ n\ n)}}{\vdash (?A \Mapsto ?B)\ (\lambda x.\ ?u\ x\ x)\ (\lambda n.\ \mathsf{le}\ n\ n)}}{\vdash ?C\ (?t\ (\lambda x.\ ?u\ x\ x))\ (\mathsf{All}\ (\lambda n.\ \mathsf{le}\ n\ n))}$$

For nonleaf inference steps, we used rules App and Abs. Note that the leaves with $?A\ x\ n$ are solved with rule Var, but the leaves with constants All and le (depicted in red) are as yet unsolved. Therefore this derivation tree yields a theorem with two hypotheses

$$\begin{aligned}&((?A \Rrightarrow ?B) \Rrightarrow ?C)\ ?t\ \mathsf{All} \longrightarrow\\&(?A \Rrightarrow ?A \Rrightarrow ?B)\ ?u\ \mathsf{le} \longrightarrow\\&\quad ?C\ (?t\ (\lambda x.\ ?u\ x\ x))\ (\mathsf{All}\ (\lambda n.\ \mathsf{le}\ n\ n)).\end{aligned}$$

In step two, we set up a proof state with the hypotheses as subgoals. The first goal is matched by (28)

$$((\mathsf{ZN} \Rrightarrow =) \Rrightarrow =)\ (\mathsf{Ball}\ \{0..\})\ \mathsf{All},$$

and the second goal by (27)

$$(\mathsf{ZN} \Rrightarrow \mathsf{ZN} \Rrightarrow =) \le \le.$$

Thus, we instantiate the variables as follows $?A \leftarrow \mathsf{ZN}$, $?B \leftarrow ?C \leftarrow (\longleftrightarrow)$, $?t \leftarrow \mathsf{Ball}\ \{0..\}$, and $?u \leftarrow \le$, which transforms the conclusion into the final equivalence theorem:

$$(\forall x_{\mathsf{int}} \in \{0..\}.\ x \le x) \longleftrightarrow (\forall n_{\mathsf{nat}}.\ n \le n) \tag{30}$$

In general, the search for matching transfer rules in the second step of the transfer algorithm is very simple: we go trough the list of transfer rules from the most newly introduced ones to older ones and if we do not succeed, we backtrack. After the tool finds a solution, we can also ask it explicitly to backtrack. This forces the tool to select an older rule from the list. Notice that we used the second step for two purposes: to discharge the hypotheses, and also (more importantly) to *synthesize* the final equivalence theorem or to be more precise its (concrete representation) left-hand side.

Let me stress the connection with typing: we do not see the ZN relation in the final equivalence theorem; it was used only during the intermediate matching/discharging of hypothesis (or typing if you want) and since the right side of the final theorem has type bool and the left side's type was synthesized (typed) to the same type, both sides were related by a trivial relation, namely by the equivalence relation $\longleftrightarrow$.

We generalize the rules App, Abs, and Var such that Γ can contain arbitrary transfer rules, not only assumptions of the form $A\ x\ y$ where A is a variable. We use the symbol $\mathcal{T}$ to denote a set of transfer rules. We are interested in statements of the form $\mathcal{T} \vdash s \longleftrightarrow t$, whose meaning is that the equivalence $s \longleftrightarrow t$ was derived from the transfer rules in $\mathcal{T}$ by the rules App, Abs, and Var.

Let us consider the following example: let $\mathcal{T}$ be a set of the transfer rules (23)–(28), then we can derive the formula (29)

$$\mathcal{T} \vdash (\forall x_{\mathsf{int}} \in \{0..\}.\, x < x + 1) \longleftrightarrow (\forall n_{\mathsf{nat}}.\, n < n + 1).$$

Let us define the set of all terms that can be proved to be equivalent with term t by using transfer rules from $\mathcal{T}$: $\mathsf{Trans}_{\mathcal{T}}(t) = \{s \mid \mathcal{T} \vdash s \longleftrightarrow t\}$. Now we can state the correctness property of the transfer algorithm.

Theorem 6. Let $\mathcal{T}$ be a set of transfer rules with the following property: if the right-hand sides of two rules from $\mathcal{T}$ are overlapping, these right-hand sides must be the same terms. Let t be a term. Then the set of all results[3] of transferring t with $\mathcal{T}$ is $\mathsf{Trans}_{\mathcal{T}}(t)$.

Proof. The property of $\mathcal{T}$ guarantees that the constants that are supposed to be replaced by fresh variables in the the first step of the transfer algorithm (during creation of the skeleton) are not in conflict and thus uniquely determined. The search for transfer rules in the second step iterates over all proof trees for all statements $\mathcal{T} \vdash s \longleftrightarrow t$ such that $s \in \mathsf{Trans}_{\mathcal{T}}(t)$. $\square$

4.3 parametrized transfer relations

Parametrized transfer relations are generalization of the notion of transfer rules to cater for transferring of propositions with nested types. Let me motivate the need for parametrized transfer relations by the fset example from the beginning of this chapter. Since the type of finite sets is defined as a subtype of the set type by `typedef`, we could use the representation function

$$\mathsf{Rep}_{\mathsf{fset}} : \alpha\ \mathsf{fset} \to \alpha\ \mathsf{set}$$

to define the transfer relation $\mathsf{SF} : \alpha\ \mathsf{set} \to \alpha\ \mathsf{fset} \to \mathsf{bool}$ as follows:

$$\mathsf{SF}\ S\ F = (\mathsf{Rep}_{\mathsf{fset}}\ F = S)$$

SF is an inverted graph of $\mathsf{Rep}_{\mathsf{fset}}$. Since we want to transfer (16) to (17), we need transfer rules for fimage and $\widetilde{\bigcup}$. With the current definition of SF, we can prove the following transfer rule for fimage:

$$((= \Mapsto =) \Mapsto \mathsf{SF} \Mapsto \mathsf{SF})\ \mathsf{image}\ \mathsf{fimage} \qquad (31)$$

Although this rule works in most situations, it is not general enough since it can only transfer between the following instances of fimage and image:

$$\begin{aligned} \mathsf{fimage}\ &: (\alpha \to \beta) \to \alpha\ \mathsf{fset} \to \beta\ \mathsf{fset} \\ \mathsf{image}\ &: (\alpha \to \beta) \to \alpha\ \mathsf{set} \to \beta\ \mathsf{set} \end{aligned}$$

3 This means that we ask the algorithm to backtrack as long as it is possible and collect all the generated results.

In other words, using this rule we are not allowed to change the element types of the used sets. However, this is exactly what we need in our example (transferring (16) to (17)) because of the nesting of the transferred types—we wanted to transfer between the following instances of fimage and image:

$$\begin{aligned} \mathsf{fimage} &: (\alpha \to \alpha\ \mathsf{fset}) \to \alpha\ \mathsf{fset} \to \alpha\ \mathsf{fset}\ \mathsf{fset} \\ \mathsf{image} &: (\alpha \to \alpha\ \mathsf{set}) \to \alpha\ \mathsf{set} \to \alpha\ \mathsf{set}\ \mathsf{set} \end{aligned}$$

With the rule (31), the transferring process will fail. Moreover, we cannot even state any transfer rule relating $\bigcup$ and $\widetilde{\bigcup}$ for similar reasons.

Luckily, the design of the Transfer tool generalizes easily to transfer relations with parameters. To allow to change the element type, we will define a parametrized version of SF as follows:

$$\mathsf{SF_p} : (\alpha \to \beta \to \mathsf{bool}) \to \alpha\ \mathsf{set} \to \beta\ \mathsf{fset} \to \mathsf{bool} \tag{32}$$

$$(\mathsf{SF_p}\ A)\ S\ F = (\exists Y_{\beta\ \mathsf{set}}.\ \mathsf{rel\_set}\ A\ S\ Y \wedge \mathsf{SF}\ Y\ F) \tag{33}$$

Recall that rel_set was defined in Figure 3 on page 70. Using the relation composition operator $\circ\circ$, we can alternatively define $\mathsf{SF_p}$ as

$$\mathsf{SF_p}\ A\ S\ F = (\mathsf{rel\_set}\ A \circ\circ \mathsf{SF})\ S\ F. \tag{34}$$

Notice that rel_set A relates α set to β set according to A and SF relates β set to β fset. The parametrized transfer relation $\mathsf{SF_p}$ allows us to state (and prove) the transfer rules for fimage and $\widetilde{\bigcup}$ that correspond to the most general type instances of these constants:

$$((A \Rrightarrow B) \Rrightarrow \mathsf{SF_p}\ A \Rrightarrow \mathsf{SF_p}\ B)\ \mathsf{image}\ \mathsf{fimage} \tag{35}$$

$$(\mathsf{SF_p}\ (\mathsf{SF_p}\ A) \Rrightarrow \mathsf{SF_p}\ A)\ \bigcup\ \widetilde{\bigcup} \tag{36}$$

While transferring (16) to (17), we use the following instances of the rules (35) and (36):

$$((= \Rrightarrow \mathsf{SF_p} =) \Rrightarrow \mathsf{SF_p} = \Rrightarrow \mathsf{SF_p}\ (\mathsf{SF_p} =))\ \mathsf{image}\ \mathsf{fimage} \tag{37}$$

$$(\mathsf{SF_p}\ (\mathsf{SF_p} =) \Rrightarrow \mathsf{SF_p} =)\ \bigcup\ \widetilde{\bigcup} \tag{38}$$

The above-described generalization of the fset transfer relation by adding parameters is an instance of a general principle. I will sketch it now. Let us assume that we work only with unary type constructors; the generalization to n-ary constructors is straightforward. For every transfer relation $T : \sigma[\beta] \to \tau[\beta] \to \mathsf{bool}$, we can define its parametrized version

$$\begin{aligned} &T_p : (\alpha \to \beta \to \mathsf{bool}) \to \sigma[\alpha] \to \tau[\beta] \to \mathsf{bool} \\ &T_p\ A\ x\ y = (\mathcal{R}\ A \circ\circ T)\ x\ y, \end{aligned} \tag{39}$$

where $\mathcal{R} : \sigma[\alpha] \to \sigma[\beta] \to \mathsf{bool}$ is a relator expression for σ. If σ is a single type constructor κ, $\mathcal{R} = \mathsf{rel}_\kappa$; if σ is a type expression, $\mathcal{R}$ is the corresponding relator expression. In the fset example (34), $\mathcal{R} = \mathsf{rel\_set}$. I implemented a procedure that automatically defines the parametrized transfer relations.

The transfer rules (35) and (36) follow a general principle as well. We can automatically prove each of them from their nonparametrized rules and from the parametricity rules for their representation constants. For example, from the nonparametrized rule (31) for fimage and the parametricity rule for image

$$((A \Mapsto B) \Mapsto \mathsf{rel\_set}\ A \Mapsto \mathsf{rel\_set}\ B)\ \mathsf{image}\ \mathsf{image}, \qquad (40)$$

we can automatically prove the parametrized version (35). As the first step, we transitively connect (40) and (31) by $\circ\circ$ and obtain:

$$\begin{aligned}&(((A \Mapsto B) \Mapsto \mathsf{rel\_set}\ A \Mapsto \mathsf{rel\_set}\ B\ )\ \circ\circ\\ &((\ = \Mapsto =) \Mapsto \mathsf{SF} \qquad\ \Mapsto \mathsf{SF} \qquad\ ))\ \mathsf{image}\ \mathsf{fimage}\end{aligned}$$

Now, by propagating $\circ\circ$ over relators and pushing it inside the relation by rules of the following style[4]

$$\mathsf{rel}^\kappa\ \overline{R} \circ\circ \mathsf{rel}^\kappa\ \overline{S} = \mathsf{rel}^\kappa\ (R_1 \circ\circ S_1) \dots (R_n \circ\circ S_n) \qquad (41)$$

and using the terminal rules $(R \circ\circ =) = (= \circ\circ R) = R$ and the definition of SF_p (34) $(\mathsf{rel\_set}\ A \circ\circ \mathsf{SF}) = \mathsf{SF}_\mathsf{p}\ A$, we can derive (35). I implemented a procedure that can do all of those steps automatically.

4.4 transfer rules with side conditions

We will generalize the notion of a transfer rule $R\ t\ t'$ by allowing side conditions on relations in R, e.g., $P\ A \longrightarrow R[A]\ t\ t'$ where A is a relation. This allows us to transfer constants that are not parametric or positively expressed, relations that are parametric only under certain conditions. That is, we want to have a notion of parametricity modulo a restricted class of relations, e.g. single valued and injective.

We will also inspect limitations of transfer rules with equalities in their transfer relations and how the transfer rules with side conditions can help.

4 Such rules hold for natural functors, as we will learn later. However the corresponding rule for the function type does not hold unconstrained. Therefore we generalized the whole procedure such that we can also use rules with side conditions. For example, the rule for the function type holds when $\overline{S}$ are equalities.

4.4.1 Conditional Parametricity

In the int/nat example, we used only those transfer rules that we specifically proved for constants on integer and natural numbers—they contain the transfer relation ZN. But it is a common situation that the term (that we want to transfer) contains polymorphic constants that are not connected to integer and natural numbers. However, those constants can still change their types while we transfer between the concrete and abstract types. Therefore we will also need transfer rules for such polymorphic constants in general. We typically use a parametricity rule for such constants as a transfer rule. For example: we want to prove that if the sum of a list of natural numbers is zero then all the numbers in the list must be zero as well—basically an iterated version of (12):[5]

$$\forall xs_{\mathsf{nat\ list}}.\ \mathsf{listsum}\ xs = 0 \longrightarrow \mathsf{list\_all}(\lambda x.\ x = 0)\ xs$$

If we use the transfer algorithm to prove that this statement is equivalent to the corresponding statement about integers, we would have to relate $\mathsf{list\_all}_{(\mathsf{int}\to\mathsf{bool})\to\mathsf{int\ list}\to\mathsf{bool}}$ to $\mathsf{list\_all}_{(\mathsf{nat}\to\mathsf{bool})\to\mathsf{nat\ list}\to\mathsf{bool}}$. This can be done by using its parametricity rule (where we substitute ZN for A):

$$\forall A_{\alpha\to\beta\to\mathsf{bool}}.\ ((A \Mapsto =) \Mapsto \mathsf{list\_all2}\ A \Mapsto =)\ \mathsf{list\_all}\ \mathsf{list\_all} \tag{42}$$

If we try to argue similarly for other polymorphic constants in the statement (=, ∀ or listsum), the proposed approach fails because these constants are not parametric. I will explain in the rest why they are not parametric and how to address this issue.

The need for transfer rules with side conditions starts right at this observation that some polymorphic functions in Isabelle are not parametric. Wadler already pointed out in his seminal paper [95, §3.4] that polymorphic equality = is not a parametric function. Let us look more closely why this is the case. Its type $\alpha \to \alpha \to \mathsf{bool}$ would suggest that the following parametricity rule should hold: $(A \Mapsto A \Mapsto =) = =$. If we unfold the function relator in it, we obtain an equivalent statement

$$\forall x\ x'\ y\ y'.\ A\ x\ x' \wedge A\ y\ y' \longrightarrow x = y \longleftrightarrow x' = y'.$$

This does obviously not hold for all relations A but only for A that is single valued and injective. Such A is called *bi-unique* in our terminology. Thus, we use the following transfer rule with a side condition:

$$\mathsf{bi\_unique}\ A \longrightarrow (A \Mapsto A \Mapsto =) = = \tag{43}$$

In fact, the statement holds also in the other direction, i.e., the relation A is bi-unique if and only if it preserves equality.

5 The function $\mathsf{list\_all}_{(\alpha\to\mathsf{bool})\to\alpha\ \mathsf{list}\to\mathsf{bool}}\ P\ xs$ is defined as $\forall x \in \mathsf{set}\ xs.\ P\ x$.

Wadler argues that = not being parametric is not a contradiction to his parametricity theorem; rather, it is a proof that polymorphic equality cannot be defined in the pure polymorphic lambda calculus. Polymorphic equality can be added as an axiomatic constant, but then parametricity will not typically hold for terms containing the constant.

Indeed, for example the function that removes the first occurrence of the given element from a list $\mathsf{remove1} : \alpha \to \alpha\ \mathsf{list} \to \alpha\ \mathsf{list}$ defined as

$$\begin{aligned}&\mathsf{remove1}\ x\ [\,] &&= [\,]\\ &\mathsf{remove1}\ x\ (y\ \#\ xs) &&= (\mathsf{if}\ x = y\ \mathsf{then}\ xs\ \mathsf{else}\ y\ \#\ \mathsf{remove1}\ x\ xs)\end{aligned}$$

uses = in its definition and therefore has this conditional transfer rule

$$\mathsf{bi\_unique}\ A \longrightarrow (A \Mapsto \mathsf{list\_all2}\ A \Mapsto \mathsf{list\_all2}\ A)\ \mathsf{remove1}\ \mathsf{remove1}.$$

The rule is not provable without the bi-uniqueness assumption.

As pointed out again by Wadler, this restriction on relations is akin to an eqtype annotation in ML, or an Eq class constraint in Haskell: in these languages, we restrict the polymorphic type of equality only to types for which equality is defined. In our relational setting, we restrict ourselves only to relations that respect equality, or equivalently that are bi-unique.

Stretching the analogy a bit further, while Haskell allows users to provide Eq instance declarations, the Transfer tool allows us to provide additional rules about bi-uniqueness that serve the same purpose:

- rules for abstract types: e.g., $\mathsf{bi\_unique}\ \mathsf{ZN}$ or $\mathsf{bi\_unique}\ \mathsf{SF}$. Every abstract type defined by typedef yields a bi-unique transfer relation.

- rules for polymorphic type constructors: e.g., $\mathsf{bi\_unique}\ A \longrightarrow \mathsf{bi\_unique}\ (\mathsf{rel\_set}\ A)$, $\mathsf{bi\_unique}\ A \longrightarrow \mathsf{bi\_unique}\ (\mathsf{list\_all2}\ A)$ or $\mathsf{bi\_unique}\ A \longrightarrow \mathsf{bi\_unique}\ \mathsf{SF_p}\ A$. Every type constructor that is a natural functor yields such a bi-uniqueness rule, see Section 4.7.

Using the above rules and the rule (43), the Transfer tool is able to relate equality on lists of integers with equality on lists of naturals, using the relation $\mathsf{list\_all2}\ \mathsf{ZN}$. It can similarly relate equality on sets, lists of sets, sets of lists, and so on.

It is usually enlightening to consider the case of the function type: under which condition in terms of A and B does it hold that $\mathsf{bi\_unique}(A \Mapsto B)$? Unlike bi-unique rules for lists or sets, the condition is not that A and B have to be bi-unique, but we have to come up with a new condition: we say that a relation is *bi-total* if it is both total and surjective. We obtain

$$\mathsf{bi\_total}\ A \longrightarrow \mathsf{bi\_unique}\ B \longrightarrow \mathsf{bi\_unique}(A \Mapsto B). \qquad (44)$$

We find a similarly prominent constant for bi-totality as the equality is for bi-uniqueness in (43): the parametricity rule of the universal quantifier requires the relation A to be bi-total:

$$\mathsf{bi\_total}\ A \longrightarrow ((A \Mapsto =) \Mapsto =)\ \mathsf{All}\ \mathsf{All} \tag{45}$$

Similarly as for bi-uniqueness, the statement holds also in the other direction, i.e., preservation of the universal quantification is equivalent to bi-totality. If we view the definition of the universal quantifier $\mathsf{All} = \lambda p_{\alpha\to\mathsf{bool}}.\ (p = (\lambda x.\ \mathsf{True}))$ in the light of the rule (44), the parametricity theorem for All is not so surprising—we compare functions for equality here.

Universal quantifiers appear in most propositions used with transfer; however, many transfer relations (including ZN) are not bi-total, but only *right-total*, i.e., surjective. In this case, the universal quantification relates to a bounded quantification, where the bound is the domain of the right-total relation:[6]

$$\mathsf{right\_total}\ A \longrightarrow ((A \Mapsto =) \Mapsto =)\ (\mathsf{Ball}\ \{x \mid \mathsf{Domp}\ A\ x\})\ \mathsf{All} \tag{46}$$

Because it is inconvenient to work with expressions such as Domp T in the transferred goal, we extended the transfer algorithm such that Domp expressions are replaced with equivalent but more convenient predicates. This is configured by registering a *transfer domain rule*: e.g.,

$$\mathsf{Domp}\ \mathsf{ZN} = (\lambda x.\ x \geq 0).$$

We provide transfer domain rules for lists and other types, for example:

$$\mathsf{Domp}\ (\mathsf{list\_all2}\ A) = \mathsf{list\_all}\ (\mathsf{Domp}\ A) \tag{47}$$

Thus we can replace, e.g., Domp (list_all2 ZN) by list_all $(\lambda x.\ x \geq 0)$. The use of Domp is not limited to the universal quantifier as there are other constants, such as UNIV, Collect or set intersection $\bigcap$, that require bi-totality in their parametricity rules. We provide more widely applicable transfer rules using Domp for those constants as well.

The last condition that we use and that is connected with = is being *right-unique*, i.e., single valued. Bi-totality, right-totality and right-uniqueness are just as bi-uniqueness preserved by many relators, including those for lists and sets (see Section 4.7). We mentioned that ZN is not bi-total but, for example, total quotients yield bi-total transfer relations; see the overview in Table 1 on page 112.

6 Strictly speaking, this rule does not capture parametricity because we do not relate different type instances of the same constant. Nevertheless, we still consider the rule to be a parametricity rule because the used transfer relation is a parametricity relation—in particular, the relation does not contain any specific transfer relations (e.g., ZN or SF).

Notice that every bi-total (or bi-unique) relation is also right-total (or right-unique). We prefer bi-rules to right-rules (if it possible), which we technically realize by registering the bi-rules later than the right-rules. Thus, we try the bi-rules during the second step of the transfer algorithm first and move to the right-rules by backtracking only if the give relation is neither bi-total nor bi-unique.

So far we have addressed only =; what about other constants? Just as Eq is not the only class constraint in Haskell, there exist other polymorphic constants belonging to type classes in Isabelle/HOL. These constants are defined by overloading. As we saw in Example 2.1 on page 18, where we overloaded 0_α for various types, we can think of 0_α as a constant that does case distinction on its type instance—a highly nonparametric operation. The approach with side conditions that we adopted because of nonparametricity of = easily generalizes to any overloaded constant: we will restrict our relations only to those that respect the overloaded constant. An example follows: $\mathsf{listsum}_{\alpha\ \mathsf{list} \to \alpha}$ sums all elements in the given list, for which it uses monoid operations 0_α and $+_{\alpha \to \alpha \to \mathsf{bool}}$. Therefore, the listsum parametricity rule requires preservation of these operations:

$$\begin{gathered} A\ 0\ 0 \longrightarrow (A \Mapsto A \Mapsto =)\ +\ + \longrightarrow \\ (\mathsf{list\_all2}\ A \Mapsto A)\ \mathsf{listsum}\ \mathsf{listsum} \end{gathered} \tag{48}$$

While using this rule in the transfer algorithm, we apply the algorithm recursively to the assumptions. Thus for example, if we use the rules (23) and (25), we can relate listsum on integers with listsum on naturals.

4.4.2 Handling Equality Relations

Many propositions that we want to transfer contain nonpolymorphic constants that remain unchanged by the transfer procedure, e.g., boolean operations. We would like to avoid the necessity for lots of trivial transfer rules such as the rule for the boolean conjunction: $(= \Mapsto = \Mapsto =)\ \wedge\ \wedge$. Instead we define a predicate is_equality A, which holds if and only if A is the equality relation on its type, and keep a database of theorems that witnesses which relators preserve equality. For the function type, the theorem looks like as follows:

$$(= \Mapsto =) = (=) \tag{49}$$

Besides functions, such theorems hold for every natural functor (e.g., lists, sets, pairs, . . .). From these theorems we straightforwardly derive that the given relator preserves the is_equality predicate. For example:

$$\mathsf{is\_equality}\ A \longrightarrow \mathsf{is\_equality}\ B \longrightarrow \mathsf{is\_equality}\ (A \Mapsto B) \tag{50}$$

We register a single reflexivity transfer rule

$$\text{is\_equality}\ A \longrightarrow A\ x\ x. \tag{51}$$

Now, if we are looking for the transfer rule for $\wedge$ in the second phase of the transfer algorithm (i.e., we are trying to discharge this goal $?A\ ?a\ \wedge$), and there does not exist any other transfer rule that would match, we use the reflexivity transfer rule (51). This sets $?a$ to $\wedge$ and leaves us with the goal is_equality $?A$. By using the is_equality preservation rules (such as (50)) first and discharging the unsolved goals by is_equality $=$, we synthesize the appropriate transfer relation $(= \Mapsto = \Mapsto =)$ for $\wedge$.

The predicate is_equality is useful for expressing transfer relations that do not correspond to the most general type instances of the related constant. Let me explain this with an example: in Section 4.3, we saw that the rule (31) was not strong enough to transfer certain goals. The standard technique how to obtain the stronger rule (35) from the weaker one is to prove parametricity of the concrete constant—here image. But some unnecessary work can be saved (especially when prototyping) if we use the weaker rules until it turns out this is not enough. In our example, we would use the following rule, which is equivalent to the weak rule (31):

$$((= \Mapsto =) \Mapsto \mathsf{SF_p} = \Mapsto \mathsf{SF_p} =)\ \text{image fimage} \tag{52}$$

But this form of the rule can lead to problems: suppose fimage : $(\alpha \to \beta) \to \alpha$ fset $\to \beta$ fset in the rule and we want to use the rule at a more specific type, say $\beta \leftarrow \alpha$ set; this can cause that discharging another goal might instantiate the goal for fimage to $((?A \Mapsto (\text{rel\_set}\ =)) \Mapsto ?C \Mapsto ?D)\ ?a$ fimage. Although rel_set $=$ is equivalent to $=$, we cannot use the rule (52). An equivalent conditional transfer rule

$$\begin{aligned}&\text{is\_equality}\ A \longrightarrow \text{is\_equality}\ B \longrightarrow\\&((A \Mapsto B) \Mapsto \mathsf{SF_p}\ A \Mapsto \mathsf{SF_p}\ B)\ \text{image fimage}\end{aligned} \tag{53}$$

avoids such difficulties. Transfer rules that mention equality relations are conveniently preprocessed in this manner: the user enters (52) but we internally derive and use (53).

4.5 proving implications instead of equivalences

The transfer proof method can replace a universal with an equivalent bounded quantifier: e.g., $(\forall n_{\mathsf{nat}}.\ n < n + 1)$ is transferred to $(\forall x_{\mathsf{int}} \in \{0..\}.\ x < x + 1)$. This yields a useful extra assumption in the new subgoal. With the transferred attribute, however, it may be preferable to start with a stronger theorem $(\forall x_{\mathsf{int}}.\ x < x + 1)$, without the bounded quantifier. In this case, the Transfer tool can prove an implication:

$$(\forall x_{\mathsf{int}}.\ x < x + 1) \longrightarrow (\forall n_{\mathsf{nat}}.\ n < n + 1) \tag{54}$$

The Transfer algorithm works exactly the same; we just need some new transfer rules that encode monotonicity. We provide rules for quantifiers and implication, using various combinations of $\longrightarrow$, $\longleftarrow$, and $=$; a few are shown here.

$$\mathsf{right\_total}\ A \longrightarrow ((A \Mapsto \longrightarrow) \Mapsto \longrightarrow)\ \mathsf{All}\ \mathsf{All} \tag{55}$$
$$\mathsf{right\_total}\ A \longrightarrow ((A \Mapsto =) \Mapsto \longrightarrow)\ \mathsf{All}\ \mathsf{All} \tag{56}$$
$$(\longleftarrow \Mapsto \longrightarrow \Mapsto \longrightarrow) \longrightarrow \longrightarrow \tag{57}$$

The derivation of (54) uses transfer rule (56); rule (55) comes into play when quantifiers are nested. These rules are applicable to relation ZN because it is right-total. Further variants of these rules (involving reverse implication) are used to transfer induction and case analysis rules, which have many nested implications and quantifiers.

Having many different transfer rules for the same constants would tend to introduce a large amount of backtracking search in step two of the transfer algorithm. To counter this, we preinstantiate some of the relation variables to $\longrightarrow$, $\longleftarrow$, or $=$, guided by a simple monotonicity analysis.

4.6 proving parametricity transfer rules

As we have already seen, parametricity rules are an important class of transfer rules. It is noteworthy to ask the question how difficult it is to prove them, or more specifically, to which extent we can automate this task. On this account, we implemented a proof method transfer_prover that allows us to automatically prove that a term t is related to t' by a given relation. The method works under the condition that we have transfer rules relating corresponding constants from t and t' and matching with the given relation. In the case of parametricity, t' is just a different type instance of t.

Thus, we can automatically prove parametricity for a derived operation, say $\mathsf{c}_{\alpha\ \tau}$, as follows: let us assume that $\mathsf{c}_{\alpha\ \tau}$ was defined as $\mathsf{c}_{\alpha\ \tau} = t$. We can prove the parametricity rule $R\ \mathsf{c}_{\alpha\ \tau}\ \mathsf{c}_{\beta\ \tau}$ by unfolding c using its definition and proving $R\ \ t\ \ t[\alpha \leftarrow \beta]$ by the transfer_prover provided there is an appropriate transfer rule for every constant in t.

To prove $R\ \ t\ \ t'$, the transfer_prover proceeds as follows: it generates the skeleton for t' and its derivation tree in the same way as I explained in Section 4.2. In this setting, we know the transfer relation R and therefore we can preinstantiate a lot of schematic relation variables in the skeleton to reduce the search space in the step two. In the step two, the tool follows the same procedure as for transfer and synthesize the left-hand size by matching with the existing transfer rules. There is an additional third step: we compare the synthesized left-hand size with t and if they differ, the tool backtracks. Why we do not preinstantiate the left-hand side in the step one (it is known too) and follow a seemingly more complicated way, I will explain with an example.

Assume we would like to prove parametricity of list_all, that is (42). The function list_all is defined as $\lambda P\ xs.\ \forall x \in \mathsf{set}\ xs.\ P\ x$, which is only syntax for $\lambda P\ xs.\ \mathsf{Ball}\ (\mathsf{set}\ xs)\ (\lambda x.\ P\ x)$. Thus, we want to prove the following equivalent theorem:

$$\begin{aligned}\forall A_{\alpha\to\beta\to\mathsf{bool}}.\ ((A \Mapsto =) \Mapsto \mathsf{list\_all2}\ A \Mapsto =)\\ (\lambda P_{\alpha\to\mathsf{bool}}\ xs_{\alpha\ \mathsf{list}}.\ \mathsf{Ball}\ (\mathsf{set}\ xs)\ (\lambda x.\ P\ x))\\ (\lambda P_{\beta\to\mathsf{bool}}\ xs_{\beta\ \mathsf{list}}.\ \mathsf{Ball}\ (\mathsf{set}\ xs)\ (\lambda x.\ P\ x))\end{aligned}$$

We infer the derivation tree for $\lambda P_{\beta\to\mathsf{bool}}\ xs_{\beta\ \mathsf{list}}.\ \mathsf{Ball}\ (\mathsf{set}\ xs)\ (\lambda x.\ P\ x)$ and obtain the following schematic theorem:

$$\begin{aligned}&(?E \Mapsto ?C \Mapsto ?f)\ ?t\ \mathsf{Ball} \longrightarrow\\ &(?D \Mapsto ?E)\ ?u\ \mathsf{set} \longrightarrow\\ &\quad(?C \Mapsto ?D \Mapsto ?f)\\ &\qquad(\lambda P\ x.\ ?t\ (?u\ x)\ P)\\ &\qquad(\lambda P_{\beta\to\mathsf{bool}}\ xs_{\beta\ \mathsf{list}}.\ \mathsf{Ball}\ (\mathsf{set}\ xs)\ (\lambda x.\ P\ x))\end{aligned}$$

Since the schematic relation $(?C \Mapsto ?D \Mapsto ?f)$ is always a first-order term, we can always easily preinstantiate it by matching it against the known transfer relation $((A \Mapsto =) \Mapsto \mathsf{list\_all2}\ A \Mapsto =)$. But this is not the case for the schematic left-hand side $\lambda P\ x.\ ?t\ (?u\ x)\ P$. The schematic variable $?t$ of the function type at the head yields an undecidable unification problem. Instead, as I already described, we synthesize the left-hand side and compare it with the expected result, which does not require the undecidable fragment of the higher-order unification.[7] The comparison is technically achieved by adding an extra assumption in the schematic

7 All the assumptions in the schematic theorem are in principle again first order terms. Because of the corner case, when the schematic transfer relation is just a single schematic variable (i.e., no function type and thus no $\Mapsto$), for example $?R$, we rewrite all the transfer rules and assumptions into an equivalent statements using an identity tag: e.g., $?R\ ?t\ c$ into $\mathsf{Rel}\ ?R\ ?t\ c$, where $\mathsf{Rel} = \mathsf{id}$.

theorem. I marked the assumption by red color. Thus, in the end we will try to discharge assumption of this schematic theorem

$$\begin{aligned}
&(?E \Rrightarrow (A \Rrightarrow =) \Rrightarrow =)\ ?t\ \mathsf{Ball} \longrightarrow\\
&(\mathsf{list\_all2}\ A \Rrightarrow ?E)\ ?u\ \mathsf{set} \longrightarrow\\
&(\lambda P\ x.\ ?t\ (?u\ x)\ P) = (\lambda P_{\alpha\to\mathsf{bool}}\ xs_{\alpha\ \mathsf{list}}.\ \mathsf{Ball}\ (\mathsf{set}\ xs)\ (\lambda x.\ P\ x)) \longrightarrow\\
&\quad ((A \Rrightarrow =) \Rrightarrow \mathsf{list\_all2}\ A \Rrightarrow =)\\
&\qquad (\lambda P\ x.\ ?t\ (?u\ x)\ P)\\
&\qquad (\lambda P_{\beta\to\mathsf{bool}}\ xs_{\beta\ \mathsf{list}}.\ \mathsf{Ball}\ (\mathsf{set}\ xs)\ (\lambda x.\ P\ x)).
\end{aligned}$$

The first two assumption are discharged by the parametricity rules for Ball (from a library) and set (generated by the `datatype` command)

$$\begin{aligned}
&(\mathsf{rel\_set}\ A \Rrightarrow (A \Rrightarrow =) \Rrightarrow =)\ \mathsf{Ball}\ \mathsf{Ball},\\
&(\mathsf{list\_all2}\ A \Rrightarrow \mathsf{rel\_set}\ A)\ \mathsf{set}\ \mathsf{set}
\end{aligned}$$

and the third assumption by the reflexivity transfer rule (51).

4.7 transferable type constructors

We have seen several times within this chapter that we need to keep track of some semantic information (or impose additional structure) on our constants and type constructors to make the transferring work. Although parametricity of constants might seem to be the most prominent one, we should not forget the additional structure that we impose on the type constructors. For example, we needed relators for type constructors to formulate parametricity rules; we also needed a couple of properties connected to type constructors, e.g., (41), (44), (47) or (49). We have treated this structure rather implicitly so far and therefore I will describe it more explicitly in this section.

A *transferable type constructor* is a type constructor equipped with a relator and a predicator[8] that satisfy certain properties. Formally, a transferable type constructor is a tuple $(\kappa, \mathsf{rel}^\kappa, \mathsf{pred}^\kappa)$, where

- κ is an n-ary type constructor,
- $\mathsf{rel}^\kappa : (\alpha_1 \to \beta_1 \to \mathsf{bool}) \to \ldots \to (\alpha_n \to \beta_n \to \mathsf{bool}) \to \overline{\alpha}\ \kappa \to \overline{\beta}\ \kappa \to \mathsf{bool}$,
- $\mathsf{pred}^\kappa : (\alpha_1 \to \mathsf{bool}) \to \ldots \to (\alpha_n \to \mathsf{bool}) \to \overline{\alpha}\ \kappa \to \mathsf{bool}$,

satisfying the following properties:

8 A predicator is to a predicate what a relator is to a relation.

- identity rule:
$$(\mathsf{rel}^\kappa = \ldots =) = (=)$$

- distribution of Domp over the relator:
$$\mathsf{Domp}\ (\mathsf{rel}^\kappa\ R_1 \ldots R_n) = \mathsf{pred}^\kappa\ (\mathsf{Domp}\ R_1) \ldots (\mathsf{Domp}\ R_n)$$

- compositionality and transitivity of the relator:
$$\mathsf{rel}^\kappa\ \overline{R} \circ\circ \mathsf{rel}^\kappa\ \overline{S} = \mathsf{rel}^\kappa\ (R_1 \circ\circ S_1) \ldots (R_n \circ\circ S_n)$$

- monotonicity of the relator:
$$R_1\ \mathcal{O}_1\ S_1 \longrightarrow \ldots \longrightarrow R_n\ \mathcal{O}_n\ S_n \longrightarrow \mathsf{rel}^\kappa\ \overline{R} \sqsubseteq \mathsf{rel}^\kappa\ \overline{S},$$
where $\mathcal{O}_i$ is either $\sqsubseteq$ or $\sqsupseteq$.

- preservation of the transfer rule side conditions by the relator:
$$\mathcal{C}_1\ R_1 \longrightarrow \ldots \longrightarrow \mathcal{C}_n\ R_n \longrightarrow \mathcal{C}\ (\mathsf{rel}^\kappa\ \overline{R}),$$
where $\mathcal{C}$ is any of these predicates: bi_total, bi_unique, right_total, right_unique, left_total, left_unique (thus, the scheme yields six rules) and $\mathcal{C}_1$ to $\mathcal{C}_n$ can be in general any combination of these six predicates but typically they are all the same as $\mathcal{C}$.

Let me clarify that the above notion of a transferable type constructor represents the maximal possible extent of the additional structure that can be provided by a user. The infrastructure that we implemented to store the semantic information is more flexible and does not strictly prescribe how much of it must be provided—the extent is configured by a user-supplied set of rules. This is a design decision motivated by two factors: 1) some type constructors can possess only a strict subset of possible properties; 2) we do not want to force the user to prove all the properties if some of them are not needed for their use case. For example, if the user does not provide a rule that the right-totality is preserved by a relator in question, transferring will fail for a certain class of problems but the user is not stopped from using the tool; if the compositionality does not hold, the user cannot use the prover for parametrization of transfer rules but the transferring process is not affected. Thus, any subset of provided rules represents a legitimate state of the tool but it might weaken its strength.

Now we will focus on the maximal extent of the structure: transferable type constructors. As I already mentioned, numerous common type constructors conform to this specification such as sets, lists, pairs or options. Of course, we would like to make our observation less intensional and find a natural class of type constructors that would be transferable.

Traytel, Popescu and Blanchette [91] came up with a notion of an n-ary *bounded natural functor* as a type constructor equipped with a map function, n set functions, a relator and a cardinal bound that satisfy certain conditions. Their motivation was to create modular definitional datatype and codatatype commands that would allow nested (co)recursion not only through other datatypes and the positive position in the function type (as in the previous implementation), but also through a broader class of well-behaved type constructors (e.g., finite sets). In their approach, the class of such type constructors is the class of bounded natural functors.

I proved that any bounded natural functor defines a transferable type constructor. This is particularly exciting since the class of bounded natural functors contains not only various nonfree types (e.g., finite sets or multisets) but most importantly all datatypes and codatatypes that are definable in Isabelle/HOL [12]. Moreover, we do not need the bound in the proof of the statement and therefore I will present a more general result here—every natural functor defines a transferable type constructor[9]. Let me start with a formal definition of a natural functor.

An n-ary natural functor is a tuple $(\kappa, \mathsf{map}^\kappa, (\mathsf{set}_i^\kappa)_{i\le n}, \mathsf{rel}^\kappa)$, where

- κ is an n-ary type constructor,
- $\mathsf{map}^\kappa : (\alpha_1 \to \beta_1) \to \ldots \to (\alpha_n \to \beta_n) \to \overline{\alpha}\ \kappa \to \overline{\beta}\ \kappa$,
- $\mathsf{set}_i^\kappa : \overline{\alpha}\ \kappa \to \alpha_i\ \mathsf{set}$,

satisfying the following properties:

- $(\kappa, \mathsf{map}^\kappa)$ is an n-ary functor:

$$\mathsf{map}^\kappa\ \mathsf{id}\ \ldots\ \mathsf{id} = \mathsf{id}$$
$$\mathsf{map}^\kappa\ (f_1 \circ g_1)\ \ldots\ (f_n \circ g_n) = \mathsf{map}^\kappa\ \overline{f} \circ \mathsf{map}^\kappa\ \overline{g}$$

- for each i the set function set_i^κ is a natural transformation into the powerset functor $(\mathsf{set}, \mathsf{image})$:

$$\mathsf{set}_i^\kappa \circ \mathsf{map}\ \overline{f} = \mathsf{image}\ f_i \circ \mathsf{set}_i^\kappa$$

- a congruence rule for the map function:

$$\frac{\forall x \in \mathsf{set}_1^\kappa\ xs.\ f_1\ x = g_1\ x \quad \ldots \quad \forall x \in \mathsf{set}_n^\kappa\ xs.\ f_n\ x = g_n\ x}{\mathsf{map}^\kappa\ \overline{f}\ xs = \mathsf{map}^\kappa\ \overline{g}\ xs}$$

- transitivity of the relator:

$$\mathsf{rel}^\kappa\ \overline{R} \circ\circ \mathsf{rel}^\kappa\ \overline{S} \sqsubseteq \mathsf{rel}^\kappa\ (R_1 \circ\circ S_1) \ldots (R_n \circ\circ S_n)$$

9 The type of sets is a natural functor, but it is obviously not bounded.

- the relator is an extension of the map function:

  $$\begin{aligned}&\mathsf{rel}^\kappa\ \overline{R}\ xs\ ys = \\ &(\exists zs.\ \mathsf{set}^\kappa_1\ zs \subseteq \{(x,y) \mid R_1\ x\ y\} \wedge \cdots \wedge \mathsf{set}^\kappa_n\ zs \subseteq \{(x,y) \mid R_n\ x\ y\} \\ &\ \wedge \mathsf{map}\ \overline{\mathsf{fst}}\ zs = xs \wedge \mathsf{map}\ \overline{\mathsf{snd}}\ zs = ys)\end{aligned}$$

Theorem 7. Let $(\kappa, \mathsf{map}^\kappa, (\mathsf{set}^\kappa_i)_{i\leq n}, \mathsf{rel}^\kappa))$ is an n-ary natural functor, then $(\kappa, \mathsf{rel}^\kappa, \mathsf{pred}^\kappa)$ is a transferable type constructor, where

$$\mathsf{pred}^\kappa\ \overline{P}\ xs = \forall x_1 \in \mathsf{set}^\kappa_1\ xs.\ P_1\ x_1 \wedge \cdots \wedge \forall x_n \in \mathsf{set}^\kappa_n\ xs.\ P_n\ x_n.$$

Proof. First of all, the basic properties of rel^κ as the identity rule, compositionality, monotonicity and the converse rule $(\mathsf{rel}^\kappa\ \overline{R})^{-1} = \mathsf{rel}^\kappa\ R_1^{-1} \ldots R_n^{-1}$ follow easily from the natural functor axioms [11, 12]. To prove preservation of the side conditions, we use their alternatives definitions:

$$\begin{aligned}\mathsf{right\_total}\ R &\longleftrightarrow R^{-1} \circ\circ R \sqsupseteq = \\ \mathsf{right\_unique}\ R &\longleftrightarrow R^{-1} \circ\circ R \sqsubseteq = \\ \mathsf{left\_total}\ R &\longleftrightarrow R \circ\circ R^{-1} \sqsupseteq = \\ \mathsf{left\_unique}\ R &\longleftrightarrow R \circ\circ R^{-1} \sqsubseteq =\end{aligned}$$

Thus, if we want to prove, for example, preservation of right-totality of a unary natural functor, we have to prove:

$$R^{-1} \circ\circ R \sqsupseteq = \longrightarrow (\mathsf{rel}\ R)^{-1} \circ\circ \mathsf{rel}\ R \sqsupseteq =$$

We use the identity rule to expand the last = to rel = and then the converse rule, compositionality and monotonicity. The same approach works for the other constraints. Preservation of bi-totality follows from the preservation of left-totality and right-totality. Analogously for bi-uniqueness.

The distribution of Domp over the relator is proved as follows: we unfold Domp using its definition and the relator using its characterization in terms of the map and set functions; after simplification, we are required to prove that the following equivalence holds for all $xs_{\overline{\alpha}\ \kappa}$:

$$\begin{aligned}&(\exists z.\ \mathsf{set}^\kappa\ z \subseteq \{(x,y).\ R\ x\ y\} \wedge \mathsf{map}^\kappa\ \mathsf{fst}\ z = xs) \longleftrightarrow \\ &(\forall x \in \mathsf{set}^\kappa\ xs.\ \exists y.\ R\ x\ y)\end{aligned}$$

This is proved by using the basic properties of the map and set functions. □

From the implementation point of view, each time when the user explicitly proves that a type constructor is a bounded natural functor (as it is done for example for types of finite sets, countable sets, multisets, association lists, or various probability mass functions), or when the user defines a datatype or a codatatype, the plugin that I implemented automatically proves that the type constructor is transferable as well.[10]

Except that we proved that every natural functor is transferable, the other main result of this section is that we can confirm the experience of the authors of the new (co)datatype commands [91]: it is helpful to *see types as richly structured objects rather than mere collections of elements.*

4.8 interfaces

The Transfer tool provides multiple user interfaces. The transfer proof method is the main one and replaces the current subgoal by a logically equivalent subgoal; typically, it replaces a goal about an abstract type by a goal about the representation type. For instance, using one of the introductory examples about finite sets

$$\begin{array}{l}\texttt{lemma}\ \forall S_{\alpha\,\mathsf{fset}}.\ \widetilde{\bigcup}\,\mathsf{fimage}\ (\lambda x_\alpha.\ \{x\})\ S = S\\ \texttt{apply}\ \mathsf{transfer},\end{array}$$

the invocation of transfer turns the lemma statement to the following equivalent proof obligation

$$\forall S_{\alpha\,\mathsf{set}}.\ \mathsf{finite}\ S \longrightarrow \bigcup \mathsf{image}\ (\lambda x_\alpha.\ \{x\})\ S = S.$$

The method internally proves an equivalence between the two subgoals (as we saw in Section 4.2) and then use it to replace the initial subgoal by the new one.

The tool also provides the transferred theorem attribute, which yields a theorem about an abstract type when given a theorem involving a representation type. For example, let us assume we stored the following theorem

$$\forall x_{\mathsf{int}}.\ x < x + 1$$

in our system under the name less−add−one. Then the following theorem less−add−one[transferred] reads

$$\forall n_{\mathsf{nat}}.\ n < n + 1.$$

As we discussed in Section 4.5, the source theorem (on the representation type) can be in general stronger then the produced theorem about the abstract type, which is also the case in this example.

10 There is not yet any infrastructure for natural functors without a bound, therefore the Transfer tool must be set up manually for the set type. Generalizing the current infrastructure in this manner is merely an engineering task.

Finally, the tool also provides the inverse to the attribute transferred, namely the attribute untransferred, which yields a theorem about a representation type given a theorem about an abstract type. Using again integer and natural numbers, let us call the following theorem nat−add−zero:

$$\forall x_{\mathsf{nat}}.\ \forall y_{\mathsf{nat}}.\ x + y = 0 \longrightarrow x = 0 \wedge y = 0$$

Then the following theorem nat−add−zero[untransferred] reads

$$\forall x_{\mathsf{int}} \in \{0..\}.\ \ \forall y_{\mathsf{int}} \in \{0..\}.\ \ x + y = 0 \longrightarrow x = 0 \wedge y = 0.$$

Also the attribute untransferred internally proves the equivalence as we saw in the case of the method transfer.

Last but not least, there are methods for debugging the transferring process to easily discover, for example, which transfer rules are missing.

4.9 limitations and future work

I will describe limitations of the Transfer tool in this section.

order of instantiations During the second phase of the transfer algorithm, when we look for transfer rules and synthesize the transferred term, the algorithm might produce a result that most users find unexpected and are not satisfied with. To demonstrate the problem, we start with a simple example:

We consider again the type of finite sets α fset and would like to transfer the following goal:[11]

$$\mathsf{finite}\ \mathsf{UNIV}_{\alpha\ \mathsf{fset}\ \mathsf{set}}$$

The expected result after applying transfer is

$$\mathsf{finite}\ \{S_{\alpha\ \mathsf{set}} \mid \mathsf{finite}\ S\}$$

but the actual result that we obtain is an unchanged goal—transfer seems to have done nothing. Let us inspect the skeleton theorem to identify what happened in this example:

$$\begin{array}{l} ?R\ ?t\ \mathsf{UNIV}_{\alpha\ \mathsf{fset}\ \mathsf{set}} \longrightarrow \\ (?R \Mapsto =)\ ?u\ \mathsf{finite} \longrightarrow \\ \quad ?u\ ?t \end{array}$$

11 If we assume that α can be instantiated only to finite types (via a type class constraint on α), the goal is provable.

This means we have to find two transfer rules; one for UNIV and one for finite. As I already mentioned in Section 4.4 on page 77, there are two transfer rules for UNIV—the first one

$$\forall A_{\alpha \to \beta \to \mathsf{bool}}.\ \mathsf{bi\_total}\ A \longrightarrow (\mathsf{rel\_set}\ A)\ \mathsf{UNIV}\ \mathsf{UNIV} \tag{58}$$

works for bi-total relations and relates $\mathsf{UNIV}_{\alpha\ \mathsf{set}}$ to $\mathsf{UNIV}_{\beta\ \mathsf{set}}$. The other

$$\mathsf{right\_total}\ A \longrightarrow (\mathsf{rel\_set}\ A)\ \{x \mid \mathsf{Domp}\ A\ x\}\ \mathsf{UNIV} \tag{59}$$

works for right-total relations and if A equals $\mathsf{SF_p} =$ (which is right-total), the rule relates $\{S_{\alpha\ \mathsf{set}} \mid \mathsf{finite}\ S\}$ to $\mathsf{UNIV}_{\alpha\ \mathsf{fset}\ \mathsf{set}}$ because it holds that $\mathsf{Domp}\ (\mathsf{SF_p} =)$ is equal to finite.

Of course we would like to use the transfer rule (59) but as we learned in Section 4.4, we will try the bi-unique rule (58) first. This requires us to discharge the assumption bi_total $?A$ and since $?A$ is still an unknown relation we can discharge the obligation by the transfer rule bi_total =. Thus $?R$ is set to the trivial relation rel_set =, which causes that $\mathsf{UNIV}_{\alpha\ \mathsf{fset}\ \mathsf{set}}$ is transferred to itself. This propagates through the whole step two and causes that transfer transfers the goal to itself and effectively does not change it at all. If we had already known that $?R$ should be $\mathsf{SF_p}\ ?B$, we could have not discharged the bi-uniqueness assumption, backtracked and chosen the rule (59).

Let us move from the concrete example to a general description of the problem: in rare cases it might happen, that 1) we are transferring a polymorphic constant (e.g., UNIV) whose transfer rules are of the form of conditional parametricity rules and 2) the partially synthesized transfer relation is yet underspecified such that we cannot decide the validity of the side conditions without instantiating the transfer relation to a more specific relation (typically by instantiating the trivial transfer relation =). This leads to results where less is transferred than the user expects.

Please notice that this problem does not reduce the theoretical strength of the tool because the user can always get the desired result by asking the tool to backtrack. The problem is a practical one: explicitly to ask a tool to backtrack does not produce robust and readable proofs.

We can formally see the output of transfer as a list of the synthesized left-hand sides where the head of the list is the result that is produced first if we apply transfer to a goal and the next elements in the list represent the results of backtracking. The limitation that we are looking at here, does not influence the set of elements in this list but the order in which the elements of the list are produced. If the order produced by the algorithm from Section 4.2 does not meet the expectation of the users in rare cases, we should try to formalize the expectation of the users.

In the motivational example, we were disappointed that transfer synthesized for $?R$ a transfer relation that is equivalent to =. Intuitively, if we can choose between transferring less or more, we go for transferring more; i.e., we are aiming at a certain concept of eagerness.

First some definitions, we say that a constant c_τ is an *atomic relation* if $\tau \leq \alpha \to \beta \to \mathsf{bool}$. We say that a constant c_τ is a *parametrized relation* if $\tau = (\alpha_1 \to \beta_1 \to \mathsf{bool}) \to \ldots \to (\alpha_n \to \beta_n \to \mathsf{bool}) \to \sigma_1[\overline{\alpha}] \to \sigma_2[\overline{\beta}] \to \mathsf{bool}$ for some σ_1, σ_2 types and where $n > 0$. Now we define a partial order $\sqsubseteq_{\mathcal{T}}$ on relations with respect to transfer rules $\mathcal{T}$ inductively as follows (T denotes an n-ary parametrized relation from $\mathcal{T}$):

$$
\begin{array}{ll}
(=) \sqsubseteq_{\mathcal{T}} R & \text{if } R \text{ is an atomic relation from } \mathcal{T} \\
(=) \sqsubseteq_{\mathcal{T}} T = \ldots = & \\
T\,\overline{R} \sqsubseteq_{\mathcal{T}} T\,\overline{S} & \text{if } R_1 \sqsubseteq_{\mathcal{T}} S_1, \ldots, R_n \sqsubseteq_{\mathcal{T}} S_n \\
R \sqsubseteq_{\mathcal{T}} T & \text{if } T = R \circ\circ S \text{ for some } S
\end{array}
$$

We also close $\sqsubseteq_{\mathcal{T}}$ under substitution. Thus, we defined = as smaller than all atomic transfer relations (e.g., ZN) and lifted this order over all relators and parametrized transfer relations in $\mathcal{T}$.

We use $\sqsubseteq_{\mathcal{T}}$ to capture our preference of the rule (59) over the rule (58) in our example since the former instantiates $?R$ to $\mathsf{SF_p}$ = and the latter to = and we have $(=) \sqsubseteq_{\mathcal{T}} \mathsf{SF_p}$ = for an appropriate $\mathcal{T}$. In a general setting, we could use $\sqsubseteq_{\mathcal{T}}$ to define a partial order on the list of left-hand sides produced by transfer and accept only the lists that observe this order. How to modify the current transfer algorithm to always observe the order while keeping the algorithm efficient is future work.

The current workarounds contain either usage of explicit backtracking, as I mentioned above, or locally changing $\mathcal{T}$ to prioritize some rules. In our example, we could locally redeclare the rule (59) as a transfer rule and thus force transfer to use it before the rule (58).

format of transfer rules The alpha and omega of the transfer tool is transfer rules. We already observed that some constants (e.g., equality or universal quantification) are not in general parametric in HOL and therefore we came up with side conditions on a transfer relation: e.g., bi-uniqueness or bi-totality. In general, the side condition under which the constant is parametric can become arbitrarily complex. The typical example is parametricity of the head function for lists $\mathsf{head}_{\alpha\,\mathsf{list} \to \alpha}$. Its type suggests the following parametricity rule

$$(\mathsf{list\_all2}\ R \Mapsto R)\ \mathsf{head}\ \mathsf{head}$$

but this theorem does not hold since head is underspecified for empty lists and in general we cannot prove $R\ (\mathsf{head}\ [])\ (\mathsf{head}\ [])$.

We would have to assume that the lists that we work with are nonempty but that would mean to break the point-free/higher-order style of our transfer rules and work with rules such as the following:

$$x \neq [\,] \longrightarrow \mathsf{list\_all2}\ R\ x\ y \longrightarrow R\ (\mathsf{head}\ x)\ (\mathsf{head}\ y)$$

Lammich's automatic procedure for data refinement [56] allows such first-order rules. In order to discharge arbitrary conditions (such as $x \neq [\,]$), the tool tries to collect facts established either locally (e.g., by a condition of an if statement) or globally (e.g., as an assumption of the whole theorem). Various automatic procedures (e.g., auto or simp) use those facts and try to discharge these arbitrary conditions within a timeout. This approach seems to work well in his setting.

Another option how to address this problem is to encode the condition in the relator: in our example we would use a variant of list_all2 that relates only nonempty lists.

It is our future work to explore possibilities how to support broader class of conditional parametricity theorems while still keeping predictive behavior of transferring.

4.10 related work

As I already mentioned, *relational parametricity* by Reynolds [83], *free theorems* by Wadler [95], and *representation independence* by Mitchell [65] were primary theoretical sources of inspiration.

First tools for transferring related theorems in theorem provers arose from the need to provide an automation for defining quotients—a ubiquitous mathematical concept. The typical task reads: how to define rational numbers as a partial quotient and prove their properties in terms of the representation type.

Much previous work has been done on formalizing quotients in theorem provers. Slotosch [85] and Paulson [78] each developed techniques for defining quotient types provided limited automation for transferring first-order properties from representation to abstract types in the form of lemmas that facilitate manual proofs. Harrison [33] implemented tools for transferring theorems automatically, although this work was still limited to first-order constants and theorems. In 2005, Homeier [39] published a design for a new HOL tool, which was the first system capable of transferring higher-order theorems.

In 2011, Kaliszyk and Urban [48] implemented a new quotient tool for Isabelle/HOL, based upon Homeier's design with some improvements. The main difference between our solution and their solution is that they do not use the relational view on transferring but more rewriting oriented approach. This leads to the following deficiency: consider α fset, a type of finite sets, this time defined as a quotient of α list. Provided that we already defined fset versions of the list functions $\mathsf{map}_{(\alpha\to\beta)\to\alpha\ \mathsf{list}\to\beta\ \mathsf{list}}$ and $\mathsf{concat}_{\alpha\ \mathsf{list}\ \mathsf{list}\to\alpha\ \mathsf{list}}$, the tool has difficulty transferring the following theorems to fset:

$$\begin{aligned}
&\mathsf{concat}\ (\mathsf{map}\ (\lambda x.\ [x])\ xs) = xs\\
&\mathsf{map}\ f\ (\mathsf{concat}\ xss) = \mathsf{concat}\ (\mathsf{map}\ (\mathsf{map}\ f)\ xss)\\
&\mathsf{concat}\ (\mathsf{map}\ \mathsf{concat}\ xsss) = \mathsf{concat}\ (\mathsf{concat}\ xsss)
\end{aligned}$$

To allow transferring over a quotient, the user has to supply a *respectfulness theorem* for every constant, a fact that the given constant respects the equivalence relation used in the definition of the quotient. Note that map occurs at several different type instances here: It is used with functions of types $\alpha \to \beta$, $\alpha \to \alpha$ list, and α list $\to \beta$ list. Unfortunately a single respectfulness theorem for map will not work in all these cases—each type instance requires a different respectfulness theorem. On top of that, the user must also prove additional *preservation lemmas*, essentially a restricted version of parametricity. These rules can be tricky to state correctly and tedious to prove. In our relational approach, we would provide only one (parametrized) transfer rule per constant (map and concat). Finally, our transfer procedure is more general and more widely applicable without so many hard-wired assumptions about quotients.

Stepping outside of the world of quotients, a lot of work has been done in different contexts as generalized rewriting or data refinement, which share a lot of similar ideas to our solution. In Coq, implementations of generalized rewriting by Coen [19] and Sozeau [86] are similar to our transfer method—in particular, Sozeau's "signatures" for higher-order functions are like our transfer rules. Sozeau's work has better support for subrelations, but our Transfer tool is more general in allowing relations over two different types.

Magaud [61] transfers Coq theorems between different types, but unlike our work, his approach is based on transforming proof terms.

Lammich's automatic procedure for data refinement [56] was inspired by our work, especially by the idea to represent types as relations. I already mentioned in the previous section that his solution supports arbitrary assumptions for transfer rules and his transfer algorithm works with first-order rules. In Coq, Cohen et al. [20] implemented a similar data refinement framework with the same objectives as Lammich's work.

Zimmermann and Herbelin [100] recently reported on their first steps to turn Cohen et al.'s work on data refinement into a general-purpose transferring plugin for Coq with similar goals as our tool. They write about our work: *Nothing going as far as their Transfer package has yet been created for Coq.*

All along it's been, "Something's being lifted from somewhere, to some thing, but what's being lifted is not clear to me, and neither are the restrictions on what the some thing can be."

— Gottfried Barrow (2013)

5 ABSTRACT TYPES UNIFORMLY: LIFTING

The main objective of the Lifting tool is to allow us to easily define operations on newly created abstract types, which is a necessary step in building a library for an abstract type. As I already elaborated on this in the introduction, there exists no formal notion of subtyping in the type system of Isabelle/HOL. Therefore when I define a new abstract type, there are no operations on this type unless I explicitly define them. Moreover, a definition is not enough. We would like to obtain some properties of the newly defined constant, for which exactly we created the Transfer tool; i.e., we want to acquire a transfer rule for the constant.

To automate this process, Lifting provides the `lift_definition` command and works with four kinds of type abstraction: type copies, subtypes, total quotients and partial quotients (see Table 1 on page 112 for an overview of these). Although we could see type copies as a special case of subtypes (and subtypes and total quotients as a special case of partial quotients) and although this is exactly what the internal construction does, I explicitly name type copies here because in some cases we can provide more specific (and thus better) procedures for them than for subtypes.

The Lifting tool is my work and builds on previous work on quotient tools by Peter Homeier, Cezary Kaliszyk and Christian Urban and is meant to conclude the effort to build a quotient tool supporting the whole type universe of higher-order logic. This chapter is partly based on my and Brian Huffman's joint paper [40].

In the next section, I will go through some motivational examples to get the first glimpse of the tool and in the light of the examples I will outline the rest of this chapter.

5.1 motivational examples

finite sets Let us recall the `typedef` command, which was introduced in Section 2.4.2: it allows us to define a new type that is isomorphic to a nonempty subset of an already existing type. Let us use it to define the type of finite sets α fset:

$$\texttt{typedef}\ \ \alpha\ \mathsf{fset} = \{S_{\alpha\ \mathsf{set}} \mid \mathsf{finite}\ S\} \quad \langle \mathit{proof} \rangle \tag{60}$$

Having defined the new abstract type fset, we would like to define some operations on it; let us say $\widetilde{\cup}_{\alpha\ \mathsf{fset} \to \alpha\ \mathsf{fset} \to \alpha\ \mathsf{fset}}$, the union operation. Let us assume that the abstraction and representation functions that we get from `typedef` are called $\mathsf{abs\_fset}_{\alpha\ \mathsf{set} \to \alpha\ \mathsf{fset}}$ and $\mathsf{rep\_fset}_{\alpha\ \mathsf{fset} \to \alpha\ \mathsf{set}}$. If we want to define $\widetilde{\cup}$ in terms of operations on the representation type (and this is what we do most of the time), there is no way to get around it than using the abstraction and representation functions as explicit coercion functions to convert between the old and the new type and vice versa:

$$X_{\alpha\ \mathsf{fset}}\ \widetilde{\cup}\ Y_{\alpha\ \mathsf{fset}} = \mathsf{abs\_fset}\ ((\mathsf{rep\_fset}\ X) \cup (\mathsf{rep\_fset}\ Y)) \tag{61}$$

Thus we defined $\widetilde{\cup}$ in terms of the set union $\cup_{\alpha\ \mathsf{set} \to \alpha\ \mathsf{set} \to \alpha\ \mathsf{set}}$. Working directly with definitions that use coercion functions (such as the definition (61)), requires to prove numerous lemmas how the coercions interact with the new function and it takes much effort to prove properties of the function. Moreover, for functions with more involved abstract types (especially using nested types), even to work out the right definition might be elaborate since it requires a complex combination of the coercions.

Let me show you how we can completely shield the user from the trauma of working with the coercions by using the Lifting tool:

$$\texttt{lift\_definition}\ \widetilde{\cup} : \alpha\ \mathsf{fset} \to \alpha\ \mathsf{fset} \to \alpha\ \mathsf{fset}\ \texttt{is}\ \cup \quad \langle \mathit{proof} \rangle$$

We specified the name of the new abstract function, its (abstract) type and which term on the representation level the new function corresponds to. We had to also provide a proof of a certain statement, which I introduce later. First, let us observe effects of the invocation of `lift_definition`. The definition that `lift_definition` used to define $\widetilde{\cup}$ is equivalent to (61) but this time it was generated by the tool. More importantly, the tool also automatically proved the following transfer rule

$$(\mathsf{SF} \Mapsto \mathsf{SF} \Mapsto \mathsf{SF})\ \cup\ \widetilde{\cup} \tag{62}$$

and if the user can provide a parametricity theorem for $\cup$, this stronger parametrized transfer rule is proved

$$(\mathsf{SF_p}\ A \Mapsto \mathsf{SF_p}\ A \Mapsto \mathsf{SF_p}\ A)\ \cup\ \widetilde{\cup}. \tag{63}$$

See Section 4.3 on page 75 for definitions of SF and $\mathsf{SF_p}$ and more about parametrized transfer rules. In fact, the Lifting tool also automatically defines SF and $\mathsf{SF_p}$ from the definition of fset (60).

The transfer rule allows us to prove properties of $\widetilde{\cup}$, e.g., commutativity:

$$\texttt{lemma}\ \forall A_{\alpha\ \mathsf{fset}}\ B_{\alpha\ \mathsf{fset}}.\ A\ \widetilde{\cup}\ B = B\ \widetilde{\cup}\ A$$
$$\texttt{apply}\ \mathsf{transfer}$$

The call of transfer turns the goal into an equivalent goal about sets

$$\forall A_{\alpha\ \mathsf{set}}\ B_{\alpha\ \mathsf{set}}.\ A \cup B = B \cup A,$$

which is a known fact from the set library and therefore easily provable. This brings us to the first important feature of the Lifting tool: *Lifting is the main provider of transfer rules for Transfer*.

Let us finally inspect the obligation that we had to prove when we used `lift_definition` to define $\widetilde{\cup}$:

$$\forall X_{\alpha\ \mathsf{set}}\ Y_{\alpha\ \mathsf{set}}.\ \mathsf{finite}\ X \longrightarrow \mathsf{finite}\ Y \longrightarrow \mathsf{finite}\ (X \cup Y) \qquad (64)$$

We call this formula a *respectfulness theorem*, which states that the representation term of the newly defined function must respect the type abstraction. In our example, we have to make us sure that if we take two elements from the subset representing fset and union them, the result still stays in the subset; in other words, forming the union of two finite sets produces a finite set. We can see the respectfulness theorem as a correctness condition under which the definition makes sense.[1]

integer numbers Let us recall the definition (1) of integer numbers:

$$\texttt{quotient\_type}\ \ \mathsf{int} = \mathsf{nat} \times \mathsf{nat}\ /\ \mathsf{intrel} \quad \langle \mathit{proof} \rangle$$

where $\mathsf{intrel}\ (x, y)\ (u, v) \longleftrightarrow x + v = u + y$. This means that int was defined as a quotient of pairs of natural numbers $\mathsf{nat} \times \mathsf{nat}$ such that if $x + v = u + y$, the pairs (x, y) and (u, v) are in the same equivalence class; e.g., $(3, 5)$ and $(0, 2)$ represent the same integer number, namely -2.

Now we want to define addition on integer numbers, $\mathsf{plus\_i}_{\mathsf{int}\to\mathsf{int}\to\mathsf{int}}$. On the concrete level, we define the addition as $\mathsf{plus\_nn}\ (x, y)\ (u, v) = (x{+}u,\ y{+}v)$, where plus_nn has the type $\mathsf{nat}{\times}\mathsf{nat} \to \mathsf{nat}{\times}\mathsf{nat} \to \mathsf{nat}{\times}\mathsf{nat}$. The respective lifted definition is straightforward:

$$\texttt{lift\_definition}\ \ \mathsf{plus\_i} : \mathsf{int} \to \mathsf{int} \to \mathsf{int}\ \mathsf{is}\ \mathsf{plus\_nn} \quad \langle \mathit{proof} \rangle \qquad (65)$$

The respectfulness theorem that we have to prove here is as follows:

$$\begin{aligned}\forall x\ x'\ y\ y'.\ \mathsf{intrel}\ x\ x' \longrightarrow \mathsf{intrel}\ y\ y' \longrightarrow \\ \mathsf{intrel}\ (\mathsf{plus\_nn}\ x\ y)\ (\mathsf{plus\_nn}\ x'\ y')\end{aligned} \qquad (66)$$

1 The role of the respectfulness theorem as a correctness condition is subtle. Looking at the logical definition of $\widetilde{\cup}$ (61), the respectfulness theorem guarantees that the value to which abs_fset is applied lies in the fragment for which abs_fset is well defined. Surprisingly, we could make a plain definition of $\widetilde{\cup}$ even if the representation term did not respect the abstraction; e.g., $X\ \widetilde{\cup}\ Y = \mathsf{abs\_fset}\ ((\mathsf{rep\_fset}\ X) \cup (\mathsf{rep\_fset}\ Y) \cup \mathsf{ind})$. Such a definition is a valid definition in Isabelle/HOL because HOL is a logic of total functions and therefore abs_fset applied to an infinite set still represents some value in the logic. But we do not know which value since the axiomatization of `typedef` abstraction functions (see Section 2.4.2 on page 19) leaves this unspecified. Therefore we cannot prove any interesting properties of such functions (e.g., a transfer rule) and want to prevent the user from making such useless definitions by the command `lift_definition`.

The high-level description of the condition is the same as in the previous example: we want to guarantee that the concrete representation respects the type abstraction. Applied to this specific example, we want plus_nn to respect the equivalence relation intrel such that the choice of different elements from the same equivalence class does not produce results from different equivalence classes.[2] Once this is proved, the Lifting tool defines plus_i as a certain combination of coercions produced by `quotient_type` and proves the respective transfer rule, which we can again use to prove properties of plus_i (for example again commutativity).

It is not a coincidence that I presented these two examples of type abstraction here: one for a subtype (α fset) and one for a total quotient (int). Although the different forms of the respectfulness theorems in these two examples and seemingly different reasons why we need them could give the impression that we need two different implementations of the Lifting tool, this is not the case. Except for some thin presentation and setup layers, internally we do not distinguish between subtypes and quotients in the Lifting tool. This is the second important feature of the Lifting tool: *Lifting treats type abstractions uniformly.*

In Section 5.2, I will explain how the Lifting tool treats type abstraction uniformly. This section presents the main theoretical results of this chapter.

The following two sections (Sections 5.3 and 5.4) describe the two above-mentioned thin layers that depend on the used type abstraction and surround the independent core. I will explain how to set up the Lifting tool and how we achieve that the respectfulness theorem for $\widetilde{\cup}$ (64) can look differently from the respectfulness theorem for plus_i (66).

Section 5.5 provides a bird's eye view on how the Transfer and Lifting tools build a platform for creating abstract types and how their layered design provides flexibility.

I will mention some interesting aspects of the implementation of the lifting algorithm in Section 5.6.

Section 5.7 prepares the ground for the next chapter by defining coercion equations that we use as code equations in Chapter 6.

In the last section (Section 5.8), we will compare the Lifting tool with previous quotient tools and conclude the chapter.

2 As in the case of subtypes, we could define plus_i in terms of a function that does not respect the equivalence classes, but then again we could not prove anything interesting about plus_i. The reason is that the representation function for a quotient maps an abstract value to the concrete value that is selected by the Hilbert choice operator from the corresponding equivalence class. This means that the concrete value is left unspecified within the equivalence class and if the definition of plus_i depended on the choice of such a value, the result would be left completely unspecified too.

5.2 lifting algorithm

We abstract from the presented examples now and give a description that covers the general case of what the Lifting tool does. The input of the lifting is a term $t : \tau_1$ on the representation level, an abstract type τ_2 and a name f of the new constant. In the first example, $t = \cup$, $\tau_1 = \alpha\ \mathsf{set} \to \alpha\ \mathsf{set} \to \alpha\ \mathsf{set}$, $\tau_2 = \alpha\ \mathsf{fset} \to \alpha\ \mathsf{fset} \to \alpha\ \mathsf{fset}$ and $f = \widetilde{\cup}$. We work generally with types τ that are composed from type constructors κ and other types $\overline{\vartheta}$. Then we write $\tau = \overline{\vartheta}\ \kappa$. Each type parameter of κ can be either covariant (we write +) or contravariant (−). For example, in the function type $\alpha \to \beta$, α is contravariant whereas β is covariant.

In this section, we will define three functions Morph^p, Relat and Trans. Morph^p is a combination of abstraction and representation functions and gives us the definition of f. The polarity superscript p (+ or −) encodes if an abstraction or a representation function should be generated. Relat is a combination of equivalence relations and allows us to describe that t behaves correctly—respects the equivalence classes. Finally, Trans is a composed transfer relation and describes how t and f are related. More formally, if the user proves the respectfulness theorem $\mathsf{Relat}(\tau_1, \tau_2)\ t\ t$, the Lifting tool will define the new constant f as $f = \mathsf{Morph}^+(\tau_1, \tau_2)\ t$ and proves the transfer rule $\mathsf{Trans}(\tau_1, \tau_2)\ t\ f$.

For now we will not distinguish between type copies, subtypes, total quotients and partial quotients. Instead we unify all these four kinds of type abstraction with a general notion of an abstract type.

Definition 5.2.1. We say that κ_2 is an *abstract type* based on κ_1 if there is a *transfer relation* $T^{\kappa_1,\kappa_2} : \overline{\vartheta}\ \kappa_1 \to \overline{\alpha}\ \kappa_2 \to \mathsf{bool}$ associated with κ_1 and κ_2 (see also Figure 4a on the next page) such that

1. T^{κ_1,κ_2} is right-total and right-unique,
2. all type variables in $\overline{\vartheta}$ are in $\overline{\alpha}$, which is a sequence of distinct type variables.

We say that $\tau_2 = \overline{\rho}\ \kappa_2$ is an *instance of an abstract type* of $\tau_1 = \overline{\sigma}\ \kappa_1$ if

1. κ_2 is an abstract type based on κ_1 given by $T^{\kappa_1,\kappa_2} : \overline{\vartheta}\ \kappa_1 \to \overline{\alpha}\ \kappa_2 \to \mathsf{bool}$,
2. $\overline{\sigma} = \theta\ \overline{\vartheta}$, where $\theta = \mathsf{match}(\overline{\rho}, \overline{\alpha})$ [3].

In the introductory examples, the Lifting tool internally generated transfer relations between the representation types and the abstract types: between $\alpha\ \mathsf{set}$ and $\alpha\ \mathsf{fset}$ for the finite set example and between $\mathsf{nat} \times \mathsf{nat}$

3 The function match is the usual matching algorithm, i.e., $\mathsf{match}(\overline{\tau}, \overline{\sigma})$ yields a substitution θ such that $\tau_i = \theta\ \sigma_i$ for all $i \in \{1, \ldots, n\}$.

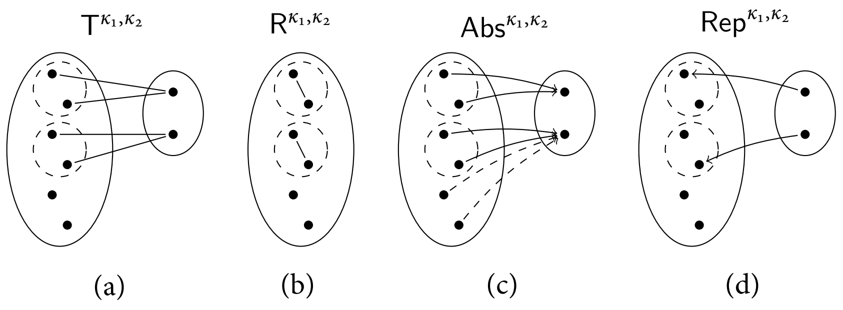

$\mathsf{T}^{\kappa_1,\kappa_2}$
$\mathsf{R}^{\kappa_1,\kappa_2}$
$\mathsf{Abs}^{\kappa_1,\kappa_2}$
$\mathsf{Rep}^{\kappa_1,\kappa_2}$
(a)
(b)
(c)
(d)

Figure 4: Components of an abstract type

and int for the integer example. In principle, such a transfer relation alone is sufficient to characterize all four kinds of type abstraction: type copies, subtypes, total and partial quotients. The other components that we can derive from every transfer relation $\mathsf{T}^{\kappa_1,\kappa_2}$ and associate with every abstract type are:

- Partial equivalence relation $\mathsf{R}^{\kappa_1,\kappa_2}$ (Figure 4b), defined as $\mathsf{R}^{\kappa_1,\kappa_2} = \mathsf{T}^{\kappa_1,\kappa_2} \circ\circ (\mathsf{T}^{\kappa_1,\kappa_2})^{-1}$.
- Abstraction function $\mathsf{Abs}^{\kappa_1,\kappa_2}$ (Figure 4c), conditionally specified by $\mathsf{T}^{\kappa_1,\kappa_2}\ a\ b \longrightarrow \mathsf{Abs}^{\kappa_1,\kappa_2}\ a = b$.
- Representation function $\mathsf{Rep}^{\kappa_1,\kappa_2}$ (Figure 4d), specified by the formula $\mathsf{T}^{\kappa_1,\kappa_2}\ (\mathsf{Rep}^{\kappa_1,\kappa_2}\ a)\ a$.

Since $\mathsf{T}^{\kappa_1,\kappa_2}$ is right-total and right-unique, there always exist some *Abs* and *Rep* functions that meet the above given specification. On the other hand, given a partial equivalence relation R, and functions *Abs* and *Rep*, the specification defines a unique T, which is right-total and right-unique.

The reflexive part of the partial equivalence relation $\mathsf{R}^{\kappa_1,\kappa_2}$ implicitly specifies which values of the concrete type are used for the construction of the abstract type.[4] The representation and abstraction functions map abstract values to concrete values and vice versa. $\mathsf{Abs}^{\kappa_1,\kappa_2}$ is underspecified outside of a range of T^{κ_1,κ_2} (dashed lines in Figure 4c) and $\mathsf{Rep}^{\kappa_1,\kappa_2}$ can select only one of the values in the corresponding class.

From a theoretical standpoint, we do not need the other components. We could build compound transfer relations for compound types and derive the other components from this relation (using the choice operator for the morphisms Morph^{p}). But it turns out that it is useful to have these other components explicitly: we need the morphisms for generating code equations (which I will explain in Chapter 6) and the equivalence relation is good for presenting the respectfulness theorem in a readable way.

4 We omitted the reflexive edges of $\mathsf{R}^{\kappa_1,\kappa_2}$ in Figure 4b.

Now we come to the key definition of this section. We derived R, *Abs* and *Rep* only for transfer relations that are associated with a type constructor. But later on, we build compound transfer relations for general types. What are R, *Abs* and *Rep* in this case? Again any functions meeting the above given specification. The following quotient predicate captures this idea and bundles all the components together.

Definition 5.2.2. We define a *quotient predicate* with the syntax $\langle ., ., ., . \rangle$ and we say that $\langle R, Abs, Rep, T \rangle$ if

1. $R = T \circ\circ T^{-1}$,
2. $T\ a\ b \longrightarrow Abs\ a = b$ and
3. $T\ (Rep\ a)\ a$.

The following definition requires that $\langle ., ., ., . \rangle$ is preserved by going through the type universe using map functions and relators.

Definition 5.2.3. We say that map^κ is a *quotient-friendly map function* for κ and rel^κ is a *quotient-friendly relator* for κ, where κ has arity n, if the assumptions $\langle R_1, m_1^+, m_1^-, T_1 \rangle, \ldots, \langle R_n, m_n^+, m_n^-, T_n \rangle$ imply

$$\begin{aligned}&\langle \mathsf{rel}^\kappa\ R_1 \ldots R_n,\\ &\ \mathsf{map}^\kappa\ m_1^{p_\kappa^1} \ldots m_n^{p_\kappa^n},\\ &\ \mathsf{map}^\kappa\ m_1^{-p_\kappa^1} \ldots m_n^{-p_\kappa^n},\\ &\ \mathsf{rel}^\kappa\ T_1 \ldots T_n \rangle\end{aligned}$$

for some $p_\kappa^1 \ldots p_\kappa^n$.[5]

Indexes p_κ^i encode which arguments of the map function are covariant ($+$) or contravariant ($-$). The $-p$ is a flipped variance of p. For example, the map function and the relator for the function type[6] are quotient friendly since the following holds:

$$\begin{aligned}&\langle R_1, Abs_1, Rep_1, T_1 \rangle \longrightarrow\\ &\langle R_2, Abs_2, Rep_2, T_2 \rangle \longrightarrow\\ &\quad \langle R_1 \Mapsto R_2, Rep_1 \mapsto Abs_2, Abs_1 \mapsto Rep_2, T_1 \Mapsto T_2 \rangle,\end{aligned} \tag{67}$$

We can see that $p_\to^1 = -$ and $p_\to^2 = +$. From now on, we will assume that every map^κ and rel^κ are quotient friendly.

5 The implementation of the lifting algorithm is more general: it allows reshuffling of arguments of rel^κ and map^κ, and map^κ can take up to $2n$ arguments because type arguments of κ can be in general covariant and contravariant at the same time, e.g., α in the type $\alpha \to \alpha$. This definition aims to be a sweet spot between generality and clarity.

6 The map function for the function type $\mapsto_{(\alpha\to\beta)\to(\gamma\to\delta)\to(\beta\to\gamma)\to\alpha\to\delta}$ is defined as $(h \mapsto g)\ f = g \circ f \circ h$.

Now we finally define Morph^p, Relat and Trans, as I promised to be the main goal of this section. First, let us define auxiliary functions morph^p, relat and trans, which will be used as the single step functions in the main inductive definition of Morph^p, Relat and Trans. Functions morph^p, relat and trans are defined for all types $\tau_1 = \overline{\sigma}\ \kappa_1$ and $\tau_2 = \overline{\rho}\ \kappa_2$ such that τ_2 is an instance of an abstract type based on τ_1, as follows:

- $\mathsf{morph}^+(\tau_1, \tau_2) = \mathsf{Abs}^{\kappa_1,\kappa_2}_{\tau_1\to\tau_2}$
- $\mathsf{morph}^-(\tau_1, \tau_2) = \mathsf{Rep}^{\kappa_1,\kappa_2}_{\tau_2\to\tau_1}$
- $\mathsf{relat}(\tau_1, \tau_2) = R^{\kappa_1,\kappa_2}_{\tau_1\to\tau_1\to\mathsf{bool}}$
- $\mathsf{trans}(\tau_1, \tau_2) = T^{\kappa_1,\kappa_2}_{\tau_1\to\tau_2\to\mathsf{bool}}$

Now we extend the simple step functions morph^p, relat and trans defined only for abstract types to functions Morph^p, Relat and Trans, which take general types τ_1 and τ_2. We extend them inductively on the structure of the two types (τ_1, τ_2) by distinguishing three cases:

- **Base case:** $\tau_1 = \tau_2$. Then we define

$$\begin{aligned}\mathsf{Morph}^p(\tau_1, \tau_2) &= \mathsf{id}_{\tau_1\to\tau_1},\\ \mathsf{Relat}(\tau_1, \tau_2) &= (=)_{\tau_1\to\tau_1\to\mathsf{bool}},\\ \mathsf{Trans}(\tau_1, \tau_2) &= (=)_{\tau_1\to\tau_1\to\mathsf{bool}}.\end{aligned}$$

- **Nonabstract type case:** $\tau_1 = \overline{\sigma}\ \kappa$ and $\tau_2 = \overline{\rho}\ \kappa$. Then we define

$$\begin{aligned}\mathsf{Morph}^p(\tau_1, \tau_2) &= \mathsf{map}^\kappa\ \mathsf{Morph}^{p^1_\kappa p}(\sigma_1, \rho_1) \ldots \mathsf{Morph}^{p^n_\kappa p}(\sigma_n, \rho_n),\\ \mathsf{Relat}(\tau_1, \tau_2) &= \mathsf{rel}^\kappa\ \mathsf{Relat}(\sigma_1, \rho_1) \ldots \mathsf{Relat}(\sigma_n, \rho_n),\\ \mathsf{Trans}(\tau_1, \tau_2) &= \mathsf{rel}^\kappa\ \mathsf{Trans}(\sigma_1, \rho_1) \ldots \mathsf{Trans}(\sigma_n, \rho_n),\end{aligned}$$

  where map^κ is a map function for κ, the term $p^i_\kappa p$ is the usual multiplication of polarities $(+ \cdot - = - \cdot + = -$ and $+ \cdot + = - \cdot - = +)$ and finally rel^κ is a relator for type κ.

- **Abstract type case:** $\tau_1 = \overline{\sigma}\ \kappa_1$, $\tau_2 = \overline{\rho}\ \kappa_2$ and $\kappa_1 \neq \kappa_2$. Additionally, we require two well-formedness conditions:
  1. κ_2 is an abstract type based on κ_1 given by $T^{\kappa_1,\kappa_2} : \overline{\vartheta}\ \kappa_1 \to \overline{\alpha}\ \kappa_2 \to \mathsf{bool}$.
  2. $\overline{\sigma} = \theta\ \overline{\vartheta}$, where $\theta = \mathsf{match}(\overline{\rho}, \overline{\alpha})$.[7]

7 Definition 5.2.1 guarantees that all type variables in $\overline{\vartheta}$ are in $\overline{\alpha}$ and thus $\overline{\rho}$ uniquely determines $\theta\ \overline{\vartheta}$.

Then we define these equations

$$\begin{aligned}
\mathsf{Morph}^{+}(\tau_1,\tau_2) &= \mathsf{morph}^{+}(\overline{\sigma}\,\kappa_1,\tau_2)\circ\mathsf{Morph}^{+}(\tau_1,\overline{\sigma}\,\kappa_1),\\
\mathsf{Morph}^{-}(\tau_1,\tau_2) &= \mathsf{Morph}^{-}(\tau_1,\overline{\sigma}\,\kappa_1)\circ\mathsf{morph}^{-}(\overline{\sigma}\,\kappa_1,\tau_2),\\
\mathsf{Relat}(\tau_1,\tau_2) &= \mathsf{Trans}(\tau_1,\overline{\sigma}\,\kappa_1)\circ\circ\,\mathsf{relat}(\overline{\sigma}\,\kappa_1,\tau_2)\\
&\quad\circ\circ\,\mathsf{Trans}(\tau_1,\overline{\sigma}\,\kappa_1)^{-1},\\
\mathsf{Trans}(\tau_1,\tau_2) &= \mathsf{Trans}(\tau_1,\overline{\sigma}\,\kappa_1)\circ\circ\,\mathsf{trans}(\overline{\sigma}\,\kappa_1,\tau_2).
\end{aligned}$$

By inspecting the above three cases, we will find out that the following two cases are not covered:

1. τ_1 or τ_2 is a type variable and $\tau_1 \neq \tau_2$ or
2. the well-formedness conditions 1. and 2. in the abstract type case are not fulfilled.

Thus, we also implicitly defined for which (τ_1, τ_2) the functions Morph^{p}, Relat and Trans are undefined and we cannot do the lifting. Notice that these cases represent inherent problems: either the types are structurally different or the required abstraction was not defined. In such cases the Lifting tool reports an error. Let us assume for the rest that we work only with such (τ_1, τ_2) that Morph^{p}, Relat and Trans are defined for.

Theorem 8. Morph^{p}, Relat and Trans have the following types:

$$\begin{aligned}
\mathsf{Morph}^{+}(\tau_1,\tau_2) &: \tau_1 \to \tau_2,\\
\mathsf{Morph}^{-}(\tau_1,\tau_2) &: \tau_2 \to \tau_1,\\
\mathsf{Relat}(\tau_1,\tau_2) &: \tau_1 \to \tau_1 \to \mathsf{bool},\\
\mathsf{Trans}(\tau_1,\tau_2) &: \tau_1 \to \tau_2 \to \mathsf{bool}.
\end{aligned}$$

Proof. By induction on defining equations of Morph^{p}, Relat and Trans. □

Thus in our context, where $t : \tau_1$, the terms $\mathsf{Relat}(\tau_1,\tau_2)\ t\ t$, $f = \mathsf{Morph}^{+}(\tau_1,\tau_2)\ t$ and $\mathsf{Trans}(\tau_1,\tau_2)\ t\ f$ are well-typed terms and f has the desired abstract type τ_2. The respectfulness theorem $\mathsf{Relat}(\tau_1,\tau_2)\ t\ t$ has to be proved by the user. We obtain the definitional theorem $f = \mathsf{Morph}^{+}(\tau_1,\tau_2)\ t$ as the result of Isabelle's internal definitional mechanism. The remaining task is to get the transfer rule $\mathsf{Trans}(\tau_1,\tau_2)\ t\ f$. The two following theorems give us the desired transfer rule.

Theorem 9. If $\langle R, Abs, Rep, T\rangle$, $f = Abs\ t$ and $R\ t\ t$, then $T\ t\ f$.

Proof. Because $R = T \circ\circ T^{-1}$, and $R\ t\ t$, we have $\exists x.\ T\ t\ x$. Let us denote this x by g. Thus $Abs\ t = g$ follows from $T\ t\ g$. But from $f = Abs\ t$ we can derive $f = g$ and thus $T\ t\ f$. □

The following Theorem 10 is the key theorem of this section: it proves that our definitions of Morph^{p}, Relat and Trans are legal, i.e., they have the desired property that they still form a (compound) abstract type, i.e., they meet the quotient predicate. First a technical lemma.

Lemma 5.2.4. The quotient predicate is preserved through the composition of abstract types: $\langle R_1, Abs_1, Rep_1, T_1\rangle$ and $\langle R_2, Abs_2, Rep_2, T_2\rangle$ imply $\langle T_1 \circ\circ R_2 \circ\circ T_1^{-1}, Abs_2 \circ Abs_1, Rep_1 \circ Rep_2, T_1 \circ\circ T_2\rangle$.

Proof. Directly from the definition of $\langle ., ., ., .\rangle$ and basic facts about functions and relations. □

Theorem 10.

$$\langle \mathsf{Relat}(\tau_1, \tau_2), \mathsf{Morph}^{+}(\tau_1, \tau_2), \mathsf{Morph}^{-}(\tau_1, \tau_2), \mathsf{Trans}(\tau_1, \tau_2)\rangle$$

Proof. By induction on defining equations of Morph^{p}, Relat and Trans:

- Base case: $\langle =, \mathsf{id}, \mathsf{id}, =\rangle$ holds.
- Nonabstract type case: $\langle ., ., ., .\rangle$ is preserved through this case as it is required by Definition 5.2.3. We obtain the assumptions of the condition in Definition 5.2.3 from the induction hypothesis.
- Abstract type case: We use Lemma 5.2.4, which shows that $\langle ., ., ., .\rangle$ is preserved through the composition of abstract types. The first assumption is obtained from the induction hypothesis (notice that we use simpler types here) and the second assumption is

  $$\langle \mathsf{relat}(\tau_1, \tau_2), \mathsf{morph}^{+}(\tau_1, \tau_2), \mathsf{morph}^{-}(\tau_1, \tau_2), \mathsf{trans}(\tau_1, \tau_2)\rangle,$$

  which holds by construction.

□

Now we can compose Theorem 9 and Theorem 10 together with the definitional theorem $f = \mathsf{Morph}^{+}(\tau_1, \tau_2)\ t$ and the respectfulness theorem $\mathsf{Relat}(\tau_1, \tau_2)\ t\ t$ and obtain the desired transfer rule $\mathsf{Trans}(\tau_1, \tau_2)\ t\ f$.

5.3 setup interface

A new abstract type κ_2 of κ_1 is registered in the Lifting tool by providing a quotient theorem $\langle R, Abs, Rep, T_{\overline{\vartheta}\ \kappa_1 \to \overline{\alpha}\ \kappa_2 \to \mathsf{bool}}\rangle$ such that $\mathsf{TV}(\overline{\vartheta}) \subseteq \mathsf{TV}(\overline{\alpha})$ and $\overline{\alpha}$ contains only distinct type variables. This defines

$$\begin{aligned} \mathsf{R}^{\kappa_1,\kappa_2} &= R,\\ \mathsf{Abs}^{\kappa_1,\kappa_2} &= Abs,\\ \mathsf{Rep}^{\kappa_1,\kappa_2} &= Rep,\\ \mathsf{T}^{\kappa_1,\kappa_2} &= T. \end{aligned}$$

Users generally will not prove the quotient theorem manually for new types because the Lifting tool automates the process for the two most common commands that can define a new abstract type—`quotient_type` and `typedef` command.

`quotient_type`: Let us recall how the `quotient_type` works: it takes as an input a (partial) equivalence relation R, defines the demanded quotient type and it provides the coercions *Abs* and *Rep* with the properties that I described in Section 2.4.3 on page 20. The tool defines the corresponding transfer relation T as the graph of the well-defined part of *Abs*: $T\ x\ y = (Abs\ x = y)$ for total quotients and $T\ x\ y = R\ x\ x \wedge (Abs\ x = y)$ for partial quotients. Using the definition of T and the properties of *Abs* and *Rep*, the tool automatically proves $\langle R, Abs, Rep, T\rangle$.

`typedef`: The `typedef` command takes a nonempty set S and defines a type isomorphic to the set, which is witnessed by the pair of coercions *Abs* and *Rep*—see Section 2.4.2 on page 19. We define T to be again the graph of the well-defined part of *Abs* but for subtypes we can do it in a more uniform way via *Rep*: $T\ x\ y = (x = Rep\ y)$ for both type copies and proper subtypes. How we define R, depends on how the set S was expressed. If $S = \mathsf{UNIV}$, we set $R = (=)$; if $S = \{x \mid P\ x\}$, we set $R = (=^{\mathsf{on}})\ P$; otherwise we set $R = (=^{\mathsf{on}})\ (\lambda x.\, x \in S)$, where $=^{\mathsf{on}}$ is the restricted equality defined as

$$=^{\mathsf{on}}\ P\ x\ y = (P\ x \wedge x = y).$$

Thus we encode a subset S by an equivalence relation that contains only a diagonal on S. Finally, the Lifting tool automatically proves the quotient predicate $\langle R, Abs, Rep, T\rangle$ (regardless of the variant of R).

During the setup, there are more theorems that get automatically proved. I will mention only the most prominent ones here—transfer rules. Since we defined the transfer relation T in this step, the tool proves properties of it such as bi-uniqueness or bi-totality. See the Table 1 on page 112 for an overview which properties for which cases of type abstraction are proved. Last but not least, if T is not bi-unique, the tool proves the following transfer rule for =:

$$(T \Mapsto T \Mapsto =)\ R\ =$$

This rule certifies that the equality on a quotient type transfers to the corresponding equivalence relation on the representation type.

The interface that we use for setting up the Lifting tool from the proof text comprises the command `setup_lifting`. The user has to supply the respective theorem: either a quotient theorem or a typedef theorem. For example, the `typedef` command in (60) generated the theorem type-definition-fset witnessing that fset is indeed a subtype. We can use this theorem to set up the Lifting tool as follows:

```
setup_lifting type-definition-fset
```

A similar theorem is generated by the quotient_type command. As we will see in Section 5.5, not only theorems from these two commands could be used. It can be any quotient theorem, for example, proved manually.

5.4 readable form of respectfulness theorems

Since the respectfulness theorem is the only proof obligation presented to the user, I also implemented a procedure that rewrites the internal point-free form into an equivalent, user-friendly, readable form, which the tool presents in lift_definition. We distinguish three cases of abstract types in this section: quotients, (proper) subtypes and type copies.

quotients The procedure is rather simple here—e.g., this is the internal form of the respectfulness theorem for plus_i (defined on page 97):

$$(\mathsf{intrel} \Mapsto \mathsf{intrel} \Mapsto \mathsf{intrel})\ \mathsf{plus\_nn}\ \mathsf{plus\_nn}$$

In order to obtain the readable form (66), we unfold the definition of $\Mapsto$.

subtypes Let us have an abstract function of the type $\sigma_1 \to \ldots \to \sigma_n \to \sigma_{n+1}$ (where σ_{n+1} is not a function) with the respectfulness theorem $(R_1 \Mapsto \ldots R_n \Mapsto R_{n+1})\ t\ t$. For the start, let us assume that every σ_i is either a subtype without nesting or a nonabstract type. Then the respectfulness theorem has a special form

$$(=^{\mathsf{on}}\ P_1 \Mapsto \ldots =^{\mathsf{on}}\ P_n \Mapsto =^{\mathsf{on}}\ P_{n+1})\ t\ t. \tag{68}$$

Using two rewriting rules

$$\begin{aligned}(=^{\mathsf{on}}\ P \Mapsto R)\ f\ f \longleftrightarrow (\forall x.\, P\ x \longrightarrow R\ (f\ x)\ (f\ x))\ \text{and}\\ =^{\mathsf{on}}\ P\ x\ x \longleftrightarrow P\ x,\end{aligned}$$

we can rewrite (68) into a more readable form

$$\forall x_1 \ldots x_n.\, P_1\ x_1 \longrightarrow \ldots \longrightarrow P_n\ x_n \longrightarrow P_{n+1}\ (t\ x_1 \ldots x_n).$$

This procedure already covers the example with a definition of union on finite sets (62). It rewrote the internal respectfulness theorem

$$(=^{\mathsf{on}}\ \mathsf{finite} \Mapsto =^{\mathsf{on}}\ \mathsf{finite} \Mapsto =^{\mathsf{on}}\ \mathsf{finite})\ \cup\ \cup$$

into a more readable form

$$\forall X_{\alpha\ \mathsf{set}}\ Y_{\alpha\ \mathsf{set}}.\ \mathsf{finite}\ X \longrightarrow \mathsf{finite}\ Y \longrightarrow \mathsf{finite}\ (X \cup Y).$$

Naturally, we would like to cover a broader class of respectfulness theorems than just theorems of the form (68): we want to have abstract types not only at the top of σ_i (e.g., α fset option) and we want to nest the abstract types (e.g., α fset fset). To achieve the generalization, we proceed as follows: we rewrite every R_i corresponding to σ_i into the form $=^{\mathsf{on}}\ t$, where t is considered to be readable, and thus rewrite the more general respectfulness theorem into the form (68). Concretely, we use the following rewriting rules to gradually move $=^{\mathsf{on}}$ to the top of the relation:

1. For every type constructor κ that is a natural functor we automatically prove and use the following rule:

$$\mathsf{rel}^{\kappa}\ (=^{\mathsf{on}}\ P) = (=^{\mathsf{on}})\ (\mathsf{pred}^{\kappa}\ P)$$

2. For the function type, we use the built-in rule

$$(= \Mapsto =^{\mathsf{on}}\ P) = (=^{\mathsf{on}})\ (\forall x.\ P\ x).$$

3. For the abstract type composition, we use the built-in rule

$$\frac{\mathsf{Domp}\ T = P_1 \qquad \mathsf{bi\_unique}\ T \qquad (T \Mapsto =)\ P_2\ P_2'}{(T \circ\circ =^{\mathsf{on}}\ P_2' \circ\circ T^{-1}) = (=^{\mathsf{on}})\ (\forall x.\ P_1\ x \wedge P_2\ x)}.$$

The third rather complicated rule deserves an explanation. We discharge the first and the second assumption by the transfer algorithm from the Transfer tool. The first assumption is used to replace Domp T by a more readable term. This is a mechanism of the Transfer tool and was explained in Section 4.4 on page 77; e.g., we replace Domp SF by finite. The last assumption requires that the defining predicate of the inner type must be parametric (for bi-unique relations) and is discharged by transfer_prover. Let me demonstrate the meaning of the last assumption by an example. Let us define the union operator on finite sets:

$$\mathtt{lift\_definition}\ \widetilde{\bigcup} : \alpha\ \mathsf{fset}\ \mathsf{fset} \to \alpha\ \mathsf{fset}\ \mathtt{is}\ \bigcup \quad \langle \mathit{proof} \rangle$$

The internal respectfulness theorem is the following beast

$$(\mathsf{rel\_set}\ \mathsf{SF} \circ\circ =^{\mathsf{on}}\ \mathsf{finite} \circ\circ (\mathsf{rel\_set}\ \mathsf{SF})^{-1} \Mapsto =^{\mathsf{on}}\ \mathsf{finite})\ \bigcup\ \bigcup,$$

which gets rewritten by our procedure to an equivalent statement, which one would expect from looking at the type of $\widetilde{\bigcup}$:

$$\forall SS_{\alpha\ \mathsf{set}\ \mathsf{set}}.\ (\forall S_{\alpha\ \mathsf{set}} \in SS.\ \mathsf{finite}\ S) \wedge \mathsf{finite}\ SS \longrightarrow \mathsf{finite}\ (\bigcup SS) \quad (69)$$

The twist is that the finite predicates that are marked red in both statements have different types: α fset set $\to$ bool vs. α set set $\to$ bool. This is the place where parametricity comes into play: these two different type instances of finite are not equivalent unless finite is parametric. At the end, notice how many different concepts we had to use to obtain (69): parametricity, transfer_prover, transfer rules, natural functors (we needed the rule for rel_set).

Let me conclude: the introduced procedure can rewrite any respectfulness theorem into a readable form under the following mild assumptions, which stem from the above-mentioned three categories of rewriting rules:

1. Nonabstract type constructors must be natural functors.[8]

2. If σ_i is a function type, the abstract type can be only in the positive position. This condition is hardly violated in practice.

3. The defining predicate of the subtype must be parametric for bi-unique[9] relations.

type copies Also here we rewrite the respectfulness theorem to a more readable form; in this case to an extremely readable form: True. I implemented a procedure that can prove the respectfulness theorem automatically if all involved abstract types are type copies. Observe that a respectfulness theorem $R\ f\ f$ is true if and only if R is reflexive, which is if and only if $= \sqsubseteq R$. I built a more general *monotonicity prover* that can prove statements such as $= \sqsubseteq R$ or $R \sqsubseteq =$.

The prover is a simple prover that repeatedly applies the following monotonicity rules:

1. Each time a total quotient $\langle R, Abs, Rep, T\rangle$ is registered, we register the following monotonicity rule:

$$\frac{}{= \sqsubseteq R}$$

2. There are trivial rules

$$\frac{}{= \sqsubseteq =} \qquad \frac{}{=^{\mathsf{on}}\ P \sqsubseteq =}$$

and rules for relations from the composition of quotients:[10]

$$\frac{\mathsf{bi\_total}\ T \qquad = \sqsubseteq R}{= \sqsubseteq T \circ\circ R \circ\circ T^{-1}} \qquad \frac{\mathsf{bi\_unique}\ T \qquad R \sqsubseteq =}{T \circ\circ R \circ\circ T^{-1} \sqsubseteq =}$$

8 Or the user provides a manually proved theorem of the form $\mathsf{rel}^{\kappa}\ (=^{\mathsf{on}}\ P) = (=^{\mathsf{on}})\ t$, where t is considered to be readable.

9 In fact, the strictly weaker left-uniqueness suffices but this is not relevant in our use case.

10 Left-totality (left-uniqueness respectively) would suffice here but this is not relevant in our use case.

3. For any type constructor κ for which we store the monotonicity rule

$$R_1\ \mathcal{O}_1\ S_1 \longrightarrow \ldots \longrightarrow R_n\ \mathcal{O}_n\ S_n \longrightarrow \mathsf{rel}^\kappa\ \overline{R} \sqsubseteq \mathsf{rel}^\kappa\ \overline{S},$$

where $\mathcal{O}_i$ is either $\sqsubseteq$ or $\sqsupseteq$ and for which we store[11] the identity rule

$$(\mathsf{rel}^\kappa = \ldots =) = (=),$$

we derive the following two rules for the prover:

$$\frac{R_1\ \mathcal{O}_1 = \qquad \ldots \qquad R_n\ \mathcal{O}_n =}{\mathsf{rel}^\kappa\ \overline{R} \sqsubseteq =}$$

$$\frac{= \mathcal{O}_1\ R_1 \qquad \ldots \qquad = \mathcal{O}_n\ R_n}{= \sqsubseteq \mathsf{rel}^\kappa\ \overline{R}}$$

Let me remind you that since the monotonicity and identity rules are automatically proved for any κ that is a natural functor or the function type, we obtain the two above-stated rules for any natural functor as well.

5.5 modular design of transfer and lifting

In the introduction, I mentioned that the Lifting tool is the main provider of transfer rules for the Transfer tool. Let me elaborate on this slogan.

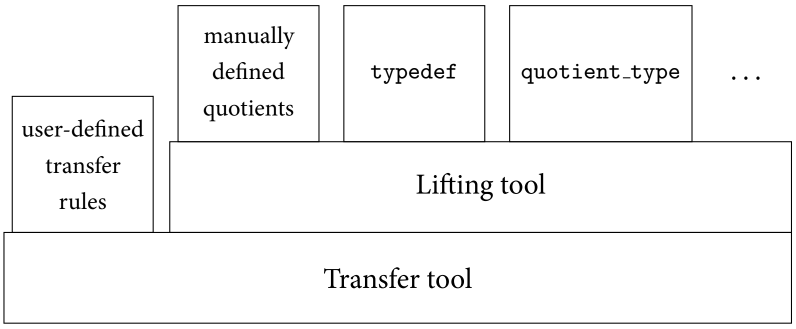


Figure 5: Modular design of tools for abstract types

When we decided to build a platform that would support creating type abstraction, we came up with a solution that uses a layered design, with multiple components and interfaces that are related as shown in Figure 5. Each component depends only on the components underneath it. At the

11 We store monotonicity and identity rules in the Transfer tool's infrastructure. See Section 4.7 on page 85.

bottom is the Transfer tool, which transfers propositions between concrete and abstract types. Note that the Transfer tool has no dependencies; it does not know anything about the representation and abstraction functions or quotient predicates. It is configured by an extensible set of transfer rules.

Above Transfer is the Lifting tool for defining constants on abstract types. It configures each new constant to work with the Transfer tool by providing a transfer rule for it. At the top are commands that configure new types to work with Lifting, such as `typedef` and `quotient_type` as I explained in Section 5.3. In principle, additional type definition commands might be implemented later. The Lifting tool is configured by quotient theorems.

The Lifting tool is the main provider of transfer rules for the Transfer tool but it is not the only one. As we saw in Chapter 4, the user can setup transferring between two existing types that were defined independently on each other, including their operations. As an example, we transferred propositions between natural and integers numbers by means of the user-defined transfer rules (23) to (28).

Similarly, `typedef` and `quotient_type` are the main providers of quotients for the Lifting tool but the user can prove manually their own quotient theorem if they need a special construction. There exist examples (e.g., the library of finite bit strings) where this flexibility is needed.

5.6 implementation

The implementation of the lifting algorithm roughly follows the abstract descriptions from Section 5.2. There are some interesting implementation aspects worth mentioning here.

Theorem 9 tells us how to obtain the transfer rule for a newly defined constant: we have to discharge the three assumptions of the theorem. We already know that we get the respectfulness theorem proved by the user and the definitional theorem from the kernel of the system. Finally, Theorem 10 gives us a recipe how to prove the quotient theorem.

I implemented a syntax-driven procedure that proves a quotient theorem for a given pair of types τ_1 and τ_2 by following the induction proof of Theorem 10: if the types τ_1 and τ_2 are the same, the procedure returns the trivial quotient theorem $\langle =, \mathsf{id}, \mathsf{id}, = \rangle$; in the abstract type case, we use a quotient theorem from the recursive call, the quotient theorem $\langle \mathsf{relat}(\tau_1, \tau_2), \mathsf{morph}^+(\tau_1, \tau_2), \mathsf{morph}^-(\tau_1, \tau_2), \mathsf{trans}(\tau_1, \tau_2) \rangle$[12] and Lemma 5.2.4; in the nonabstract case, the procedure uses a theorem that certifies that the quotient predicate is preserved by the type constructor

12 The theorem is an instance of the quotient theorem registered by the user for the abstract type in question—see Section 5.3.

in question, i.e., a theorem in the style of Definition 5.2.3 on page 101. We already saw such a theorem for the function type ((67) on page 101). Another example of such a theorem this time for α list:

$$\begin{gathered}\langle R, Abs, Rep, T\rangle \longrightarrow \\ \langle \mathsf{list\_all2}\ R, \mathsf{map}\ Abs, \mathsf{map}\ Rep, \mathsf{list\_all2}\ T\rangle\end{gathered} \tag{70}$$

These theorems are either already part of the library (e.g., for the function type) or are proved automatically (for any natural functor) or must be provided by the user.

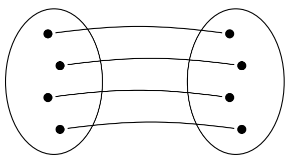
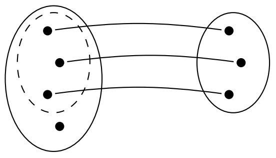
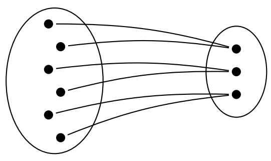
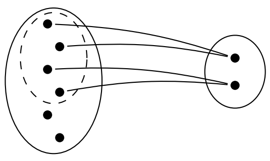
Now comes the main twist. Notice that we do not provide the statement of the quotient theorem and let the procedure prove it, but we provide only a pair of types as an input and the procedure *synthesize* and proves the corresponding quotient theorem. Thus we do not only prove the compound quotient theorem to derive the transfer rule but also to synthesize the terms $\mathsf{Morph}^p(\tau_1, \tau_2)$, $\mathsf{Relat}(\tau_1, \tau_2)$ and $\mathsf{Trans}(\tau_1, \tau_2)$ as a side effect. Thanks to this approach, I managed to get a simple implementation, which does not suffer from various technical limitations of the original quotient tool [48].[13]

5.7 coercion equations

There is another category of theorems that the `lift_definition` command proves. They show how the newly defined abstract function interacts with the coercions of the abstract type. I claimed in the introduction that we want to shield the user from the coercion functions and this section does not contradict the objective. The users do not usually encounter the coercion equations. Their main role is that they can be used as code equations for the code generator under some restrictions. The code generator is a central component in Isabelle/HOL and is used in a lot of projects for program and algorithm verification. I will explain the role of the Lifting tool in code generation in a separate chapter—Chapter 6.

For the present, I will define the format of the coercion equations and sketch how to prove them. Let us assume that the abstract function is called f and its type is $\sigma_1 \to \ldots \to \sigma_n \to \sigma_{n+1}$, where σ_{n+1} is not a function type. From Section 5.2 we know that f was defined as $f = Abs\ t$, the respectfulness theorem was $R\ t\ t$ and there is a corresponding coercion Rep such that $\langle R, Abs, Rep, T\rangle$ for some transfer relation T. Now using our knowledge of the lifting algorithm and of the quotient theorem for

13 The main advantage is that we do not need any formal notion of covariant and contravariant parameters of a type constructor and also any complicated procedures for building terms. All of this is implicitly encoded in the quotient theorems from Definition 5.2.3 on page 101—we can see this in the concrete examples (67) and (70).

	total equivalence relation	partial equivalence relation
trivial relation (subset of =)	**type copy**  transfer relation: bi-unique, bi-total rep_eq: + abs_eq: + example: mappings (α, β) mapping = $\alpha \rightharpoonup \beta$[14]	**subtype**  transfer relation: bi-unique, right-total rep_eq: + abs_eq: ~ example: finite sets α fset = $\{S_{\alpha\ \mathsf{fset}} \mid$ finite $S\}$
nontrivial relation	**total quotient**  transfer relation: right-unique, bi-total rep_eq: − abs_eq: + example: integer numbers int = nat × nat/intrel	**partial quotient**  transfer relation: right-unique, right-total rep_eq: − abs_eq: ~ example: rational numbers α rat = int × int/ratrel

+ ... yes − ... no ~ ... only with assumptions

Table 1: Categorization of type abstractions

the function type (67), we can conclude that $R = R_1 \Mapsto \ldots \Mapsto R_n \Mapsto R_{n+1}$, $Abs = Rep_1 \mapsto \cdots \mapsto Rep_n \mapsto Abs_{n+1}$ and $Rep = Abs_1 \mapsto \cdots \mapsto Abs_n \mapsto Rep_{n+1}$ for some R_i, Abs_i and Rep_i, whose form is not important here.

There exist two coercion equations associated with f:

- *Representation function equation* has form

$$Rep_{n+1}\ (f\ x_1 \ldots x_n) = t\ (Rep_1\ x_1) \ldots (Rep_n\ x_n).$$

- *Abstraction function equation* has form

$$f\ (Abs_1\ x_1) \ldots (Abs_n\ x_n) = Abs_{n+1}\ (t\ x_1 \ldots x_n).$$

representation function equation. By unfolding the map function $\mapsto$ in the definition of f and using simple facts we get $Rep_{n+1}\ (f\ x_1 \ldots x_n) = Rep_{n+1}\ (Abs_{n+1}\ t')$, where $t' = t\ (Rep_1\ x_1) \ldots (Rep_n\ x_n)$. From the

14 The type $\alpha \rightharpoonup \beta$ is syntax for $\alpha \rightarrow \beta$ option.

respectfulness theorem it follows that $R_{n+1}\ (Rep_{n+1}\ (Abs_{n+1}\ t'))\ t'$. If we were able to prove that the relation R_{n+1} is a subset of the equality, we could conclude that $Rep_{n+1}\ (Abs_{n+1}\ t') = t'$ and obtain the desired equation $Rep_{n+1}\ (f\ x_1 \dots x_n) = t\ (Rep_1\ x_1) \dots (Rep_n\ x_n)$. The monotonicity prover from Section 5.4 can prove that $R_{n+1} \sqsubseteq =$ if Rep_{n+1} and Abs_{n+1} represent a subtype or a type copy.

abstraction function equation. It holds that $R\ (Rep\ f)\ t$. By unfolding $\Mapsto$ in R and $\mapsto$ in Rep and using the fact that $\forall x\ y.\ R_{n+1}\ (Rep_{n+1}\ x)\ y \longrightarrow x = Abs_{n+1}\ y$, we get this conditional equation $R_1\ x_1\ x_1 \longrightarrow \dots \longrightarrow R_n\ x_n\ x_n \longrightarrow f\ (Abs_1\ x_1) \dots (Abs_n\ x_n) = Abs_{n+1}\ (t\ x_1 \dots x_n)$. We will try to discharge the assumptions by the monotonicity prover. If R_1 to R_n are relations that are composed from relators that preserve reflexivity (e.g., any natural functor relator) and the abstract types that are involved are total (i.e., a type copy or a total quotient), the monotonicity prover succeeds for every assumption and gives us a plain equation.[15]

Overall, we can generate representation function equations for type copies and subtypes, and abstraction function equations with extra assumptions for any abstract type. We can discharge the extra assumptions for total abstract types and thus obtain plain equations. See Table 1 for an overview, for which type abstractions which coercion equations we get.

5.8 related work

The related work coincides with the related work about quotients that I presented in Chapter 4. Slotosch [85], Paulson [78] and Harrison [33] implemented tools that can define first-order functions on quotients. In 2005, Homeier [39] published the first system capable of defining higher-order functions. In 2011, Kaliszyk and Urban [48] identified a limitation of Homeier's tool: although it can work with higher-order functions, it still does not support a composition of quotient types. For example, the union operator on finite sets (defined as a quotient on lists) $\widetilde{\bigcup} : \alpha\ \mathsf{fset}\ \mathsf{fset} \to \alpha\ \mathsf{fset}$ cannot be defined by Homeier's tool.

Kaliszyk and Urban's tool improved on Homeier's tool: it can produce the right definition for functions with nested quotients in their types (e.g., $\widetilde{\bigcup}$) but the compound equivalence relation for nested quotients was defined differently from our solution. In the Abstract type case in the Lifting algorithm on page 102, they would use the following definition:

$$\mathsf{Relat}(\tau_1, \tau_2) = \mathsf{Relat}(\tau_1, \overline{\sigma}\ \kappa_1) \circ\circ \mathsf{relat}(\overline{\sigma}\ \kappa_1, \tau_2) \circ\circ \mathsf{Relat}(\tau_1, \overline{\sigma}\ \kappa_1)$$

15 The function relator $\Mapsto$ does not preserve reflexivity in the negative position (but it still preserves (=), i.e., type copies). This does not limit us in practice, as there is hardly a function with a functional parameter with an abstract type in a negative position.

Although Theorem 8 still holds, Theorem 10 cannot be proved as Kaliszyk and Urban noticed themselves [48, end of §2]: *Unfortunately a general quotient theorem ... would not be true in general.* The reason is that their version of the lemma stating that the quotient predicate is preserved through the composition of abstract types (Lemma 5.2.4 in our setting)

$$
\begin{aligned}
&\langle R_1, Abs_1, Rep_1, T_1\rangle \longrightarrow\\
&\langle R_2, Abs_2, Rep_2, T_2\rangle \longrightarrow\\
&\quad\langle R_1 \circ\circ R_2 \circ\circ R_1, Abs_2 \circ Abs_1, Rep_1 \circ Rep_2, T_1 \circ\circ T_2\rangle
\end{aligned}
$$

does not hold. Thus their quotient tool still does not cover the whole possible type universe because it requires to prove unprovable respectfulness theorems for some functions (using composed quotient types) that still respect the corresponding equivalence relations. The Lifting tool does not suffer from this problem.

This is the main *theoretical contribution* of my work on the Lifting tool: considering tools for quotients in higher-order logic, the Lifting tool is the first tool that covers the whole type universe (especially higher-order types and compositions of quotients). This claim is supported by Theorem 10. The definition of an abstract function fails only when this is inherently impossible: either the representation and abstract types are structurally different or the required abstraction is not defined.

I will quickly list other differences of the Lifting tool (and also the Transfer tool) from the tool by Kaliszyk and Urban:

- The modular design yields flexibility, i.e., Lifting is not limited to types defined by `quotient_type` and similarly Transfer to constants defined by `lift_definition`.
- The Lifting tool supports arbitrary type constructors, rather than only covariant ones plus the hard-coded function type.
- The respectfulness theorem is generated by the tool and is presented in the user-readable form. Moreover, the theorem is automatically discharged for type copies.
- The Lifting tool proves the coercion equations and is thus integrated with the code generator.

The problems are solved, not by giving new information, but by arranging what we have known since long.

— Ludwig Wittgenstein (1953)

6 USE CASE: DATA REFINEMENT

Data refinement is an important technique that allows us to reason abstractly and yet obtain efficient code. Our methodology reduces data refinement to code generation by means of only small enlargement of the trusted code base thanks to using abstract types at various levels.

6.1 background

Algorithm (or program) verification is one of the main usage scenarios of Isabelle/HOL. In this scenario, the user defines a specification of an algorithm and proves desired correctness properties of it. The last step is to extract executable code from the specification by using the *code generator*, which guarantees that the generated code possesses the same correctness properties as the specification.

Isabelle/HOL supports code generation for a number of functional programming languages (SML, OCaml, Haskell, Scala). Basically, equational theorems in HOL, called *code equations*, are translated into function definitions in the target languages. A mathematical treatment of this translation process, including correctness proofs, can be found elsewhere [29]. We stay on the level of code equations here and do not need to worry about the further translation steps. The semantics of the generated code is that we will see evaluations of it as rewriting steps in a certain higher-order rewrite system that is an abstraction of all the target languages. The key correctness property with respect to this semantics is that any rewriting step $s \rightsquigarrow t$ in this system must be a provable equality $s = t$ in HOL.

When we want to generate code for some function f, any list of equations of the form $f \ \ldots = \ldots$ (with pattern matching on the lhs) can (in principle) serve as code equations, not just the original definition of f. Thus we are free to define a second, more efficient function g, prove $f(x) = g(x)$, and use this equation (together with the ones for g) as the code equations for f. Algebraic datatypes in HOL are turned into equivalent algebraic datatypes in the target language. Interestingly, the correctness proof revealed that in fact any function in HOL can in principle become a constructor function in the target language (but of course not a defined function at the same time).

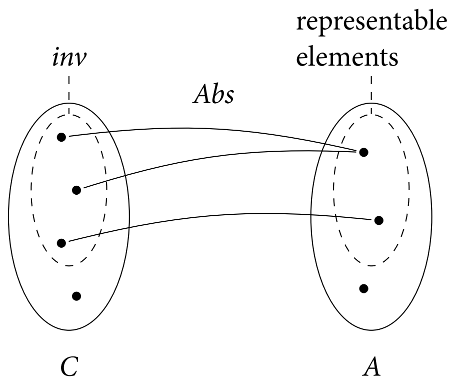


Figure 6: Data refinement

Algorithm verification is most convenient at a high level of abstraction, reasoning about data in terms of sets, functions and other mathematical concepts. However, when running the verified code we want to replace sets and functions by lists and trees, to make them efficiently executable. This replacement is called *data refinement*, and the ideal theorem prover should do this fully automatically once we prove that the concrete representation is adequate.

More formally, data refinement replaces an *abstract* datatype A by a more *concrete* one C in the generated code. The typical example is the implementation of sets by lists. The concrete type is also called the *implementation* or *representation*. Refining A by C requires an *abstraction function* $Abs : C \to A$ (e.g., mapping $[1, 2]$ to $\{1, 2\}$) and an *invariant* $inv : C \to \mathsf{bool}$ (e.g., ruling out lists with duplicates). The basic picture is shown in Figure 6.

The standard approach is to demand that *Abs* is a homomorphism: for every operation $f \in \Sigma$ (the primitive operations that need implementing) on the abstract type and its concrete implementation f' it must be shown that $f(Abs(x)) = Abs(f'(x))$. A system supporting data refinement on this basis will require the user to prove the homomorphism property for all operations to ensure soundness of the refinement step. This means that a new trusted component is added to the system: the refinement manager. A typical example for this approach is the KIV system [82].

When we design an extension of our proof assistant, we always try to find a sweet spot between trustworthiness and usability. We measure trustworthiness in terms of how much code must be trusted, how complex the code is and how complex the correctness argument for the code is. We followed the same approach when we planned to add support for data refinement into the code generator.

In order to add as little trusted code as possible, we turn the standard approach to data refinement on its head: Rather than check that the correct homomorphism theorems have been proved before code is generated, the homomorphism theorems themselves are the glue code between f and f'. More precisely, we instruct the code generator to view A as an algebraic datatype with the single (uninterpreted!) constructor $\mathit{Abs} : C \to A$. Now $f(\mathit{Abs}(x)) = \mathit{Abs}(f'(x))$ is a code equation that performs pattern matching on Abs to turn a call of f into a call of f'. This is the key point of our approach: We generate code for the actual function f, not for some other function f' for which some additional theorems show that it implements f in the correct manner. This form of data refinement is completely automatic: Once a particular refinement of A by C has been set up, generating code involving functions on A involves no further input by the user. Moreover, this approach is covered by the original semantics and no new trusted extension is needed.

Unfortunately it breaks down once we have a nontrivial invariant and can only prove $\mathit{inv}(c) \longrightarrow f(\mathit{Abs}(c)) = \mathit{Abs}(f'(c))$. This is a conditional equation and thus unsuitable for generating code. As I stressed a couple of times in this text, we can introduce a new *subtype* defined by *inv* to make the equation unconditional and thus to make our trick work even when we have a nontrivial invariant. At this point we need to introduce a minimal extension of the code generator to execute functions defined on subtypes. This is the content of Section 6.2.

We call a type σ *basic* if σ contains an abstract type constructor either only at the top or not at all. More formally: let the abstract type be $\overline{\alpha}\ \kappa$ then either $\sigma = \overline{\tau}\ \kappa$ where each τ_i does not contain κ or σ does not contain κ at all.

To keep our methodology simple, as well as its correctness proof, our approach assumes two restrictions:

1. The refinement happens on a per type constructor basis, i.e., the type that is being refined has to be of the form $\overline{\alpha}\ \kappa$. We propose this workaround: if we want to refine a general type expression τ (e.g., $\alpha \to \beta$ option), we will create a *type copy* of τ and transfer our specification to the new type. This is the subject of Section 6.3.

2. Using the extension for invariants and given a function $f : \sigma_1 \to \ldots \to \sigma_n \to \sigma$ (where σ is not a function type) that is being executed as primitive operation, the return type σ must be basic. Thus σ can be α fset but not α fset option. A workaround: I implemented a fully automated procedure that defines a provably equivalent specification for f such that f is not a primitive operation anymore and can be executed, provided the outer nonabstract type in σ is a (co)datatype. The procedure introduces *auxiliary subtypes* that are isomorphic to the types violating the restriction and defines appropriate projection functions that conform the restriction. See Section 6.4 for details.

Thus we can keep the sweet spot between trustworthiness and usability: the trusted extension (and its proof of correctness) is small and at the same time its restrictions can be lifted both thanks to abstract types.

Needless to say, we use the Transfer and Lifting tools in all cases when we use abstract types in our methodology. Let me note that we should *not* view the role of Transfer and Lifting in our approach as follows: there is a methodology for reducing data refinement to code generation and later somebody implemented an automation for a part of it to make the life of the users easier. We treat the automation rather as an *integral* part of the methodology because it makes our solution viable in practice.

The code generator together with the methodology for data refinement was developed by Florian Haftmann and Tobias Nipkow. Alexander Krauss came up with the idea to use quotient tools for automating parts of Florian and Tobias's approach. My contribution was to make Alexander's idea come into existence and to develop the technique to lift one of the main limitation of the approach, the return type limitation (Section 6.4). This chapter is based on our joint paper [28].

6.2 data refinement with invariants

I will explain how to perform data refinement with invariants by first considering the case without invariants and then later bring invariants into play. The last part shows how to automate our approach by the Transfer and Lifting tools.

6.2.1 Standard Method without Invariants

The standard example for refinement without invariants is the implementation of sets by lists with no restrictions on the order or multiplicity of the elements in the lists.

The relation between lists and sets is an instance of Figure 6 where $C = \alpha\ \mathsf{list}$, $A = \alpha\ \mathsf{set}$, *inv* is true everywhere (every list is a valid representation) and $Abs = \mathsf{set}$, a predefined function that returns the set of elements in a list. Infinite sets are, in general, not representable.

As explained in Section 6.1, we will now consider the abstraction function set as the single constructor of type α set for code generation purposes. Arbitrary constants (of an appropriate type) can be turned into datatype constructors in the generated code. We call such constants *pseudo constructors*. Like ordinary constructors, they have no defining code equations, but other code equations can use them in patterns on the left-hand side. There are no particular logical properties that such pseudo constructors have to satisfy—they do not have to be injective or exhaust the abstract type. We have to only instruct the code generator to view the function $\mathsf{set}_{\alpha\ \mathsf{list}\to\alpha\ \mathsf{set}}$ as a pseudo constructor.

In the generated code, the type α set will become a datatype whose elements are, in fact, lists, but wrapped up in the constructor set. For the primitive set operations, we can easily prove alternative equations that pattern-match on set. Here are some examples:

$$\begin{array}{lcl}
\{\} & = & \mathsf{set}\ [] \\
\mathsf{Set.insert}\quad x\ (\mathsf{set}\ xs) & = & \mathsf{set}\ (\mathsf{List.insert}\ x\ xs) \\
\mathsf{Set.remove}\ x\ (\mathsf{set}\ xs) & = & \mathsf{set}\ (\mathsf{List.removeAll}\ x\ xs)
\end{array}$$

If we register these theorems as code equations, the code generator will use them instead of the original definition of the function involved. Thus if we want to execute Set.insert 4 (Set.insert 2 $\{\}$), we obtain the following evaluation by using the above-stated code equations:

$$\begin{array}{r}
\mathsf{Set.insert}\ 4\ (\mathsf{Set.insert}\ 2\ \{\}) \rightsquigarrow \\
\mathsf{Set.insert}\ 4\ (\mathsf{Set.insert}\ 2\ (\mathsf{set}\ [])) \rightsquigarrow \\
\mathsf{Set.insert}\ 4\ (\mathsf{set}\ (\mathsf{List.insert}\ 2\ [])) \rightsquigarrow \\
\mathsf{Set.insert}\ 4\ (\mathsf{set}\ [2]) \rightsquigarrow \\
\mathsf{set}\ [4, 2]
\end{array}$$

The function set is a datatype constructor and thus the term set $[4, 2]$ is already in normal form and represents the set $\{2, 4\}$.

This technique, which we call *standard method*, allows the replacement of one type by another type with surprising ease, based purely on the equational semantics of the code generator. The description shows here why the refinement can happen only on a per type constructor basis: when we instruct the code generator to see $Abs : C \to A$ as a pseudo constructor, it produces the datatype definition

$$\texttt{datatype}\ \ A = Abs\ C$$

in the code and therefore A must be a type of the form $\overline{\alpha}\ \kappa$. Moreover, we require that for every $f \in \Sigma$ of the type $\tau_1 \to \cdots \to \tau_n \to \tau$ (where τ is not a function type) it holds that all τ_i are basic. Otherwise, we could not use pattern matching in the code equation for f.

6.2.2 Adding Invariants

Implementing sets by lists with possibly repeated elements is inefficient. Therefore we now impose the invariant that all elements of the representing lists are distinct and call such lists *distinct lists*. The situation is again the one in Figure 6 on page 116 with $C = \alpha$ list, $A = \alpha$ set, $Abs =$ set, but

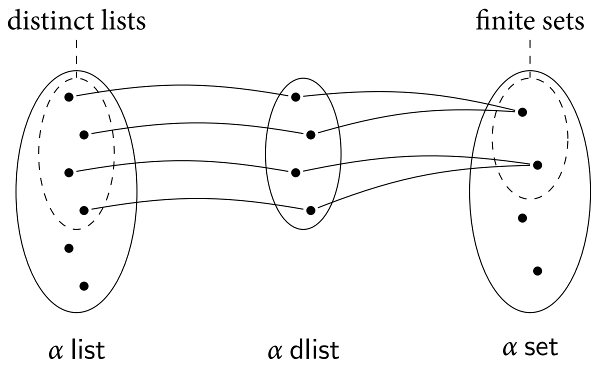


Figure 7: Sets by distinct lists using α dlist

now inv = distinct, which is a predefined function that tests if all elements of a list are distinct.

But now there is the problem that the function set can also be applied to lists that are not distinct. As a consequence, some equations for the primitive set operations only hold conditionally, for example

$$\mathsf{distinct}\ xs \longrightarrow \mathsf{Set.remove}\ x\ (\mathsf{set}\ xs) = \mathsf{set}\ (\mathsf{List.remove1}\ x\ xs).$$

This conditional theorem will be rejected as a code equation by the code generator. For soundness reasons, the precondition cannot simply be dropped, but without it, the theorem does not hold because List.remove1 removes at most one occurrence of x from xs and not all of them like List.removeAll. Our solution is to introduce an intermediate type α dlist for distinct lists (see Figure 7). Thus we split the implementation into two steps: the new *subtype* step from α list to α dlist, where α dlist is a new type that is isomorphic to a subset of α list, the distinct lists, followed by the basic data refinement of α set by α dlist, which does not involve an invariant anymore and can be dealt with by the standard method.

The new subtype with an invariant is defined by `typedef`:

$$\mathtt{typedef}\ \ \alpha\ \mathsf{dlist} = \{xs_{\alpha\ \mathsf{list}} \mid \mathsf{distinct}\ xs\}$$

We assume that the command produced the coercions

$$\begin{aligned} &\mathsf{list} \quad : \alpha\ \mathsf{dlist} \to \alpha\ \mathsf{list}\ \text{and} \\ &\mathsf{Dlist} : \alpha\ \mathsf{list} \to \alpha\ \mathsf{dlist}, \end{aligned}$$

which were defined by the following axiomatization, which is equivalent to the standard one presented in Section 2.4.2:

$$\mathsf{Dlist}\ (\mathsf{list}\ dxs) = dxs \tag{71}$$

$$\mathsf{distinct}\ xs \longleftrightarrow \mathsf{list}\ (\mathsf{Dlist}\ xs) = xs \tag{72}$$

Using the two coercions, we can define all primitive operations on dlist by using the corresponding operations on list, as we already learned in Chapter 5. For example, this is the definition of Dlist.remove : $\alpha \to \alpha$ dlist $\to \alpha$ dlist:

$$\text{Dlist.remove } x \ dxs = \text{Dlist } (\text{List.remove1 } x \ (\text{list } dxs)) \tag{73}$$

Then we bridge the gap between α set and α dlist by a new pseudo constructor dset : α dlist $\to \alpha$ set defined as

$$\text{dset } dxs = \text{set } (\text{list } dxs).$$

If we assume that we already have all primitive operations on α dlist together with the necessary properties, it is again straightforward to prove code equations implementing set operations, for example for Set.remove:

$$\text{Set.remove } x \ (\text{dset } xs) = \text{dset } (\text{Dlist.remove } x \ xs)$$

Therefore what is left is the question how to implement dlist operations by list operations. Using dlist as a pseudo constructor as in the standard method runs into the same problem as before:

$$\text{distinct } xs \longrightarrow \text{Dlist.remove } x \ (\text{Dlist } xs) = \text{Dlist } (\text{List.remove1 } x \ xs)$$

is only provable under the assumption distinct xs. Therefore we try the definition of Dlist.remove (73) itself as a code equation. Now we need to execute the function list on the rhs and face the same problem:

$$\text{list } (\text{Dlist } xs) = xs \tag{74}$$

is only provable for distinct xs. Therefore we extend the code generator for this special case as follows. We register the theorem (71) in the code generator, which instructs the code generator to make Dlist a pseudo constructor and to turn the composition around and make (74) a code equation, although it is not a theorem.

The justification is a meta-theoretic one: we ensure that in code equations, Dlist is only applied to distinct lists, for which (74) is provable. This property of Dlist will be guaranteed by a check that Dlist is only applied to the result of operations on lists that have been proved to preserve the invariant. We guarantee this property by restricting the user to registering only a theorem called a *code certificate* but not the actual code equation. For Dlist.remove the code certificate looks as follows:

$$\text{list } (\text{Dlist.remove } x \ dxs) = \text{List.remove1 } x \ (\text{list } dxs) \tag{75}$$

The code generator checks that the rhs does not contain the pseudo constructor Dlist and derives the actual code equation (73) from it (this is a direct consequence of (71)). The point of the code certificate is that it entails preservation of the invariant. Indeed: when we use the code equation (73), it is true that we introduce a term where Dlist is applied to List.remove1 x (list dxs), but since

$$\begin{aligned}
&\text{list } (\text{Dlist } (\text{List.remove1 } x \ (\text{list } dxs)) \\
&= \text{list } (\text{Dlist.remove } x \ (\text{Dlist } (\text{list } dxs))) && \text{by (73)} \\
&= \text{List.remove1 } x \ (\text{list } (\text{Dlist } (\text{list } dxs)) && \text{by (75)} \\
&= \text{List.remove1 } x \ (\text{list } dxs) && \text{by (71),}
\end{aligned}$$

we can conclude distinct (List.remove1 x (list dxs)) by (72).[1]

This concludes the presentation of code generation for dlist. We will straightforwardly generalize this description to a formal correctness proof.

Let us consider functions $Abs : C \to A$ and $Rep : A \to C$ such that $Abs\ (Rep\ y) = y$ and $inv : C \to \text{bool}$ such that $inv\ x \longleftrightarrow Rep\ (Abs\ x) = x$. We assume that the result type of all functions in Σ is basic.

The format for the code certificate is now

$$\psi\ (f\ \overline{y}) \ = \ t \tag{76}$$

where ψ is Rep (if $\tau = (\dots)\kappa$) or the identity (otherwise). The free variables of t must be contained in $\overline{y}$. The code generator turns this into $f\ \overline{y} = \varphi\ t$ (by a proof step), where φ is Abs (if $\tau = (\dots)\kappa$) or the identity (otherwise). The only liberty that the code generator takes is that it turns the theorem $Abs\ (Rep\ y) = y$ into the nontheorem $Rep\ (Abs\ x) = x$. Of course the latter is implied by $inv\ x$, and we will show that $inv\ s$ holds for all terms $Abs\ s$ that may arise during a computation. But this requires a careful proof (see below). The following table summarizes the behavior of the code generator.

E	E'
$Rep\ (f\ \overline{y}) = t$	$f\ \overline{y} = Abs\ t$
$Abs\ (Rep\ y) = y$	$Rep\ (Abs\ x) = x$

Let E be the set of all code equations at the point when the code generator is invoked and let E' be the result of the translation shown in the table above. That is, most equations are moved from E to E' unchanged, but $Rep\ (f\ \overline{y}) = t$ and $Abs\ (Rep\ y) = y$ are translated as above. Moreover, the code generator enforces that Abs must not occur on the rhs of any equation in E. (This is not a restriction because if one really needed an operation that behaved like Abs one could define it separately from Abs to avoid confusion.)

1 This is not a coincidence. It can be shown that (75) and the theorem distinct $xs \longrightarrow$ distinct(List.remove1 x xs) are equivalent if we use the properties of list and Dlist (71) and (72) and the definition of Dlist.remove (73).

In [29] correctness of the code generation process is shown by interpreting code equations as higher-order rewrite rules and proving that code generation preserves the reduction behavior. Our translation from E to E' is a first step that happens before the steps considered in [29]. We will now prove correctness of that first step by relating the equational theory of E (written $E \vdash u = v$) with reduction in E'. Notation $E' \vdash u \rightarrow v$ means that there is a rewrite step from u to v using either a rule from E' or β-reduction.

We call a term t *invariant* iff (i) $E \vdash \mathit{Rep}\ (\mathit{Abs}\ s) = s$ for all subterms $(\mathit{Abs}\ s)$ of t and (ii) every occurrence of *Abs* in t is applied to an argument.

Lemma 6.2.1. If u is invariant and $E' \vdash u \rightarrow^* v$, then v is invariant.

Proof. By induction on the length of the reduction sequence. In each step, we need to check invariance of newly created *Abs* terms. Because user-provided code equations with *Abs* on the rhs are forbidden, only the derived code equation $f\ \overline{y} = \mathit{Abs}\ t$ can introduce a new *Abs* term, namely $\mathit{Abs}\ t$ itself, where *Abs* is applied and for which we have $E \vdash \mathit{Rep}\ (\mathit{Abs}\ t) = \mathit{Rep}\ (\mathit{Abs}\ (\mathit{Rep}\ (f\ \overline{y}))) = \mathit{Rep}\ (f\ \overline{y}) = t$. Invariance is preserved by β-reduction because it cannot create new *Abs* terms because all *Abs* must already be applied to arguments. □

Lemma 6.2.2. If u is invariant and $E' \vdash u \rightarrow^* v$, then $E \vdash u = v$.

Proof. By induction on the length of the reduction sequence and by the previous lemma. □

Thus we know that if we start with an invariant term, reduction with E' only produces equations that are already provable in E. Invariance of the initial term is enforced by Isabelle very easily: the initial term must not contain *Abs*.

6.2.3 Using Transfer and Lifting

I will demonstrate how we can automate the construction of the intermediate type from the previous section by the Transfer and Lifting tools.

First of all, we have to set up the lifting infrastructure, which is done by a theorem generated by `typedef` for α dlist:

`setup_lifting` type-definition-dlist

This typical boilerplate command already registers α dlist as an abstract datatype with a constructor Dlist in the code generator.

Then every operation is lifted by `lift_definition` command, e.g.:

`lift_definition` remove : $\alpha \rightarrow \alpha$ dlist $\rightarrow \alpha$ dlist `is` List.remove1

After we prove the respectfulness theorem

$$\forall x_\alpha\ xs_{\alpha\ \mathsf{list}}.\ \mathsf{distinct}\ xs \longrightarrow \mathsf{distinct}\ (\mathsf{List.remove1}\ x\ xs)$$

the command produces the obligatory definition (equivalent to (73)) and a transfer rule. More importantly, we automatically obtain the following representation function (coercion) equation (see Section 5.7 on page 111):

$$\mathsf{list}\ (\mathsf{Dlist.remove}\ x\ dxs) = \mathsf{List.remove1}\ x\ (\mathsf{list}\ dxs),$$

which is exactly the code certificate (75). The Lifting tool automatically registers the representation function equation as the code certificate.

If the return type of an abstract function is basic, the representation function equation provided by the Lifting tool has exactly the form (76), which is required by the code generator. Thus except for proving that an operation on the concrete level preserves the invariant—and this is in general unavoidable—everything else is fully automatic.

Let me note here that if we want to create a total quotient type with executable operations, we can use the abstract function equations as code equations by employing the standard method with *Abs* as a constructor.

6.3 data refinement for type expressions

In this section, we will tackle the limitation of the code generator that the type that is being refined has to be of the form $\overline{\alpha}\,\kappa$. The type of *maps* $\alpha \to \beta$ option does not have this form, yet one would still like to refine it by some efficient type of tables. Thus we introduce a new type (α, β) mapping that is a copy of $\alpha \to \beta$ option. It can be refined further, for example, by red-black trees using the techniques from Section 6.2. (See Figure 8 on the facing page for the complete picture.) The implementation type rbt-impl is just a plain datatype of binary trees with a color in each node; on top of it, the subtype rbt of well-shaped trees satisfying the invariant of red-black trees is defined. This example represents the most general form of data refinement discussed in this chapter.

But now all definitions using $\alpha \to \beta$ option must be lifted to the new type (α, β) mapping. Of course, this has to be done for primitive operations on maps such as a lookup or an update only once for all. But it also has to be done for all other definitions using these primitive functions. The reason is that one has to provide for such derived operations new code equations that use primitive operations of (α, β) mapping and not $\alpha \to \beta$ option. On the other hand, no code equations have to be provided for the primitive operations on (α, β) mapping in this phase because these will be provided later on in the phase described in Section 6.2. Of course, it is also possible to base the formalization on the lifted type

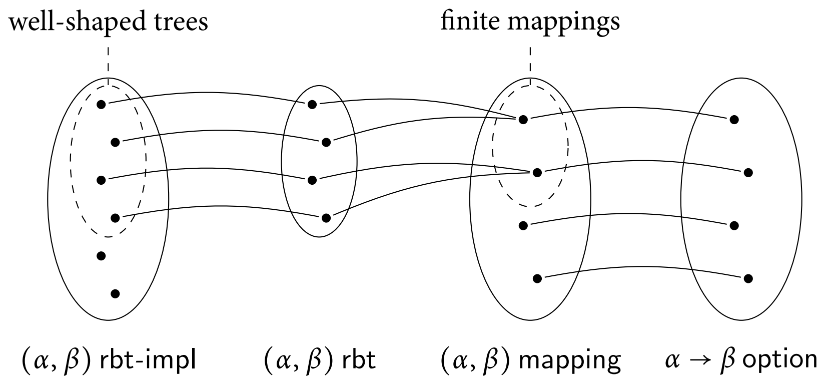

well-shaped trees
finite mappings
(α, β) rbt-impl
(α, β) rbt
(α, β) mapping
$\alpha \to \beta$ option

Figure 8: Maps implemented by red-black trees

(α, β) mapping from the beginning but this contradicts the very idea of data refinement.

The complications of this general setting are as follows: For a start, you do not obtain code for f but for some f'. Moreover you have to refine every function f to some f', not just the primitive ones, and you have to look carefully at the definition of f' (typically produced by `lift_definition`) and at the abstraction relations involved to convince yourself that f and f' are in the desired relationship. But it is not quite as bad as this. As soon as you define a derived function h where $\alpha \to \beta$ option is no longer present in the type of h but that still uses maps inside its body, you no longer need to lift h to some h', but you still have to prove a code equation for h itself that uses mappings internally. This can be done again by transfer.

Having mentioned `lift_definition` and transfer, I will show how we again use Lifting and Transfer to automate moving the specification from $\alpha \to \beta$ option to (α, β) mapping. First, (α, β) mapping is defined as a copy of $\alpha \to \beta$ option and all primitive operations on maps are lifted:

`typedef` (α, β) mapping = $\mathsf{UNIV}_{(\alpha \to \beta\ \mathsf{option})\ \mathsf{set}}$ $\langle proof \rangle$

`lift_definition` empty : (α, β) mapping `is` $\lambda\_.$ None .
`lift_definition` lookup : (α, β) mapping $\to \alpha \to \beta$ option
  `is` $\lambda m\ k.\ m\ k$.
`lift_definition` update : $\alpha \to \beta \to (\alpha, \beta)$ mapping $\to (\alpha, \beta)$ mapping
  `is` $\lambda k\ v\ m.\ m(k \mapsto v)$.

We showed only 3 such operations here but in reality there are more of them. Notice that we did not have to prove a respectfulness theorem in the `lift_definition` command as this is proved automatically if we work with type copies.

Now let us assume we used maps in our formalization to implement a special data type that behaves like a multiset and the multiplicity of elements is limited. Now we can implement an insert for this data structure that ensures that if the limit is reached, the map is not changed.

```
definition insert-lim : (α → nat option) → α → nat → α → nat option
  where insert-lim m k lim = case m k of
    Some n ⇒ if n < lim then m(k ↦ n + 1) else m
    None ⇒ m(k ↦ 1)
```

We use `lift_definition` to define a copy of insert-lim that operates on the code generation type (α, nat) mapping

```
lift_definition insert-lim′ :
  (α, nat) mapping → α → nat → (α, nat) mapping is insert-lim .
```

Function insert-lim′ is defined in terms of the original function insert-lim with the help of the coercions between maps and mappings. In contrast to the situation in Section 6.2, we cannot use this definition as a code equation because it goes in the wrong direction: it reduces a computation on mappings to maps. The desired code equation for insert-lim′ is proved by transfer from the definition of the original function.

```
lemma insert-lim′ m k lim = case Mapping.lookup m k of
  Some n ⇒ if n < lim then Mapping.update k (n + 1) m else m
  None ⇒ Mapping.update k 1 m
by transfer (fact insert-lim-def)
```

It is inconvenient that one has to write down the lifted code equation even if the proof is trivial thanks to the Transfer and Lifting tools. We could use the transferred attribute to transfer theorems in the other direction, i.e., from the concrete level to the abstract level and thus we would not have to write down the lifted code equation at all. But there is the problem that if we go in this direction, it is not clear which parts of a term should really be transferred. The transferred attribute can eagerly transfer all terms from the $\alpha \to \beta$ option to the (α, β) mapping level according to the transfer rules. But the user might want some subterms to remain maps. This would require some mechanism that allows users to annotate a term and say which parts should not be transferred. This is work in progress. Transferring from (α, β) mapping to $\alpha \to \beta$ option instead is unambiguous: all occurrences of mapping are replaced.

6.4 compound return types

We say that a list *xs* is a sublist of a list *ys* if *xs* can be obtained by removing some elements from *ys*. Let us assume that we defined a function sublists : α list → α list list such that sublists *xs* is equal to a list of all sublists of *xs*. Since every sublist of a distinct list must still be distinct, we can lift sublists to the dlist-level:

$$\texttt{lift\_definition}\ \mathsf{sublists} : \alpha\ \mathsf{dlist} \to \alpha\ \mathsf{dlist\ list}\ \texttt{is}\ \mathsf{List.sublists}$$

I omitted the proof of the respectfulness theorem since we are mostly interested in the representation function equation:

$$\mathsf{map\ list}\ (\mathsf{Dlist.sublists}\ xs) = \mathsf{List.sublists}\ (\mathsf{list}\ xs)$$

This equation cannot be used as a code certificate because it does not comply with the form (76)—the top symbol of the left-hand side is not a single representation function but a compound term map list as the return type α dlist list of Dlist.sublists is not basic.

Not being able to execute functions with nonbasic return types might be particularly annoying since there exists a nonnegligible number of functions that use such return types for example for modeling partiality (α option) or combining several results ($\alpha \times \beta$). We could extend the code generator by the notion of a map function and require some restrictions on the map functions to allow us to extend our syntactic correctness proof as well. This is all, in principle, achievable, but this would enlarge the trusted code base (and the complexity of the proof), which, as I stated in Section 6.1, is not desirable. Instead, we work around the limitation.

How can we execute sublists if it has a nonbasic return type? Notice that the return-type restriction holds only for primitive functions (functions from Σ). My workaround allows us to execute sublists as a derived function (i.e., no data refinement needed) by providing a code equation that executes sublists in terms of some other primitive operations for which the return-type restriction holds.

In order to achieve this, we define an auxiliary type that is isomorphic to α dlist list, the type that breaks the restriction.

$$\texttt{typedef}\ \ \alpha\ \mathsf{dlist\text{-}list} = \{xss_{\alpha\ \mathsf{list\ list}} \mid \mathsf{list\_all\ distinct}\ xss\}$$

Observe the definition—we did not define α dlist-list operationally as a type copy of α dlist list but the two types are still isomorphic.

Now we define another lifted version of List.sublists called sublists′, which is the same as sublists except for the return type:

$$\texttt{lift\_definition}\ \mathsf{sublists}' : \alpha\ \mathsf{dlist} \to \alpha\ \mathsf{dlist\text{-}list}\ \texttt{is}\ \mathsf{List.sublists}$$

The following two theorems are the internal respectfulness theorems for sublists′ and sublists:

$$(=^{\mathsf{on}}\ \mathsf{distinct} \mapsto\!\!> =^{\mathsf{on}}\ (\mathsf{list\_all\ distinct}))\ \mathsf{List.sublists\ List.sublists}$$
$$(=^{\mathsf{on}}\ \mathsf{distinct} \mapsto\!\!> \mathsf{list\_all2}\ (=^{\mathsf{on}}\ \mathsf{distinct}))\ \mathsf{List.sublists\ List.sublists}$$

The two above-stated theorems are equivalent as the following equation

$$\mathsf{list\_all2}\ (=^{\mathsf{on}}\ P) = =^{\mathsf{on}}\ (\mathsf{list\_all}\ P) \tag{77}$$

holds for any P and since we already store this equation in Lifting's infrastructure (see Section 5.4), we can automatically derive the respectfulness theorem for sublists′ from the respectfulness theorem for sublists.

Notice that sublists′ conforms to the return-type restriction and is therefore executable. Thus we reduced our task to the following task: Find a function $Rep_{\alpha\ \mathsf{dlist\text{-}list} \to \alpha\ \mathsf{dlist\ list}}$ such that:

1. sublists dxs = Rep (sublists′ dxs)

2. Rep is executable

It is not a surprise that the function Rep that we use is the bijection between α dlist-list and α dlist list. We define it as a lifted identity:

`lift_definition` Rep : α dlist-list → α dlist list `is` $\mathsf{id}_{\alpha\ \mathsf{list\ list} \to \alpha\ \mathsf{list\ list}}$

To discharge the respectfulness theorem for Rep we again use (77). The function Rep fulfills 1. because we can show by transfer that 1. is equivalent to List.sublists xs = id (List.sublists xs), which is a trivial fact.

The last step is to make Rep executable. We cannot use the representation function equation since Rep does not have a basic return type either. Instead, we provide an alternative code equation. The idea is simple: we recursively destruct the list that is a representation of an α dlist-list value and construct a list of type α dlist list by applying the Dlist constructor to the elements of the destructed list. Ideally we would use lifted pattern matching to implement Rep but this does not work because of the higher-order flavor of pattern matching in HOL[2]. Fortunately, it is computer science folklore that we can rewrite any datatype pattern matching to an equivalent first-order term that is a combination of if-then-else clauses and datatype selectors and discriminators. This form works for

2 The pattern matching is only syntax sugar in HOL and is realized by functions called case combinators; e.g., $\mathsf{case}_{\mathsf{list}} : \alpha \to (\beta \to \beta\ \mathsf{list} \to \alpha) \to \beta\ \mathsf{list} \to \alpha$. If we lift this combinator to the type $\alpha \to (\beta\ \mathsf{dlist} \to \beta\ \mathsf{dlist\text{-}list} \to \alpha) \to \beta\ \mathsf{dlist\text{-}list} \to \alpha$, we obtain in the representation function equation abstraction functions for dlist and dlist-list (as the types are in negative position), which is not allowed by the code generator.

us. Let us assume we already obtained the lifted version of selectors and discriminators for α dlist-list from respective functions on α list:[3]

$$\begin{aligned}\mathsf{head}' &: \alpha\ \mathsf{dlist\text{-}list} \to \alpha\ \mathsf{dlist}\\ \mathsf{tail}' &: \alpha\ \mathsf{dlist\text{-}list} \to \alpha\ \mathsf{dlist\text{-}list}\\ \mathsf{isNil}' &: \alpha\ \mathsf{dlist\text{-}list} \to \mathsf{bool}\end{aligned}$$

Having that, we provide this code equation for Rep

$$\mathsf{Rep}\ dxs = \mathsf{if\ isNil}'\ dxs\ \mathsf{then}\ [\,]\ \mathsf{else}\ (\mathsf{head}'\ dxs)\ \#\ (\mathsf{Rep}\ (\mathsf{tail}'\ dxs)),$$

which can be proved by employing transfer and proving the equivalent formula, which is a basic property of selectors, discriminators and constructors on α list:

$$\mathsf{id}\ xs = \mathsf{if\ isNil}\ xs\ \mathsf{then}\ [\,]\ \mathsf{else}\ (\mathsf{head}\ xs)\ \#\ (\mathsf{id}\ (\mathsf{tail}\ xs))$$

Since head′, tail′ and isNil′ conform to the return-type restriction, Rep is executable. The following reduction sequence should give us an idea how Rep reduces an α dlist-list value to an α dlist list value (Dlist-list is the constructor for α dlist-list):

$$\begin{aligned}&\mathsf{Rep}\ (\mathsf{Dlist\text{-}list}\ [xs_1\ \ldots, xs_n]) =\\ &\mathsf{Rep}\ (\mathsf{Dlist\text{-}list}\ (xs_1\ \#\ldots\#\ xs_n\ \#\ [\,])) \rightsquigarrow\\ &\mathsf{Dlist}\ xs_1\ \#\ \mathsf{Rep}\ (\mathsf{Dlist\text{-}list}\ (xs_2\ \#\ldots\#\ xs_n\ \#\ [\,])) \rightsquigarrow\\ &\mathsf{Dlist}\ xs_1\ \#\ \mathsf{Dlist}\ xs_2\ \#\ \mathsf{Rep}\ (\mathsf{Dlist\text{-}list}\ (xs_3\ \#\ldots\#\ xs_n\ \#\ [\,])) \rightsquigarrow\\ &\ldots \rightsquigarrow\\ &\mathsf{Dlist}\ xs_1\ \#\ldots\#\ \mathsf{Dlist}\ xs_n\ \#\ \mathsf{Rep}\ (\mathsf{Dlist\text{-}list}\ [\,]) \rightsquigarrow\\ &\mathsf{Dlist}\ xs_1\ \#\ldots\#\ \mathsf{Dlist}\ xs_n\ \#\ [\,] =\\ &[\mathsf{Dlist}\ xs_1, \ldots, \mathsf{Dlist}\ xs_n]\end{aligned}$$

This ends the presentation for sublists and its return type α dlist list. Since I did the presentation for the case where the outer nonabstract type is a recursive datatype with two constructors (the list type), it is not hard to work out the general algorithm from the concrete example. I will only sketch the general case and point to particularities that did not arise in the example.

First, let me define more precisely which return types are supported. The construction works for any τ that is inductively defined as follows:

- τ is a type variable.

3 The respectfulness theorem for a discriminator is always proved automatically by the monotonicity prover. The selectors must be totalized (what is head [ ]?) by selected values such that the invariant holds for them and thus the respectfulness theorems are provable. The totalization is done automatically in a principled way.

- $\tau = \tau_1 \dots \tau_n\ \kappa$, where κ is an abstract type constructor and $\tau_1 \dots \tau_n$ do not contain abstract type constructors, i.e., nesting of abstract types is not allowed. For example, int dlist is allowed whereas int dlist dlist not.

- $\tau = \tau_1 \dots \tau_n\ \kappa$, where κ is a type constructor defined as a (co)datatype whose constructors do not have the function type or non-free datatypes as arguments. Therefore the datatype α rose-tree = RTree α (α rose-tree list) is allowed whereas the datatype hfset = HFset (hfset fset) not since there are no destructors and constructors for fset.

Thus we work not only with return types of the shape $A\ \kappa$, where A is an abstract type and $\alpha\ \kappa$ is a datatype, but also with the datatypes that are nested, which is something that we did not consider in the introductory example. Let us execute a function f whose return type has the following form $A\ \kappa_1\ \dots\ \kappa_n$, where $\kappa_1 \dots \kappa_n$ are unary (co)datatype type constructors. We again define an auxiliary type

$$\texttt{typedef}\ A\text{-}\kappa_1\text{-}\dots\text{-}\kappa_n = \{x \mid (\mathsf{pred}_{\kappa_n} \circ \dots \circ \mathsf{pred}_{\kappa_1})\ \mathit{inv}\ x\},$$

where pred_{κ_i} are respective predicators for κ_i (we define them for any natural functor, see Section 4.7) and inv the defining predicate for A. We define f' as before and use n theorems like (77) for every κ_i to prove the respectfulness theorem for f'. The main difference happens when we define selectors for $A\text{-}\kappa_1\text{-}\dots\text{-}\kappa_n$. Let us say that the datatype $\alpha\ \kappa_n$ was defined as $\alpha\ \kappa_n = \mathsf{C}_1\ \alpha \mid \mathsf{C}_2\ (\alpha\ \kappa_n\ \tau)$. Thus we define two selectors:

$$\begin{aligned}\pi_1 &: A\text{-}\kappa_1\text{-}\dots\text{-}\kappa_n \to A\ \kappa_1 \dots \kappa_{n-1}\\ \pi_2 &: A\text{-}\kappa_1\text{-}\dots\text{-}\kappa_n \to A\text{-}\kappa_1\text{-}\dots\text{-}\kappa_n\ \tau\end{aligned}$$

In both cases, the functions do not conform to the return-type restriction. This is the place where we run our construction recursively on these two functions. I will skip other details and just state that this recursion is always well founded (we recurse on type expressions, not on values of these types). The rest of the construction follows the introductory example analogously—definition of Rep, its code equation and the code equation f = Rep f'. The whole construction is fully automatic and does not require any additional input from the user.

The last thing that is worth mentioning is that we store the auxiliary types and their associated Rep functions such that we can reuse them when we encounter the same return type but for a different function.

Future work comprises loosening restrictions on the supported types in the described workaround: we want to support nesting of abstract types.

6.5 related work

Data refinement is a perennial topic that was first considered by Hoare more than 40 years ago [36], who already introduced abstraction functions and invariants. This principle of data refinement became an integral part of the model oriented specification language VDM [47] (and was later generalized to nondeterministic operations [35, 71]). In the first-order context of universal algebra it was shown that there are always fully abstract models such that any concrete implementation can be shown correct with a homomorphism [70].

The standard method for data refinement was already presented in Haftmann and Nipkow's paper about the code generator [29] in the form of an example. The whole infrastructure (including the treatment of invariants) had been used for a few years before it was properly published [28].

As described in Section 6.2, all occurrences of the type being refined (e.g., α set) are refined by the same type (α list). However, our infrastructure does not, by itself, enforce this. Lochbihler [60] developed an extension that builds on top of our infrastructure to support multiple representations. He exploits the fact that there can be multiple pseudo constructors for any type and organizes them by means of type classes. In fact, Isabelle's default refinement of sets supports cofinite sets by using a second pseudo constructor coset : α list $\to$ α set, where coset xs = −set xs ("−" is the set complement here).

In Chapter 4, I already mentioned Lammich's work [56] for data refinement in Isabelle/HOL and the similar work of Cohen et al. [20] in Coq. Lammich's approach has some similarity with Section 6.3: you do not obtain code for f but for some f' that is in a certain relationship to f. As a result, he can work with a more general notion of refinement supporting (for example) nondeterministic operations and multiple implementations of the same type. A difference is that his system proves invariance preservation for derived functions as an explicit theorem by using means of a procedure similar to our transfer_prover whereas we define a new abstract type and let the type checker do the work. In a nutshell, his is a general framework for heavy duty data refinement, ours is a lightweight infrastructure for completely transparent but more limited data refinement.

ACL2 supports data refinement, too. In the Mu-calculus case study [49], Manolios shows how to implement sets by lists (using a congruence on lists) while Greve *et al.* [27] explain how to deal with invariants. The details are rather different from our work because ACL2 is untyped.

In Coq, besides the already mentioned approach of Cohen et al. [20], one can use the older approach of parametrized modules [22]: perform the development inside the context of a specification of finite sets (or whatever abstract type you have), and later instantiate the module with some implementation of finite sets that has been proved to satisfy the finite set axioms. The drawback is that you do not really work with the actual abstract type (e.g., sets), but some axiomatization of it, which may not have the same nice syntax and proof support.

Always do what you are afraid to do.
— Ralph Waldo Emerson (1841)

7 USE CASE: FROM TYPES TO SETS

HOL types are naturally interpreted as nonempty sets—this intuition is reflected in the type definition rule for the HOL-based systems (including Isabelle/HOL), where a new type can be defined whenever a nonempty set is given. However, in HOL this definition mechanism cannot be applied inside proof contexts. In this chapter, I propose a more expressive type-definition rule that addresses this. The new expressive power opens the opportunity for a translation tool that relativizes type-based statements to more flexible set-based variants. This tool is an interesting use case of the Transfer tool.

This is joint work with Andrei Popescu and this chapter is partly based on our workshop paper [55].

7.1 motivation

Let us recall the motivational example from Chapter 1. Let $\mathsf{P} : \alpha\ \mathsf{list} \to \mathsf{bool}$ be a fixed constant (whose definition is not important here). We will consider the following statements, where we extend the syntax introduced in Section 2.2 and explicitly quantify over types at the outermost level:

$$\forall \alpha.\ \exists xs_{\alpha\ \mathsf{list}}.\ \mathsf{P}\ xs \tag{78}$$

$$\forall \alpha.\ \forall A_{\alpha\ \mathsf{set}}.\ A \neq \varnothing \longrightarrow (\exists xs \in \mathsf{lists}\ A.\ \mathsf{P}\ xs) \tag{79}$$

The formula (79) is a relativized form of (78), quantifying not only over all types α, but also over all their nonempty subsets A, and correspondingly relativizing the quantification over all lists to quantification over the lists from A. Therefore we call statements as (78) *type based* whereas statements as (79) *set based*.

Type-based theorems have advantages over the set-based ones. First, they are more concise (compare (78) and (79)). Second, the HOL types encode properties more implicitly and are therefore more rigid compared to explicitly expressed sets (i.e., formulas) and therefore proof automation also usually works better for them. Overall, it is much easier to reason about type-based statements than set-based ones. On the other hand, the set-based theorems provide more flexibility since they allow us to reason about mathematical structures that are defined only on a subset of the whole type. For example, a function $f_{\tau\to\sigma}$ can be injective only on some $A_{\tau\ \mathsf{set}}$ or be a measure only on a subset. We are forced to come up with workarounds if we want to use theorems from a type-based library of injective functions or measures in such cases. Thus, while it is much easier to reason about type-based statements such as (78), the set-based statements such as (79) are more general and easier to apply.

In the most striking cases, the relativization was created manually. For example, the constant $\mathsf{inj\text{-}on}\ A\ f = (\forall x\ y \in A.\ f\ x = f\ y \longrightarrow x = y)$ together with a small library about functions being injective only on a set. Often the users want to convert whole type-based libraries to set-based ones. For example, my colleague Immler writes about his formalization experience in his master thesis [43, §5.7]: *The main reason why we had to introduce this new type* [of finite maps] *is that almost all topological properties are formalized in terms of type classes, i.e., all assumptions have to hold on the whole type universe. It feels like a cleaner approach* [would be] *to relax all necessary topological definitions and results from types to sets because other applications might profit from that, too.*

In different HOL-based systems, there are numerous theories [4, 16, 18, 37, 62] that are developed as set based altogether from beginning (usually formalizations of algebraic structures where the carrier is explicit) or that require such set-based theories. This is of course an alternative if one is willing to pay the price.

Ideally—and this is what I propose here—the users would develop their theories in a type-based fashion, and then export the main theorems as set-based statements. Unfortunately, the HOL systems currently do not allow for this. From the set-theoretic semantics point of view, the statements (78) and (79) are equivalent. However, from a theorem proving perspective, they are quite different— assuming that (78) is a theorem, one cannot prove (79). Indeed, in a proof attempt of (79), one would fix a nonempty set A and, to invoke (78), one would need to define a new type corresponding to A, an action not currently allowed inside a proof context.

The problem of types vs sets bites even stronger when it comes to tool writing: for example, the new (co)datatype tool [12] maintains a notion of a bounded natural functor, which in the unary case is a type constructor $\alpha\ \mathsf{F}$ together with a functorial map function $\mathsf{Fmap} : (\alpha \to \beta) \to \alpha\ \mathsf{F} \to \beta\ \mathsf{F}$. For technical reasons, some key facts proved by the tool (e.g., those involving algebras and coalgebras for F) require the more flexible set-based variant—this is done via an internalization of F to sets, $\mathsf{Fin} : \alpha\ \mathsf{set} \to \alpha\ \mathsf{F}\ \mathsf{set}$.[1] The development would be dramatically simplified if one could focus on the type-based counterparts for the intermediate lemmas, and only export the set-based version of the main results at the end.

1 Fin is to F what lists is to list.

7.2 proposal of a logic extension: local typedef

To address the above, we propose extending the HOL logic with a new rule for type definitions with the following properties:

- It enables type definitions to be emulated inside proofs while avoiding the introduction of dependent types by a simple syntactic check.
- It is natural and sound w.r.t. our ground semantics (Section 3.4.4) as well as the standard HOL semantics à la Pitts [81].

Let us recall that the current Isabelle/HOL type definition mechanism, introduced in Section 2.4.2 on page 19, introduces three new axioms that postulate that the newly defined type is isomorphic to the given set A via mappings *Abs* and *Rep*. Let the notation ${}_{\alpha}(A \approx \beta)^{Abs}_{Rep}$ denote the formula that is a conjunction of these three axioms and thus states that $A_{\alpha\ \mathsf{set}}$ is isomorphic to β, which is witnessed by the two mappings *Abs* and *Rep*. Thus, when the user issues a command `typedef` $\tau = S_{\sigma\ \mathsf{set}}$, as we know, the system introduces a new type τ and two constants $\mathsf{Abs}^{\tau} : \sigma \to \tau$ and $\mathsf{Rep}^{\tau} : \tau \to \sigma$ and adds the following axiom into the theory: $S \neq \varnothing \longrightarrow {}_{\sigma}(S \approx \tau)^{\mathsf{Abs}^{\tau}}_{\mathsf{Rep}^{\tau}}$. The user is required to discharge the goal $S \neq \varnothing$, after which the definitional theorem ${}_{\sigma}(S \approx \tau)^{\mathsf{Abs}^{\tau}}_{\mathsf{Rep}^{\tau}}$ is inferred.

Before we proceed to carry out a formal proof that the new rule is sound, we will start with a less formal description to motivate the formulation of the rule and understand the intuition behind it. Let us take a purely semantic perspective and ignore the rank-1 polymorphism for a minute. Then the principle behind type definitions simply states that for all types α and nonempty subsets A of them, there exists a type β isomorphic to A:

$$\forall \alpha.\ \forall A_{\alpha\ \mathsf{set}}.\ A \neq \varnothing \longrightarrow \exists \beta.\ \exists Abs_{\alpha \to \beta}\ Rep_{\beta \to \alpha}.\ {}_{\alpha}(A \approx \beta)^{Abs}_{Rep} \qquad (\star)$$

The typedef mechanism can be regarded as the result of applying a sequence of standard rules for connectives and quantifiers to $(\star)$ in a more expressive logic (notationally, we use Gentzen's sequent calculus):

1. Left $\forall$ rule of α and A with given type σ and term $S_{\sigma\ \mathsf{set}}$ (both provided by the user), and left implication rule:

$$\dfrac{\dfrac{\Gamma \vdash S \neq \varnothing \qquad \Gamma, \exists\beta\ Abs\ Rep.\ {}_{\sigma}(S \approx \beta)^{Abs}_{Rep} \vdash \varphi}{\Gamma, (\star) \vdash \varphi}\ \forall_L, \forall_L, \longrightarrow_L}{\Gamma \vdash \varphi}\ \text{Cut of } (\star)$$

2. Left $\exists$ rule for β, *Abs* and *Rep*, introducing some new/fresh type τ, and functions Abs^{τ} and Rep^{τ}:

$$\dfrac{\Gamma \vdash S \neq \varnothing \qquad \dfrac{\Gamma,\ {}_\sigma(S \approx \tau)^{\mathsf{Abs}^\tau}_{\mathsf{Rep}^\tau} \vdash \varphi}{\Gamma,\ \exists\beta\ Abs\ Rep.\ {}_\sigma(S \approx \beta)^{Abs}_{Rep} \vdash \varphi}\ \exists_L, \exists_L, \exists_L}{\dfrac{\Gamma, (\star) \vdash \varphi}{\Gamma \vdash \varphi}\ \text{Cut of } (\star)}\ \forall_L, \forall_L, \longrightarrow_L$$

The user further discharges $\Gamma \vdash S \neq \varnothing$, and therefore the overall effect of this chain is the sound addition of ${}_\sigma(S \approx \tau)^{\mathsf{Abs}^\tau}_{\mathsf{Rep}^\tau}$ as an extra assumption when trying to prove an arbitrary fact φ.

What we propose is to use a variant of the above (with fewer instantiations) as an actual rule:

- In step 1. we do not ask the user to provide concrete σ and $S_{\sigma\ \mathsf{set}}$, but work with a type σ and a term $A_{\sigma\ \mathsf{set}}$ that can contain type and term *variables*.

- In step 2., we only apply the left $\exists$ rule to the type β and introduce a fresh type *variable* β

We obtain:

$$\dfrac{\Gamma \vdash A \neq \varnothing \qquad \dfrac{\Gamma,\ \exists Abs\ Rep.\ {}_\sigma(A \approx \beta)^{Abs}_{Rep} \vdash \varphi}{\Gamma,\ \exists\beta\ Abs\ Rep.\ {}_\sigma(A \approx \beta)^{Abs}_{Rep} \vdash \varphi}\ [\beta \text{ fresh}]\ \exists_L}{\dfrac{\Gamma, (\star) \vdash \varphi}{\Gamma \vdash \varphi}\ \text{Cut of } (\star)}\ \forall_L, \forall_L, \longrightarrow_L$$

In a notation closer to Isabelle/HOL, the overall rule, written (LT) as in "Local Typedef", looks as follows:

$$\dfrac{\Gamma \vdash A \neq \varnothing \qquad \Gamma \vdash (\exists Abs\ Rep.\ {}_\sigma(A \approx \beta)^{Abs}_{Rep}) \longrightarrow \varphi}{\Gamma \vdash \varphi}\ [\beta \notin A, \varphi, \Gamma]\ (\mathrm{LT})$$

This rule allows us to locally assume that there is a type β isomorphic to an arbitrary nonempty set A. The syntactic check $\beta \notin A, \varphi, \Gamma$ prevents an introduction of a dependent type (since A can contain term variables in general).

The above discussion merely shows that (LT) is morally correct and more importantly *natural* in the sense that it is an instance of a more general principle, namely the rule $(\star)$.

Now we will proceed to a formal proof of (LT)'s soundness. Let us fix a signature Σ and a well-formed definitional theory D over Σ for the rest of this section.

Theorem 11. Any deduction consisting of the deduction rules of Isabelle/HOL and the (LT) rule is sound.

Proof. The overall structure of the proof is as follows:

- We construct a model $\mathcal{I}$ of the theory D according to Theorem 3 on page 42.
- We extend Σ and $\mathcal{I}$ into Σ' and $\mathcal{I}'$ such that the interpretation of ground types in $\mathcal{I}'$ is closed under subsets.
- We show that (LT) preserves $\mathcal{I}'$.

Let us construct the extended signature $\Sigma' \supseteq \Sigma$ and the interpretation $\mathcal{I}'$ of Σ' such that $\mathcal{I}'$ extends $\mathcal{I}$, written $\mathcal{I} \leq \mathcal{I}'$.[2] Moreover, we require that the interpretation of ground types in $\mathcal{I}'$ is closed under subsets, which is formally defined as follows: for every $\theta : \mathsf{TVar} \to \mathsf{GType}_{\Sigma'}$, $\mathcal{I}'$-compatible valuation ξ and a term $A : \sigma\ \mathsf{set}$ it holds that

$$[\theta(A)]^{\mathcal{I}'}(\xi) \neq \emptyset \longrightarrow \exists \tau \in \mathsf{GType}_{\Sigma'}.\ [\tau]^{\mathcal{I}'} = [\theta(A)]^{\mathcal{I}'}(\xi). \qquad (80)$$

Let Σ_0 denote Σ and $\mathcal{I}_0$ denote $\mathcal{I}$. We proceed as follows: for every ground type $\tau \in \mathsf{GType}_{\Sigma_0}$, and for every nonempty set $S \subseteq [\tau]^{\mathcal{I}_0}$ we extend Σ_0 with a fresh type constructor τ^S without type parameters and call the extended signature Σ_1. We will define Σ_1's model $\mathcal{I}_1$ such that $\mathcal{I}_0 \leq \mathcal{I}_1$. We start with $\mathcal{I}_0$ and interpret the new types by sets that produced them in the first place: $[\tau^S]^{\mathcal{I}_1} := S$. Moreover, we have to define interpretation of the other newly introduced ground items in $\mathsf{GType}_{\Sigma_1}$ and $\mathsf{GCInst}_{\Sigma_1}$, namely ground types and ground constant instances that contain τ^S for some S in their types. We will interpret them by using the same well-founded recursion principle as in the proof of Theorem 3, where we constructed a model for a well-formed definitional theory. Since $\mathsf{TV}(u) \supseteq \mathsf{TV}(v)$ for every definition $u \equiv v$, any ground item depending on a newly introduced ground item must be new as well. Thus defining new items in $\mathcal{I}_1$ does not change the interpretation of the old ones. To sum up, $\mathcal{I}_1$ is a model of $\Sigma_1 \supseteq \Sigma_0$ and $\mathcal{I}_0 \leq \mathcal{I}_1$.

We can see the above-described construction as a single step in an iterative process (i.e., replace 0 by i and 1 by $i+1$) and create $\Sigma_2, \Sigma_3, \ldots$ and their models $\mathcal{I}_2, \mathcal{I}_3, \ldots$ such that $\Sigma_i \subseteq \Sigma_{i+1}$ and $\mathcal{I}_i \leq \mathcal{I}_{i+1}$. Let us take a limit of this process and define $\Sigma' = \bigcup_{i=0}^{\infty} \Sigma_i$ and $\mathcal{I}' = \bigcup_{i=0}^{\infty} \mathcal{I}_i$. The limits Σ' and $\mathcal{I}'$ are extensions of Σ and $\mathcal{I}$ (i.e., $\Sigma \subseteq \Sigma'$ and $\mathcal{I} \leq \mathcal{I}'$), and $\mathcal{I}'$ is a model of Σ'. Moreover, Σ' and $\mathcal{I}'$ have the desired closure property (80). Let $A : \sigma\ \mathsf{set}$ and $\theta(\sigma) = \sigma'$. Because our semantics guarantees that $[\theta(A)]^{\mathcal{I}'}(\xi) : [\sigma'\ \mathsf{set}]^{\mathcal{I}'} = \mathcal{P}([\sigma']^{\mathcal{I}'})$ and $[\theta(A)]^{\mathcal{I}'}(\xi)$ is a nonempty set,

2 If $\mathcal{I}_1$ is an interpretation of Σ_1 and $\mathcal{I}_2$ is an interpretation of Σ_2 such that $\Sigma_1 \subseteq \Sigma_2$, we define $\mathcal{I}_1 \leq \mathcal{I}_2$ to mean $[u]^{\mathcal{I}_1} = [u]^{\mathcal{I}_2}$ for all $u \in \mathsf{GType}^{\bullet}_{\Sigma_1} \cup \mathsf{GCInst}^{\bullet}_{\Sigma_1}$.

there must exist $\tau \in \mathsf{GType}^{\Sigma'}$ such that $[\tau]^{\mathcal{I}'} = [\theta(A)]^{\mathcal{I}'}(\xi)$ due to the construction of $\mathcal{I}'$.

As $\mathcal{I}'$ extends $\mathcal{I}$, $\mathcal{I}'$ is a model of D. Theorem 2 guarantees that $\mathcal{I}'$ is preserved by the deduction rules of Isabelle/HOL. Now we prove that $\mathcal{I}'$ is preserved by the (LT) rule as well.

Let us assume that $\mathcal{I}'$ is a model of the assumptions of (LT), i.e.,

$$\begin{aligned}&\mathcal{I}' \models (\Gamma, A \neq \varnothing) \text{ and}\\&\mathcal{I}' \models (\Gamma, (\exists Abs\ Rep.\, {}_{\sigma}(A \approx \beta)^{Abs}_{Rep}) \longrightarrow \varphi).\end{aligned}$$

Let us fix ground substitution $\theta : \mathsf{TVar} \to \mathsf{GType}^{\Sigma'}$ and $\mathcal{I}'$-compatible valuation ξ such that $[\theta(\psi)]^{\mathcal{I}'}(\xi) = \mathsf{true}$ for all $\psi \in \Gamma$. Then we obtain:

$$[\theta(A \neq \varnothing)]^{\mathcal{I}'}(\xi) = \mathsf{true} \tag{81}$$

$$[\theta((\exists Abs\ Rep.\, {}_{\sigma}(A \approx \beta)^{Abs}_{Rep}) \longrightarrow \varphi)]^{\mathcal{I}'}(\xi) = \mathsf{true} \tag{82}$$

From (81) and the definition of our semantics we obtain

$$[\theta(A)]^{\mathcal{I}'}(\xi) \neq \varnothing. \tag{83}$$

Using semantics of $\longrightarrow$ and the fact $\beta \notin A, \varphi, \Gamma$, we can derive from (82)

$$\begin{aligned}&(\exists \tau \in \mathsf{GType}^{\Sigma'}.\, [\theta[\tau/\beta](\exists Abs\ Rep.\, {}_{\sigma}(A \approx \beta)^{Abs}_{Rep})]^{\mathcal{I}'}(\xi) = \mathsf{true})\\&\quad\longrightarrow [\theta(\varphi)]^{\mathcal{I}'}(\xi) = \mathsf{true}.\end{aligned} \tag{84}$$

The antecedent of (84) is true: using the closure property of $\mathcal{I}'$ (80) (together with (83)), we obtain $\tau \in \mathsf{GType}_{\Sigma'}$ such that $[\tau]^{\mathcal{I}'} = [\theta(A)]^{\mathcal{I}'}(\xi)$. The ground type τ is the witness for the existential quantifier in (84) since

$$[\exists Abs\ Rep.\, {}_{\theta(\sigma)}(\theta(A) \approx \tau)^{Abs}_{Rep})]^{\mathcal{I}'}(\xi) = \mathsf{true}$$

means that $[\theta(A)]^{\mathcal{I}'}(\xi)$ must be isomorphic to itself due to the interpretation of τ (recall $[\tau]^{\mathcal{I}'} = [\theta(A)]^{\mathcal{I}'}(\xi)$).

Thus $[\theta(\varphi)]^{\mathcal{I}'}(\xi) = \mathsf{true}$ and therefore $\mathcal{I}' \models (\Gamma, \varphi)$. As Σ' and $\mathcal{I}'$ are extensions of Σ and $\mathcal{I}$, and Γ and φ use only symbols from Σ, we obtain $\mathcal{I} \models (\Gamma, \varphi)$. To conclude, we cannot prove False by using the theory D together with the deduction rules of Isabelle/HOL and the (LT) rule. $\square$

The (LT) rule is also sound with respect to the rules of the original HOL by Gordon and to its standard semantics by Pitts [81]. In this case, the proof is easier because the construction of Σ' and $\mathcal{I}'$ with the closure property is not necessary. As we saw in Section 3.4.3 on page 36, in the standard semantics, we interpret type variables by sets directly by a type valuation $\theta : \mathsf{TVar} \to \mathcal{U}$ and do not go through the intermediate step of ground types. But then the analog of the closure property (80) in Pitts's semantics is a trivial property:

$$\forall \theta.\ \forall \xi.\ [A](\theta)(\xi) \neq \varnothing \longrightarrow \exists B \in \mathcal{U}.\ B = [A](\theta)(\xi)$$

The extra work of building Σ' and $\mathcal{I}'$ is not a coincidence. In Section 3.4.3, we moved the border between syntax and semantics to adapt the semantics to the syntactic nature of overloading to make the explanation of overloading more intuitive. Since the nature of the (LT) rule is more semantic, we can observe that the semantic part is intruding into syntax—we had to extend D's signature by names for any set that could be an interpretation of a type. But this complication is much smaller than alternatively using the standard semantics for overloading.

7.3 translation algorithm

The rule (LT) allows us to handle the motivating example from Section 7.1. We assume (78) is a theorem, and wish to prove (79). We fix α and $A_{\alpha\ \mathsf{set}}$ and assume $A \neq \varnothing$. Applying (LT), we obtain a type β (represented by a fresh type variable) such that $\exists Abs\ Rep.\ {}_{\alpha}(A \approx \beta)^{Abs}_{Rep}$, from which we obtain Abs and Rep such that ${}_{\alpha}(A \approx \beta)^{\mathsf{Abs}}_{\mathsf{Rep}}$. From this, (78) with α instantiated to β, and the definition of lists, we obtain

$$\exists xs_{\beta\ \mathsf{list}} \in \mathsf{lists}\ (\mathsf{UNIV}_{\beta\ \mathsf{set}}).\ \mathsf{P}_{\beta\ \mathsf{list}\to\mathsf{bool}}\ xs.$$

Furthermore, using that Abs and Rep are isomorphisms between $A_{\alpha\ \mathsf{set}}$ and $\mathsf{UNIV}_{\beta\ \mathsf{set}}$, we obtain

$$\exists xs_{\alpha\ \mathsf{list}} \in \mathsf{lists}\ A_{\alpha\ \mathsf{set}}.\ \mathsf{P}_{\alpha\ \mathsf{list}\to\mathsf{bool}}\ xs,$$

as desired.[3]

The above relativization process would be really useful only if automated. This is a perfect application for the Transfer tool. As we learned in Chapter 4, Transfer can both automatically synthesize the relativized statement (e.g., (79) from above) and prove it from the original statement (e.g., (78) from above).

We will consider a general case now and use Transfer to automate the relativization process. Let us start with a type-based theorem

$$\forall \alpha.\ \varphi[\alpha], \tag{85}$$

where $\varphi[\alpha]$ is a formula containing α. We fix α and $A_{\alpha\ \mathsf{set}}$, assume $A \neq \varnothing$ and fix a fresh type variable β. We also assume $\exists Abs\ Rep.\ {}_{\alpha}(A \approx \beta)^{Abs}_{Rep}$, from which we obtain Abs and Rep such that

$${}_{\alpha}(A \approx \beta)^{\mathsf{Abs}}_{\mathsf{Rep}}. \tag{86}$$

3 We silently assume parametricity of the quantifier $\exists$ and P.

We use the Lifting setup infrastructure (see Section 5.3 on page 104) together with (86) to create setup for Transfer—most importantly a transfer relation $\mathsf{T} : \alpha \to \beta \to \mathsf{bool}$ is defined. Since β is a subtype of α, the relation T is bi-unique and right-total (see Table 1 on page 112).

The following theorem, (85) with α instantiated to β,

$$\varphi[\beta] \tag{87}$$

is the input for Transfer. We will transfer the theorem along the relation T and thus replace the type β by the related type α. Let us assume that we have appropriate transfer rules for this translation step. Roughly speaking, this means that for every constant $c_{\tau[\beta]}$[4] from φ, there exists a transfer rule that can relate $c_{\tau[\beta]}$ to some $c^{\mathsf{on}}_{\tau[\alpha]}$ using T. As we saw in Chapter 4, we do not store such specific rules. We rather obtain them as instances of more general parametricity rules. Such a rule for c looks in general as follows:

$$\begin{aligned}&\forall \alpha\ \beta.\ \forall T_{\alpha\to\beta\to\mathsf{bool}}.\ \mathsf{bi\_unique}\ T \longrightarrow \mathsf{right\_total}\ T \longrightarrow\\ &\quad \mathcal{R}[T]\ c^{\mathsf{on}}_{\tau[\alpha]}\ c_{\tau[\beta]}\end{aligned} \tag{88}$$

By instantiating α, β and T for α, β, and T and discharging the side conditions (as T is bi-unique and right-total) we obtain the desired relation between c and its relativized version c^{on}. All of this happens automatically during the transfer algorithm.

Let us consider three concrete examples of rules of the form (88). The transfer rule for = (43)

$$\begin{aligned}&\forall \alpha\ \beta.\ \forall T_{\alpha\to\beta\to\mathsf{bool}}.\ \mathsf{bi\_unique}\ T \longrightarrow\\ &\quad (T \Mapsto T \Mapsto =)\ (=_{\alpha\to\alpha\to\mathsf{bool}})\ (=_{\beta\to\beta\to\mathsf{bool}})\end{aligned}$$

transfers = to its relativized version, which is only a different type instance of =, whereas the transfer rule for universal quantification (46)

$$\begin{aligned}&\forall \alpha\ \beta.\ \forall T_{\alpha\to\beta\to\mathsf{bool}}.\ \mathsf{right\_total}\ T \longrightarrow\\ &\quad ((T \Mapsto =) \Mapsto =)\ (\mathsf{Ball}\ \{x_\alpha \mid \mathsf{Domp}\ T\ x\})\ \mathsf{All}_{(\beta\to\mathsf{bool})\to\mathsf{bool}}\end{aligned}$$

replaces All by a different constant, namely the bounded quantification Ball. The term $\{x_\alpha \mid \mathsf{Domp}\ T\ x\}$ gets automatically replaced by the domain of T in the transfer algorithm (see the description in Section 4.4 on page 80); in our settings $\{x_\alpha \mid \mathsf{Domp}\ \mathsf{T}\ x\} = A$. Thus, using the above rule, we can replace "$\forall x_\beta$" with "$\forall x_\alpha \in A_{\alpha\ \mathsf{set}}$". The last example is the rule for the predicate $\mathsf{inj}_{(\alpha\to\beta)\to\mathsf{bool}}$, whose relativization is $\mathsf{inj\text{-}on}_{\alpha\ \mathsf{set}\to(\alpha\to\beta)\to\mathsf{bool}}$:

$$\begin{aligned}&\forall \alpha\ \beta.\ \forall T_{\alpha\to\beta\to\mathsf{bool}}.\ \mathsf{bi\_unique}\ T \longrightarrow \mathsf{right\_total}\ T \longrightarrow\\ &\quad ((T \Mapsto =) \Mapsto =)\ (\mathsf{inj\text{-}on}\ \{x_\alpha \mid \mathsf{Domp}\ T\ x\})\ \mathsf{inj}\end{aligned}$$

4 The expression $\tau[\beta]$ denotes a type expression containing β.

Overall, Transfer will use rules of the form (88) to replace all cs in φ by c^{on}s and thus proves the equivalence

$$\varphi[\beta] \longleftrightarrow \varphi^{\mathsf{on}}[\alpha, A_{\alpha\,\mathsf{set}}],$$

where $\varphi^{\mathsf{on}}[\alpha, A_{\alpha\,\mathsf{set}}]$ is the synthesized relativization of $\varphi[\beta]$. We discharge the left-hand side by (87) and obtain $\varphi^{\mathsf{on}}[\alpha, A_{\alpha\,\mathsf{set}}]$ as a theorem.

Since $\varphi^{\mathsf{on}}[\alpha, A_{\alpha\,\mathsf{set}}]$ does not contain β, we can use the (LT) rule to remove the assumption (86) ${}_{\alpha}(A \approx \beta)^{\mathsf{Abs}}_{\mathsf{Rep}}$ and to obtain the final result:

$$\forall \alpha.\ \forall A_{\alpha\,\mathsf{set}}.\ A \neq \varnothing \longrightarrow \varphi^{\mathsf{on}}[\alpha, A]$$

This theorem is the set-based version of $\forall \alpha.\ \varphi[\alpha]$. Except for the parametricity rules of the form (88), which are usually a part of a library, no further input is required and the whole process runs automatically.

7.4 further extensions

There are however Isabelle-specific complications in the isomorphic journey between types and sets: axiomatic type classes and overloading. I will explain in this section how these two features are in conflict with the algorithm described above and how to circumvent these complications.

7.4.1 Local Type Classes

The first complication is the implicit assumptions on types given by the axiomatic type classes. We have not dealt with axiomatic type classes in this thesis yet. I will briefly explain the essence of this mechanism. We can annotate type variables with type classes, which are essentially predicates on types. For example, α_{finite} means that α can be instantiated only with a type that we proved to fulfill the conditions of the type class finite, namely that the type must contain finitely many elements.

For example, let us modify (85) to speak about types of class finite:

$$\forall \alpha_{\mathsf{finite}}.\ \varphi[\alpha_{\mathsf{finite}}] \tag{89}$$

In the translation algorithm we instantiated α with the type variable β to obtain the formula (87). This step breaks the algorithm from the previous section since β is not a member of the type class finite. Even if we assumed finite $\mathsf{UNIV}_{\beta\,\mathsf{set}}$, we could not make a type variable β locally a member of finite. This operation is not allowed in Isabelle.

In order to relativize (89), we first need a version of it which internalizes the type class assumption

$$\forall \alpha.\ \mathsf{finite}(\alpha) \longrightarrow \varphi[\alpha], \tag{90}$$

where $\mathsf{finite}(\alpha)$ is a term of type bool, which is true if and only if α is a finite type.[5] Assuming $\mathsf{finite}(\beta)$ allows us to instantiate α by β and thus execute the original algorithm. We obtain the set-based version of (89)

$$\forall \alpha.\ \forall A_{\alpha\ \mathsf{set}}.\ \mathsf{finite}\ A \longrightarrow A \neq \emptyset \longrightarrow \varphi^{\mathsf{on}}[\alpha, A]$$

since the relativization of $\mathsf{finite}(\beta)$ is $\mathsf{finite}\ A$.

The internalization of type classes (inferring (90) from (89)) is already supported by the kernel of Isabelle—thus no further work is required from us. The rule for internalization of type classes is a result of the work by Haftmann and Wenzel [30, 99].

7.4.2 Local Overloading

Having addressed implicit assumptions on types given by axiomatic type classes, the only hurdle that could prevent us from relativizing a formula $\forall \alpha.\ \varphi[\alpha]$ is if a parametricity rule à la (88) could not be proved for some c in φ. This might happen if c were an overloaded constant, which will explained be later. But where do the overloaded constants come from?

The mechanism of type classes in Isabelle gets really useful when we are allowed to associate operations with a type class to achieve Haskell-like type classes. This is technically achieved by using an overloaded constant (implicitly representing the associated operation) in the definition of the corresponding axiomatic type class. In this respect, the type class finite from the previous section is somewhat special since there are no operations associated with it, i.e., we did not use any overloaded constant in its definition.

Therefore we use semigroups as the running example in this section since semigroups require an associated operation—multiplication. First, let us notice how a general specification of a semigroup would look. It would contain a nonempty set $A_{\alpha\ \mathsf{set}}$, a binary operation $f_{\alpha\to\alpha\to\alpha}$ such that A is closed under f, and a proof of the specific property of semigroups that f is associative on A. We capture the last property by the predicate

$$\mathsf{semigroup}^{\mathsf{on}}_{\mathsf{with}}\ A\ f = (\forall x\ y\ z \in A.\ f\ (f\ x\ y)\ z = f\ x\ (f\ y\ z)),$$

which we read along the paradigm: *a structure* on *the set A* with *operations* $f_1, \ldots, f_n$.

The realization of semigroups by a type class in Isabelle is somewhat more specific. The type σ can belong to the type class semigroup if $\mathsf{semigroup}(\sigma)$ is provable, where

$$\mathsf{semigroup}(\alpha)\ \text{iff}\ \forall x_\alpha\ y_\alpha\ z_\alpha.\ (x * y) * z = x * (y * z). \qquad (91)$$

5 This is Wenzel's approach [99] to represent axiomatic type classes by internalizing them as predicates on types, i.e., constants of type $\forall \alpha.$ bool. As this particular type is not allowed in Isabelle, Wenzel uses instead α itself $\to$ bool, where α itself is a singleton type.

Notice that the associated multiplication operation is represented by the *global* overloaded constant $*_{\alpha\to\alpha\to\alpha}$, which will cause the complication as already mentioned.

Let us relativize $\forall\alpha_{\mathsf{semigroup}}.\ \varphi[\alpha_{\mathsf{semigroup}}]$ now. We fix a nonempty set A, a binary f such that A is closed under f and assume $\mathsf{semigroup}^{\mathsf{on}}_{\mathsf{with}}\ A\ f$. As before, we locally define β to be isomorphic to A and obtain the respective isomorphisms Abs and Rep.

Having defined β, we want to prove that β belongs into semigroup. Using the approach from the previous section, this goal translates into proving $\mathsf{semigroup}(\beta)$, which requires that the overloaded constant $*_{\beta\to\beta\to\beta}$ used in the definition of semigroup (see (91)) must be isomorphic to f on A. In other words, we have to locally define $*_{\beta\to\beta\to\beta}$ to be a projection of f onto β, i.e., $x_\beta * y_\beta$ must equal $\mathsf{Abs}(f\ (\mathsf{Rep}\ x)\ (\mathsf{Rep}\ y))$. Although we can locally "define" a new constant (fix a fresh term variable c and assume $c = t$), we cannot overload the global symbol $*$ locally for β. This is not supported by Isabelle. In other words, there is no hope that we could relate $*_{\beta\to\beta\to\beta}$ and $f_{\alpha\to\alpha\to\alpha}$ by a transfer rule.

Since the global overloaded constants is the offending problem, we will cope with the complication by compiling out the overloaded constant $*$ from

$$\forall\alpha.\ \mathsf{semigroup}(\alpha) \longrightarrow \varphi[\alpha] \tag{92}$$

by the dictionary construction as follows: whenever $c = \ldots * \ldots$ (i.e., c was defined in terms of $*$ and thus depends implicitly on the overloaded meaning of $*$), define $c_{\mathsf{with}}\ f = \ldots f \ldots$ and use it instead of c. The parameter f plays a role of the dictionary here: whenever we want to use c_{with}, we have to explicitly specify how to perform multiplication in c_{with} by instantiating f. That is to say, the implicit meaning of $*$ in c was made explicit by f in c_{with}. Using this approach, we obtain:

$$\forall\alpha.\ \forall f_{\alpha\to\alpha\to\alpha}.\ \mathsf{semigroup}_{\mathsf{with}}\ f \longrightarrow \varphi_{\mathsf{with}}[\alpha, f], \tag{93}$$

where $\mathsf{semigroup}_{\mathsf{with}}\ f_{\alpha\to\alpha\to\alpha} = (\forall x_\alpha\ y_\alpha\ z_\alpha.\ f\ (f\ x\ y)\ z = f\ x\ (f\ y\ z))$ and similarly for φ_{with}. For now, we assume that (93) is a theorem and look at how it helps us to finish the relativization and later we will explain how to derive (93) as a theorem.

Given (93), we will instantiate α with β and obtain

$$\forall f_{\beta\to\beta\to\beta}.\ \mathsf{semigroup}_{\mathsf{with}}\ f \longrightarrow \varphi_{\mathsf{with}}[\beta, f].$$

Recall that the quantification over all functions of type $\beta \to \beta \to \beta$ is isomorphic to the bounded quantification over all functions of type $\alpha \to \alpha \to \alpha$ under which $A_{\alpha\ \mathsf{set}}$ is closed.[6] The difference after compiling

6 We are talking about the transfer rule (46), which we discussed in Section 7.3. This time we quantify over functions of type $\beta \to \beta \to \beta$ and therefore the bound is "all functions of type $\alpha \to \alpha \to \alpha$ closed under A".

out the overloaded constant $*$ is that now we are isomorphically relating two bounded (local) variables from the quantification and not a global constant $*$ to a local variable.

Thus we reduced the relativization once again to the original algorithm and can obtain the set-based version

$$\forall\alpha.\ \forall A_{\alpha\ \mathsf{set}}.\ A \neq \varnothing \longrightarrow \forall f_{\alpha\to\alpha\to\alpha}.\ (\forall x_\alpha\ y_\alpha \in A.\ f\ x\ y \in A) \longrightarrow \\ \mathsf{semigroup}^{\mathsf{on}}_{\mathsf{with}}\ A\ f \longrightarrow \varphi^{\mathsf{on}}_{\mathsf{with}}[\alpha, A, f].$$

Let us get back to the dictionary construction. Its detailed description can be found, for example, in the paper by Krauss and Schropp [50]. We will outline the process only informally here. Our task is to compile out an overloaded constant $*$ from a term s. As a first step, we transform s into $s_{\mathsf{with}}[*/f]$ such that $s = s_{\mathsf{with}}[*/f]$ and such that unfolding the definitions of all constants in s_{with} does not yield $*$ as a subterm. We proceed for every constant c in s as follows: if c has no definition, we do not do anything. If c was defined as $c = t$, we first apply the construction recursively on t and obtain t_{with} such that $t = t_{\mathsf{with}}[*/f]$; thus $c = t_{\mathsf{with}}[*/f]$. Now we define a new constant $c_{\mathsf{with}}\ f = t_{\mathsf{with}}$. As $c_{\mathsf{with}}\ * = c$, we replace c in s by $c_{\mathsf{with}}\ *$. At the end, we obtain $s = s_{\mathsf{with}}[*/f]$ as a theorem. Notice that this procedure produces s_{with} that does not semantically depends on $*$ only if there is no type in s that depends on $*$.

Thus the above-described step applied to (92) produces

$$\forall\alpha.\ \mathsf{semigroup}_{\mathsf{with}}\ *_{\alpha\to\alpha\to\alpha} \longrightarrow \\ \varphi_{\mathsf{with}}[\alpha, f_{\alpha\to\alpha\to\alpha}][*_{\alpha\to\alpha\to\alpha}/f_{\alpha\to\alpha\to\alpha}]. \tag{94}$$

To finish the dictionary construction, we replace every occurrence of $*_{\alpha\to\alpha\to\alpha}$ by a universally quantified variable $f_{\alpha\to\alpha\to\alpha}$ and obtain (93). This derivation step is not currently allowed in Isabelle.

Now we will formulate the extension of the logic in the form of a rule that allows us to perform the derivation of (93) from (94). First, let us recall that $\rightsquigarrow^{\downarrow +}$ is the transitive and substitutive closure of the constant/type dependency relation $\rightsquigarrow$ from Section 3.4.1. Additionally, we define Δ_c as the set of all types for which c was overloaded, i.e., $\Delta_c = \{\sigma \mid \text{there is a definition } c_\sigma \equiv v \text{ for some } v\}$. We say that a type or a constant instance u is in φ, written $u \in \varphi$, if $u \in \mathsf{consts}^\bullet(\varphi) \cup \mathsf{types}^\bullet(\varphi)$.

Now we can formulate the Unoverloading rule (UO):

$$\frac{\varphi[c_\sigma/x_\sigma]}{\forall x_\sigma.\ \varphi}\ [\neg(p \rightsquigarrow^{\downarrow +} c_\sigma) \text{ for any } p \in \varphi;\ \sigma \not\leq \Delta_c] \tag{UO}$$

This means that we can replace a constant c_σ by a universally quantified variable x_σ under these two side conditions:

1. All types and constant instances in φ do not semantically depend on c_σ through a chain of constant and type definitions. The constraint is fulfilled by the first step of the dictionary construction unless there is a type depending on c_σ.

2. There is no matching definition for c_σ. In our use case, c_σ is always a type-class operation with its most general type (e.g., $*_{\alpha\to\alpha\to\alpha}$). As we overload a type-class operation only for strictly more specific types (such as $*_{\mathsf{nat}\to\mathsf{nat}\to\mathsf{nat}}$), the condition $\sigma \not\leq \Delta_c$ must be fulfilled.

The proof of the rule's soundness is still an open question.

Conjecture 1. The rule (UO) is sound.

Proof idea. The truth of a formula φ containing c_σ should not be affected by any existing definition of c_τ with $\tau < \sigma$ and therefore, since c_σ is unconstrained (and thus uninterpreted) outside of such τs, it behaves like a term variable x_σ in φ. $\square$

In the next section, I will summarize how the extensions for type classes and overloading work together with the algorithm from Section 7.3.

7.4.3 General Case

Let us assume that Υ is a type class depending on the overloaded constants $*_1, \ldots, *_n$, written $\overline{*}$. We write $A \downarrow \overline{f}$ to mean that A is closed under operations $f_1, \ldots, f_n$.

The following derivation tree shows how we derive from the type-based theorem $\vdash \forall \alpha_\Upsilon.\ \varphi[\alpha_\Upsilon]$ (the topmost formula in the tree) its set-based version (the bottommost formula). Explanation of the derivation steps follows after the tree.

$$
\begin{array}{cl}
\vdash \forall \alpha_\Upsilon.\ \varphi[\alpha_\Upsilon] & \\ \hline
\vdash \forall \alpha.\ \Upsilon(\alpha) \longrightarrow \varphi[\alpha] & (1) \\ \hline
\vdash \forall \alpha.\ \Upsilon_{\mathsf{with}}\ \overline{*}[\alpha] \longrightarrow \varphi_{\mathsf{with}}[\alpha, \overline{f}][\overline{*}/\overline{f}] & (2) \\ \hline
\vdash \forall \alpha.\ \forall \overline{f}[\alpha].\ \Upsilon_{\mathsf{with}}\ \overline{f} \longrightarrow \varphi_{\mathsf{with}}[\alpha, \overline{f}] & (3) \\ \hline
A_{\alpha\ \mathsf{set}} \neq \varnothing,\ {}_{\alpha}(A \approx \beta)^{Abs}_{Rep} \vdash \forall \alpha.\ \forall \overline{f}[\alpha].\ \Upsilon_{\mathsf{with}}\ \overline{f} \longrightarrow \varphi_{\mathsf{with}}[\alpha, \overline{f}] & (4) \\ \hline
A_{\alpha\ \mathsf{set}} \neq \varnothing,\ {}_{\alpha}(A \approx \beta)^{Abs}_{Rep} \vdash \forall \overline{f}[\beta].\ \Upsilon_{\mathsf{with}}\ \overline{f} \longrightarrow \varphi_{\mathsf{with}}[\beta, \overline{f}] & (5) \\ \hline
\begin{array}{l} A_{\alpha\ \mathsf{set}} \neq \varnothing, \\ {}_{\alpha}(A \approx \beta)^{Abs}_{Rep} \end{array} \vdash \forall \overline{f}[\alpha].\ A \downarrow \overline{f} \longrightarrow \Upsilon^{\mathsf{on}}_{\mathsf{with}}\ A\ \overline{f} \longrightarrow \varphi^{\mathsf{on}}_{\mathsf{with}}[\alpha, A, \overline{f}] & (6) \\ \hline
A_{\alpha\ \mathsf{set}} \neq \varnothing \vdash \forall \overline{f}[\alpha].\ A \downarrow \overline{f} \longrightarrow \Upsilon^{\mathsf{on}}_{\mathsf{with}}\ A\ \overline{f} \longrightarrow \varphi^{\mathsf{on}}_{\mathsf{with}}[\alpha, A, \overline{f}] & (7) \\ \hline
\begin{array}{l} \vdash \forall \alpha.\ \forall A_{\alpha\ \mathsf{set}}. \\ \qquad A \neq \varnothing \longrightarrow \forall \overline{f}[\alpha].\ A \downarrow \overline{f} \longrightarrow \Upsilon^{\mathsf{on}}_{\mathsf{with}}\ A\ \overline{f} \longrightarrow \varphi^{\mathsf{on}}_{\mathsf{with}}[\alpha, A, \overline{f}] \end{array} & (8)
\end{array}
$$

Derivation steps:

(1) The class internalization from Section 7.4.1.

(2) The first step of the dictionary construction from Section 7.4.2.

(3) The Unoverloading rule (UO) from Section 7.4.2.

(4) We fix fresh α, $A_{\alpha\,\mathsf{set}}$, β, $Abs_{\alpha\to\beta}$ and $Rep_{\beta\to\alpha}$. We assume that A is nonempty and that β is isomorphic to A.

(5) We instantiate α in the conclusion with β.

(6) Relativization along the isomorphism between β and A by the Transfer tool—see Section 7.3.

(7) Since Abs and Rep are present only in ${}_{\alpha}(A \approx \beta)^{Abs}_{Rep}$, we can existentially quantify over them and replace the hypothesis with $\exists Abs\ Rep.\ {}_{\alpha}(A \approx \beta)^{Abs}_{Rep}$, which we discharge by the Local Typedef rule from Section 7.2, as β is not present elsewhere either (the previous step (6) removed all occurrences of β in the conclusion).

(8) We move all hypotheses into the conclusion and quantify over all fixed variables.

The whole process depends only on the (conditional) parametricity rules used in the step (6). We can provide such rules for a broad class of constants as we are transferring over right-total and bi-unique relations. Such rules are usually part of a library and no additional input is required from the user. The only restriction stems from the step (2). We cannot perform the dictionary construction for types depending on overloaded constants unless we want to compile out such types as well.

7.5 conclusion

We are currently experimenting with a prototype implementation of the new rules and their integration with Lifting and Transfer—the implementation of the (LT) and (UO) rules is already available from my website [69]. A rigorous proof of the soundness of the (UO) rule is future work.

But even without the (UO) rule in place, we can still relativize a large class of formulas: any formula without overloaded constants. We believe that the Local Typedef rule, which is semantically well justified and apparently quite useful, is a good candidate for HOL citizenship.

Notice that with the (UO) rule in place, we can address one of the long-standing user complaints that they are not allowed to provide, for example, two different orders for the same type when using the type class for orders. With our tool, they can still enjoy advantages of type classes and overloading while proving abstract properties about orders and only export the final product as a set-based theorem, which quantifies over all possible orders. According to the motto: *Prove easily and be still flexible.*

The question: is this good news or bad news? And the answer is: Yes!
— Barry Schwartz (2005)

8 CONCLUSION

Interactive theorem provers have always aimed to make proving less error prone and less demanding. This thesis provides an advancement on this front. Our work allows for sound and more abstract reasoning and thus makes proving in Isabelle/HOL more trustworthy and easier.

8.1 results

Isabelle/HOL has contained user-defined types since its creation. Our goal was to promote user-defined types as a tool for abstraction.

Our first contribution is to have made the type definitional mechanism consistent with the overloading constant definitions. We defined decidable criteria under which these definitions cannot create a cycle—the offending dependency. We identified the standard HOL semantics [81] as not being suitable for overloading (which has a syntactic notion) and therefore created a novel semantics, which is a natural syntactic-semantic blend. Our model of Isabelle/HOL is the first explanation of the correct interplay of the two definitional mechanisms after a series of works on this topic [30, 50, 75, 99]. This work put Isabelle/HOL onto a firm foundational ground to a relief of its users.

Although the concrete (technical) contribution here is novel (new semantics, first model for typedef with overloading), our achievement seen in a broader context of interactive provers is not revolutionary at all. We are just catching up with a group of provers that have already clarified their foundations [81], in some cases even mechanically [34, 52, 67]. In our eyes, we are settling a debt concerning clarity of Isabelle's logical foundations and trustworthiness of its implementation. Of course, this has good reasons: our logic is more complex and we put more energy into other aspects of Isabelle and created indisputably a system with the highest usability among interactive provers. Let us shift our focus back to the kernel of our system. In this regard, our work is meant to be a beginning not an end product on this front—I will elaborate on this in the future work section.

On the practical side, our main contribution was a development of two tools, Transfer and Lifting, providing support for creating libraries of abstract types.

Transfer is a working implementation of ideas successively developed by Reynolds [83], Wadler [95] and Mitchell [65]. Transfer supports transferring of theorems between any pair of types that can be set in a relation and for which we have transfer rules relating corresponding operations on these types. Various automation procedures ease the burden of proving: the transfer prover can prove parametricity for derived operations and the integration with the (co)datatype tool [12] proves setup theorems for natural functors. Our work confirms that it is useful to treat types as objects with a rich structure rather than sets of elements. Compared with previous solutions based on the rewriting approach [33, 39, 48, 78], Transfer is more general and requires less input from the users—the hallmark of representing related types as relations and using parametricity.

Lifting is an improvement on previous quotient tools [39, 48]. Lifting defines operations on different kinds of abstract types (subtypes and quotients) by using a uniform algorithm. The main contribution is that it is the first quotient tool that does not impose any limitation on the higher-order type of the lifted operations. Our automation can discharge the correctness condition of the definition for type copies and rewrites its statement into a readable form in other cases. The integration with the (co)datatype tool again provides setup theorems. The integration with the code generator animates our methodology for reducing data refinement to code generation.

Adding the layered design into our solution caters for flexibility: Transfer is not limited only to operations defined by Lifting, and Lifting is not limited only to types defined by the command `quotient_type`.

The principles on which Transfer and Lifting are based are not tightly bound to Isabelle/HOL. Thus we consider our results to be of general interest to the whole HOL community. Our detailed description allows for recreating similar tools at the very least in other HOL-based provers.

The Transfer and Lifting tools have become popular tools. One of Isabelle's advanced users gave me an anecdotal evidence [42]: *If I had not been able to use the type of finite sets together with Transfer and Lifting, I would have had to use lists to model finite sets and would have hated myself every day.* An inspection of the Isabelle2015 distribution [44] and the Archive of Formal Proofs [5] reveals around 3500 uses of methods and commands from the Transfer and Lifting tools.

Our interpretation of the above is that the Isabelle users started enjoying abstract types thanks to the invaluable help of our tools. Thus we achieved our goal to make types in Isabelle/HOL a powerful and useful tool for abstraction. Overall, our work means that we can overcome limitations of an elementary type system by building a layer of automation around it.

This brings us to a broader perspective of improving the technology of interactive theorem proving. There have been numerous other works [10, 12, 51, 63, 93, 96–98] that made proving in Isabelle/HOL more scalable and more productive. Thanks to this advancement, the users can and dare to create larger and more demanding formalizations [14, 23, 45, 57, 59, 77, 88, 90], which in return opens new challenges. This is a never-ending process. In this respect, our work is just a small stone put into a much larger mosaic, a small step in a long run.

Let me also mention what our tools cannot do. At the very beginning of my thesis, I praised abstraction as a fruitful way of working. Let us not forget how easily mathematicians' minds can move between different abstractions; sometimes as easy as our eyes can refocus from our laptops to a distant tree. We are still far from this effortless freedom in our theorem provers, although most if not every user would agree that an ideal prover should support such refocusing. Transfer and Lifting still work only on a small scale and do not provide what we could call *reasoning modulo isomorphism*. We are only at the start of using our tools as black boxes to build more complicated tools. My procedure to work around the return-type limitation of the code generator or my proposal to move theorems from types to sets are first swallows that *will* hopefully make a summer.

8.2 future work

Here I propose how to address most of the limitations of our work and suggest interesting follow-up projects.

8.2.1 Foundations and Trustworthiness

conservativity We proved only consistency of Isabelle/HOL. Consistency is a crucial, but rather weak property—a suitable notion of conservativeness (perhaps in the style of Wenzel [99], but covering type definitions as well) is left as future work. Overloading is again the challenge here.

framework for logical extensions Our original plan was to create a general framework that would allow us to prove easily that a certain extension of the logic (for example the Local Typedef rule) constitutes a conservative extension. Our idea was that one would prove that the extension preserves a certain invariant (motivated mainly by overloading) rather than to inspect its interaction with all other rules in the logic. This plan proved to be overambitious as the first step.

mechanical verification To strengthen confidence in our results, we wish to mechanically verify the proofs from Chapter 3 in Isabelle, especially the algorithm from Section 3.5 for deciding well-formedness of a theory. Verification of the current kernel all the way down to the implementation level would be extremely laborious task with the current proving technology due to the kernel's complexity. But we can try to reduce the size of the kernel.

reducing the kernel The advantage of reducing the size of the kernel would be twofold: we make the kernel more trustworthy by reducing the trusted code and we make the potential verification of what is left more feasible. We propose the following steps:

- The `typedef` command in Isabelle/HOL takes as an input a term of type α set. This means that the set type must be introduced before the `typedef` command and therefore axiomatically. After all, we saw this in Section 2.3, where we introduced α set as a type copy of $\alpha \to$ bool by two axioms mem_Collect_eq and Collect_mem_eq. We could change `typedef` to take a term of type $\alpha \to$ bool, as in the original HOL by Gordon [81], and then define α set by using it.

- The kernel of Isabelle contains a powerful rule for higher-order resolution, which runs as a subroutine a higher-order unifier, a complex piece of code. We could compute the unification outside of the kernel and only certify in the kernel that the supplied substitution is a unifier.

- The mechanism of axiomatic type classes is a part of the kernel. Wenzel [99] already showed in 1997 how to compile out this mechanism by reflecting type classes as predicates on types and interpreting affected parts of Isabelle's type system (order-sorted unification and type inference) as a logical reasoning in a simple fragment of the propositional logic. Following Wenzel's proposal is rather an engineering challenge—performance and compatibility.

8.2.2 Transfer and Lifting

limitations of transfer I mentioned two limitations of Transfer in Section 4.9: 1) Under some rare conditions, transfer does not synthesize a goal that would represent the most eager way of transferring. The desired result can be obtained only by backtracking. We want to improve the transfer algorithm such that this does not happen. 2) Some useful conditional transfer rules are not expressible in our current framework. We want to generalize the notion of a transfer rule with side conditions.

lifting restrictions of the code generator In Section 6.4, I presented the algorithm for lifting the return-type restriction of the code generator. The presented solution does not support composition of abstract types. It seems that parametricity (for left-unique relations) of the defining predicate of the outer type suffices. It is an open question if this condition is also necessary. Another topic would be to implement a similar algorithm for functions that do not need invariants. Here a dual restriction is required: the type arguments of a function must be basic.

from types to sets We want to finish the project of translating type-based theorem to set-based theorems (Chapter 7). Besides implementation of the translation algorithm, this requires a proof of soundness of the Unoverloading rule (UO).

reasoning modulo isomorphism I have already mentioned this project in the previous section. Our ideal is to move whole specifications and theories between different types. More speculatively, we could envision a mechanism that would work as follows: if a user wanted to apply some method to a goal, it would be recognized that we need to change a view to an isomorphic view of the goal to make the method succeed.

BIBLIOGRAPHY

[1] A Consistent Foundation for Isabelle/HOL - A Correction Patch. URL: http://www21.in.tum.de/~kuncar/documents/patch.html.

[2] M. Adams. Introducing HOL Zero - (Extended Abstract). In K. Fukuda, J. v. d. Hoeven, M. Joswig, and N. Takayama, editors, *ICMS 2010*. Vol. 6327, in LNCS, pp. 142–143. Springer, 2010.

[3] A. Anand and V. Rahli. Towards a Formally Verified Proof Assistant. In G. Klein and R. Gamboa, editors, *ITP 2014*. Vol. 8558, in LNCS, pp. 27–44. Springer, 2014.

[4] J. Aransay, C. Ballarin, and J. Rubio. A Mechanized Proof of the Basic Perturbation Lemma. *J. autom. reasoning*, 40(4):271–292, 2008.

[5] Archive of Formal Proofs. URL: http://afp.sf.net/.

[6] R. D. Arthan. Some Mathematical Case Studies in ProofPower–HOL. In K. Slind, editor, *TPHOLs 2004 (Emerging Trends)*, in School of Computing, pp. 1–16. University of Utah, 2010.

[7] B. Barras. Coq en Coq. Tech. rep. (3026). INRIA, 1996.

[8] B. Barras. Sets in Coq, Coq in Sets. *J. Formalized Reasoning*, 3(1):29–48, 2010.

[9] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Of *Texts in Theoretical Computer Science. An EATCS Series.* Springer, 2004.

[10] J. C. Blanchette. Automatic Proofs and Refutations for Higher-Order Logic. PhD thesis. Institut für Informatik, Technische Universität München, 2012.

[11] J. C. Blanchette, M. Desharnais, L. Panny, A. Popescu, and D. Traytel. *Defining (Co)datatypes and Primitively (Co)recursive Functions in Isabelle/HOL*, 2015. URL: https://isabelle.in.tum.de/dist/Isabelle2015/doc/datatypes.pdf.

[12] J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel. Truly Modular (Co)datatypes for Isabelle/HOL. In G. Klein and R. Gamboa, editors, *ITP 2014*. Vol. 8558, in LNCS, pp. 93–110. Springer, 2014.

[13] J. C. Blanchette, A. Popescu, and D. Traytel. Foundational extensible corecursion: a proof assistant perspective. In K. Fisher and J. H. Reppy, editors, *ICFP 2015*, pp. 192–204. ACM, 2015.

[14] T. Bourke, R. J. v. Glabbeek, and P. Höfner. A Mechanized Proof of Loop Freedom of the (Untimed) AODV Routing Protocol. In F. Cassez and J. Raskin, editors, *ATVA 2014*. Vol. 8837, in LNCS, pp. 47–63. Springer, 2014.

[15] A. Bove, P. Dybjer, and U. Norell. A Brief Overview of Agda - A Functional Language with Dependent Types. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs 2009*. Vol. 5674, in LNCS, pp. 73–78. Springer, 2009.

[16] H. Chan and M. Norrish. Mechanisation of AKS Algorithm: Part 1 – The Main Theorem. In C. Urban and X. Zhang, editors, *ITP 2015*. Vol. 9236, in LNCS, pp. 117–136. Springer, 2015.

[17] A. Church. A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

[18] A. R. Coble. Formalized Information-Theoretic Proofs of Privacy Using the HOL4 Theorem-Prover. In N. Borisov and I. Goldberg, editors, *PETS 2008*. Vol. 5134, in LNCS, pp. 77–98. Springer, 2008.

[19] C. S. Coen. A Semi-reflexive Tactic for (Sub-)Equational Reasoning. In J. Filliâtre, C. Paulin-Mohring, and B. Werner, editors, *TYPES 2004*. Vol. 3839, in LNCS, pp. 98–114. Springer, 2004.

[20] C. Cohen, M. Dénès, and A. Mörtberg. Refinements for Free! In G. Gonthier and M. Norrish, editors, *CPP 2013*. Vol. 8307, in LNCS, pp. 147–162. Springer, 2013.

[21] M. Dénès. [Coq-Club] Propositional extensionality is inconsistent in Coq. Archived at `https://sympa.inria.fr/sympa/arc/coq-club/2013-12/msg00119.html`. Dec. 12, 2013.

[22] J.-C. Filliâtre and P. Letouzey. Functors for Proofs and Programs. In, *ESOP 2004*. Vol. 2986, in LNCS, pp. 370–384. Springer, 2004.

[23] P. Gammie, A. L. Hosking, and K. Engelhardt. Relaxing safely: verified on-the-fly garbage collection for x86-TSO. In D. Grove and S. Blackburn, editors, *PLDI 2015*, pp. 99–109. ACM, 2015.

[24] M. J. C. Gordon and T. F. Melham, eds. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[25] M. Gordon. From LCF to HOL: a short history. In G. D. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pp. 169–186. The MIT Press, 2000.

[26] A. Grabowski, A. Kornilowicz, and A. Naumowicz. Mizar in a Nutshell. *J. Formalized Reasoning*, 3(2):153–245, 2010.

[27] D. Greve, M. Kaufmann, P. Manolios, J. Moore, S. Ray, J. Ruiz-Reina, R. Sumners, D. Vroon, and M. Wilding. Efficient execution in an automated reasoning environment. *J. Funct. Program.*, 18:15–46, 2008.

[28] F. Haftmann, A. Krauss, O. Kunčar, and T. Nipkow. Data Refinement in Isabelle/HOL. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *ITP 2013*. Vol. 7998, in LNCS, pp. 100–115. Springer, 2013.

[29] F. Haftmann and T. Nipkow. Code Generation via Higher-Order Rewrite Systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *FLOPS 2010*. Vol. 6009, in LNCS, pp. 103–117. Springer, 2010.

[30] F. Haftmann and M. Wenzel. Constructive Type Classes in Isabelle. In T. Altenkirch and C. McBride, editors, *TYPES 2006*. Vol. 4502, in LNCS, pp. 160–174. Springer, 2006.

[31] F. Haftmann and M. Wenzel. Private communication. Oct. 2014.

[32] J. Harrison. HOL Light: A Tutorial Introduction. In M. K. S. and A. J. Camilleri, editors, *FMCAD '96*. Vol. 1166, in LNCS, pp. 265–269. Springer, 1996.

[33] J. Harrison. *Theorem proving with the real numbers*. Of *CPHC/BCS distinguished dissertations*. Springer, 1998.

[34] J. Harrison. Towards Self-verification of HOL Light. In U. Furbach and N. Shankar, editors, *IJCAR 2006*. Vol. 4130, in LNCS, pp. 177–191. Springer, 2006.

[35] J. He, C. Hoare, and J. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, *ESOP '86*. Vol. 213, in LNCS, pp. 187–196. Springer, 1986.

[36] C. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1:271–281, 1972.

[37] J. Hölzl and A. Heller. Three Chapters of Measure Theory in Isabelle/HOL. In M. C. J. D. v. Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *ITP 2011*. Vol. 6898, in LNCS, pp. 135–151. Springer, 2011.

[38] J. Hölzl, F. Immler, and B. Huffman. Type Classes and Filters for Mathematical Analysis in Isabelle/HOL. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *ITP 2013*. Vol. 7998, in LNCS, pp. 279–294. Springer, 2013.

[39] P. V. Homeier. A Design Structure for Higher Order Quotients. In J. Hurd and T. F. Melham, editors, *TPHOLs 2005*. Vol. 3603, in LNCS, pp. 130–146. Springer, 2005.

[40] B. Huffman and O. Kunčar. Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In G. Gonthier and M. Norrish, editors, *CPP 2013*. Vol. 8307, in LNCS, pp. 131–146. Springer, 2013.

[41] B. Huffman and C. Urban. A New Foundation for Nominal Isabelle. In M. Kaufmann and L. C. Paulson, editors, *ITP 2010*. Vol. 6172, in LNCS, pp. 35–50. Springer, 2010.

[42] L. Hupel. Private communication. Nov. 2015.

[43] F. Immler. Generic Construction of Probability Spaces for Paths of Stochastic Processes. MA thesis. Institut für Informatik, Technische Universität München, 2012.

[44] Isabelle2015. URL: http://isabelle.in.tum.de/website-Isabelle2015/.

[45] IsaFoR/CeTA - An Isabelle/HOL Formalization of Rewriting for Certified Termination Analysis. URL: http://cl-informatik.uibk.ac.at/software/ceta/.

[46] Issues of The Cyclicity Checker. URL: http://www21.in.tum.de/~kuncar/documents/issues/.

[47] C. B. Jones. *Software Development. A Rigourous Approach*. Prentice Hall, 1980.

[48] C. Kaliszyk and C. Urban. Quotients revisited for Isabelle/HOL. In W. C. Chu, W. E. Wong, M. J. Palakal, and C. Hung, editors, *SAC 2011*, pp. 1639–1644. ACM, 2011.

[49] M. Kaufmann, P. Manolios, and J. S. More. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.

[50] A. Krauss and A. Schropp. A Mechanized Translation from Higher-Order Logic to Set Theory. In M. Kaufmann and L. C. Paulson, editors, *ITP 2010*. Vol. 6172, in LNCS, pp. 323–338. Springer, 2010.

[51] A. Krauss, C. Sternagel, R. Thiemann, C. Fuhs, and J. Giesl. Termination of Isabelle Functions via Termination of Rewriting. In M. C. J. D. v. Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *ITP 2011*. Vol. 6898, in LNCS, pp. 152–167. Springer, 2011.

[52] R. Kumar, R. Arthan, M. O. Myreen, and S. Owens. HOL with Definitions: Semantics, Soundness, and a Verified Implementation. In G. Klein and R. Gamboa, editors, *ITP 2014*. Vol. 8558, in LNCS, pp. 308–324. Springer, 2014.

[53] O. Kunčar. Correctness of Isabelle's Cyclicity Checker: Implementability of Overloading in Proof Assistants. In X. Leroy and A. Tiu, editors, *CPP 2015*, pp. 85–94. ACM, 2015.

[54] O. Kunčar and A. Popescu. A Consistent Foundation for Isabelle/HOL. In C. Urban and X. Zhang, editors, *ITP 2015*. Vol. 9236, in LNCS, pp. 234–252. Springer, 2015.

[55] O. Kunčar and A. Popescu. From Types to Sets in Isabelle/HOL. In, *Isabelle Workshop 2014*, 2014. URL: http://www21.in.tum.de/~kuncar/documents/kuncar-popescu-itp2014.pdf.

[56] P. Lammich. Automatic Data Refinement. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *ITP 2013*. Vol. 7998, in LNCS, pp. 84–99. Springer, 2013.

[57] P. Lammich and A. Lochbihler. The Isabelle Collections Framework. In M. Kaufmann and L. C. Paulson, editors, *ITP 2010*. Vol. 6172, in LNCS, pp. 339–354. Springer, 2010.

[58] K. R. M. Leino and M. Moskal. Co-induction Simply - Automatic Co-inductive Proofs in a Program Verifier. In C. B. Jones, P. Pihlajasaari, and J. Sun, editors, *FM 2014*. Vol. 8442, in LNCS, pp. 382–398. Springer, 2014.

[59] A. Lochbihler. Java and the Java Memory Model - A Unified, Machine-Checked Formalisation. In H. Seidl, editor, *ESOP 2012*. Vol. 7211, in LNCS, pp. 497–517. Springer, 2012.

[60] A. Lochbihler. Light-Weight Containers for Isabelle: Efficient, Extensible, Nestable. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *ITP 2013*. Vol. 7998, in LNCS, pp. 116–132. Springer, 2013.

[61] N. Magaud. Changing Data Representation within the Coq System. In D. A. Basin and B. Wolff, editors, *TPHOLs 2003*. Vol. 2758, in LNCS, pp. 87–102. Springer, 2003.

[62] M. Maggesi. A formalisation of metric spaces in HOL Light. Presented at the Workshop Formal Mathematics for Mathematicians. CICM 2015. Published online. 2015. URL: http://www.cicm-conference.org/2015/fm4m/FMM_2015_paper_3.pdf.

[63] D. Matichuk, M. Wenzel, and T. C. Murray. An Isabelle Proof Method Language. In G. Klein and R. Gamboa, editors, *ITP 2014*. Vol. 8558, in LNCS, pp. 390–405. Springer, 2014.

[64] C. McBride et al. [HoTT] Newbie questions about homotopy theory and advantage of UF/Coq. Archived at http://article.gmane.org/gmane.comp.lang.agda/6106. Aug. 1, 2014.

[65] J. C. Mitchell. Representation Independence and Data Abstraction. In, *POPL '86*, pp. 263–276. ACM, 1986.

[66] O. Müller, T. Nipkow, D. v. Oheimb, and O. Slotosch. HOLCF= HOL+LCF. *J. Funct. Program.*, 9(2):191–223, 1999.

[67] M. O. Myreen and J. Davis. The Reflective Milawa Theorem Prover Is Sound - (Down to the Machine Code That Runs It). In G. Klein and R. Gamboa, editors, *ITP 2014*. Vol. 8558, in LNCS, pp. 421–436. Springer, 2014.

[68] W. Naraschewski and M. Wenzel. Object-Oriented Verification Based on Record Subtyping in Higher-Order Logic. In J. Grundy and M. C. Newey, editors, *TPHOLs '98*. Vol. 1479, in LNCS, pp. 349–366. Springer, 1998.

[69] New Rules for HOL. URL: `http://www21.in.tum.de/~kuncar/documents/new_rules.html`.

[70] T. Nipkow. Are Homomorphisms Sufficient for Behavioural Implementations of Deterministic and Nondeterministic Data Types? In F. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *STACS '87*. Vol. 247, in LNCS, pp. 260–271. Springer, 1987.

[71] T. Nipkow. Non-Deterministic Data Types: Models and Implementations. *Acta Informatica*, 22:629–661, 1986.

[72] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Vol. 2283 of *LNCS*. Springer, 2002.

[73] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Part of the Isabelle2015 distribution, 2015. URL: `https://isabelle.in.tum.de/dist/Isabelle2015/doc/tutorial.pdf`.

[74] T. Nipkow and G. Snelting. Type Classes and Overloading Resolution via Order-Sorted Unification. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*. Vol. 523, in LNCS, pp. 1–14. Springer, 1991.

[75] S. Obua. Checking Conservativity of Overloaded Definitions in Higher-Order Logic. In F. Pfenning, editor, *RTA 2006*. Vol. 4098, in LNCS, pp. 212–226. Springer, 2006.

[76] L. C. Paulson. A fixedpoint approach to (co)inductive and (co)datatype definitions. In G. D. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pp. 187–212. MIT Press, 2000.

[77] L. C. Paulson. A Mechanised Proof of Gödel's Incompleteness Theorems Using Nominal Isabelle. *J. Autom. Reasoning*, 55(1):1–37, 2015.

[78] L. C. Paulson. Defining functions on equivalence classes. *ACM trans. comput. log.*, 7(4):658–675, 2006.

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

[80] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[81] A. Pitts. In. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. M. J. C. Gordon and T. F. Melham, editors. Cambridge University Press, 1993. part The HOL Logic, pp. 191–232.

[82] W. Reif, G. Schellhorn, and K. Stenzel. Interactive Correctness Proofs for Software Modules Using KIV. In, *COMPASS '95*, pp. 151–162. IEEE, 1995.

[83] J. C. Reynolds. Types, Abstraction and Parametric Polymorphism. In, *IFIP Congress*, pp. 513–523, 1983.

[84] N. Shankar, S. Owre, and J. M. Rushby. *PVS Tutorial*. Computer Science Laboratory, SRI International, 1993.

[85] O. Slotosch. Higher Order Quotients and their Implementation in Isabelle HOL. In E. L. Gunter and A. P. Felty, editors, *TPHOLs '97*. Vol. 1275, in LNCS, pp. 291–306. Springer, 1997.

[86] M. Sozeau. A New Look at Generalized Rewriting in Type Theory. *J. Formalized Reasoning*, 2(1):41–62, 2009.

[87] M. Sozeau and N. Oury. First-Class Type Classes. In O. A. Mohamed, C. A. Muñoz, and S. Tahar, editors, *TPHOLs 2008*. Vol. 5170, in LNCS, pp. 278–293. Springer, 2008.

[88] The CAVA Project - Computer Aided Verification of Automata. URL: https://cava.in.tum.de/.

[89] The HOL4 Theorem Prover. URL: http://hol.sourceforge.net/.

[90] The seL4 Microkernel. URL: https://sel4.systems/.

[91] D. Traytel, A. Popescu, and J. C. Blanchette. Foundational, Compositional (Co)datatypes for Higher-Order Logic: Category Theory Applied to Theorem Proving. In, *LICS 2012*, pp. 596–605. IEEE, 2012.

[92] C. Urban. Nominal Techniques in Isabelle/HOL. *J. Autom. Reasoning*, 40(4):327–356, 2008.

[93] C. Urban and C. Kaliszyk. General Bindings and Alpha-Equivalence in Nominal Isabelle. *Logical Methods in Computer Science*, 8(2), 2012.

[94] P. Wadler and S. Blott. How to Make Ad-hoc Polymorphism Less Ad Hoc. In, *POPL '89*, pp. 60–76. ACM, 1989.

[95] P. Wadler. Theorems for Free! In, *FPCA '89*, pp. 347–359. ACM, 1989.

[96] M. Wenzel. Asynchronous User Interaction and Tool Integration in Isabelle/PIDE. In G. Klein and R. Gamboa, editors, *ITP 2014*. Vol. 8558, in LNCS, pp. 515–530. Springer, 2014.

[97] M. Wenzel. System description: Isabelle/jEdit in 2014. In C. Benzmüller and B. W. Paleo, editors, *UITP 2014*. Vol. 167, in EPTCS, pp. 84–94, 2014.

[98] M. Wenzel. Isabelle/Isar — a Versatile Environment for Human-Readable Formal Proof Documents. PhD thesis. Institut für Informatik, Technische Universität München, 2002.

[99] M. Wenzel. Type Classes and Overloading in Higher-Order Logic. In E. L. Gunter and A. P. Felty, editors, *TPHOLs '97*. Vol. 1275, in LNCS, pp. 307–322. Springer, 1997.

[100] T. Zimmermann and H. Herbelin. Automatic and Transparent Transfer of Theorems along Isomorphisms in the Coq Proof Assistant. *CoRR*, abs/1505.05028, 2015.