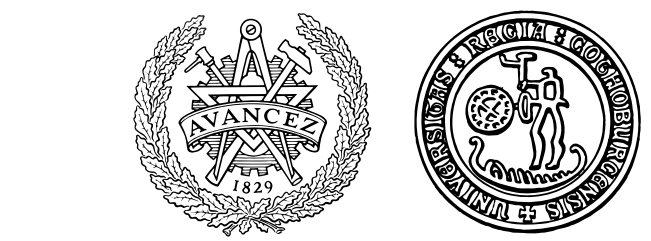
Thesis for the Degree of Doctor of Philosophy

A Theory of Parametric Polymorphism and an Application

A formalisation of parametric polymorphism within and about dependent type-theory, and an application to property-based testing.

Jean-Philippe Bernardy



Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden 2011

A Theory of Parametric Polymorphism and an Application
A formalisation of parametric polymorphism within and about dependent type-theory, and an application to property-based testing.
Jean-Philippe Bernardy
ISBN 978-91-7385-514-3

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 3195
ISSN 0346-718X

Technical Report 77D
ISSN 1653-1787

Department of Computer Science and Engineering
Functional Programming Research Group

Chalmers University of Technology and
Göteborg University
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 10 00

Printed at Chalmers
Göteborg, Sweden 2011

A Theory of Parametric Polymorphism and an Application
A formalisation of parametric polymorphism within and about dependent type-theory, and an application to property-based testing.

Jean-Philippe Bernardy
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

This thesis revisits the well-known notion of parametric polymorphism in the light of modern developments in type-theory. Additionally, applications of parametric polymorphism are also presented.

The first part of the thesis presents a theoretical investigation of the semantics of parametric polymorphism of and within type-theories with dependent types. It is shown how the meaning of polymorphic, possibly dependent, types can be reflected within type-theory itself, via a simple syntactic transformation. This self-referential property opens the door to internalise the transformation in type-theory, and we study one possible way to do so. We also examine how the translation relates to various specific features of type-theory, such as proof irrelevance and realizability.

The second part is concerned an application of parametric polymorphism relevant to software engineers. We present a schema to reduce polymorphic properties to equivalent monomorphic properties, for the purpose of testing. Our proof uses parametricity and properties of initial algebras.

Keywords: Types, Polymorphism, Dependent types, Parametricity, Logical relations, Realizability, Testing, Haskell type-classes

This thesis is based on the work contained in the following papers.

I. Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson (2010). "Parametricity and Dependent Types". In: *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*. Baltimore, Maryland: ACM, pp. 345–356

II. Jean-Philippe Bernardy and Marc Lasson (2011). "Realizability and Parametricity in Pure Type Systems". In: *Foundations Of Software Science And Computational Structures*. Ed. by Martin Hofmann. Vol. 6604. Lecture Notes in Computer Science. Springer, pp. 108–122

III. Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen (2010). "Testing Polymorphic Properties". In: *European Symposium on Programming*. Ed. by Andrew Gordon. Vol. 6012. Lecture Notes in Computer Science. Springer, pp. 125–144

Contents

Acknowledgements

Starting doctoral studies after spending many years away from the academic world is not an obvious thing to do. My warmest thanks go to the people who made this change of direction possible, and supported it. Without them this thesis would never have been written. In particular, Raymond Devillers and Darius Blasband recommended me to Sibylle Schupp who trusted me enough to accept to channel funds into my direction and provided precious supervision during the first year of my studies.

Many thanks also go to Patrik Jansson, who has supervised me for the last three years of my studies. His constant support, interest and guidance have been invaluable. Koen Claessen accepted to co-supervise this work, and his unparalleled enthusiasm easily blew away any cloud of doubt crossing my way. My co-authors also deserve the deepest gratitude, as each of them brought invaluable expertise and broadened my scientific horizons.

All of this work has been greatly facilitated by the environment provided by the department, whose members, from doctoral students to professors, constitute an amazing reservoir of knowledge, brilliance, and wisdom. The less academic aspects of Chalmers are on par, and the administrative staff gets at least part of the credit for it.

I also wish to thank Stephanie Weirich for accepting the role of faculty opponent, as well as the members of the grading committee: Janis Voigtländer, Patricia Johann and Catarina Coquand.

There are more antecedents to this thesis than space allows me to explicitly acknowledge in writing. Even though I must stop here, they shall rest assured that they are not forgotten. But enough looking back! I want to also thank you, reader, for taking some of the ideas from this volume with you, allowing them to live further on.

Introduction

1 Background

1.1 Programming Computers: an Art?

One of the foundational texts of computer science, "The Art of Computer Programming" (Knuth, 1997), is prefaced with these words:

> Here is your book, the one your thousands of letters have asked us to publish. It has taken us years to do, checking and rechecking countless recipes to bring you only the best, only the interesting, only the perfect. Now we can say, without a shadow of a doubt, that every single one of them, if you follow the directions to the letter, will work for you exactly as well as it did for us, even if you have never cooked before. — McCall's Cookbook (1963)

Such an introduction might give the impression that programming is not quite a science, but rather an art, not in the noble sense of creating attractive artifacts (music, painting, sculpture, etc.) but in the pragmatic sense of a collection of techniques learned by practice and observation (as in Sun Tzu's "The Art of War").

The cookbook analogy remains as suggestive today as it was in the past millennium: it seems that programming remains a business of trial and error, mostly guided by empirical experience rather than rigorous scientific inquiry.

A possible reason for this state of affairs is that programming has more emphasis on the computer rather than the problem it is supposed to solve. Practitioners of the art focus too quickly on the sequence of operations that the computer should perform, at the expense of improving their own understanding of the problem at hand. Often this shift of attention results in software errors, because it is difficult to keep a good overview of a program when one thinks of it as millions of elementary instructions.

1.2 Intuitionistic Programming

We believe that programs are first and foremost constructions of the mind; and essential to their correctness is therefore the ability of the programming environment to support expressing the intuition of the programmer as naturally as possible. What this might mean in technical terms is subjective. We remark however that the ability to express oneself naturally is also an important property of logical systems; and thus we attempt to transpose some lessons learned from that field. We adopt the view that intuitionistic type-theory (Martin-Löf, 1984) is the right framework to express oneself logically, and therefore that an ideal programming language should incorporate that framework[1].

In more detail, two aspects of programming environments are essential to the construction of correct programs: *abstraction*, and *types*.

1.2.1 Abstraction

Over the years, proverbial pieces of wisdom have emerged to help programmers to focus on their intent rather than particulars of implementation. A popular one instructs: — Don't repeat yourself! — compelling the programmer to avoid repeating an implementation pattern multiple times. Sticking to this principle prevents from scattering the implementation of a single idea over the whole program, forcing to encode it as directly as possible into the programming language.

Pierce (2002) translates "Don't repeat yourself" into the following more technical terms:

> Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by *abstracting out* the varying parts.

However, abstracting-out common parts is feasible only if the host programming language allows it. The binary codes that a computer can natively understand, (as well as their direct symbolic representations) typically do not allow any kind of abstraction — one is forced to encode high-level ideas, necessarily obscuring the intent.

More evolved[2] programming languages allow extracting parts of programs and using them whenever needed simply by invoking them by name. These program pieces program can be parametrized by simple values (they abstract over the actual data to handle), then one call these *functions*. For

[1] Hence the title of this section.

[2] Dare we say more intelligently designed?

example, a list-sorting function can take the list of items to sort as a parameter, and returns the sorted list as result. With the advent of FORTRAN (Metcalf and Reid, 1990) and similar languages, such first-order abstraction became a commodity available to the majority of programmers.

Further along the line, functional programming languages such as Haskell (Marlow, 2010) or ML (Milner, Tofte, and Harper, 1990) allow to abstract over functions themselves, allowing great freedom of abstraction.

1.2.2 Types

Another language feature which is essential to support construction of correct programs are *types*. In a nutshell, types enable the programmer to keep track of the structure of data and computation in a way that is checkable by the computer itself. Effectively, they act as contracts between the implementer of the function and its users. Simple type systems such as that of FORTRAN focus on the interpretation of bit patterns as integers or rational values. For example, the compiler will warn the user of a function whenever they attempt to feed a 64-bit integer to a function expecting a 16-bit one.

In languages with higher-order abstraction, such as Haskell or ML, one can abstract over functions themselves. Types then specify the type of the functions expected as parameters; and thus the type system becomes much more expressive.

Advanced type systems, such as that of Agda (Norell, 2007), can capture any functional property of data and computation: whether a list is sorted, that a relational database corresponds to a given schema, etc. (Oury and Swierstra (2008) provide more examples.) If type-checking is performed statically, when the program is compiled, it then amounts to proving that such properties hold for all executions of the program, independently of its input.

The contributions of this thesis are within the realm of functional programming with rich type systems. We believe that such programming languages are the best environments currently available to construct correct programs, because they allow to:

- freely abstract parts of the program as desired, and conversely freely combine simple, well-understood functions into more complex parts; and

- precisely express the intention of the programmer for each function using types.

1.3 Parametric Polymorphism

Having briefly introduced the notion of types and abstraction (we refer the reader to, for example, Pierce (2002) for an extensive discussion), we are ready to introduce the topic of this thesis: parametric polymorphism. Indeed, parametric polymorphism combines these two notions, it is *abstraction over types*. The rest of this section uses an example to illustrate the idea.

Consider the following function, written in a Haskell-like language. It inserts an integer into a suitable position within a sorted list.

```
insert : Int → List Int → List Int
insert x [ ] = [x]
insert x (y :: t) = if   x ⩽ y
                    then x :: y :: t
                    else y :: insert x t
```

The above function does its job adequately, but is unsatisfactory in at least one aspect: it is not as abstract as possible. For starters, it uses a *concrete* comparison function, whereas in principle it could use any notion of ordering. Therefore, the particular ordering used can be abstracted, to yield the following definition:

```
insert : (Int → Int → Bool) → Int → List Int → List Int
insert leq x [ ] = [x]
insert leq x (y :: t) = if   leq x y
                        then x :: y :: t
                        else y :: insert leq x t
```

This version is more abstract, but misses one crucial aspect of abstraction. Cursory analysis of the function definition reveals that it is in fact completely independent of the *type* of the elements handled. The same function would work just as well with, for example, characters instead of integers. This can be captured by adding another parameter to the function. The type of this parameter is $\star$, the type of types. Since the rest of the type is dependent on this type argument, it must be named there. The resulting syntax is illustrated in the following piece of code.

```
insert : (a : ⋆) → (a → a → Bool) → a → List a → List a
insert a leq x [ ] = [x]
insert a leq x (y :: t) = if   leq x y
                          then x :: y :: t
                          else y :: insert a leq x t
```

The kind of abstraction introduced by the above example is often called *parametric polymorphism*, and is the focus of the remainder of this volume. Before moving on, we shall stress an important characteristic of $\star$, which

remains implicit in the above example. The type of types ($\star$) is *itself* abstract. That is, a function with a parameter a of type $\star$ is forbidden to behave differently depending on the actual argument given for a. (Still, merely passing along a's actual value, or values of type a, is allowed.) The prototypical language capturing the concept of parametric polymorphism is the polymorphic lambda-calculus (Reynolds, 1974). It was developed independently by Girard (1972). A perhaps more pedagogical introduction to polymorphism is given by Cardelli and Wegner (1985).

1.4 The relational interpretation of types

As we have previously mentioned, types are a form of contract between the implementer of the function and its users. By giving a polymorphic type to a function, implementers promise to handle values abstractly. In return, the function can be used in a wide variety of applications.

The meaning of the contract can be captured by logical propositions. For example, the type-declaration of insert gives rise to the following proposition[3]:

$$
\begin{aligned}
&\forall\, a_1\, a_2 \;\to\; (a_R : a_1 \to a_2 \to \star) \to \\
&\forall\, o_1\, o_2 \;\to\; (\forall\, x_1\, x_2 \to a_R\, x_1\, x_2 \to \\
&\qquad\qquad \forall\, y_1\, y_2 \to a_R\, y_1\, y_2 \to o_1\, x_1\, y_1 \;==\; o_2\, x_2\, y_2) \to \\
&\forall\, x_1\, x_2 \;\to\; a_R\, x_1\, x_2 \to \\
&\forall\, xs_1\, xs_2 \to \mathsf{Indexwise}\, a_R\, xs_1\, xs_2 \to \\
&\mathsf{Indexwise}\, a_R\, (\mathsf{insert}\, a_1\, o_1\, x_1\, xs_1)\, (\mathsf{insert}\, a_2\, o_2\, x_2\, xs_2)
\end{aligned}
$$

In the above, a_R refers to any relation between the types a_1 and a_2; and Indexwise lifts the relation to lists, index-wise. The proposition is a theorem, and its proof can be extracted mechanically from the definition of insert. In fact, every implementation of insert respecting the contract imposed by the type would yield a valid proof. This means that the above theorem holds for any function that has the same type as insert — we know it holds even before looking at the definition of the function it concerns. Hence, the propositions coming from the relational interpretation are sometimes called "free theorems".

The rules to interpret System F types as relations in second order logic, as well as the statement that every object must satisfy parametricity, were given by Reynolds (1983). This interpretation makes the contract (as-type) accessible to reasoning in a logical framework. Wadler (1989) has taken advantage of this result to derive useful theorems from the type of usual functional programs. Much of this dissertation is concerned with extending this idea to more complex type systems.

[3]The method to derive it is explained in Paper I.

1.5 Programming languages with dependent types

System F is the core type system of many mainstream functional languages, but recent developments have seen the rise of programming languages based on even richer type systems. A critical feature of such systems is that the right-hand-side of a function type can depend on the variable bound by the left-hand-side. A prototypical example is a lookup function in a tuple of given length n.

$$\mathsf{lookup} : (\mathsf{a} : \star) \to (\mathsf{i} : \mathbb{N}) \to (\mathsf{n} : \mathbb{N}) \to (\mathsf{i} < \mathsf{n}) \to \mathsf{Vec}\,\mathsf{a}\,\mathsf{n} \to \mathsf{a}$$

Syntactically, the quantification over types ($(\mathsf{a} : \star) \to \ldots$) is unified with the formation of function types ($(\mathsf{i} : \mathbb{N}) \to \ldots$). This syntactical unification is the central feature of so called Pure Type Systems (PTS) (Barendregt, 1992). A consequence of such a treatment is that the programming language also becomes a powerful logical framework.[4] Propositions can simply be written down in the language of types; and inhabitants of the type play the role of proofs. This unification of types and propositions is central in the next chapter. With the introduction of dependency on values, types can express any proposition, and thus they become a very precise specification tool. This means that it is possible both to *specify* and *implement* safety-critical applications in a single framework.

Languages featuring dependent types include Agda (Norell, 2007), which is inspired by Martin-Löf type theory (Martin-Löf, 1984). The proof assistant Coq, which is based on the calculus of inductive constructions (Coquand and Huet, 1986; Werner, 1994), has recently gained popularity as a framework for expressing verified programs (Leroy, 2009; Chlipala et al., 2009). Some aspects of dependent types have also made their way into the Glasgow Haskell Compiler (the most used Haskell compiler).

2 Contents

The rest of the dissertation is comprised of self-contained papers which discuss various aspects of the notion of parametric polymorphism. We give here a brief overview that can serve as a reading guide.

2.1 Paper I: Parametricity and dependent types

In Paper I, the concern is to generalise Reynolds' abstraction theorem for systems with dependent types. Furthermore, if the framework is sufficiently rich (as Coq or Agda are), the relational interpretation of terms can be expressed in the same framework as the terms themselves.

[4] Conversely, proof assistants can be used as programming languages.

Technically, for any PTS used as a programming language, there is a PTS that can be used as a logic for parametricity. Types in the source PTS are translated to relations (expressed as types) in the target system. Similarly, values of a given type are translated to proofs that the values satisfy the relational interpretation.

We also show that the assumption that every term satisfies the parametricity condition generated by its type is consistent with the generated logic.

A consequence is that, for every function written by the programmer, the theorem that it satisfies its parametricity condition comes *for free*, and can be used right along with the original function in a proof assistant. The net effect is that properties of the type that are given by polymorphism become available for showing the correctness of the program.

2.2 Paper II: Realizability and Parametricity

Reynolds' interpretation of types essentially acts as an embedding of System F into second-order logic. The corresponding projection removes much information from logical formulas to obtain types of System F. The projection was developed earlier by Girard (1972). The connection between Reynolds' and Girard's interpretations was identified by Wadler (2007).

Whereas Paper I shows how Reynolds' *embedding* can be generalised to dependent types, Paper II shows how Girard's *projection* can be generalised. The paper also reveals parallels between parametricity and Krivine-style realizability.

Additionally, Paper II addresses two shortcomings of Paper I. First, we show how to handle PTSs with a finite number of sorts: it is not necessary to extend the PTS of the programming language with an infinite hierarchy to express the relational interpretation of its terms, as we do in Paper I. Second, the proof of the abstraction theorem is more elegant: by separating the treatments of terms and types, the proof becomes more structured and shorter than that presented in Paper I. These two improvements come at the cost of a longer definition for the relational interpretation of terms.

2.3 Paper III: Testing Polymorphic Properties

2.3.1 Property-based Testing

Besides typing, testing is another useful technique to aid in producing reliable software. It comes into play when precise typing is either not possible (the type system is too weak to express the property of interest) or not practical (for example one does not know yet the right properties to capture in the type). While testing isolated cases can be necessary to get a grip on the program behaviour, it is often better to express general

properties about functions and let a tool generate test cases to search for counter-examples (Hughes, 2007).

2.3.2 Testing Polymorphic Code

Polymorphic properties are not so easy to test in that way though: because testing can only be applied to concrete values, one must generate monomorphic instances. The problem is to pick the right set of monomorphic instances to test on. Many approaches are possible, from the most conservative to the most daring:

1. test on every possible type;
2. test on a type with infinitely many elements; or
3. test on a type with just a few elements (like booleans).

The issue can be quite confusing, and we have anecdotal evidence that testers often do the wrong thing.

For years, QuickCheck practice has been to use the second approach. While relatively safe, this method is quite wasteful: one may potentially perform many redundant tests. In order to speed up testing, Runciman, Naylor, and Lindblad (2008) suggest the third method. However, if a too small type is chosen, the generator will consistently miss whole classes of test cases, giving a false sense of security to the users.

We propose a technique to automatically derive a monomorphic instance of a polymorphic property aiming at minimising the amount of redundant testing. We prove, via the relational interpretation of types, that such a constructed instance covers all the cases possible in the polymorphic case. In terms of the hierarchy presented above, our technique is as safe as 1. and more efficient than 2.

Our framework effectively enables testers to take advantage of parametric polymorphism without requiring them to work with the interpretation of types as relations, nor leaving them to wonder how to apply it to their particular problem.

2.4 Statement of personal contribution

The papers in this dissertation are coauthored with other people. My contributions to the papers are as follows:

I. **Proofs for free – Parametricity for dependent types**

   The main contributions of this paper are shared with my co-authors. Other contributions are solely due to me, such as

- the introduction of coloured pure type systems and the use of colours in the formulation of the relational interpretation,
- the internalisation of parametricity,
- the irrelevant interpretation of propositions,
- the proof that identity proofs are extensionally equivalent to the identity.

Ross Paterson is entirely responsible for the proof of correctness of the inductive parametric interpretation of inductive families.

II. **Realizability and Parametricity in Pure Type Systems**

In this paper, my main technical contributions are

- the proof of the abstraction theorem (which is simpler than the proof presented in the previous paper, at least in some respects), and
- the theorem reducing realizability to parametricity, as well as its (straightforward) proof.

The rest of the contributions of the paper are shared with Marc Lasson, except for the proofs showing that the logical system P^2 is well-behaved (e.g. its normalisation property), which are entirely due to him.

III. **Testing Polymorphic Properties**

The technical contributions in this paper are mine. Patrik Jansson pointed out that the translation to our canonical testing type can be expressed as embedding-projection pairs. Most examples are due to Koen Claessen.

Paper I

Proofs for Free - Parametricity for Dependent Types

This paper is an extended and revised version of a paper which appeared in the proceedings of the International Conference on Functional Programming, 2010, under the title "Parametricity and Dependent Types". The present version is under consideration for publication in the Journal of Functional Programming.

Proofs for Free - Parametricity for Dependent Types

Jean-Philippe Bernardy, Patrik Jansson, Ross Paterson

Abstract

Reynolds' abstraction theorem shows how a typing judgement in System F can be translated into a relational statement (in second order predicate logic) about inhabitants of the type.

We obtain a similar result for pure type systems: for any PTS used as a programming language, there is a PTS that can be used as a logic for parametricity. Types in the source PTS are translated to relations (expressed as types) in the target. Similarly, values of a given type are translated to proofs that the values satisfy the relational interpretation. We extend the result to inductive families.

We also show that the assumption that every term satisfies the parametricity condition generated by its type is consistent with the generated logic. Our proof gives a computationally meaningful way to interpret that assumption.

1 Introduction

Types are used in many parts of computer science to keep track of different kinds of values and to keep software from going wrong. Starting from the presentation of the simply typed lambda calculus by Church (1940), we have seen a steady flow of typed languages and calculi. With increasingly rich type systems came more refined properties about well-typed terms. In his *abstraction theorem* Reynolds (1983) defined a relational interpretation of System F types, and showed that interpretations of a well-typed term in related contexts yield related results. In "Theorems for Free" Wadler (1989) observed that if a type has no free variables, the relational interpretation can thus be viewed as a *parametricity* property satisfied by all terms of that type. Almost twenty years ago Barendregt (1992) described a common framework for a large family of calculi with expressive types: Pure Type Systems (PTSs). By the Curry–Howard correspondence, the calculi in the PTS family can be seen both as programming languages and as logics. The more advanced calculi go beyond System F and include full dependent types and support expressing datatypes.

Recent work (Takeuti, 2004; Johann and Voigtländer, 2006; Neis, Dreyer, and Rossberg, 2009; Vytiniotis and Weirich, 2010) has developed parametricity results for several such calculi, but not in a common framework. In this paper, we apply and extend Reynolds' idea to a large class

of PTSs and we provide a framework which unifies previous descriptions of parametricity and forms a basis for future studies of parametricity in specific type systems. As a by-product, we get parametricity for dependently-typed languages. This paper is an extended and revised version of (Bernardy, Jansson, and Paterson, 2010). Our specific contributions are:

- An extension of the PTS framework to capture explicit syntax (Section 2).
- A concise definition of the translation of types to relations (Definition 9), which yields parametricity propositions for PTSs.
- A formulation (and a proof) of the abstraction theorem for PTSs (Theorem 11). A remarkable feature of the theorem is that the translation from types to relations and the translation from terms to proofs are unified.
- An extension of the translation to inductive definitions (Section 4), and its proof of correctness.
- A formulation of an axiom schema able to internalise the abstraction theorem in the target PTS. The axiom schema is proved consistent, thanks to a translation to the PTS without the axioms (Section 5).
- A specialisation of the general framework to constructs such as propositions, type classes and constructor classes (Section 6).
- A demonstration by example of how to derive free theorems for (and as) dependently-typed functions (sections 3.3, 4 and 6).

Our examples use a notation close to that of Agda (Norell, 2007), for greater familiarity for users of dependently-typed functional programming languages. The notation takes advantage of the "implicit syntax" feature, making the examples easy to read.

2 Pure type systems, with colour

In this section we introduce the notion of coloured pure type systems, which is an extension of PTS (as described by Barendregt (1992, sec. 5.2)). The colours capture the fact that various flavours of quantification use different syntax. We introduce our notation along the way, as well as our running example type systems. While mere PTSs are sufficient for (most of) the technical results of this paper, colours allow to emphasise the structure of our translation from programs to their relational counterparts. A two-colour PTS is also extensively used in our examples.

Definition 1 (Syntax of terms)**.** A PTS is a type system over a λ-calculus with the following syntax:

$$\begin{array}{rcll} \mathcal{T} & = & \mathcal{C} & \text{constant} \\ & | & \mathcal{V} & \text{variable} \\ & | & \mathcal{T}\,\mathcal{T} & \text{application} \\ & | & \lambda\mathcal{V}{:}\mathcal{T}.\,\mathcal{T} & \text{abstraction} \\ & | & \forall\mathcal{V}{:}\mathcal{T}.\,\mathcal{T} & \text{dependent function space} \end{array}$$

We often write $(\mathsf{x}{:}A) \to B$ for $\forall\mathsf{x}{:}A.\,B$, and sometimes just $A \to B$ when x does not occur free in B. We use different fonts to indicate what category a meta-syntactic variable ranges over. Sans-serif roman (like x) is used for $\mathcal{V}$, fraktur (like $\mathfrak{c}$) for $\mathcal{C}$ and italics (like A) for $\mathcal{T}$. As an exception, the letters s and t are used for the subset $\mathcal{S}$ of $\mathcal{C}$ introduced in the next paragraph.

The typing judgement of a PTS is parametrised over a *specification* $S = (\mathcal{S}, \mathcal{A}, \mathcal{R})$, where $\mathcal{S} \subseteq \mathcal{C}$, $\mathcal{A} \subseteq \mathcal{C} \times \mathcal{S}$ and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$. The set $\mathcal{S}$ specifies the sorts, $\mathcal{A}$ the axioms (an axiom $(\mathfrak{c}, s) \in \mathcal{A}$ is often written $\mathfrak{c} : s$), and $\mathcal{R}$ specifies the typing rules of the function space. A rule (s_1, s_2, s_2), where the second and third sorts coincide, is often written $s_1 \rightsquigarrow s_2$.

An attractive feature of PTSs is that the syntax for types and values is unified. It is the type of a term that tells how to interpret it (as a value, type, kind, etc.).

The λ-cube Barendregt (1992) defined a family of calculi each with $\mathcal{S} = \{\star, \Box\}$, $\mathcal{A} = \{\star : \Box\}$ and $\mathcal{R}$ a selection of rules of the form $s_1 \rightsquigarrow s_2$, for example:

- The (monomorphic) λ-calculus has $\mathcal{R}_\lambda = \{\star \rightsquigarrow \star\}$, corresponding to ordinary (value-level, non-dependent) functions.
- System F has $\mathcal{R}_\mathrm{F} = \mathcal{R}_\lambda \cup \{\Box \rightsquigarrow \star\}$, adding (impredicative) universal quantification over types (thus including functions from types to values).
- System Fω has $\mathcal{R}_{\mathrm{F}\omega} = \mathcal{R}_\mathrm{F} \cup \{\Box \rightsquigarrow \Box\}$, adding type-level functions.
- The Calculus of Constructions (CC) has $\mathcal{R}_\mathrm{CC} = \mathcal{R}_{\mathrm{F}\omega} \cup \{\star \rightsquigarrow \Box\}$, adding dependent types (functions from values to types).

Here $\star$ and $\Box$ are conventionally called the sorts of *types* and *kinds* respectively.

Notice that F is a subsystem of Fω, which is itself a subsystem of CC. (We say that $S_1 = (\mathcal{S}_1, \mathcal{A}_1, \mathcal{R}_1)$ is a subsystem of $S_2 = (\mathcal{S}_2, \mathcal{A}_2, \mathcal{R}_2)$ when $\mathcal{S}_1 \subseteq \mathcal{S}_2$, $\mathcal{A}_1 \subseteq \mathcal{A}_2$ and $\mathcal{R}_1 \subseteq \mathcal{R}_2$.) In fact, the λ-cube is so named because the lattice of the subsystem relation between all the systems forms a cube, with CC at the top.

Sort hierarchies Difficulties with impredicativity[1] have led to the development of type systems with an infinite hierarchy of sorts. The "pure" part of such a system can be captured in the following PTS, which we name I_ω.

Definition 2 (I_ω)**.** I_ω is a PTS with this specification:

- $\mathcal{S} = \{\star_i \mid i \in \mathbb{N}\}$
- $\mathcal{A} = \{\star_i : \star_{i+1} \mid i \in \mathbb{N}\}$
- $\mathcal{R} = \{(\star_i, \star_j, \star_{\max(i,j)}) \mid i, j \in \mathbb{N}\}$

Compared to the monomorphic λ-calculus, $\star$ has been expanded into the infinite hierarchy $\star_0, \star_1, \ldots$ In I_ω, the sort $\star_0$ (abbreviated $\star$) is called the sort of types. Type constructors, or type-level functions have type $\star \to \star$. Terms like $\star$ (representing the set of types) and $\star \to \star$ (representing the set of type constructors) have type $\star_1$ (the sort of kinds). Terms like $\star_1$ and $\star \to \star_1$ have type $\star_2$, and so on.

Although the infinite sort hierarchy was introduced to avoid impredicativity, they can in fact coexist, as Coquand (1986) has shown. For example, in the Generalised Calculus of Constructions (CC_ω) of Miquel (2001), impredicativity exists for the sort $\star$ (conventionally called the sort of *propositions*), which lies at the bottom of the hierarchy.

Definition 3 (CC_ω)**.** CC_ω is a PTS with this specification:

- $\mathcal{S} = \{\star\} \cup \{\Box_i \mid i \in \mathbb{N}\}$
- $\mathcal{A} = \{\star : \Box_0\} \cup \{\Box_i : \Box_{i+1} \mid i \in \mathbb{N}\}$
- $\mathcal{R} = \{\star \rightsquigarrow \star, \star \rightsquigarrow \Box_i, \Box_i \rightsquigarrow \star \mid i \in \mathbb{N}\} \cup$
  $\{(\Box_i, \Box_j, \Box_{\max(i,j)}) \mid i, j \in \mathbb{N}\}$

In the above definition, impredicativity is implemented by the rules of the form $\Box_i \rightsquigarrow \star$.

Both CC and I_ω are subsystems of CC_ω, with $\star_i$ in I_ω corresponding to $\Box_i$ in CC_ω. Because $\Box$ in CC corresponds to $\Box_0$ in CC_ω, we often abbreviate $\Box_0$ as $\Box$.

Many dependently-typed programming languages and proof assistants are based on variants of I_ω or CC_ω, often with the addition of inductive definitions (Dybjer, 1994; Paulin-Mohring, 1993). Such tools include Agda (Norell, 2007), Coq (The Coq development team, 2010) and Epigram (McBride and McKinna, 2004).

[1] It is inconsistent with strong sums (Coquand, 1986).

2.1 PTS as logical framework

Another use for PTSs is as logical frameworks: types correspond to propositions and terms to proofs. This correspondence extends to all aspects of the systems and is widely known as the Curry-Howard isomorphism. The judgement $\vdash p : P$ means that p is a *witness*, or *proof* of the proposition P. If the judgement holds (for some p) we say that P is *inhabited*.

In the logical system reading, an inhabited type corresponds to a tautology and dependent function types correspond to universal quantification. A predicate P over a type A has the type $A \to s$, for some sort s: a value a satisfies the predicate whenever the type $P\,a$ is inhabited. Similarly, binary relations between values of types A_1 and A_2 have type $A_1 \to A_2 \to s$.

For this approach to be safe, it is important that the system be *consistent*. In fact, the particular systems used here even exhibit the stronger normalisation property: each each witness p reduces to a normal form.

In fact, in I_ω and similarly rich type systems, one may both represent programs and logical formulae about them. In the following sections, we make full use of this property: we encode programs and parametricity statements about them in the same type system.

2.2 Explicit Syntax: Coloured Pure Type Systems

In PTSs, the syntax of terms is completely uniform. However, some systems usually presented as PTSs still use different syntax for the various forms of quantifications. For example, traditional presentations of System F use a different syntax for the quantification over individuals ($\star \rightsquigarrow \star$) than for the quantification over types ($\Box \rightsquigarrow \star$). A common case is to use the symbols $\forall$ and Λ for quantification and abstraction over types, and $\to$ and λ for individuals. Additionally, brackets are often used to mark type application. While the flavour of quantification can always be recovered from a type derivation, the advantage of explicit syntax is that it is possible to identify which flavour is used merely by looking at the term. Moreover, a type-derivation tree might not be available.

The purpose of Coloured Pure Type Systems is to capture explicit syntax in a parametrised way. A colour annotation is added to the syntax of application, abstraction and product, and a colour component is added to $\mathcal{R}$. A rule (k, s_1, s_2, s_2) is often written $s_1 \overset{k}{\rightsquigarrow} s_2$. Note that a single colour may be assigned to multiple rules. (In the electronic version of this document, colours are sometimes rendered visually.) The corresponding typing rules ensure that the colours are matched (Figure 1.1).

Erasure of colour yields a plain (monochrome) PTS; and erasure of colour in a valid coloured derivation tree yields a valid derivation tree in the monochrome PTS. Therefore, useful properties of PTSs (such as subject

$$
\begin{array}{rcll}
\mathcal{T}_{\mathcal{K}} & = & \mathcal{C} & \text{constant} \\
& | & \mathcal{V} & \text{variable} \\
& | & \mathcal{T} \bullet_{\mathcal{K}} \mathcal{T} & \text{application} \\
& | & \lambda^{\mathcal{K}} \mathcal{V} : \mathcal{T}.\ \mathcal{T} & \text{abstraction} \\
& | & \forall^{\mathcal{K}} \mathcal{V} : \mathcal{T}.\ \mathcal{T} & \text{dependent function space}
\end{array}
$$

$$\frac{}{\vdash \mathfrak{c} : s}\ \mathfrak{c} : s \in \mathcal{A} \qquad \textsc{axiom}$$

$$\frac{\Gamma \vdash A : s}{\Gamma, \mathsf{x}{:}A \vdash \mathsf{x} : A} \qquad \textsc{start}$$

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, \mathsf{x}{:}C \vdash A : B} \qquad \textsc{weakening}$$

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, \mathsf{x}{:}A \vdash B : s_2}{\Gamma \vdash (\forall^k \mathsf{x}{:}A.\ B) : s_3}\ (k, s_1, s_2, s_3) \in \mathcal{R} \qquad \textsc{product}$$

$$\frac{\Gamma \vdash F : (\forall^k \mathsf{x}{:}A.\ B) \qquad \Gamma \vdash a : A}{\Gamma \vdash F \bullet_k a : B[\mathsf{x} \mapsto a]} \qquad \textsc{application}$$

$$\frac{\Gamma, \mathsf{x}{:}A \vdash b : B \qquad \Gamma \vdash (\forall^k \mathsf{x}{:}A.\ B) : s}{\Gamma \vdash (\lambda^k \mathsf{x}{:}A.\ b) : (\forall^k \mathsf{x}{:}A.\ B)} \qquad \textsc{abstraction}$$

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s \qquad B =_\beta B'}{\Gamma \vdash A : B'} \qquad \textsc{conversion}$$

Figure 1.1: CPTS syntax for the set of colours $\mathcal{K}$, and typing rules of the CPTS with specification $(\mathcal{K}, \mathcal{S}, \mathcal{A}, \mathcal{R})$. The only change with respect to the standard PTS definition is the addition of colour annotations in product, application and abstraction.

reduction, substitution, etc.) are retained in CPTSs.

2.2.1 A colour for logic

Earlier in this section, we have outlined how PTSs can be used to represent concepts like propositions and proofs. One may want to use special syntax for PTS constructs when the proposition-as-types interpretation is intended: even though propositions and types are syntactically unified in PTSs, it can be useful to make the intent explicit. Therefore a special colour might be reserved for the purpose of expressing logical formulae in some CPTSs. A possible choice of concrete syntax is the following, reminiscent of naive set theory.

$$\begin{array}{rcll} \mathcal{T}_{\text{logic}} & = & \ldots & \\ & | & \mathcal{T} \in \mathcal{T} & \text{(reverse application)} \\ & | & \{\mathcal{V}:\mathcal{T}|\mathcal{T}\} & \text{(abstraction)} \\ & | & \forall\mathcal{V}:\mathcal{T}.\ \mathcal{T} & \text{(quantification)} \end{array}$$

2.2.2 A colour for implicit syntax

Many proof assistants and dependently-typed programming languages (including Agda, Coq and LEGO) provide so-called "implicit" syntax. The rationale for the feature is that, in the presence of precise type information, some parts of terms (applications or abstractions) can be fully inferred by the type-checker. In such cases, the user might want to actually leave out such parts the the terms. It is convenient to do so by marking certain quantifications as "implicit". Then, the presence of the corresponding applications and abstractions can be inferred by the type-checker.

Such marking can be precisely modelled by a two-colour PTS: one colour for regular syntax, and one for "implicit-syntax". (Typically every rule is available in both colours.) The syntax of CPTS does not allow for omission of terms though, so it can be used only for terms whose omitted parts have been filled in by the type-checker. Miquel (2001, p. 1.3.2) gives a detailed overview of two-colour PTSs used for implicit syntax.

3 The relational interpretation

In this section we present the core contribution of this paper: the relational interpretation of a term, as a syntactic translation from terms (in a source PTS) to terms (in a target PTS). As we will see in Section 3.3, it is a generalisation of the classical rules given by Reynolds (1983), extended to application and abstraction.

3.1 From programming language to logic

For a particular source CPTS S, we shall require a target CPTS S^t, where the relational counterparts of the source terms can be expressed. The use of colour is double: first, to emphasise the structure of the relational interpretation (which is less apparent in the monochrome case) and second to support application to systems with implicit syntax. For a first approximation, we assume that the only constants in S are sorts. We return to the general case in Section 4.

Definition 4 (reflecting system)**.** A CPTS $S^t = (\mathcal{K}^t, \mathcal{S}^t, \mathcal{A}^t, \mathcal{R}^t)$ *reflects* a CPTS $S = (\mathcal{K}, \mathcal{S}, \mathcal{A}, \mathcal{R})$ if S is a subsystem of S^t and

1. there is a colour $0 \in \mathcal{K}^t$, used for relation (or predicate) construction. Annotations for this colour are consistently omitted in the remainder of the section.
2. there are two functions $\cdot_i$ and $\cdot_r$ from $\mathcal{K}$ to $\mathcal{K}^t$
3. for each sort $s \in \mathcal{S}$,
   - $\mathcal{S}^t$ contains sorts s', $\lceil s \rceil$, $\lceil s' \rceil$ and $\lceil s'' \rceil$
   - $\mathcal{A}^t$ contains $s : s'$, $\lceil s \rceil : \lceil s' \rceil$ and $\lceil s' \rceil : \lceil s'' \rceil$
   - $\mathcal{R}^t$ contains $s \rightsquigarrow \lceil s' \rceil$ and $s' \rightsquigarrow \lceil s'' \rceil$
4. for each axiom $s : t \in \mathcal{A}$, $\lceil s' \rceil = \lceil t \rceil$.
5. for each rule $(k, s_1, s_2, s_3) \in \mathcal{R}$,
   - $(k, s_1, s_2, s_3) \in \mathcal{R}^t$
   - $s_1 \overset{k_i}{\rightsquigarrow} \lceil s_3 \rceil \in \mathcal{R}^t$
   - $(k_r, \lceil s_1 \rceil, \lceil s_2 \rceil, \lceil s_3 \rceil) \in \mathcal{R}^t$

Remark 5. *The above definition is intuitively justified as follows.*

1. *The colour* 0 *is used for formation of parametricity predicates.*
2. *For each colour* $k \in \mathcal{K}$*,*
   - *the colour* k_i *is used for universal quantification over individuals in logical formulas;*
   - *the colour* k_r *is used for quantifications over propositions in the target system.*
3. *For each sort* s*, the sort* $\lceil s \rceil$ *is the sort of parametricity propositions about types in* s*, and must exist in* $\mathcal{S}^t$*. One can see* $\lceil \cdot \rceil$ *as a function from* $\mathcal{S}$ *to* $\mathcal{S}^t$*.*

For each input sort, the relational interpretation creates redexes to check predicate membership[2]*. This requires*

- *each input sort s to be typeable (i.e. inhabit another sort s' — in the above definition we consistently use s' for a sort that s inhabits);*
- *two extra sorts in the target system ($\lceil s' \rceil$, $\lceil s'' \rceil$) on top of $\lceil s \rceil$;*
- *rules to allow for the formation of predicates.*

4. *The following two relations between sorts must commute.*
   - *axiomatic inhabitation ($\mathcal{A}$);*
   - *correspondence between a sort of types and a sort of relational propositions ($\lceil \cdot \rceil$).*
5. *For each type-formation rule of the input system, there is:*
   - *a copy of the rule in the target system;*
   - *a formation rule for quantification over individuals;*
   - *a formation rule for relational-propositions, exactly mirroring that of the input system.*

Example 1. The system CC_ω reflects each of the systems in the λ-cube, with $\lceil s \rceil = s$ and $k_i = k_r = k$.

Definition 6 (reflective)**.** We say that S is reflective if S reflects itself with $\lceil s \rceil = s$ and $k_i = k_r = k$.

Example 2. Note that both I_ω and CC_ω are reflective. Therefore we can write programs in these systems and derive valid statements about them, within the same PTS.

3.2 From types to relations, from terms to proofs

Definition 7 (renaming)**.** The term A_i is obtained by replacing each free variable x in the term A by a variable x_i.

Definition 8 (replication)**.** Given a natural number n (implicit from the context), $\overline{A}$ stands for n terms A_i, each obtained by renaming, as defined above. Correspondingly, $\overline{\mathsf{x}:A}$ stands for n bindings $(\mathsf{x}_i : A_i)$. If replication is used in a binder (abstraction or dependent function space), then the binder is also replicated.

[2] In Paper II we do away with this infelicity at the cost of a longer definition of the relational interpretation.

Definition 9 ($[\![\cdot]\!]$, translation from types to relations)**.** We define a mapping $[\![\cdot]\!]$ from $\mathcal{T}_{\mathcal{K}}$ to $\mathcal{T}_{\mathcal{K}^t}$ as follows:

$$\begin{aligned}
[\![s]\!] &= \lambda \overline{x{:}s}.\ \overline{x} \to \lceil s \rceil \\
[\![x]\!] &= x_{\mathsf{R}} \\
[\![\forall^k x{:}A.\ B]\!] &= \lambda \overline{f{:}(\forall^k x{:}A.\ B)}.\ \forall^{k_i} \overline{x{:}A}.\ \forall^{k_r} x_{\mathsf{R}}{:}[\![A]\!]\,\overline{x}.\ [\![B]\!]\ (\overline{f \bullet_k x}) \\
[\![F \bullet_k a]\!] &= [\![F]\!] \bullet_{k_i} \overline{a} \bullet_{k_r} [\![a]\!] \\
[\![\lambda^k x{:}A.\ b]\!] &= \lambda^{k_i} \overline{x{:}A}.\ \lambda^{k_r} x_{\mathsf{R}}{:}[\![A]\!]\,\overline{x}.\ [\![b]\!]
\end{aligned}$$

Note that for each variable x free in A, the translation $[\![A]\!]$ has free variables $x_1, \ldots, x_n$ and x_{R}. There is a corresponding replication of variables bound in contexts, which is made explicit in the following definition.

Definition 10 (translation of contexts)**.**

$$[\![\Gamma, x{:}A]\!] = [\![\Gamma]\!], \overline{x{:}A}, x_{\mathsf{R}}{:}[\![A]\!]\,\overline{x}$$

Note that each tuple $\overline{x : A}$ in the translated context must satisfy the relation $[\![A]\!]$, as witnessed by x_{R}. Thus, one may interpret $[\![\Gamma]\!]$ as n related environments.

We can then state our main result:

Theorem 11 (abstraction)**.**

$$\Gamma \vdash_S A : B \implies [\![\Gamma]\!] \vdash_{S^t} [\![A]\!] : [\![B]\!]\,\overline{A}$$

Proof. By induction on the derivation. Details of the proof are given in appendix A, page 51. □

The above theorem can be read in two ways. A direct reading is as a typing judgement about translated terms: if A has type B, then $[\![A]\!]$ has type $[\![B]\!]\,\overline{A}$. The more fruitful reading is as an abstraction theorem for pure type systems: if A has type B in environment Γ, then n interpretations $\overline{A}$ in related environments $[\![\Gamma]\!]$ are related by $[\![B]\!]$. Further, $[\![A]\!]$ is a witness of this proposition *within the type system*. In particular, closed terms are related to themselves: $\vdash A : B \implies \vdash [\![A]\!] : [\![B]\!]\,A \ldots A$.

3.3 Examples: the λ-cube

In this section, we show that $[\![\cdot]\!]$ specialises to the rules given by Reynolds (1983) to read a System F type as a relation. Having shown that our framework can explain parametricity theorems for System-F-style types, we move on to progressively higher-order constructs. In these examples, the binary version of parametricity is used (arity $n = 2$). We work here in

monochrome systems: $\mathcal{K} = \{1\}$, and annotations are omitted. Using Definition 4 one can verify that the following system reflects System F. Note that the sorts $\Box_1$ and $\Box_2$ come from the need of typing membership tests, and correspond to the sorts with the same name in CC_ω.

- $\mathcal{S} = \{\star, \Box, \Box_1, \lceil\star\rceil, \lceil\Box\rceil, \lceil\Box\rceil_1, \lceil\Box\rceil_2\}$
- $\mathcal{A} = \{\star : \Box, \Box : \Box_1, \lceil\star\rceil : \lceil\Box\rceil, \lceil\Box\rceil : \lceil\Box_1\rceil, \lceil\Box_1\rceil : \lceil\Box_2\rceil\}$
- $\mathcal{R} = \{\star \rightsquigarrow \star, \Box \rightsquigarrow \star, \star \rightsquigarrow \lceil\Box\rceil, \Box \rightsquigarrow \lceil\Box_1\rceil, \Box_1 \rightsquigarrow \lceil\Box_2\rceil, \lceil\star\rceil \rightsquigarrow \lceil\star\rceil, \lceil\Box\rceil \rightsquigarrow \lceil\star\rceil\}$

Types to relations Note that, by definition,

$$[\![\star]\!]\,\mathsf{T}_1\,\mathsf{T}_2 \;=\; \mathsf{T}_1 \to \mathsf{T}_2 \to \lceil\star\rceil$$

Here we use $\lceil\star\rceil$ on the right side as the sort of propositions. This means that types are translated to relations (as desired).

Function types Applying our translation to non-dependent function types, we get:

$$[\![A \to B]\!] : [\![\star]\!]\,(A \to B)\,(A \to B)$$
$$[\![A \to B]\!]\,\mathsf{f}_1\,\mathsf{f}_2 \;=\; \forall \mathsf{a}_1 : A.\ \forall \mathsf{a}_2 : A.\ [\![A]\!]\,\mathsf{a}_1\,\mathsf{a}_2 \to [\![B]\!]\,(\mathsf{f}_1\,\mathsf{a}_1)\,(\mathsf{f}_2\,\mathsf{a}_2)$$

That is, functions are related iff they take related arguments into related outputs. The rule is often found in the literature written in set-theoretic notation, which can be recovered by using the syntax described in Section 2.2.1 for the colours 0 and 1_i, and special treatment of pairs:

$$[\![A \to B]\!] = \{(f_1, f_2) | \forall a_1, a_2 . (a_1, a_2) \in [\![A]\!] \to (f_1\,a_1, f_2\,a_2) \in [\![B]\!]\}$$

Type schemes System F includes universal quantification of the form

$$\forall \mathsf{A} : \star.\ B.$$

Applying $[\![\cdot]\!]$ to this type expression yields:

$$[\![\forall \mathsf{A} : \star.\ B]\!] : [\![\star]\!]\,(\forall \mathsf{A} : \star.\ B)\,(\forall \mathsf{A} : \star.\ B)$$
$$[\![\forall \mathsf{A} : \star.\ B]\!]\,\mathsf{g}_1\,\mathsf{g}_2 \;=\; \forall \mathsf{A}_1 : \star.\ \forall \mathsf{A}_2 : \star.\ \forall \mathsf{A}_\mathsf{R} : [\![\star]\!]\,\mathsf{A}_1\,\mathsf{A}_2.\ [\![B]\!]\,(\mathsf{g}_1\,\mathsf{A}_1)\,(\mathsf{g}_2\,\mathsf{A}_2)$$

In words, polymorphic values are related iff instances at related types are related. Note that because A may occur free in B, the variables A_1, A_2 and A_R may occur free in $[\![B]\!]$.

Type constructors With the addition of the rule $\Box \rightsquigarrow \Box$, one can construct terms of type $\star \to \star$, which are sometimes known as type constructors, type formers or type-level functions. As Voigtländer (2009*b*) remarks, extending Reynolds-style parametricity to support type constructors appears to be folklore. Such folklore can be precisely justified by our framework by applying $[\![\cdot]\!]$ to obtain the relational counterpart of type constructors:

$$[\![\star \to \star]\!] : [\![\Box]\!]\,(\star \to \star)\,(\star \to \star)$$
$$[\![\star \to \star]\!]\,\mathsf{F}_1\,\mathsf{F}_2 \;=\; \forall \mathsf{A}_1 : \star.\ \forall \mathsf{A}_2 : \star.\ [\![\star]\!]\,\mathsf{A}_1\,\mathsf{A}_2 \to [\![\star]\!]\,(\mathsf{F}_1\,\mathsf{A}_1)\,(\mathsf{F}_2\,\mathsf{A}_2)$$

That is, a term of type $[\![\star \to \star]\!]\,\mathsf{F}_1\,\mathsf{F}_2$ is a (polymorphic) function converting a relation between any types A_1 and A_2 to a relation between $\mathsf{F}_1\,\mathsf{A}_1$ and $\mathsf{F}_2\,\mathsf{A}_2$, a *relational action*. For the target system to accept the above, the rules $\Box \rightsquigarrow \lceil\Box\rceil$ and $\lceil\Box\rceil \rightsquigarrow \lceil\Box\rceil$ must also be added there.

Dependent functions In a system with the rule $\star \rightsquigarrow \Box$, value variables may occur in dependent function types like $\forall x : A.\ B$, which we translate as follows:

$$[\![\forall x : A.\ B]\!] : [\![\star]\!]\,(\forall x : A.\ B)\,(\forall x : A.\ B)$$
$$[\![\forall x : A.\ B]\!]\,\mathsf{f}_1\,\mathsf{f}_2 \;=\; \forall x_1 : A.\ \forall x_2 : A.\ \forall x_R : [\![A]\!]\,x_1\,x_2.\ [\![B]\!]\,(\mathsf{f}_1\,x_1)\,(\mathsf{f}_2\,x_2)$$

Here, the target system is extended with the rule $\lceil\star\rceil \rightsquigarrow \lceil\Box\rceil$. The rule $\star \rightsquigarrow \lceil\Box\rceil$ is also required, but already in the system, as it is required in by the source axiom $\star : \Box$ as well.

Proof terms We have used $[\![\cdot]\!]$ to turn types into relations, but we can also use it to turn terms into proofs of abstraction properties. As a simple example, the relation corresponding to the type $\mathsf{T} = (\mathsf{A} : \star) \to \mathsf{A} \to \mathsf{A}$, namely

$$\begin{aligned}[\![\mathsf{T}]\!]\,\mathsf{f}_1\,\mathsf{f}_2 \;=\; &\forall \mathsf{A}_1 : \star.\ \forall \mathsf{A}_2 : \star.\ \forall \mathsf{A}_R : [\![\star]\!]\,\mathsf{A}_1\,\mathsf{A}_2.\\ &\forall x_1 : \mathsf{A}_1.\ \forall x_2 : \mathsf{A}_2.\ \mathsf{A}_R\,x_1\,x_2 \to \mathsf{A}_R\,(\mathsf{f}_1\,\mathsf{A}_1\,x_1)\,(\mathsf{f}_2\,\mathsf{A}_2\,x_2)\end{aligned}$$

states that functions of type T map related inputs to related outputs, for any relation. From a term $\mathsf{id} = \lambda \mathsf{A} : \star.\,\lambda x : \mathsf{A}.\,x$ of this type, by the abstraction theorem we obtain a term $[\![\mathsf{id}]\!] : [\![\mathsf{T}]\!]\,\mathsf{id}\,\mathsf{id}$, that is, a proof of the abstraction property:

$$[\![\mathsf{id}]\!]\,\mathsf{A}_1\,\mathsf{A}_2\,\mathsf{A}_R\,x_1\,x_2\,x_R \;=\; x_R$$

We return to proof terms in Section 4.3 after introducing datatypes.

3.4 Implicit syntax

In the following sections, our examples are written using Agda syntax, and take advantage of the implicit syntax feature. The following colour-set is used: $\mathcal{K} = \{e, i\}$ (e = explicit colour; i = implicit colour). Rather than using colour annotations, the following (Agda-style) concrete syntax is used.

Definition 12 (Agda-like syntax for two-colour PTS)**.**

$$\begin{array}{rcll} \mathcal{T} & = & \mathcal{C} & \text{constant} \\ & | & \mathcal{V} & \text{variable} \\ & | & \mathcal{T}\,\mathcal{T} & \text{application} \\ & | & \lambda\mathcal{V}{:}\mathcal{T}.\ \mathcal{T} & \text{abstraction} \\ & | & (\mathcal{V}{:}\mathcal{T}) \to \mathcal{T} & \text{dependent function space} \\ & | & \mathcal{T}\,\{\mathcal{T}\} & \text{implicit application} \\ & | & \lambda\{\mathcal{V}{:}\mathcal{T}\}.\ \mathcal{T} & \text{implicit abstraction} \\ & | & \{\mathcal{V}{:}\mathcal{T}\} \to \mathcal{T} & \text{implicit dependent function space} \end{array}$$

Additionally, implicit abstraction and application may be left out when the context allows it. We use the following colour-mappings:

$$\begin{array}{ll} 0 \mapsto e & \\ i_r \mapsto e & i_i \mapsto i \\ e_r \mapsto e & e_i \mapsto i \end{array}$$

In this case, $[\![\cdot]\!]$ specialises as follows, and the abstraction theorem is unchanged.

Definition 13 (translation from types to relations, specialised)**.**

$$\begin{array}{rcl} [\![s]\!] & = & \lambda\overline{\mathsf{x}{:}s}.\ \overline{\mathsf{x}} \to \lceil s \rceil \\ [\![\mathsf{x}]\!] & = & \mathsf{x}_\mathsf{R} \\ [\![(\mathsf{x}{:}A) \to B]\!] & = & \lambda\overline{\mathsf{f}{:}((\mathsf{x}{:}A) \to B)}.\ \{\overline{\mathsf{x}{:}A}\} \to (\mathsf{x}_\mathsf{R}{:}[\![A]\!]\,\overline{\mathsf{x}}) \to [\![B]\!]\,(\overline{\mathsf{f}\,\mathsf{x}}) \\ [\![F\,a]\!] & = & [\![F]\!]\,\{\overline{a}\}\,[\![a]\!] \\ [\![\lambda\mathsf{x}{:}A.\ b]\!] & = & \lambda\{\overline{\mathsf{x}{:}A}\}.\ \lambda\mathsf{x}_\mathsf{R}{:}[\![A]\!]\,\overline{\mathsf{x}}.\ [\![b]\!] \\ [\![\{\mathsf{x}{:}A\} \to B]\!] & = & \lambda\overline{\mathsf{f}{:}(\{\mathsf{x}{:}A\} \to B)}.\ \{\overline{\mathsf{x}{:}A}\} \to (\mathsf{x}_\mathsf{R}{:}[\![A]\!]\,\overline{\mathsf{x}}) \to [\![B]\!]\,(\overline{\mathsf{f}\,\{\mathsf{x}\}}) \\ [\![F\,\{a\}]\!] & = & [\![F]\!]\,\{\overline{a}\}\,[\![a]\!] \\ [\![\lambda\{\mathsf{x}{:}A\}.\ b]\!] & = & \lambda\{\overline{\mathsf{x}{:}A}\}.\ \lambda\mathsf{x}_\mathsf{R}{:}[\![A]\!]\,\overline{\mathsf{x}}.\ [\![b]\!] \end{array}$$

The usage of implicit syntax in the translation is not innocent: it is carefully designed to take advantage of the type-inference mechanism to allow shorter expressions of the translations. For example, $[\![\mathsf{id}]\!]$, generated from $\mathsf{id} : \mathsf{T}$ can now hide four out of six abstractions:

$$[\![\mathsf{id}]\!]\,\mathsf{A}_\mathsf{R}\,\mathsf{x}_\mathsf{R} \;=\; \mathsf{x}_\mathsf{R}$$

In general, we observe the following:

Observation 14. *For any term A of type B, given the type annotation $[\![A]\!] : [\![B]\!]\,\overline{A}$, then arguments may be omitted at every implicit application in the expansion of $[\![A]\!]$. Every implicit abstraction is inferable as well.*

4 Constants and datatypes

While the above development assumes input pure type systems with $\mathcal{C} = \mathcal{S}$, it is possible to add constants to the system and retain parametricity, as long as each constant is parametric. That is, for each new axiom $\vdash_\mathcal{S} \mathfrak{c} : A$ (where $\mathfrak{c}$ is an arbitrary constant and A an arbitrary term not a mere sort) we require a term $[\![\mathfrak{c}]\!]$ such that the judgement $\vdash_{\mathcal{S}^t} [\![\mathfrak{c}]\!] : [\![A]\!]\,\bar{\mathfrak{c}}$ holds. (Assuming additional β-conversion rules involving those constants preserve types.)

One source of constants in many languages is datatype definitions. In the rest of this section we investigate the implications of the parametricity conditions on datatypes, and give two translation schemes for inductive families (as an extension of I_ω).

4.1 Parametricity and elimination

Reynolds (1983) and Wadler (1989) assume that each type constant $\mathsf{K} : \star$ is translated to the identity relation. This definition is certainly compatible with the condition required by Theorem 11 for such constants: $[\![\mathsf{K}]\!] : [\![\star]\!]\,\mathsf{K}\,\mathsf{K}$, but so are many other relations. Are we missing some restriction for constants? This question might be answered by resorting to a translation to pure terms via Church encodings (Böhm and Berarducci, 1985), as Wadler (2007) does. However, in the hope to shed a different light on the issue, we give another explanation, using our machinery.

Consider a base type, such as $\mathsf{Bool} : \star$, equipped with constructors $\mathsf{true} : \mathsf{Bool}$ and $\mathsf{false} : \mathsf{Bool}$. In order to derive parametricity theorems in a system containing such a constant Bool, we must define $[\![\mathsf{Bool}]\!]$, satisfying $\vdash [\![\mathsf{Bool}]\!] : [\![\star]\!]\,\overline{\mathsf{Bool}}$. What are the restrictions put on the term $[\![\mathsf{Bool}]\!]$? First, we must be able to define $[\![\mathsf{true}]\!] : [\![\mathsf{Bool}]\!]\,\overline{\mathsf{true}}$. Therefore, $[\![\mathsf{Bool}]\!]\,\overline{\mathsf{true}}$ must be inhabited. The same reasoning holds for the false case.

Second, to write any useful program using Booleans, a way to test their value is needed. This may be done by adding a constant

$$\mathsf{if} : \mathsf{Bool} \to (\mathsf{A} : \star) \to \mathsf{A} \to \mathsf{A} \to \mathsf{A}$$

such that $\mathsf{if}\,\mathsf{true}\,\mathsf{A}\,\mathsf{x}\,\mathsf{y} \longrightarrow_\beta \mathsf{x}$ and $\mathsf{if}\,\mathsf{false}\,\mathsf{A}\,\mathsf{x}\,\mathsf{y} \longrightarrow_\beta \mathsf{y}$.

Now, if a program uses if, we must also define $[\![\mathsf{if}]\!]$ of type

$$[\![\mathsf{Bool} \to (\mathsf{A} : \star) \to \mathsf{A} \to \mathsf{A} \to \mathsf{A}]\!]\,\overline{\mathsf{if}}$$

for parametricity to work. Let us expand the type of $[\![\mathsf{if}]\!]$ and attempt to give a definition case by case:

$$\begin{aligned}
&[\![\mathsf{if}]\!] : \{\mathsf{b}_1\,\mathsf{b}_2 : \mathsf{Bool}\} \to (\mathsf{b}_\mathsf{R} : [\![\mathsf{Bool}]\!]\,\mathsf{b}_1\,\mathsf{b}_2) \to \\
&\qquad \{\mathsf{A}_1\,\mathsf{A}_2 : \star\} \to (\mathsf{A}_\mathsf{R} : [\![\star]\!]\,\mathsf{A}_1\,\mathsf{A}_2) \to \\
&\qquad \{\mathsf{x}_1 : \mathsf{A}_1\} \to \{\mathsf{x}_2 : \mathsf{A}_2\} \to (\mathsf{x}_\mathsf{R} : \mathsf{A}_\mathsf{R}\,\mathsf{x}_1\,\mathsf{x}_2) \to \\
&\qquad \{\mathsf{y}_1 : \mathsf{A}_1\} \to \{\mathsf{y}_2 : \mathsf{A}_2\} \to (\mathsf{y}_\mathsf{R} : \mathsf{A}_\mathsf{R}\,\mathsf{y}_1\,\mathsf{y}_2) \to \\
&\qquad \mathsf{A}_\mathsf{R}\,(\mathsf{if}\,\mathsf{b}_1\,\mathsf{A}_1\,\mathsf{x}_1\,\mathsf{y}_1)\,(\mathsf{if}\,\mathsf{b}_2\,\mathsf{A}_2\,\mathsf{x}_2\,\mathsf{y}_2) \\
&[\![\mathsf{if}]\!]\,\{\mathsf{true}\}\,\{\mathsf{true}\}\,\mathsf{b}_\mathsf{R}\,_\,\mathsf{x}_\mathsf{R}\,\mathsf{y}_\mathsf{R} = \mathsf{x}_\mathsf{R} \\
&[\![\mathsf{if}]\!]\,\{\mathsf{true}\}\,\{\mathsf{false}\}\,\mathsf{b}_\mathsf{R}\,_\,\mathsf{x}_\mathsf{R}\,\mathsf{y}_\mathsf{R} = ?_{tf} \\
&[\![\mathsf{if}]\!]\,\{\mathsf{false}\}\,\{\mathsf{true}\}\,\mathsf{b}_\mathsf{R}\,_\,\mathsf{x}_\mathsf{R}\,\mathsf{y}_\mathsf{R} = ?_{ft} \\
&[\![\mathsf{if}]\!]\,\{\mathsf{false}\}\,\{\mathsf{false}\}\,\mathsf{b}_\mathsf{R}\,_\,\mathsf{x}_\mathsf{R}\,\mathsf{y}_\mathsf{R} = \mathsf{y}_\mathsf{R}
\end{aligned}$$

(From this example onwards, we use a layout convention to ease the reading of translated types: each triple of arguments, corresponding to one argument in the original function, is written on its own line if space permits.)

In order to complete the above definition, we must provide a type-correct expression for each question mark. For $?_{tf}$, this means that we must construct an expression of type $\mathsf{A}_\mathsf{R}\,\mathsf{x}_1\,\mathsf{y}_2$. Neither $\mathsf{x}_\mathsf{R} : \mathsf{A}_\mathsf{R}\,\mathsf{x}_1\,\mathsf{x}_2$ nor $\mathsf{y}_\mathsf{R} : \mathsf{A}_\mathsf{R}\,\mathsf{y}_1\,\mathsf{y}_2$ can help us here. The only liberty left is in $\mathsf{b}_\mathsf{R} : [\![\mathsf{Bool}]\!]\,\mathsf{true}\,\mathsf{false}$. If we let $[\![\mathsf{Bool}]\!]\,\mathsf{true}\,\mathsf{false}$ be falsity ($\bot$), then this case can never be reached and we need not give an equation for it. This reasoning holds symmetrically for $?_{ft}$. Therefore, we have the restrictions:

$$\begin{aligned}
&[\![\mathsf{Bool}]\!]\,\mathsf{x}\,\mathsf{x} = \text{some inhabited type} \\
&[\![\mathsf{Bool}]\!]\,\mathsf{x}\,\mathsf{y} = \bot \quad \text{if } \mathsf{x} \neq \mathsf{y}
\end{aligned}$$

We have some freedom regarding picking "some inhabited type", so we choose $[\![\mathsf{Bool}]\!]\,\mathsf{x}\,\mathsf{x}$ to be truth ($\top$), making $[\![\mathsf{Bool}]\!]$ an encoding of the identity relation.

An intuition behind parametricity is that, when programs "know" more about a type, the parametricity condition becomes stronger. The above example illustrates how this intuition can be captured within our framework.

4.2 Inductive families

Many languages permit datatype declarations like those in Figure 1.2. Dependently typed languages typically allow the return types of constructors to have different arguments, yielding *inductive families* (Paulin-Mohring, 1993; Dybjer, 1994) such as the family Vec, in which the type is indexed by the number of elements.

```
data ⊥ : ⋆ where
  -- no constructors
data ⊤ : ⋆ where
  tt : ⊤
data Bool : ⋆ where
  false : Bool
  true  : Bool
data ℕ : ⋆ where
  zero : ℕ
  succ : ℕ → ℕ
```

```
data List (A : ⋆) : ⋆ where
  nil  : List A
  cons : A → List A → List A
data Vec (A : ⋆) : ℕ → ⋆ where
  nilV  : Vec A zero
  consV : A → (n : ℕ) → Vec A n →
          Vec A (succ n)
data Σ (A : ⋆) (B : A → ⋆) : ⋆ where
  _,_ : (a : A) → B a → Σ A B
data _ ≡ _ {A : ⋆} (a : A) : A → ⋆ where
  refl : a ≡ a
```

Figure 1.2: Examples of simple datatypes and inductive families

Data family declarations of sort s ($\star$ in the examples) have the typical form:[3]

$$\begin{array}{c}\textbf{data}\ \mathfrak{T}\,(\mathsf{a}{:}A) : \forall \mathsf{n}{:}N.\ s\ \textbf{where}\\ \mathfrak{c} : \forall \mathsf{b}{:}B.\ (\forall \mathsf{x}{:}X.\ \mathfrak{T}\,\mathsf{a}\,i) \to \mathfrak{T}\,\mathsf{a}\,v\end{array}$$

Arguments of the type constructor $\mathfrak{T}$ may be either parameters a, which scope over the constructors and are repeated at each recursive use of $\mathfrak{T}$, or indices n, which may vary between uses. Data constructors $\mathfrak{c}$ have non-recursive arguments b, whose types are otherwise unrestricted, and recursive arguments with types of a constrained form ($\mathfrak{T}$ cannot appear in X), which cannot be referred to in the other terms.

Such a declaration can be interpreted as a simultaneous declaration of formation and introduction constants

$$\begin{array}{l}\mathfrak{T} : \forall \mathsf{a}{:}A.\ \forall \mathsf{n}{:}N.\ s\\ \mathfrak{c}\ : \forall \{\mathsf{a}{:}A\}.\ \forall \mathsf{b}{:}B.\ (\forall \mathsf{x}{:}X.\ \mathfrak{T}\,\mathsf{a}\,i) \to \mathfrak{T}\,\mathsf{a}\,v\end{array}$$

and also an eliminator to analyse values of that type:

$$\begin{array}{l}\mathfrak{T}\text{-elim} : \forall \{\mathsf{a}{:}A\}.\\ \qquad\qquad \forall \mathsf{P}{:}(\forall \mathsf{n}{:}N.\ \mathfrak{T}\,\mathsf{a}\,\mathsf{n} \to s).\\ \qquad\qquad \mathsf{Case}_{\mathfrak{c}} \to \forall \mathsf{n}{:}N.\ \forall \mathsf{t}{:}\mathfrak{T}\,\mathsf{a}\,\mathsf{n}.\ \mathsf{P}\,\mathsf{n}\,\mathsf{t}\end{array}$$

where the type $\mathsf{Case}_{\mathfrak{c}}$ of the case for each constructor $\mathfrak{c}$ is

$$\forall \mathsf{b}{:}B.\ \forall \mathsf{u}{:}(\forall \mathsf{x}{:}X.\ \mathfrak{T}\,\mathsf{a}\,i).\ (\forall \mathsf{x}{:}X.\ \mathsf{P}\,i\,(\mathsf{u}\,\mathsf{x})) \to \mathsf{P}\,v\,(\mathfrak{c}\,\{\mathsf{a}\}\,\mathsf{b}\,\mathsf{u})$$

[3] We show only one of each element here, but the generalisation to arbitrary numbers is straightforward.

with beta-equivalences (one for each constructor c):

$$\mathfrak{T}\text{-elim}\,\{\mathsf{a}\}\,\mathsf{P}\,\mathsf{e}\,v\,(\mathsf{c}\,\{\mathsf{a}\}\,\mathsf{b}\,\mathsf{u}) = \mathsf{e}\,\mathsf{b}\,\mathsf{u}\,(\lambda \mathsf{x}{:}X.\ \mathfrak{T}\text{-elim}\,\{\mathsf{a}\}\,\mathsf{P}\,\mathsf{e}\,i\,(\mathsf{u}\,\mathsf{x})) \tag{1.1}$$

We shall often use corresponding pattern matching definitions instead of these eliminators (Coquand, 1992).

For example, the definition of List in Figure 1.2 gives rise to the following constants:

$$\begin{array}{ll}
\mathsf{List} & : (\mathsf{A} : \star) \to \star \\
\mathsf{nil} & : \{\mathsf{A} : \star\} \to \mathsf{List\ A} \\
\mathsf{cons} & : \{\mathsf{A} : \star\} \to \mathsf{A} \to \mathsf{List\ A} \to \mathsf{List\ A} \\
\mathsf{List\text{-}elim} & : \{\mathsf{A} : \star\} \to (\mathsf{P} : \mathsf{List\ A} \to \star) \to \\
& \quad \mathsf{P\ nil} \to \\
& \quad ((\mathsf{x} : \mathsf{A}) \to (\mathsf{xs} : \mathsf{List\ A}) \to \mathsf{P\ xs} \to \mathsf{P}\,(\mathsf{cons\ x\ xs})) \to \\
& \quad (\mathsf{ys} : \mathsf{List\ A}) \to \mathsf{P\ ys}
\end{array}$$

In the following sections, we consider two ways to define a proof term $[\![\mathsf{c}]\!] : [\![T]\!]\,\mathsf{c}\ldots\mathsf{c}$ for each constant $\mathsf{c} : T$ introduced by the data definition.

4.3 Deductive-style translation

First, we define each proof as a term (using pattern matching to simplify the presentation). We begin with the translation of the equation for each constructor:

$$[\![\mathfrak{T}\text{-elim}\ \mathsf{a}\ \mathsf{P}\ \mathsf{e}\ v]\!]\ \overline{(\mathsf{c}\,\{\mathsf{a}\}\,\mathsf{b}\,\mathsf{u})}\ ([\![\mathsf{c}]\!]\,\{\overline{\mathsf{a}}\}\,\mathsf{a}_\mathsf{R}\,\{\overline{\mathsf{b}}\}\,\mathsf{b}_\mathsf{R}\,\{\overline{\mathsf{u}}\}\,\mathsf{u}_\mathsf{R}) = [\![RHS]\!]$$

for $RHS = \mathsf{e}\,\mathsf{b}\,\mathsf{u}\,(\lambda \mathsf{x}{:}X.\ \mathfrak{T}\text{-elim}\,\{\mathsf{a}\}\,\mathsf{P}\,\mathsf{e}\,i\,(\mathsf{u}\,\mathsf{x}))$. To turn this into a pattern matching definition of $\mathfrak{T}$-elim, we need a suitable definition of $[\![\mathsf{c}]\!]$, and similarly for the constructors in v. The only arguments of $[\![\mathsf{c}]\!]$ not already in scope are b_R and u_R, so we package them as a dependent pair, because the type of u_R may depend on that of b_R. Writing $(\mathsf{x} : A) \times B$ for $\Sigma\,A\,(\lambda \mathsf{x}{:}A.\ B)$, we define

$$\begin{array}{lll}
[\![\mathfrak{T}]\!] : [\![\forall \mathsf{a}{:}A.\ \forall \mathsf{n}{:}N.\ s]\!]\,\overline{\mathfrak{T}} & & \\
[\![\mathfrak{T}]\!]\,\{\overline{\mathsf{a}}\}\,\mathsf{a}_\mathsf{R}\,\{\overline{v}\}\,[\![v]\!]\ \ \overline{(\mathsf{c}\,\{\mathsf{a}\}\,\mathsf{b}\,\mathsf{u})} & = & (\mathsf{b}_\mathsf{R} : [\![B]\!]\,\overline{b}) \times [\![\forall \mathsf{x}{:}X.\ \mathfrak{T}\,\mathsf{a}\,i]\!]\,\overline{\mathsf{u}} \\
[\![\mathfrak{T}]\!]\,\{\overline{\mathsf{a}}\}\,\mathsf{a}_\mathsf{R}\,\{\overline{\mathsf{u}}\}\,\mathsf{u}_\mathsf{R}\ \ \overline{\mathsf{t}} & = & \bot \\
[\![\mathsf{c}]\!] : [\![\forall \mathsf{a}{:}A.\ \forall \mathsf{b}{:}B.\ (\forall \mathsf{x}{:}X.\ \mathfrak{T}\,\mathsf{a}\,i) \to \mathfrak{T}\,\mathsf{a}\,v]\!]\,\overline{\mathsf{c}} & & \\
[\![\mathsf{c}]\!]\,\{\overline{\mathsf{a}}\}\,\mathsf{a}_\mathsf{R}\,\{\overline{\mathsf{b}}\}\,\mathsf{b}_\mathsf{R}\,\{\overline{\mathsf{u}}\}\,\mathsf{u}_\mathsf{R} & = & (\mathsf{b}_\mathsf{R}, \mathsf{u}_\mathsf{R})
\end{array}$$

and the translation of $\mathfrak{T}$-elim becomes

$$[\![\mathfrak{T}\text{-elim}\ \mathsf{a}\ \mathsf{P}\ \mathsf{e}\ v]\!]\ \overline{(\mathsf{c}\,\{\mathsf{a}\}\,\mathsf{b}\,\mathsf{u})}\ (\mathsf{b}_\mathsf{R}, \mathsf{u}_\mathsf{R}) = [\![RHS]\!]$$

Because $[\![\mathfrak{T}]\!]$ yields $\bot$ unless the constructors match, these clauses provide complete coverage.

Booleans To get an intuition of the meaning of the above translation scheme we proceed to apply it to a number of examples. For Booleans, the generic schema specialises to the definitions given in Section 4.1.

Lists and vectors From the definition of List in Figure 1.2, we have the constant $\mathsf{List} : \star \to \star$, so List is an example of a type constructor, and thus $[\![\mathsf{List}]\!]$ is a relation transformer. The relation transformer we get by applying our scheme is exactly that given by Wadler (1989): lists are related iff their lengths are equal and their elements are related point-wise.

$$
\begin{array}{l}
[\![\mathsf{List}]\!] : [\![\star \to \star]\!]\ \mathsf{List}\ \mathsf{List} \\
[\![\mathsf{List}]\!]\ \mathsf{A_R}\ \mathsf{nil}\ \quad\quad \mathsf{nil}\ \quad\quad = \top \\
[\![\mathsf{List}]\!]\ \mathsf{A_R}\ (\mathsf{cons}\ \mathsf{x_1}\ \mathsf{xs_1})\ (\mathsf{cons}\ \mathsf{x_2}\ \mathsf{xs_2}) = \mathsf{A_R}\ \mathsf{x_1}\ \mathsf{x_2}\ \times\ [\![\mathsf{List}]\!]\ \mathsf{A_R}\ \mathsf{xs_1}\ \mathsf{xs_2} \\
[\![\mathsf{List}]\!]\ \mathsf{A_R}\ \_\ \quad\quad \_\ \quad\quad = \bot \\
[\![\mathsf{nil}]\!] : [\![(\mathsf{A} : \star) \to \mathsf{List\ A}]\!]\ \mathsf{nil}\ \mathsf{nil} \\
[\![\mathsf{nil}]\!]\ \mathsf{A_R} = \mathsf{tt} \\
[\![\mathsf{cons}]\!] : [\![(\mathsf{A} : \star) \to \mathsf{A} \to \mathsf{List\ A} \to \mathsf{List\ A}]\!]\ \mathsf{cons}\ \mathsf{cons} \\
[\![\mathsf{cons}]\!]\ \mathsf{A_R}\ \mathsf{x_R}\ \mathsf{xs_R} = (\mathsf{x_R}, \mathsf{xs_R})
\end{array}
$$

(We use $\top$ for nullary constructors as it is the identity of $\times$.) The translations of the constants of Vec are given in Figure 1.3.

List rearrangements The first example of a parametric type examined by Wadler (1989) is the type of list rearrangements: $\mathsf{R} = (\mathsf{A} : \star) \to \mathsf{List\ A} \to \mathsf{List\ A}$. Intuitively, functions of type R know nothing about the actual argument type A, and therefore they can only produce the output list by taking elements from the input list. In this section we recover that result as an instance of Theorem 11.

Applying the translation to R yields:

$$
\begin{array}{l}
[\![\mathsf{R}]\!] : \mathsf{R} \to \mathsf{R} \to \star \\
[\![\mathsf{R}]\!]\ \mathsf{r_1}\ \mathsf{r_2} = \{\mathsf{A_1}\ \mathsf{A_2} : \star\} \to (\mathsf{A_R} : [\![\star]\!]\ \mathsf{A_1}\ \mathsf{A_2}) \to \\
\qquad\qquad \{\mathsf{xs_1} : \mathsf{List\ A_1}\} \to \{\mathsf{xs_2} : \mathsf{List\ A_2}\} \to (\mathsf{xs_R} : [\![\mathsf{List}]\!]\ \mathsf{A_R}\ \mathsf{xs_1}\ \mathsf{xs_2}) \to \\
\qquad\qquad [\![\mathsf{List}]\!]\ \mathsf{A_R}\ (\mathsf{r_1}\ \mathsf{A_1}\ \mathsf{xs_1})\ (\mathsf{r_2}\ \mathsf{A_2}\ \mathsf{xs_2})
\end{array}
$$

In words: two list rearrangements $\mathsf{r_1}$ and $\mathsf{r_2}$ are related iff for all types $\mathsf{A_1}$ and $\mathsf{A_2}$ with relation $\mathsf{A_R}$, and for all lists $\mathsf{xs_1}$ and $\mathsf{xs_2}$ point-wise related by $\mathsf{A_R}$, the resulting lists $\mathsf{r_1\ A_1\ xs_1}$ and $\mathsf{r_2\ A_2\ xs_2}$ are also point-wise related by $\mathsf{A_R}$. By Theorem 11, $[\![\mathsf{R}]\!]\ \mathsf{r}\ \mathsf{r}$ holds for any term r of type R. This means that applying r preserves (point-wise) any relation existing between input lists

$$
\begin{array}{l}
[\![\mathsf{Vec}]\!] : [\![(\mathsf{A} : \star) \to \mathbb{N} \to \star]\!]\ \overline{\mathsf{Vec}} \\
[\![\mathsf{Vec}]\!]\ \mathsf{A_R}\ \mathsf{n_R}\ \mathsf{nilV} \qquad\qquad\quad \mathsf{nilV} \ = \ \top \\
[\![\mathsf{Vec}\,]\!]\ \mathsf{A_R}\ (\mathsf{succ}\ \mathsf{n}_1)\ (\mathsf{succ}\ \mathsf{n}_2)\ \mathsf{n_R}\ (\mathsf{consV}\ \mathsf{n}_1\ \mathsf{x}_1\ \mathsf{xs}_1)\ (\mathsf{consV}\ \mathsf{n}_2\ \mathsf{x}_2\ \mathsf{xs}_2) \ = \\
\quad \mathsf{A_R}\ \mathsf{x}_1\ \mathsf{x}_2 \ \times \ (\mathsf{n_R} : [\![\mathsf{Nat}]\!]\ \mathsf{n}_1\ \mathsf{n}_2) \ \times \ [\![\mathsf{Vec}]\!]\ \mathsf{A_R}\ \mathsf{n_R} \\
[\![\mathsf{Vec}]\!]\ \mathsf{A_R}\ \mathsf{n_R}\ \mathsf{xs}_1 \qquad\qquad\quad \mathsf{xs}_2 \ = \ \bot \\
\\
[\![\mathsf{nilV}]\!] : [\![\{\mathsf{A} : \star\} \to \mathsf{Vec\ A\ zero}]\!]\ \overline{\mathsf{nilV}} \\
[\![\mathsf{nilV}]\!]\ \mathsf{A_R} \ = \ \mathsf{tt} \\
\\
[\![\mathsf{consV}]\!] : [\![\{\mathsf{A} : \star\} \to \mathsf{A} \to (\mathsf{n} : \mathbb{N}) \to \mathsf{Vec\ A\ n} \to \mathsf{Vec\ A\ (succ\ n)}]\!]\ \overline{\mathsf{consV}} \\
[\![\mathsf{consV}]\!]\ \mathsf{A_R}\ \mathsf{x_R}\ \mathsf{n_R}\ \mathsf{xs_R} \ = \ (\mathsf{x_R}, (\mathsf{n_R}, \mathsf{xs_R})) \\
\\
[\![\mathsf{Vec\text{-}elim}]\!] : [\![\ \{\mathsf{A} : \star\} \to \\
\qquad\qquad (\mathsf{P} : (\mathsf{n} : \mathbb{N}) \to \mathsf{Vec\ n\ A} \to \star) \to \\
\qquad\qquad (\mathsf{en} : \mathsf{P\ zero\ (nilV\ A)}) \to \\
\qquad\qquad (\mathsf{ec} : \qquad (\mathsf{x} : \mathsf{A}) \to (\mathsf{n} : \mathbb{N}) \to (\mathsf{xs} : \mathsf{Vec\ n\ A}) \to \\
\qquad\qquad\qquad\qquad\ \mathsf{P\ n\ xs} \to \mathsf{P\ (succ\ n)\ (consV\ x\ n\ xs)}) \to \\
\qquad\qquad (\mathsf{n} : \mathbb{N}) \to (\mathsf{v} : \mathsf{Vec\ n\ A}) \to \mathsf{P\ n\ v}]\!]\ \overline{\mathsf{Vec\text{-}elim}} \\
[\![\mathsf{Vec\text{-}elim}]\!]\ \mathsf{A_R}\ \mathsf{P_R}\ \mathsf{en_R}\ \mathsf{ec_R}\ \_ \ \{\mathsf{nilV}\} \qquad\quad \{\mathsf{nilV}\}\ \_ \ = \ \mathsf{en_R} \\
[\![\mathsf{Vec\text{-}elim}]\!]\ \mathsf{A_R}\ \mathsf{P_R}\ \mathsf{en_R}\ \mathsf{ec_R}\ \mathsf{n_R}\ \{\mathsf{consV}\ \mathsf{x}_1\ \mathsf{n}_1\ \mathsf{xs}_1\}\ \{\mathsf{consV}\ \mathsf{x}_2\ \mathsf{n}_2\ \mathsf{xs}_2\} \\
\qquad\qquad (\mathsf{x_R}, (\mathsf{n_R}, \mathsf{xs_R})) \\
\qquad\qquad = \ \mathsf{ec_R}\ \mathsf{x_R}\ \mathsf{n_R}\ \mathsf{xs_R}\ ([\![\mathsf{Vec\text{-}elim}]\!]\ \mathsf{A_R}\ \mathsf{P_R}\ \mathsf{en_R}\ \mathsf{ec_R}\ \mathsf{n_R}\ \mathsf{xs_R})
\end{array}
$$

Figure 1.3: Deductive translation of Vec constants. ($[\![\mathbb{N}]\!]$ is the identity relation.)

of equal length. By specialising $\mathsf{A_R}$ to a function ($\mathsf{A_R\,a_1\,a_2 = f\,a_1 \equiv a_2}$) we obtain the well-known result:

$$\begin{aligned}&(\mathsf{A_1\,A_2} : \star) \to (\mathsf{f} : \mathsf{A_1} \to \mathsf{A_2}) \to (\mathsf{xs} : \mathsf{List\,A_1}) \to \\ &\quad \mathsf{map\,f\,(r\,A_1\,xs) \equiv r\,A_2\,(map\,f\,xs)}\end{aligned}$$

(This form relies on the facts that $[\![\mathsf{List}]\!]$ preserves identities and composes with map.)

Dependent pair It might be worthwhile noting that a dependent pair translates to another dependent pair. That is, a pair $(a, b) : \Sigma\ A\ B$ translates to

$$([\![a]\!], [\![b]\!]) : [\![\Sigma]\!][\![A]\!][\![B]\!](a_1, b_1)(a_2, b_2)$$

where

$$\begin{aligned}&[\![\Sigma]\!] : \{\mathsf{A_1\,A_2} : \star\}\,(\mathsf{A_R} : [\![\star]\!]\,\mathsf{A_1\,A_2}) \\ &\qquad \{\mathsf{B_1} : \mathsf{A_1} \to \star\}\,\{\mathsf{B_2} : \mathsf{A_2} \to \star\} \\ &\qquad (\mathsf{B_R} : \{\mathsf{a_1} : \mathsf{A_1}\}\,\{\mathsf{a_2} : \mathsf{A_2}\} \to \mathsf{A_R\,a_1\,a_2} \to [\![\star]\!]\,(\mathsf{B_1\,a_1})\,(\mathsf{B_2\,a_2})) \\ &[\![\Sigma]\!]\,\mathsf{A_R\,B_R}\,(\mathsf{a_1}, \mathsf{b_1})\,(\mathsf{a_2}, \mathsf{b_2}) = \Sigma\,(\mathsf{A_R\,a_1\,a_2})\,(\lambda\,\mathsf{a_R} \to \mathsf{B_R\,a_R\,b_1\,b_2})\end{aligned}$$

Proof terms We have seen that applying $[\![\cdot]\!]$ to a type yields a parametricity property for terms of that type, and by Theorem 11 we can also apply $[\![\cdot]\!]$ to a term of that type to obtain a proof of the property. As an example, consider a rearrangement function odds that returns every second element from a list:

$$\begin{aligned}&\mathsf{odds} : (\mathsf{A} : \star) \to \mathsf{List\,A} \to \mathsf{List\,A} \\ &\mathsf{odds\,A\,nil} &&= \mathsf{nil} \\ &\mathsf{odds\,A\,(cons\,x\,nil)} &&= \mathsf{cons\,x\,nil} \\ &\mathsf{odds\,A\,(cons\,x\,(cons\,\_\,xs))} &&= \mathsf{cons\,x\,(odds\,A\,xs)}\end{aligned}$$

Any list rearrangement function must satisfy the parametricity condition $[\![\mathsf{R}]\!]$ seen above, and $[\![\mathsf{odds}]\!]$ is a proof that odds satisfies parametricity. Expanding it yields:

$$\begin{aligned}&[\![\mathsf{odds}]\!] : [\![(\mathsf{A} : \star) \to \mathsf{List\,A} \to \mathsf{List\,A}]\!]\,\mathsf{odds\,odds} \\ &[\![\mathsf{odds}]\!]\,\mathsf{A_R}\,\{\mathsf{nil}\}\,\{\mathsf{nil}\}\,\_ = \mathsf{tt} \\ &[\![\mathsf{odds}]\!]\,\mathsf{A_R}\,\{\mathsf{cons\,x_1\,nil}\}\,\{\mathsf{cons\,x_2\,nil}\}\,(\mathsf{x_R}, \_) = (\mathsf{x_R}, \mathsf{tt}) \\ &[\![\mathsf{odds}]\!]\,\mathsf{A_R}\,\{\mathsf{cons\,x_1\,(cons\,\_\,xs_1)}\}\,\{\mathsf{cons\,x_2\,(cons\,\_\,xs_2)}\}\,(\mathsf{x_R}, (\_, \mathsf{xs_R})) = \\ &\quad (\mathsf{x_R}, [\![\mathsf{odds}]\!]\,\mathsf{A_R}\,\{\mathsf{xs_1}\}\,\{\mathsf{xs_2}\}\,\mathsf{xs_R})\end{aligned}$$

We see that $[\![\mathsf{odds}]\!]$ performs essentially the same computation as odds, on two lists in parallel. However, instead of building a new list, it keeps track of the relations (in the R-subscripted variables). This behaviour stems from the last two cases in the definition of $[\![\mathsf{odds}]\!]$. Performing such a computation is enough to prove the parametricity condition.

4.4 Inductive-style translation

Inductive definitions offer another way of defining the translations $[\![\mathfrak{c}]\!]$ of the constants associated with a datatype, an *inductive* definition in contrast to the *deductive* definitions of the previous section. Given an inductive family

$$\begin{array}{l}\textbf{data}\ \mathfrak{T}\,(\mathsf{a}:A):K\ \textbf{where}\\ \quad \mathfrak{c}:C\end{array}$$

by applying our translation to the components of the **data**-declaration, we obtain an inductive family that defines the relational counterparts of the original type $\mathfrak{T}$ and its constructors $\mathfrak{c}$ at the same time:

$$\begin{array}{l}\textbf{data}\ [\![\mathfrak{T}]\!]\,([\![\mathsf{a}:A]\!]):[\![K]\!]\,\overline{(\mathfrak{T}\,\mathsf{a})}\ \textbf{where}\\ \quad [\![\mathfrak{c}]\!]:[\![C]\!]\,\overline{(\mathfrak{c}\,\{\mathsf{a}\})}\end{array}$$

It remains to supply a proof term for the parametricity of the elimination constant $\mathfrak{T}$-elim. If the inductive family has the form

$$\begin{array}{l}\textbf{data}\ \mathfrak{T}\,(\mathsf{a}:A):\forall\mathsf{n}:N.\ s\ \textbf{where}\\ \quad \mathfrak{c}:\forall\mathsf{b}:B.\ (\forall\mathsf{x}:X.\ \mathfrak{T}\,\mathsf{a}\,i)\to\mathfrak{T}\,\mathsf{a}\,v\end{array}$$

then its translation is

$$\begin{array}{l}\textbf{data}\ [\![\mathfrak{T}]\!]\,(\overline{\mathsf{a}:A})\,(\mathsf{a}_\mathsf{R}:[\![A]\!]\,\bar{\mathsf{a}}):\{\overline{\mathsf{n}:N}\}\to(\mathsf{n}_\mathsf{R}:[\![N]\!]\,\bar{\mathsf{n}})\to\overline{\mathfrak{T}\,\mathsf{a}\,\mathsf{n}}\to\lceil s\rceil\ \textbf{where}\\ \quad [\![\mathfrak{c}]\!]:\{\overline{\mathsf{b}:B}\}\to(\mathsf{b}_\mathsf{R}:[\![B]\!]\,\bar{\mathsf{b}})\to[\![(\forall\mathsf{x}:X.\ \mathfrak{T}\,\mathsf{a}\,i)\to\mathfrak{T}\,\mathsf{a}\,v]\!]\,(\mathfrak{c}\,\{\mathsf{a}\}\,\overline{b})\end{array}$$

and the elimination constant for $[\![\mathfrak{T}]\!]$ to sort $\lceil s_e\rceil$ has type

$$\begin{array}{l}[\![\mathfrak{T}]\!]\text{-elim}:\{\overline{\mathsf{a}:A}\}\to\{\mathsf{a}_\mathsf{R}:[\![A]\!]\,\bar{\mathsf{a}}\}\to\\ \qquad(\mathsf{Q}:\{\overline{\mathsf{n}:N}\}\to(\mathsf{n}_\mathsf{R}:[\![N]\!]\,\bar{\mathsf{n}})\to(\overline{\mathsf{t}:\mathfrak{T}\,\mathsf{a}\,\mathsf{n}})\to[\![\mathfrak{T}\,\mathsf{a}\,\mathsf{n}]\!]\,\bar{\mathsf{t}}\to\lceil s_e\rceil)\to\\ \qquad\mathsf{Case}_{[\![\mathfrak{c}]\!]}\to\\ \qquad\{\overline{\mathsf{n}:N}\}\to(\mathsf{n}_\mathsf{R}:[\![N]\!]\,\bar{\mathsf{n}})\to(\overline{\mathsf{t}:\mathfrak{T}\,\mathsf{a}\,\mathsf{n}})\to(\mathsf{t}_\mathsf{R}:[\![\mathfrak{T}\,\mathsf{a}\,\mathsf{n}]\!]\,\bar{\mathsf{t}})\to\mathsf{Q}\,\{\bar{\mathsf{n}}\}\,\mathsf{n}_\mathsf{R}\,\bar{\mathsf{t}}\,\mathsf{t}_\mathsf{R}\end{array}$$

where $\mathsf{Case}_{[\![\mathfrak{c}]\!]}$ is

$$\begin{array}{l}\{\overline{\mathsf{b}:B}\}\to(\mathsf{b}_\mathsf{R}:[\![B]\!]\,\bar{\mathsf{b}})\to\\ \{\mathsf{u}:(\mathsf{x}:X)\to\mathfrak{T}\,\mathsf{a}\,i\}\to(\mathsf{u}_\mathsf{R}:[\![(\mathsf{x}:X)\to\mathfrak{T}\,\mathsf{a}\,i]\!]\,\bar{\mathsf{u}})\to\\ (\{\overline{\mathsf{x}:X}\}\to(\mathsf{x}_\mathsf{R}:[\![X]\!]\,\bar{\mathsf{x}})\to\mathsf{Q}\,\{\vec{i}\}\,[\![i]\!]\,\overline{(\mathsf{u}\,\mathsf{x})}\,[\![\mathsf{u}\,\mathsf{x}]\!])\to\\ \mathsf{Q}\,\{\overline{v}\}\,[\![v]\!]\,\overline{(\mathfrak{c}\,\{\mathsf{a}\}\,\mathsf{b}\,\mathsf{u})}\,[\![\mathfrak{c}\,\{\mathsf{a}\}\,\mathsf{b}\,\mathsf{u}]\!]\end{array}$$

Therefore the proof $[\![\mathfrak{T}\text{-elim}]\!]$ can be defined using $[\![\mathfrak{T}]\!]$-elim and $\mathfrak{T}$-elim as follows:

$$\begin{array}{l}[\![\mathfrak{T}\text{-elim}]\!]:[\![\forall\{\mathsf{a}:A\}.\ \forall\mathsf{P}:(\forall\mathsf{n}:N.\ \mathfrak{T}\,\mathsf{a}\,\mathsf{n}\to s).\ \forall\mathsf{e}:\mathsf{Case}_\mathfrak{c}.\\ \qquad\qquad\forall\mathsf{n}:N.\ \forall\mathsf{t}:\mathfrak{T}\,\mathsf{a}\,\mathsf{n}.\ \mathsf{P}\,\mathsf{n}\,\mathsf{t}]\!]\,\overline{\mathfrak{T}\text{-elim}}\\ [\![\mathfrak{T}\text{-elim}\,\{\mathsf{a}\}\,\mathsf{P}\,\mathsf{e}]\!]=[\![\mathfrak{T}]\!]\text{-elim}\,\{\bar{\mathsf{a}}\}\,\{\mathsf{a}_\mathsf{R}\}\,\mathsf{Q}\,\mathsf{f}\end{array}$$

where

$$\mathsf{Q}\,\{\overline{n}\}\,n_R\,\overline{t}\,t_R = [\![\mathsf{P}\,n\,t]\!]\,\overline{(\mathfrak{T}\text{-elim}\,\{a\}\,\mathsf{P}\,e\,n\,t)} \tag{1.2}$$

$$f\,\{\overline{b}\}\,b_R\,\{\overline{u}\}\,u_R = [\![e\,b\,u]\!]\,\{\overline{(\lambda x{:}X.\ \mathfrak{T}\text{-elim}\,\{a\}\,\mathsf{P}\,e\,i\,(u\,x))}\} \tag{1.3}$$

We proceed to check that f has the right return type. Because

$$e\,b\,u : ((x{:}X) \to \mathsf{P}\,i\,(u\,x)) \to \mathsf{P}\,v\,(c\,\{a\}\,b\,u)$$

we have (by the abstraction theorem)

$$\begin{aligned}[\![e\,b\,u]\!] : {} & \{\overline{p{:}(x{:}X) \to \mathsf{P}\,i\,(u\,x)}\} \to \\ & (\{\overline{x{:}X}\} \to (x_R{:}[\![X]\!]\,\overline{x}) \to [\![\mathsf{P}\,i\,(u\,x)]\!]\,\overline{(p\,x)}) \to \\ & [\![\mathsf{P}\,v\,(c\,\{a\}\,b\,u)]\!]\,\overline{(e\,b\,u\,p)}\end{aligned}$$

and hence the type of $f\,\{\overline{b}\}\,b_R\,\{\overline{u}\}\,u_R$ is:

$$\begin{aligned}& (\{\overline{x{:}X}\} \to (x_R{:}[\![X]\!]\,\overline{x}) \to [\![\mathsf{P}\,i\,(u\,x)]\!]\,\overline{(\mathfrak{T}\text{-elim}\,\{a\}\,\mathsf{P}\,e\,i\,(u\,x))}) \to \\ & [\![\mathsf{P}\,v\,(c\,\{a\}\,b\,u)]\!]\,\overline{(e\,b\,u\,(\lambda x{:}X.\ \mathfrak{T}\text{-elim}\,\{a\}\,\mathsf{P}\,e\,i\,(u\,x)))} \\ = {} & \{\text{ datatype equation (1.1) from page 29 }\} \\ & (\{\overline{x{:}X}\} \to (x_R{:}[\![X]\!]\,\overline{x}) \to [\![\mathsf{P}\,i\,(u\,x)]\!]\,\overline{(\mathfrak{T}\text{-elim}\,\{a\}\,\mathsf{P}\,e\,i\,(u\,x))}) \to \\ & [\![\mathsf{P}\,v\,(c\,\{a\}\,b\,u)]\!]\,\overline{(\mathfrak{T}\text{-elim}\,\{a\}\,\mathsf{P}\,e\,v\,(c\,\{a\}\,b\,u))} \\ = {} & \{\text{ definition of Q (1.2) }\} \\ & (\{\overline{x{:}X}\} \to (x_R{:}[\![X]\!]\,\overline{x}) \to \mathsf{Q}\,\{\overline{i}\}\,[\![i]\!]\,(\overline{u\,x})\,[\![u\,x]\!]) \to \\ & \mathsf{Q}\,\{\overline{v}\}\,[\![v]\!]\,\overline{(c\,\{a\}\,b\,u)}\,[\![c\,\{a\}\,b\,u]\!]\end{aligned}$$

Deductive and inductive-style translations define the same relation, but the objects witnessing the instances of the inductively defined-relation record additional information, namely which rules are used to prove membership of the relation. However, since the same constructor never appears in more than one case of the inductive definition, that additional content can be recovered from a witness of the deductive-style; therefore the two styles are isomorphic.

Booleans Applying the above scheme to the **data**-declaration of Bool (from Figure 1.2), we obtain:

$$\begin{aligned}&\textbf{data}\ [\![\mathsf{Bool}]\!] : [\![\star]\!]\,\overline{\mathsf{Bool}}\ \textbf{where} \\ &\quad [\![\mathsf{true}]\!] : [\![\mathsf{Bool}]\!]\,\overline{\mathsf{true}} \\ &\quad [\![\mathsf{false}]\!] : [\![\mathsf{Bool}]\!]\,\overline{\mathsf{false}}\end{aligned}$$

The main difference from the deductive-style definition is that it is possible, by analysis of a value of type $[\![\mathsf{Bool}]\!]$, to recover the arguments of the relation (either all true, or all false).

The elimination constant for Bool is

$$\mathsf{Bool\text{-}elim} : (\mathsf{P} : \mathsf{Bool} \to \star) \to \mathsf{P}\,\mathsf{true} \to \mathsf{P}\,\mathsf{false} \to (\mathsf{b} : \mathsf{Bool}) \to \mathsf{P}\,\mathsf{b}$$

Similarly, our new type $[\![\mathsf{Bool}]\!]$ (with $n = 2$) has an elimination constant with the following type:

$$\begin{aligned}[\![\mathsf{Bool}]\!]\text{-}\mathsf{elim} : {} & (\mathsf{C} : \mathsf{a}_1\,\mathsf{a}_2 : \mathsf{Bool} \to [\![\mathsf{Bool}]\!]\,\mathsf{a}_1\,\mathsf{a}_2 \to \star) \to \\ & \mathsf{C}\,\mathsf{true}\,\mathsf{true}\,[\![\mathsf{true}]\!] \to \mathsf{C}\,\mathsf{false}\,\mathsf{false}\,[\![\mathsf{false}]\!] \to \\ & \{\,\mathsf{b}_1\,\mathsf{b}_2 : \mathsf{Bool}\,\} \to (\mathsf{b}_R : [\![\mathsf{Bool}]\!]\,\mathsf{b}_1\,\mathsf{b}_2) \to \mathsf{C}\,\mathsf{b}_1\,\mathsf{b}_2\,\mathsf{b}_R\end{aligned}$$

As an instance of the above scheme, we can define $[\![\mathsf{Bool\text{-}elim}]\!]$ using the elimination constants $[\![\mathsf{Bool}]\!]$ and $[\![\mathsf{Bool}]\!]$-elim as follows

$$\begin{aligned}&[\![\mathsf{Bool\text{-}elim}]\!] : \\ &\quad \{\,\mathsf{P}_1\,\mathsf{P}_2 : \mathsf{Bool} \to \star\,\} \to (\mathsf{P}_R : [\![\mathsf{Bool} \to \star]\!]\,\mathsf{P}_1\,\mathsf{P}_2) \to \\ &\quad \{\,\mathsf{x}_1 : \mathsf{P}_1\,\mathsf{true}\,\} \to \{\,\mathsf{x}_2 : \mathsf{P}_2\,\mathsf{true}\,\} \to (\mathsf{P}_R\,[\![\mathsf{true}]\!]\,\mathsf{x}_1\,\mathsf{x}_2) \to \\ &\quad \{\,\mathsf{y}_1 : \mathsf{P}_1\,\mathsf{false}\,\} \to \{\,\mathsf{y}_2 : \mathsf{P}_2\,\mathsf{false}\,\} \to (\mathsf{P}_R\,[\![\mathsf{false}]\!]\,\mathsf{y}_1\,\mathsf{y}_2) \to \\ &\quad \{\,\mathsf{b}_1\,\mathsf{b}_2 : \mathsf{Bool}\,\} \to (\mathsf{b}_R : [\![\mathsf{Bool}]\!]\,\mathsf{b}_1\,\mathsf{b}_2) \to \\ &\quad \mathsf{P}_R\,\mathsf{b}_R\,(\mathsf{Bool\text{-}elim}\,\mathsf{P}_1\,\mathsf{x}_1\,\mathsf{y}_1\,\mathsf{b}_1) \\ &\qquad\quad (\mathsf{Bool\text{-}elim}\,\mathsf{P}_2\,\mathsf{x}_2\,\mathsf{y}_2\,\mathsf{b}_2) \\ &[\![\mathsf{Bool\text{-}elim}]\!]\,\{\,\mathsf{P}_1\,\}\,\{\,\mathsf{P}_2\,\}\,\mathsf{P}_R\,\{\,\mathsf{x}_1\,\}\,\{\,\mathsf{x}_2\,\}\,\mathsf{x}_R\,\{\,\mathsf{y}_1\,\}\,\{\,\mathsf{y}_2\,\}\,\mathsf{y}_R \\ &\quad = [\![\mathsf{Bool}]\!]\text{-}\mathsf{elim} \\ &\qquad\quad (\lambda\,\mathsf{b}_1\,\mathsf{b}_2\,\mathsf{b}_R \to \mathsf{P}_R\,\mathsf{b}_R\,(\mathsf{Bool\text{-}elim}\,\mathsf{P}_1\,\mathsf{x}_1\,\mathsf{y}_1\,\mathsf{b}_1) \\ &\qquad\qquad\qquad\qquad\quad (\mathsf{Bool\text{-}elim}\,\mathsf{P}_2\,\mathsf{x}_2\,\mathsf{y}_2\,\mathsf{b}_2)) \\ &\qquad\quad \mathsf{x}_R\,\mathsf{y}_R\end{aligned}$$

Lists For List, as introduced in Figure 1.2, we have the following deductive translation:

$$\begin{aligned}&\mathbf{data}\ [\![\mathsf{List}]\!]\,([\![\mathsf{A} : \star]\!]) : [\![\star]\!]\,\overline{(\mathsf{List}\,\mathsf{A})}\ \mathbf{where} \\ &\quad [\![\mathsf{nil}]\!] \quad : [\![\mathsf{List}\,\mathsf{A}]\!]\,\overline{\mathsf{nil}} \\ &\quad [\![\mathsf{cons}]\!] : [\![\mathsf{A} \to \mathsf{List}\,\mathsf{A} \to \mathsf{List}\,\mathsf{A}]\!]\,\overline{\mathsf{cons}}\end{aligned}$$

or after expansion (for $n = 2$):

$$\begin{aligned}&\mathbf{data}\ [\![\mathsf{List}]\!]\,\{\,\mathsf{A}_1\,\mathsf{A}_2 : \star\,\}\,(\mathsf{A}_R : [\![\star]\!]\,\mathsf{A}_1\,\mathsf{A}_2) : \\ &\qquad\quad \mathsf{List}\,\mathsf{A}_1 \to \mathsf{List}\,\mathsf{A}_2 \to \star\ \mathbf{where} \\ &\quad [\![\mathsf{nil}]\!] \quad : [\![\mathsf{List}]\!]\,\mathsf{A}_R\,\mathsf{nil}\,\mathsf{nil} \\ &\quad [\![\mathsf{cons}]\!] : \{\,\mathsf{x}_1 : \mathsf{A}_1\,\} \to \{\,\mathsf{x}_2 : \mathsf{A}_2\,\} \to (\mathsf{x}_R : \mathsf{A}_R\,\mathsf{x}_1\,\mathsf{x}_2) \to \\ &\qquad\quad \{\,\mathsf{xs}_1 : \mathsf{List}\,\mathsf{A}_1\,\} \to \{\,\mathsf{xs}_2 : \mathsf{List}\,\mathsf{A}_2\,\} \to (\mathsf{xs}_R : [\![\mathsf{List}]\!]\,\mathsf{A}_R\,\mathsf{xs}_1\,\mathsf{xs}_2) \to \\ &\qquad\quad [\![\mathsf{List}]\!]\,\mathsf{A}_R\,(\mathsf{cons}\,\mathsf{x}_1\,\mathsf{xs}_1) \\ &\qquad\qquad\qquad\quad (\mathsf{cons}\,\mathsf{x}_2\,\mathsf{xs}_2)\end{aligned}$$

The above definition encodes the same relational action as that given in Section 4.3. Again, the difference is that the *derivation* of a relation between lists xs_1 and xs_2 is available as an object of type $[\![\mathsf{List}]\!]\,\mathsf{A}_R\,\mathsf{xs}_1\,\mathsf{xs}_2$.

$$
\begin{array}{l}
\textbf{data}\ [\![\mathsf{Vec}]\!]\ ([\![\mathsf{A} : \star]\!]) : [\![\mathbb{N} \to \star]\!]\ \overline{(\mathsf{Vec}\ \mathsf{A})}\ \textbf{where} \\
\quad [\![\mathsf{nilV}]\!] \quad : [\![\mathsf{Vec}\ \mathsf{A}\ \mathsf{zero}]\!]\ \overline{\mathsf{nilV}} \\
\quad [\![\mathsf{consV}]\!] : [\![\{\mathsf{x} : \mathsf{A}\} \to (\mathsf{n} : \mathbb{N}) \to \mathsf{Vec}\ \mathsf{A}\ \mathsf{n} \to \mathsf{Vec}\ \mathsf{A}\ (\mathsf{succ}\ \mathsf{n})]\!]\ \overline{\mathsf{consV}} \\
\textbf{data}\ [\![\mathsf{Vec}]\!]\ \{\mathsf{A}_1\ \mathsf{A}_2 : \star\}\ (\mathsf{A}_R : \mathsf{A}_1 \to \mathsf{A}_2 \to \star) : \\
\qquad \{\mathsf{n}_1\ \mathsf{n}_2 : \mathbb{N}\} \to (\mathsf{n}_R : [\![\mathbb{N}]\!]\ \mathsf{n}_1\ \mathsf{n}_2) \to \\
\qquad \mathsf{Vec}\ \mathsf{A}_1\ \mathsf{n}_1 \to \mathsf{Vec}\ \mathsf{A}_2\ \mathsf{n}_2 \to \star\ \textbf{where} \\
\quad [\![\mathsf{nilV}]\!] \quad : [\![\mathsf{Vec}]\!]\ \mathsf{A}_R\ [\![\mathsf{zero}]\!]\ \mathsf{nilV}\ \mathsf{nilV} \\
\quad [\![\mathsf{consV}]\!] : \{\mathsf{x}_1 : \mathsf{A}_1\} \to \{\mathsf{x}_2 : \mathsf{A}_2\} \to (\mathsf{x}_R : \mathsf{A}_R\ \mathsf{x}_1\ \mathsf{x}_2) \to \\
\qquad \{\mathsf{n}_1\ \mathsf{n}_2 : \mathbb{N}\} \to (\mathsf{n}_R : [\![\mathbb{N}]\!]\ \mathsf{n}_1\ \mathsf{n}_2) \to \\
\qquad \{\mathsf{xs}_1 : \mathsf{Vec}\ \mathsf{A}_1\ \mathsf{n}_1\} \to \{\mathsf{xs}_2 : \mathsf{Vec}\ \mathsf{A}_2\ \mathsf{n}_2\} \to \\
\qquad\quad (\mathsf{xs}_R : [\![\mathsf{Vec}]\!]\ \mathsf{A}_R\ \mathsf{n}_R\ \mathsf{xs}_1\ \mathsf{xs}_2) \to \\
\qquad [\![\mathsf{Vec}]\!]\ \mathsf{A}_R\ ([\![\mathsf{succ}]\!]\ \mathsf{n}_R)\ (\mathsf{consV}\ \mathsf{x}_1\ \mathsf{n}_1\ \mathsf{xs}_1)\ (\mathsf{consV}\ \mathsf{x}_2\ \mathsf{n}_2\ \mathsf{xs}_2)
\end{array}
$$

Figure 1.4: Inductive translation of Vec, both before and after expansion.

Proof terms The proof term for the list-rearrangement example can be constructed in a similar way to the inductive one. The main difference is that the target lists are also built and recorded in the $[\![\mathsf{List}]\!]$ structure. In short, this version has more of a computational flavour than the inductive version.

$$
\begin{array}{lcl}
[\![\mathsf{odds}]\!] : [\![(\mathsf{A} : \star) \to \mathsf{List}\ \mathsf{A} \to \mathsf{List}\ \mathsf{A}]\!]\ \mathsf{odds}\ \mathsf{odds} & & \\
[\![\mathsf{odds}]\!]\ \mathsf{A}_R\ [\![\mathsf{nil}]\!] & = & [\![\mathsf{nil}]\!] \quad \mathsf{A}_R \\
[\![\mathsf{odds}]\!]\ \mathsf{A}_R\ ([\![\mathsf{cons}]\!]\ \mathsf{x}_R\ [\![\mathsf{nil}]\!]) & = & [\![\mathsf{cons}]\!]\ \mathsf{A}_R\ \mathsf{x}_R\ ([\![\mathsf{nil}]\!] \quad \mathsf{A}_R) \\
[\![\mathsf{odds}]\!]\ \mathsf{A}_R\ ([\![\mathsf{cons}]\!]\ \mathsf{x}_R\ ([\![\mathsf{cons}]\!]\ \_\ \mathsf{xs}_R)) & = & [\![\mathsf{cons}]\!]\ \mathsf{A}_R\ \mathsf{x}_R\ ([\![\mathsf{odds}]\!]\ \mathsf{A}_R\ \mathsf{xs}_R)
\end{array}
$$

Vectors We can apply the same translation method to inductive families. For example, Figures 1.4 and 1.5 give the translation of the family Vec, corresponding to lists indexed by their length. The relation obtained by applying $[\![\cdot]\!]$ encodes that vectors are related if their lengths are the same and if their elements are related point-wise. The difference with the List version is that the equality of lengths is encoded in $[\![\mathsf{consV}]\!]$ as an $\mathbb{N}$ (identity) relation.

5 Internalisation

We know that free theorems hold for any term of S (and these theorems are expressible and provable in S^t). Unfortunately, users of the logical system S^t cannot take advantage of that fact: they have to redo the proofs for every new program (even though the proof is derivable, thanks to $[\![\cdot]\!]$). We would like the instances of the abstraction theorem to come truly for

$$
\begin{aligned}
&[\![\,\mathsf{Vec\text{-}elim}\,]\!] : [\![\,\{\mathsf{A} : \star\} \to \\
&\qquad (\mathsf{P} : (\mathsf{n} : \mathbb{N}) \to \mathsf{Vec\ n\ A} \to \star) \to \\
&\qquad (\mathsf{en} : \mathsf{P\ zero\ (nilV\ A)}) \to \\
&\qquad (\mathsf{ec} : (\mathsf{x} : \mathsf{A}) \to (\mathsf{n} : \mathbb{N}) \to (\mathsf{xs} : \mathsf{Vec\ n\ A}) \to \\
&\qquad\qquad \mathsf{P\ n\ xs} \to \mathsf{P\ (succ\ n)\ (consV\ A\ x\ n\ xs)}) \to \\
&\qquad (\mathsf{n} : \mathbb{N}) \to (\mathsf{v} : \mathsf{Vec\ n\ A}) \to \mathsf{P\ n\ v}]\!]\ \overline{\mathsf{Vec\text{-}elim}} \\
&[\![\,\mathsf{Vec\text{-}elim\ A\ P\ en\ ec}]\!] \;=\; [\![\mathsf{Vec}]\!]\text{-}\mathsf{elim}\ \mathsf{A}_\mathsf{R} \\
&\quad (\lambda\,[\![\mathsf{n} : \mathbb{N}, \mathsf{v} : \mathsf{Vec\ n\ A}]\!]\,.\,[\![\mathsf{P\ n\ v}]\!]\ \overline{(\mathsf{Vec\text{-}elim\ A\ P\ en\ ec\ v})}) \\
&\quad \mathsf{en}_\mathsf{R} \\
&\quad (\lambda\,[\![\mathsf{x} : \mathsf{A}, \mathsf{n} : \mathbb{N}, \mathsf{xs} : \mathsf{Vec\ n\ A}]\!]\,.\,[\![\mathsf{ec\ x\ n\ xs}]\!]\ \overline{(\mathsf{Vec\text{-}elim\ A\ P\ en\ ec\ xs})})
\end{aligned}
$$

Figure 1.5: Proof term for Vec-elim using the inductive-style definitions.

free: that is, extend S^t with a suitable construct that makes parametricity arguments available for every program in S. To do so, we construct a new system S^t_p, which is the system S^t extended with following axiom schema:

Axiom 15 (parametricity)**.** For every closed type B of sort s ($\vdash_S B : s$), assume

$$\mathsf{param}_B : \forall^{k_i}\mathsf{x}{:}B.\ [\![B]\!]\,\mathsf{x}\ldots\mathsf{x}$$

The consistency of the new system remains to be shown. This can be done via a sound translation from S^t_p to S^t. A first attempt would be to extend β-reduction rules with

$$\mathsf{param}_B\ A \longrightarrow_\beta [\![A]\!]$$

Unfortunately, the above fails if A is an open term, because $[\![A]\!]$ contains occurrences of the variable x_R, which is not bound in the context of $\mathsf{param}_B\ A$. Therefore we need a more complex interpretation. Even with a more complex interpretation accounting for free variables in A, we need to stick to closed types. Indeed, if the type B were to contain free variables, the type of param_B would not be well-scoped.

Parametricity witnesses Our attempt to show consistency by giving a local interpretation of the parametricity principle failed. Therefore, we instead can do a "global" transformation of a closed term in S^t_p to a term in S^t.

The idea is to transform the program such that, whenever a variable $(\mathsf{x} : A)$ is bound, a witness $(\mathsf{x}_\mathsf{R} : [\![A]\!]\,\mathsf{x}\ldots\mathsf{x})$ that x satisfies the parametricity condition is bound at the same time. This means that functions are modified to take an additional argument witnessing that the original arguments are parametric. This additional argument is then used to interpret occurrences

of x in the argument of param_B. At every application, the parametricity witness can be reconstructed, using the $[\![\cdot]\!]$ translation of the original argument. For example, the fragment

$$\mathsf{value} \;=\; \mathsf{p}\,(\mathsf{suc}\,\mathsf{m})$$

in context

$$\begin{array}{l}\mathbb{N} \;:\star\\ \mathsf{suc} : \mathbb{N} \to \mathbb{N}\\ \mathsf{m} \;:\mathbb{N}\\ \mathsf{X} \;:\lceil\star\rceil\\ \mathsf{p} \;:\mathbb{N} \to \mathsf{X}\end{array}$$

would be translated to:

$$\mathsf{value} \;=\; \mathsf{p}\,(\mathsf{suc}\,\mathsf{m})\,([\![\mathsf{suc}\,\mathsf{m}]\!])$$

in context

$$\begin{array}{ll}\mathbb{N} & :\star\\ [\![\,\mathbb{N}\,]\!] & : \mathbb{N} \to \mathbb{N} \to \lceil\star\rceil\\ \mathsf{suc} & : \mathbb{N} \to \mathbb{N}\\ [\![\mathsf{suc}]\!] & : [\![\,\mathbb{N} \to \mathbb{N}\,]\!]\,\mathsf{suc}\,\mathsf{suc}\\ \mathsf{m} & :\mathbb{N}\\ [\![\mathsf{m}]\!] & : [\![\,\mathbb{N}\,]\!]\,\mathsf{m}\,\mathsf{m}\\ \mathsf{X} & :\lceil\star\rceil\\ \mathsf{p} & : (\mathsf{n} : \mathbb{N}) \to [\![\mathbb{N}]\!]\,\mathsf{n}\,\mathsf{n} \to \mathsf{X}\end{array}$$

General case In the rest of the section, we define the translation $\langle\!|\cdot|\!\rangle$ from terms of S^t_p to terms of S^t. The translation is similar to $[\![\cdot]\!]$, with a number of differences:

- The new translation deals with a richer language: there is a structure in the space of sorts, which can be either of the form s or $\lceil s\rceil$. Further, it does not duplicate the bindings whose type are not in the source language (the sort is of the form $\lceil s\rceil$). Therefore it behaves differently depending on this sort, and using sorts, we must therefore distinguish two parts of the PTS: one (the source language of $[\![\cdot]\!]$), which deals with programs and types of sort s and another which deals with parametricity proofs and propositions of sort $\lceil s\rceil$ (the target language).

- The translation does not transform types to relations.

- The new translation does not replicate the bindings: it adds at most one additional binding, regardless of the arity of param. A consequence is that the renaming operation (Definition 7) must be modified, such that occurrences of variables bound in bindings processed by $\langle\!\|\cdot\|\!\rangle$ are not renamed.

As hinted above, $\langle\!\|\cdot\|\!\rangle$ does not work on all possible system S^t. The precise set of restrictions is as follows.

Definition 16 (Restrictions for internalisation)**.**

1. Let $\lceil\mathcal{S}\rceil = \mathcal{S}^t - \mathcal{S}$. If $s \in \mathcal{S}$, then $\lceil s\rceil \in \lceil\mathcal{S}\rceil$. This ensures that the sorts of types of the sources language can always be distinguished from the sorts of propositions.[4]

2. If $(k, s_1, s_2, s_3) \in \mathcal{R}^t$ and $s_3 \in \mathcal{S}$, then $s_1 \in \mathcal{S}$ and $s_2 \in \mathcal{S}$. This ensures terms and types of the source language can contain no propositions of parametricity nor their proofs.

3. Let $K_v \subseteq K$ and $K_w = K - K_v$. (In the following we will use the meta-syntactic variable a for colours in the first group and b for colours in the second one.) If $(k, s_1, s_2, s_3) \in \mathcal{R}$ then $s_1 \in \mathcal{S} \leftrightarrow k \in K_v$.

   This ensures that quantifications over terms in the input language can be recognised syntactically from quantifications over parametricity propositions and proofs. This requirement is for convenience only, as suitable colours can be inferred from a typing derivation.

4. For each rule $s_1 \overset{v}{\rightsquigarrow} \lceil s_2\rceil$ there must be a colour $t_v \in K_w$ and a rule $\lceil s_1\rceil \overset{t_v}{\rightsquigarrow} \lceil s_2\rceil$.

For example, the system described in Section 3.3 satisfies these conditions. In the following, we assume that param_B is always saturated. Doing so causes no loss of generality: η-expansion can be applied to obtain the desired form. We define the translation $\langle\!\|\cdot\|\!\rangle$ from terms typed in S^t_p to terms of S^t as follows.

[4]This restriction rules out (non-trivial) reflective systems.

Definition 17 (Compilation of param)**.**

$$
\begin{array}{rll}
\langle\!|s|\!\rangle &= s & \\
\langle\!|\mathsf{x}|\!\rangle &= \mathsf{x} & \\
\langle\!|\mathsf{param}_B\, F\, A_0 \ldots A_l|\!\rangle &= [\![F]\!]\, A_0 \ldots A_l & \\
\hline
\langle\!|(\mathsf{x}{:}A) \xrightarrow{v} B|\!\rangle &= (\mathsf{x}{:}A) \xrightarrow{v} (\mathsf{x}_\mathsf{R}{:}[\![A]\!]\,\mathsf{x}\ldots\mathsf{x}) \xrightarrow{t_v} \langle\!|B|\!\rangle & \\
\langle\!|\lambda^v \mathsf{x}{:}A.\ b|\!\rangle &= \lambda^v \mathsf{x}{:}A.\ \lambda^{t_v} \mathsf{x}_\mathsf{R}{:}[\![A]\!]\,\mathsf{x}\ldots\mathsf{x}.\ \langle\!|b|\!\rangle & \\
\langle\!|F \bullet_v a|\!\rangle &= \langle\!|F|\!\rangle \bullet_v a \bullet_{t_v} [\![a]\!] & (\dagger) \\
\hline
\langle\!|(\mathsf{x}{:}A) \xrightarrow{w} B|\!\rangle &= (\mathsf{x}{:}\langle\!|A|\!\rangle) \xrightarrow{w} \langle\!|B|\!\rangle & \\
\langle\!|\lambda^w \mathsf{x}{:}A.\ b|\!\rangle &= \lambda^w \mathsf{x}{:}\langle\!|A|\!\rangle.\ \langle\!|b|\!\rangle & \\
\langle\!|F \bullet_w a|\!\rangle &= \langle\!|F|\!\rangle \bullet_w \langle\!|a|\!\rangle & (*) \\
\hline
\langle\!|\Gamma, \mathsf{x}{:}A|\!\rangle &= \langle\!|\Gamma|\!\rangle, \mathsf{x}{:}A, \mathsf{x}_\mathsf{R}{:}[\![A]\!]\,\mathsf{x}\ldots\mathsf{x} & \text{if } \Gamma \vdash A : s \\
\langle\!|\Gamma, \mathsf{x}{:}A|\!\rangle &= \langle\!|\Gamma|\!\rangle, \mathsf{x}{:}\langle\!|A|\!\rangle & \text{if } \Gamma \vdash A : \lceil s \rceil
\end{array}
$$

Lemma 18. *Assuming $s \in \mathcal{S}$, then*

1. *if $\Gamma \vdash_{S^t} B : s$, then* param *cannot appear in B and*
2. *if $\Gamma \vdash_{S^t} A : B$, then* param *cannot appear in A.*

Proof. The proof is done by simultaneous induction on the typing derivations.

In the base case, a constant cannot be param, because its type has a sort in $\lceil s \rceil$.

In the induction cases, we take advantage of the restriction on rules to ensure that subterms also satisfy the conditions of the lemma. □

Theorem 19. *All occurrences of* param *are removed by $\langle\!| \cdot |\!\rangle$.*

Proof. The proof is done by induction on terms.

- The base case (param_B) removes occurrences.
- No other occurrences are introduced. In particular, in the line marked with an asterisk $(*)$; the argument of sort $\lceil s \rceil$ (which may contain param) is not duplicated. In line marked $(\dagger)$, the term a cannot contain any occurrence of param, as shown by Lemma 18.

□

Theorem 20 (soundness)**.** *$\langle\!| \cdot |\!\rangle$ translates valid judgements in S^t_p to valid judgements in S^t.*

$$\Gamma \vdash_{S^t_p} A : B \Rightarrow \langle\!|\Gamma|\!\rangle \vdash_{S^t} \langle\!|A|\!\rangle : \langle\!|B|\!\rangle$$

Proof idea. The proof proceeds by induction on the typing derivation. □

Computational interpretation We have previously discussed how the computational content of a term $[\![A]\!]$ is essentially the same as that of A (Section 3.3, Section 4.3). The same can be said about the translation $\langle\!| \cdot |\!\rangle$. Indeed, at most, it merely doubles abstractions and applications. Hence, $\langle\!| \cdot |\!\rangle$ can be used as a pass of a compiler for a language featuring the param construct.

6 Applications

Sections 3 and 4 contain simple applications of our setting. More applications of parametricity on programs expressible in System F are shown in Paper III. In this section we see how elaborate constructions can be handled. All examples of this section fit within the system I_ω augmented with inductive definitions.

6.1 A library for applications

Applying $[\![\cdot]\!]$ by hand to non-trivial examples can be tedious. The reader eager to experiment is suggested to use computer aids. One possible tool is that of Böhme (2007) which computes the relational interpretation of any Haskell type. Unfortunately, the above tool has not been extended to support dependent types. To generate the examples for this paper, we have used an Agda library (Bernardy, 2010) instead. An advantage of the library approach is that one can use a single framework to write programs and reason using free theorems about them.

6.2 Proof irrelevance and parametricity

In this section we show that any two proofs of a given proposition can be treated as related. In a predicative system with inductive families, such as Agda, there are at least two ways to represent propositions. A common choice is to use $\star$ for the sort of propositions, as we suggest in Section 2.1. One issue is then that quantification over types in $\star$ is in $\star_1$, hence not a proposition. The issue can be side-stepped by encoding propositions in a universe like the following, where quantification using π yields a proposition in the same universe.

```
data Prop : ⋆₁ where
  top  : Prop
  bot  : Prop
```

$$
\begin{aligned}
&\_\wedge\_ : \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop}\\
&\pi \quad : (\mathsf{A} : \star) \to (\mathsf{f} : \mathsf{A} \to \mathsf{Prop}) \to \mathsf{Prop}
\end{aligned}
$$

One can then construct proposition objects, for example an ordering between naturals

$$
\begin{aligned}
&\_<\_ : \mathbb{N} \to \mathbb{N} \to \mathsf{Prop}\\
&\mathsf{n} \quad < \mathsf{zer} \quad = \mathsf{bot}\\
&\mathsf{zer} \quad < \mathsf{suc}\,\mathsf{n} \;=\; \mathsf{top}\\
&\mathsf{suc}\,\mathsf{m} < \mathsf{suc}\,\mathsf{n} \;=\; \mathsf{m} < \mathsf{n}
\end{aligned}
$$

or the predicate that n is the biggest natural:

$$
\begin{aligned}
&\mathsf{supremum} : \mathbb{N} \to \mathsf{Prop}\\
&\mathsf{supremum}\,\mathsf{n} \;=\; \pi\,\mathbb{N}\,(\lambda\,\mathsf{m} \to \mathsf{m} \leqslant \mathsf{n})
\end{aligned}
$$

The intention is for propositions to be interpreted as the set of their proofs:

$$
\begin{aligned}
&\mathsf{Proof} : \mathsf{Prop} \to \star\\
&\mathsf{Proof}\,\mathsf{top} \quad\;\; = \top\\
&\mathsf{Proof}\,\mathsf{bot} \quad\;\; = \bot\\
&\mathsf{Proof}\,(\mathsf{a} \wedge \mathsf{b}) \;=\; \mathsf{Proof}\,\mathsf{a} \;\times\; \mathsf{Proof}\,\mathsf{b}\\
&\mathsf{Proof}\,(\pi\,\mathsf{A}\,\mathsf{f}) \;=\; (\mathsf{a} : \mathsf{A}) \to \mathsf{Proof}\,(\mathsf{f}\,\mathsf{a})
\end{aligned}
$$

but to enable changing the parametricity translation of proofs, we will instead just postulate an abstract $\mathsf{Proof} : \mathsf{Prop} \to \star$ and a few constants. We choose constants so that proofs (terms of type $\mathsf{Proof}\,\mathsf{p}$ for some $\mathsf{p} : \mathsf{Prop}$) only can interact in limited ways with programs ($\mathsf{a} : \mathsf{A} : \star$). We allow standard proof constructions: introduction and elimination of π and $\wedge$ and introduction of $\top$. Additionally, given any proof of falsity, we allow to construct a program of an arbitrary type.

$$
\begin{aligned}
&\mathsf{app} : (\mathsf{A} : \star) \to (\mathsf{f} : \mathsf{A} \to \mathsf{Prop}) \to \mathsf{Proof}\,(\pi\,\mathsf{A}\,\mathsf{f}) \to (\mathsf{a} : \mathsf{A}) \to \mathsf{Proof}\,(\mathsf{f}\,\mathsf{a})\\
&\mathsf{abs} : (\mathsf{A} : \star) \to (\mathsf{f} : \mathsf{A} \to \mathsf{Prop}) \to ((\mathsf{a} : \mathsf{A}) \to \mathsf{Proof}\,(\mathsf{f}\,\mathsf{a})) \to \mathsf{Proof}\,(\pi\,\mathsf{A}\,\mathsf{f})\\
&\mathsf{proj1} : (\mathsf{a}\,\mathsf{b} : \mathsf{Prop}) \to \mathsf{Proof}\,(\mathsf{a} \wedge \mathsf{b}) \to \mathsf{Proof}\,\mathsf{a}\\
&\mathsf{proj2} : (\mathsf{a}\,\mathsf{b} : \mathsf{Prop}) \to \mathsf{Proof}\,(\mathsf{a} \wedge \mathsf{b}) \to \mathsf{Proof}\,\mathsf{b}\\
&\mathsf{pair} : (\mathsf{a}\,\mathsf{b} : \mathsf{Prop}) \to \mathsf{Proof}\,\mathsf{a} \to \mathsf{Proof}\,\mathsf{b} \to \mathsf{Proof}\,(\mathsf{a} \wedge \mathsf{b})\\
&\mathsf{obvious} : \mathsf{Proof}\,\mathsf{top}\\
&\mathsf{botElim} : \mathsf{Proof}\,\mathsf{bot} \to (\mathsf{A} : \star) \to \mathsf{A}
\end{aligned}
$$

A consequence of this restriction is that the structure of proofs is *irrelevant* in the meaning of programs. The reason is that programs cannot assume that the structure of a proof follows that of the proposition being examined.

Note that programs could depend on the structure of proofs if we were to use the *definition* of Proof given above, and that in that case, our relational interpretation would translate proofs to witnesses that these are related. For example, given the type of a lookup function in a list bound by length:

$$\mathsf{lk} : \{\mathsf{A} : \star\} \to (\mathsf{n} : \mathbb{N}) \to (\mathsf{xs} : \mathsf{List}\,\mathsf{A}) \to \mathsf{Proof}\,(\mathsf{n} < \mathsf{len}\,\mathsf{xs}) \to \mathsf{A}$$

one gets the following relation, which carries an assumption $\mathsf{p_R}$ requiring the proofs $\mathsf{p_1}$ and $\mathsf{p_2}$ to be related. That assumption would have a complicated formulation if we had taken the standard interpretation of the set of proofs.

$$\begin{aligned}
&[\![\mathsf{lk}]\!] : \{\mathsf{A_1}\,\mathsf{A_2} : \star\}\,(\mathsf{A_R} : \mathsf{A_1} \to \mathsf{A_2} \to \star)\\
&\quad \{\mathsf{n_1}\,\mathsf{n_2} : \mathbb{N}\}\,(\mathsf{n_R} : [\![\mathsf{Nat}]\!]\,\mathsf{n_1}\,\mathsf{n_2})\\
&\quad \{\mathsf{xs_1} : \mathsf{List}\,\mathsf{A_1}\}\,\{\mathsf{xs_2} : \mathsf{List}\,\mathsf{A_2}\}\,(\mathsf{xs_R} : [\![\mathsf{List}]\!]\,\mathsf{A_R}\,\mathsf{xs_1}\,\mathsf{xs_2})\\
&\quad \{\mathsf{p_1} : \mathsf{Proof}\,(\mathsf{n_1} < \mathsf{len}\,\mathsf{xs_1})\}\\
&\quad \{\mathsf{p_2} : \mathsf{Proof}\,(\mathsf{n_2} < \mathsf{len}\,\mathsf{xs_2})\}\\
&\quad (\mathsf{p_R} : [\![\mathsf{n} < \mathsf{len}\,\mathsf{xs}]\!]\,\mathsf{p_1}\,\mathsf{p_2}) \to\\
&\quad \mathsf{A_R}\,(\mathsf{lk}\,\mathsf{n_1}\,\mathsf{xs_1}\,\mathsf{p_1})\,(\mathsf{lk}\,\mathsf{n_1}\,\mathsf{xs_1}\,\mathsf{p_1})
\end{aligned}$$

However, by axiomatising Proof, we can pick any translation $[\![\mathsf{Proof}]\!]$ which also satisfies the other axioms. In fact, we can assert that all proofs are related:

$$\begin{aligned}
&[\![\mathsf{Proof}]\!] : [\![\mathsf{proposition} \to *]\!]\,\mathsf{Proof}\,\mathsf{Proof}\\
&[\![\mathsf{Proof}]\!]\,_\,\mathsf{x_1}\,\mathsf{x_2} \;=\; \top
\end{aligned}$$

The assumptions requiring proofs to be related then reduce to $\top$; effectively disappearing (because values of singleton types like $\top$ can always be inferred).

For the above overriding to be sound, one needs to provide a translation of app, abs, proj1, proj2, pair, obvious and botElim respecting the parametricity condition. All but the last are easy to translate: their results are proofs, so the result type of their translation is $\top$. Hence, constant functions returning tt do the job. Translating botElim can seem more tricky: because it has a proof as argument, the assertion that all proofs are related makes $[\![\mathsf{botElim}]\!]$ potentially more difficult to write, as it has one less assumption to work with. However, it already has two proofs of falsity as arguments, so the relational witness is superfluous.

$$\begin{aligned}
&[\![\mathsf{botElim}]\!] : (\mathsf{b_1} : \mathsf{Proof\ bot}) \to (\mathsf{b_2} : \mathsf{Proof\ bot}) \to \top \to\\
&\qquad\qquad [\![(\mathsf{A} : \star) \to \mathsf{A}]\!]\,(\mathsf{botElim}\,\mathsf{b_1})\,(\mathsf{botElim}\,\mathsf{b_2})\\
&[\![\mathsf{botElim}]\!]\,\mathsf{b_1}\,\mathsf{b_2} \;=\; \mathsf{botElim}\,\mathsf{b_1}\,([\![(\mathsf{A} : \star) \to \mathsf{A}]\!]\,(\mathsf{botElim}\,\mathsf{b_1})\,(\mathsf{botElim}\,\mathsf{b_2}))
\end{aligned}$$

In summary, assuming proof-irrelevance, proof arguments do not strengthen parametricity conditions in useful ways. One often (but not always) does not care about the *actual* proof of a proposition, but merely that it exists. In that case, knowing that two proofs are related adds no information.

Irrelevant interpretation of propositions in CC_ω In the system CC_ω, propositions are naturally encoded as inhabitants of the sort $\star$; because

the system provides the impredicative rules $\Box_i \rightsquigarrow \star$. The interpretation of proposition as trivial relations done in the previous section can be transported to CC_ω.

This yields another way for CC_ω to reflect itself:

- $\lceil \Box_i \rceil = \star$,
- $\lceil \star \rceil$ removed.

In that case, $[\![\cdot]\!]$ depends on the particular product rule corresponding to the construct it is being applied to (see Section 5 for another example of a transformation with similar behaviour).

Definition 21 ($[\![\cdot]\!]$ in CC_ω, capturing irrelevance of $\star$)**.**

$$\begin{aligned}[\![\Box_i]\!] &= \lambda \overline{x{:}s}.\ \bar{x} \to \star \\ [\![x]\!] &= x_R\end{aligned}$$

For $(\Box_i, \Box_j, \Box_{i \sqcup j})$,

$$\begin{aligned}[\![\forall x{:}A.\ B]\!] &= \lambda \overline{f{:}(\forall x{:}A.\ B)}.\ \forall \overline{x{:}A}.\ \forall x_R{:}[\![A]\!]\,\bar{x}.\ [\![B]\!]\,(\overline{f\,x}) \\ [\![F\,a]\!] &= [\![F]\!]\,\bar{a}\,[\![a]\!] \\ [\![\lambda x{:}A.\ b]\!] &= \lambda \overline{x{:}A}.\ \lambda x_R{:}[\![A]\!]\,\bar{x}.\ [\![b]\!]\end{aligned}$$

For $\star \rightsquigarrow \Box_i$,

$$\begin{aligned}[\![\forall x{:}A.\ B]\!] &= \lambda \overline{f{:}(\forall x{:}A.\ B)}.\ \forall \overline{x{:}A}.\ [\![B]\!]\,(\overline{f\,x}) \\ [\![F\,a]\!] &= [\![F]\!]\,\bar{a} \\ [\![\lambda x{:}A.\ b]\!] &= \lambda \overline{x{:}A}.\ [\![b]\!]\end{aligned}$$

Note that all types of sort $\star$ and their inhabitants are removed by the above translation. Therefore neither the case for $\star$ nor the case for constructs depending on the rule $\Box_i \rightsquigarrow \star$ need to be given.

6.3 Type classes

What if a function is not parametrised over all types, but only types equipped with decidable equality? One way to model this difference in a pure type system is to add an extra parameter to capture the extra constraint. For example, a function `nub` : `Nub` removing duplicates from a list may be given the following type:

$$\mathsf{Nub} \;=\; (\mathsf{A} : \star) \to \mathsf{Eq}\,\mathsf{A} \to \mathsf{List}\,\mathsf{A} \to \mathsf{List}\,\mathsf{A}$$

The equality requirement itself may be modelled as a mere comparison function: $\mathsf{Eq}\,\mathsf{A} \;=\; \mathsf{A} \to \mathsf{A} \to \mathsf{Bool}$. In that case, the parametricity statement is amended with an extra requirement on the relation between types, which expresses that eq_1 and eq_2 must respect the A_R relation. Formally:

$$\begin{aligned}
&[\![\mathsf{Eq}\,\mathsf{A}]\!]\,\mathsf{eq}_1\,\mathsf{eq}_2 \;=\; \{\mathsf{a}_1 : \mathsf{A}_1\} \to \{\mathsf{a}_2 : \mathsf{A}_2\} \to \mathsf{A}_\mathsf{R}\,\mathsf{a}_1\,\mathsf{a}_2 \to \\
&\qquad\qquad\qquad\quad \{\mathsf{b}_1 : \mathsf{A}_1\} \to \{\mathsf{b}_2 : \mathsf{A}_2\} \to \mathsf{A}_\mathsf{R}\,\mathsf{b}_1\,\mathsf{b}_2 \to \\
&\qquad\qquad\qquad\quad [\![\mathsf{Bool}]\!]\,(\mathsf{eq}_1\,\mathsf{a}_1\,\mathsf{b}_1)\,(\mathsf{eq}_2\,\mathsf{a}_2\,\mathsf{b}_2) \\
&[\![\mathsf{Nub}]\!]\,\mathsf{n}_1\,\mathsf{n}_2 \;= \\
&\quad \{\mathsf{A}_1\,\mathsf{A}_2 : \star\} \to (\mathsf{A}_\mathsf{R} : [\![\star]\!]\,\mathsf{A}_1\,\mathsf{A}_2) \to \\
&\quad \{\mathsf{eq}_1 : \mathsf{Eq}\,\mathsf{A}_1\} \to \{\mathsf{eq}_2 : \mathsf{Eq}\,\mathsf{A}_2\} \to [\![\mathsf{Eq}\,\mathsf{A}]\!]\,\mathsf{eq}_1\,\mathsf{eq}_2 \to \\
&\quad \{\mathsf{xs}_1 : \mathsf{List}\,\mathsf{A}_1\} \to \{\mathsf{xs}_2 : \mathsf{List}\,\mathsf{A}_2\} \to [\![\mathsf{List}\,\mathsf{A}]\!]\,\mathsf{xs}_1\,\mathsf{xs}_2 \to \\
&\quad [\![\mathsf{List}]\!]\,\mathsf{A}_\mathsf{R}\,(\mathsf{n}_1\,\mathsf{A}_1\,\mathsf{eq}_1\,\mathsf{xs}_1)\,(\mathsf{n}_2\,\mathsf{A}_2\,\mathsf{eq}_2\,\mathsf{xs}_2)
\end{aligned}$$

So far, this is just confirming the informal description in Wadler (1989). But with access to full dependent types, one might wonder: what if we model equality more precisely, for example by requiring eq to be reflexive?

$$\begin{aligned}
\mathsf{Eq}'\,\mathsf{A} \;&=\; (\mathsf{eq} : \mathsf{A} \to \mathsf{A} \to \mathsf{Bool}) \times \mathsf{Refl}\,\mathsf{eq} \\
\mathsf{Refl}\,\mathsf{eq} \;&=\; (\mathsf{x} : \mathsf{A}) \to \mathsf{eq}\,\mathsf{x}\,\mathsf{x} \equiv \mathsf{true}
\end{aligned}$$

In the case of Eq′, the parametricity condition does not become more exciting. It merely requires the proofs of reflexivity at A_1, A_2 to be related. This extra condition adds nothing new, as seen in Section 6.2.

The observations drawn from this simple example can be generalised: type-classes may be encoded as their dictionary of methods (Wadler and Blott, 1989), ignoring their laws. Indeed, even if a type class has associated laws, they have little impact on the parametricity results.

6.4 Constructor classes

Having seen how to apply our framework both to type constructors and type classes, we now apply it to types quantified over a type constructor, with constraints.

Voigtländer (2009*b*) provides many such examples, using the Monad constructor class. They fit well in our framework, but here we show the simpler example of Functors, which already captures the essence of constructor classes.

$$\begin{aligned}
&\mathsf{Functor} : \star_1 \\
&\mathsf{Functor} \;=\; (\mathsf{F} : \star \to \star) \times ((\mathsf{X}\,\mathsf{Y} : \star) \to (\mathsf{X} \to \mathsf{Y}) \to \mathsf{F}\,\mathsf{X} \to \mathsf{F}\,\mathsf{Y})
\end{aligned}$$

Our translation readily applies to the above definition, and yields the following relation between functors:

$$
\begin{aligned}
&[\![\mathsf{Functor}]\!] : \mathsf{Functor} \to \mathsf{Functor} \to \star_1 \\
&[\![\mathsf{Functor}]\!]\,(\mathsf{F}_1, \mathsf{map}_1)\,(\mathsf{F}_2, \mathsf{map}_2) \\
&\quad = (\mathsf{F}_R : \{\mathsf{A}_1\,\mathsf{A}_2 : \star\} \to (\mathsf{A}_R : \mathsf{A}_1 \to \mathsf{A}_2 \to \star) \to (\mathsf{F}_1\,\mathsf{A}_1 \to \mathsf{F}_2\,\mathsf{A}_2 \to \star)) \times \\
&\qquad (\ \{\mathsf{X}_1\,\mathsf{X}_2 : \star\} \to (\mathsf{X}_R : \mathsf{X}_1 \to \mathsf{X}_2 \to \star) \to \\
&\qquad\ \{\mathsf{Y}_1\,\mathsf{Y}_2 : \star\} \to (\mathsf{Y}_R : \mathsf{Y}_1 \to \mathsf{Y}_2 \to \star) \to \\
&\qquad\ \{\mathsf{f}_1 : \mathsf{X}_1 \to \mathsf{Y}_1\} \to \{\mathsf{f}_2 : \mathsf{X}_2 \to \mathsf{Y}_2\} \to \\
&\qquad\ (\{\mathsf{x}_1 : \mathsf{X}_1\} \to \{\mathsf{x}_2 : \mathsf{X}_2\} \to \mathsf{X}_R\,\mathsf{x}_1\,\mathsf{x}_2 \to \mathsf{Y}_R\,(\mathsf{f}_1\,\mathsf{x}_1)\,(\mathsf{f}_2\,\mathsf{x}_2)) \to \\
&\qquad\ \{\mathsf{y}_1 : \mathsf{F}_1\,\mathsf{X}_1\} \to \{\mathsf{y}_2 : \mathsf{F}_2\,\mathsf{X}_2\} \to (\mathsf{y}_R : \mathsf{F}_R\,\mathsf{X}_R\,\mathsf{y}_1\,\mathsf{y}_2) \to \\
&\qquad\ \mathsf{F}_R\,\mathsf{Y}_R\,(\mathsf{map}_1\,\mathsf{f}_1\,\mathsf{y}_1)\,(\mathsf{map}_2\,\mathsf{f}_2\,\mathsf{y}_2)\ )
\end{aligned}
$$

In words, the translation of a functor is the product of a relation transformer (F_R) between functors F_1 and F_2, and a witness that map_1 and map_2 preserve relations.

Such Functors can be used to define a generic fold operation, which typically takes the following form:

$$
\begin{aligned}
&\textbf{data}\ \mu\,((\mathsf{F}, \mathsf{map}) : \mathsf{Functor}) : \star\ \textbf{where} \\
&\quad \mathsf{In} : \mathsf{F}\,(\mu\,(\mathsf{F}, \mathsf{map})) \to \mu\,(\mathsf{F}, \mathsf{map}) \\
&\mathsf{fold} : ((\mathsf{F}, \mathsf{map}) : \mathsf{Functor}) \to (\mathsf{A} : \star) \to (\mathsf{F}\,\mathsf{A} \to \mathsf{A}) \to \mu\,(\mathsf{F}, \mathsf{map}) \to \mathsf{A} \\
&\mathsf{fold}\,(\mathsf{F}, \mathsf{map})\,\mathsf{A}\,\phi\,(\mathsf{In}\,\mathsf{d}) \ =\ \phi\,(\mathsf{map}\,(\mu\,(\mathsf{F}, \mathsf{map}))\,\mathsf{A}\,(\mathsf{fold}\,(\mathsf{F}, \mathsf{map})\,\mathsf{A}\,\phi)\,\mathsf{d})
\end{aligned}
$$

Note that the μ datatype is not strictly positive, so its use would be prohibited in many dependently-typed languages to avoid inconsistency. However, if one restricts oneself to well-behaved functors (yielding strictly positive types), then consistency is restored both in the source and target systems, and the parametricity condition derived for fold is valid.

One can see from the type of fold that it behaves uniformly over $(\mathsf{F}, \mathsf{map})$ as well as over A. By applying $[\![\cdot]\!]$ to fold and its type, this observation can be expressed (and justified) formally and used to reason about fold. Further, every function defined using fold, and in general any function parametrised over any functor enjoys the same kind of property.

Gibbons and Paterson (2009) previously made a similar observation in a categorical setting, showing that fold is a natural transformation between higher-order functors. Their argument heavily relies on categorical semantics and the universal property of fold, while our type-theoretical argument uses the type of fold as a starting point and directly obtains a parametricity property. However some additional work is required to obtain the equivalent property using natural transformations and horizontal compositions from the parametricity property.

6.5 Type equality

Equality between types A and B can be expressed by the following relation, named after Leibniz, which asserts that any proof involving A can be converted to a proof involving B.

```
Equal : ⋆ → ⋆ → ⋆₁
Equal A B = (P : ⋆ → ⋆) → P A → P B
```

An intuitive reading of the type of Equal suggests that inhabitants of that type can only be polymorphic identity functions. Indeed, conversions from P A to P B, for an arbitrary P, cannot depend on the actual values. We would like to apply the axiom of parametricity to recover a formal proof of that result.

Before doing so, we will do a practice round on the similar, but simpler problem of showing that functions of type Id must be (extensionally) the identity function.

```
Id = (A : ⋆) → A → A
```

Using parametricity with arity $n = 1$, we have:

```
paramId : (f : Id) →
          {A : Set} (AR : A → Set)
          {x : A} → (xR : AR x) →
          AR (f A x)
```

Then we can instantiate A_R with the predicate of "being equal to x, the input of f"; and its proof x_R with reflexivity of equality to obtain the desired result.

```
theorem : (f : Id) → (A : ⋆) → (x : A) → x ≡ f A x
theorem f A x = paramId f (_ ≡ _ x) refl
```

The proof of our original proposition follows the same pattern, with a single complication. Because Equal A B is an open term, our parametricity axiom cannot be applied to it directly. There is a simple trick that allows us to proceed though: bind the variables in a dependent pair and apply the axiom to that type. Parametricity then gives us:

```
SomeEqual = (A : ⋆) × (B : ⋆) × Equal A B
paramSomeEqual : (s : SomeEqual) → ⟦SomeEqual⟧ s
```

where

```
⟦Equal⟧ {A} AR {B} BR = λ (e : Equal A B) →
  {P : ⋆ → ⋆} → (PR : {X : Set} → (X → Set) → P X → Set)
  {p : P A} → PR AR x₁ →
  PR BR (e f p)
⟦SomeEqual⟧ (A, B, e) =
  (AR : A → ⋆₁) ×
  (BR : B → ⋆₁) ×
  (⟦Equal⟧ AR BR e)
```

Using this instantiation of the parametricity axiom, we can proceed as in the Id case, with three differences.

- The instantiation of the predicate constructor $\mathsf{P_R}$ takes an extra argument p, which we ignore.
- Because the input and output type are syntactically different, we use heterogeneous equality ($\_ \cong \_$).
- We ignore the predicates $\mathsf{A_R}$ and $\mathsf{B_R}$ constructed by $\mathsf{param}_\cdot$ in the record of type $[\![\mathsf{SomeEqual}]\!]$.

$$\begin{aligned}&\mathsf{theorem} : \forall\,(\mathsf{A}\,\mathsf{B} : \star) \to (\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,\mathsf{B}) \to (\mathsf{P} : \star \to \star)\,(\mathsf{x} : \mathsf{P}\,\mathsf{A}) \to \mathsf{x} \cong \mathsf{e}\,\mathsf{P}\,\mathsf{x}\\ &\mathsf{theorem}\,\mathsf{A}\,\mathsf{B}\,\mathsf{e}\,\mathsf{P}\,\mathsf{x} = \mathsf{q}\\ &\quad \textbf{where}\,(\_,\_,\mathsf{q}) = (\mathsf{param}_{\mathsf{SomeEqual}}\,(\mathsf{A},\mathsf{B},\mathsf{e})\,\{\mathsf{P}\}\,(\lambda\,\mathsf{p} \to ((\_ \cong \_)\,\mathsf{x}))\,\mathsf{refl})\end{aligned}$$

Some points are worth emphasising:

- It is possible to get a result about an open term, even though our axiom only handles closed terms. Still, we get a concrete result (the above theorem) that does not involve any occurrence of the parametricity axiom. This happens because the function constructing predicates $(\lambda\,\mathsf{p} \to ((\_ \cong \_)\,\mathsf{x}))$ precisely discards those occurrences.
- The result is already exposed by Vytiniotis and Weirich (2010), but it is remarkable that its proof is one line long given our framework.
- Because the equality $\_ \cong \_$ is heterogeneous, deriving a substitution principle from it requires Streicher's Axiom K (Hofmann and Streicher, 1996).

  In consequence, it seems that one cannot derive that all proofs of equality are equal from the axiom of parametricity.

7 Discussion

7.1 Related work

Some of the many studies of parametricity have already been mentioned and analysed in the rest of the paper. In this section we compare our work to only a couple of the most relevant pieces of work.

One direction of research is concerned with parametricity in extensions of System F. Our work is directly inspired by Vytiniotis and Weirich (2010), which extend parametricity to (an extension of) Fω: indeed, Fω can be seen as a PTS with one more product rule than System F.

Before that, Takeuti (2004) attempted to extend CC with parametricity. Takeuti asserted parametricity at all types, in a similar way as we do here, in fact extending similar axiom schemes for System F by Plotkin and Abadi

(1993). For each $\alpha : \Box$ and $P : \alpha$, Takeuti defined a relational interpretation $\langle P \rangle$ and a kind $(\!| P : \alpha |\!)$ such that $\langle P \rangle : (\!| P : \alpha |\!)$. Then for each type $T : \star$, he postulated an axiom $\mathsf{param}_T : (\forall x : T.\ \langle T \rangle\ x\ x)$, conjecturing that such axioms did not make the system inconsistent. For closed terms P, Takeuti's translations $\langle P \rangle$ and $(\!| P : \alpha |\!)$ resemble our $[\![P]\!]$ and $[\![\alpha]\!]\ \overline{P}$ respectively (with $n = 2$), but the pattern is obscured by an error in the translation rule for the product $\Box \rightsquigarrow \star$. His omission of a witness x_R for the relationship between values x_1 and x_2 in the rules corresponding to the product $\star \rightsquigarrow \Box$ appears to correspond to a computationally-irrelevant interpretation of $\star$, as we present in Section 6.2.

In previous work (Bernardy, Jansson, and Paterson, 2010) we have shown that the relational interpretation can be generalised to PTSs. Here we extend the results in multiple ways:

- we have annotated the relational interpretation with colours, clarifying the role of each type of quantification, and showing how the translation can take advantage of systems with implicit syntax (Section 3);
- we have proven that our previous inductive relational interpretation of inductive families is correct (Section 4.4);
- we show that part of the meta-theory of parametricity can be internalised into a PTS and that the theory remains consistent (for an important class of systems) (Section 5);
- we have argued in detail why one can assume that two proofs of a given proposition are always related (Section 6.2);
- we have shown on an example that the support of Σ types allows to get results for open types, even with an axiom schema restricted to closed types (Section 6.5);
- we allow for the source and target system to be different.

Bernardy and Lasson (2011) have shown how to construct a logic for parametricity for an arbitrary source PTS (Definition 4) which is as consistent as the source PTS.

Besides supporting more sorts and function spaces, an orthogonal extension of parametricity theory is to support impure features in the system. For example, (Johann and Voigtländer, 2006) studied how explicit strictness modifies parametricity results. It is not obvious how to support such extensions in our framework.

Another direction of research is concerned with better understanding of parametricity. Here we shall mention only (Wadler, 2007), which gives a particularly lucid presentation of the abstraction theorem, as the inverse

of Girard's Representation theorem (Girard, 1972): Reynolds gives an embedding from System Fto second order logic, while Girard gives the corresponding projection. Our version of the abstraction theorem differs in the following aspects from that of Wadler (and to our knowledge all others):

1. Instead of targeting a logic, we target its *propositions-as-types* interpretation, expressed in a PTS.
2. We abstract from the details of the systems, generalising to a class of PTSs.
3. We add that the translation function used to interpret types as relations can also be used to interpret terms as witnesses of those relations. In short, the $[\![A]\!]$ part of $\Gamma \vdash A : B \implies [\![\Gamma]\!] \vdash [\![A]\!] : [\![B]\!]\,\overline{A}$ is new. This additional insight depends heavily on using the interpretation of propositions as types.

The question of how Girard's projection generalises to arbitrary PTSs naturally arises, and is addressed by Bernardy and Lasson (2011).

It also appears that the function $[\![\cdot]\!]$ (for the unary case) has been discovered independently by Monnier and Haguenauer (2010), for a very different purpose. They use $[\![\cdot]\!]$ as a compilation function from CC to a language with singleton types as the sole way to express dependencies from values to types. Their goal is to enforce phase-distinction between compile-time and run-time. Type preservation of the translation scheme is the main formal property presented by Monnier and Haguenauer. We remark that this property is a specialisation of our abstraction theorem for CC. Another lesson learnt from this parallel is that the unary $[\![\cdot]\!]$ generates singleton types.

7.2 Future work

Our explanation of parametricity for dependent types has opened a whole range of interesting topics for future work.

We should investigate whether our framework can be applied (and extended if need be) to more exotic systems, for example those incorporating strictness annotations (seq) or non-termination.

We gave an interpretation of the axiom of parametricity as a compilation pass to a language not requiring the axiom. It would also be interesting to, instead, extend the β-reduction rules to support the axiom.

The target PTS that we constructed has typed individuals, whereas many logics for parametricity have untyped individuals. Girard's representation theorem shows that, in System F, such type information can be recovered and is therefore not essential. It would be worthwhile to generalise that result to arbitrary PTSs.

We presented only simple examples. Applying the results to more substantial applications should be done as well. In particular, we hope that our results opens the door to a more streamlined way of getting free theorems for domain-specific programming languages. One would proceed along the following steps:

1. model the domain-specific languages within a dependently-typed language;
2. use $[\![\cdot]\!]$ to obtain parametricity properties of any function of interest;
3. prove correctness by using the properties.

We think that the above process is an economical way to work with parametricity for extended type systems. Indeed, developing languages with exotic type systems as an embedding in a dependently-typed language is increasingly popular (Oury and Swierstra, 2008), and that is the first step in the above process. By providing an automatic second step, we hope to spare language designers the effort to adapt Reynolds' abstraction theorem for new type systems in an ad-hoc way.

Acknowledgements Thanks to Andreas Abel, Thierry Coquand, Nils Anders Danielsson, Peter Dybjer, Marc Lasson, Guilhem Moulin, Ulf Norell, Nicolas Pouillard, Janis Voigtländer, Stephanie Weirich and anonymous reviewers for providing us with very valuable feedback.

A Proof of the abstraction theorem

In this appendix we sketch the proof of our main theorem, using the following lemmas:

Lemma 22 (translation preserves β-reduction)**.**

$$A \longrightarrow_\beta^* A' \implies [\![A]\!] \longrightarrow_\beta^* [\![A']\!]$$

Proof sketch. The proof proceeds by induction on the derivation of $A \longrightarrow_\beta^* A'$. The interesting case is where the term A is a β-redex $(\lambda x\colon B.\ C)\ b$. That case relies on the way $[\![\cdot]\!]$ interacts with substitution:

$$[\![b[x \mapsto C]]\!] = [\![b]\!][\overline{x} \mapsto \overline{C}][x_R \mapsto [\![C]\!]]$$

The remaining cases are congruences. □

Lemma 23 ($[\![s]\!]$ is well-typed)**.** *For each sort $s \in \mathcal{S}$ we have $\vdash [\![s]\!] : \overline{s} \to \lceil s \rceil'$ in S^t.*

	$\boxed{\Gamma \vdash A : B}$	$\Longrightarrow$	$\boxed{[\![\Gamma]\!] \vdash [\![A]\!] : [\![B]\!]\ \overline{A}}$
axiom	$\vdash s : s'$		$\vdash (\lambda \overline{x{:}s}.\ \overline{x} \to \lceil s \rceil) : \overline{s} \to \lceil s' \rceil$
start	$\dfrac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A}$		$\dfrac{[\![\Gamma]\!] \vdash [\![A]\!] : \overline{A} \to \lceil s \rceil}{[\![\Gamma]\!], \overline{x{:}A}, x_R : [\![A]\!]\ \overline{x} \vdash x_R : [\![A]\!]\ \overline{x}}$
weakening	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x{:}C \vdash A : B}$		$\dfrac{[\![\Gamma]\!] \vdash [\![A]\!] : [\![B]\!]\ \overline{A} \quad [\![\Gamma]\!] \vdash [\![C]\!] : \overline{C} \to \lceil s \rceil}{[\![\Gamma]\!], \overline{x{:}C}, x_R : [\![C]\!]\ \overline{x} \vdash [\![A]\!] : [\![B]\!]\ \overline{A}}$
product	$\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash (\forall^k x{:}A.\ B) : s_3}$		$\dfrac{[\![\Gamma]\!] \vdash [\![A]\!] : \overline{A} \to \lceil s_1 \rceil \quad [\![\Gamma]\!], \overline{x{:}A}, x_R : [\![A]\!]\ \overline{x} \vdash [\![B]\!] : \overline{B} \to \lceil s_2 \rceil}{[\![\Gamma]\!] \vdash (\lambda f : \overline{(\forall^k x{:}A.\ B)}.\ \forall^{k_i} \overline{x{:}A}.\ \forall^{k_r} x_R : [\![A]\!]\ \overline{x}.\ [\![B]\!]\ (\overline{f\ x})) : (\overline{\forall^k x{:}A.\ B}) \to \lceil s_3 \rceil}$
application	$\dfrac{\Gamma \vdash F : (\forall^k x{:}A.\ B) \quad \Gamma \vdash a : A}{\Gamma \vdash F \bullet_k A : B[x \mapsto a]}$		$\dfrac{[\![\Gamma]\!] \vdash [\![F]\!] : (\forall^{k_i} \overline{x{:}A}.\ \forall^{k_r} x_R : [\![A]\!]\ \overline{x}.\ [\![B]\!]\ (\overline{F \bullet_k x})) \quad [\![\Gamma]\!] \vdash [\![a]\!] : [\![A]\!]\ \overline{a}}{[\![\Gamma]\!] \vdash [\![F]\!] \bullet_{k_i} \overline{a} \bullet_{k_r} [\![a]\!] : [\![B[x \mapsto a]]\!]\ (\overline{F \bullet_k a})}$
abstraction	$\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2 \quad \Gamma, x{:}A \vdash b : B}{\Gamma \vdash (\lambda^k x{:}A.\ b) : (\forall^k x{:}A.\ B)}$		$\dfrac{[\![\Gamma]\!] \vdash [\![A]\!] : \overline{A} \to \lceil s_1 \rceil \quad [\![\Gamma]\!], \overline{x{:}A}, x_R : [\![A]\!]\ \overline{x} \vdash [\![B]\!] : \overline{B} \to \lceil s_2 \rceil \quad [\![\Gamma]\!], \overline{x{:}A}, x_R : [\![A]\!]\ \overline{x} \vdash [\![b]\!] : [\![B]\!]\ \overline{b}}{[\![\Gamma]\!] \vdash (\lambda^{k_i} \overline{x{:}A}.\ \lambda^{k_r} x_R : [\![A]\!]\ \overline{x}.\ [\![b]\!]) : (\forall^{k_i} \overline{x{:}A}.\ \forall^{k_r} x_R : [\![A]\!]\ \overline{x}.\ [\![B]\!]\ \overline{b})}$
conversion	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_\beta B'}{\Gamma \vdash A : B'}$		$\dfrac{[\![\Gamma]\!] \vdash [\![A]\!] : [\![B]\!]\ \overline{A} \quad [\![\Gamma]\!] \vdash [\![B']\!] : \overline{B'} \to \lceil s \rceil \quad [\![B]\!] =_\beta [\![B']\!]}{[\![\Gamma]\!] \vdash [\![A]\!] : [\![B']\!]\ \overline{A}}$

Figure 1.6: Outline of a proof of Theorem 11 by induction over the derivation of $\Gamma \vdash A : B$.

Proof. From the requirements for a sort $s \in \mathcal{S}$ in Definition 4 we can infer (in S^t)

$$\dfrac{\dfrac{\dfrac{}{\vdash s : s'}}{\overline{x{:}s} \vdash x_i : s}\,\text{st} \qquad \dfrac{\dfrac{}{\vdash \lceil s \rceil : \lceil s \rceil'} \qquad \dfrac{}{\vdash s : s'}}{\overline{x{:}s} \vdash \lceil s \rceil : \lceil s \rceil'}\,\text{wk}}{\overline{x{:}s} \vdash \overline{x} \to \lceil s \rceil : \lceil s \rceil'}\; s \rightsquigarrow \lceil s \rceil' \qquad \dfrac{\dfrac{}{\vdash s : s'} \qquad \dfrac{}{\vdash \lceil s \rceil' : \lceil s \rceil''}}{\vdash \overline{s} \to \lceil s \rceil' : \lceil s \rceil''}\; s' \rightsquigarrow \lceil s \rceil''}{\vdash (\lambda \overline{x{:}s}.\ \overline{x} \to \lceil s \rceil) : \overline{s} \to \lceil s \rceil'}\,\text{abs}$$

□

Theorem 24 (abstraction)**.** *If the PTS S^t reflects S,*

$$\Gamma \vdash_S A : B \implies [\![\Gamma]\!] \vdash_{S^t} [\![A]\!] : [\![B]\!]\, \overline{A}$$

Proof sketch. A derivation of $[\![\Gamma]\!] \vdash [\![A]\!] : [\![B]\!]\, \overline{A}$ in S^t is constructed by induction on the derivation of $\Gamma \vdash A : B$ in S, using the syntactic properties of PTSs. We have one case for each typing rule: each type rule translates to a portion of a corresponding relational typing judgement, as shown in Figure 1.6.

For concision, the proof sketch uses a variant form of the abstraction rule; equivalence of the two systems follows from Barendregt (1992, Lemma 5.2.13). The conversion case uses Lemma 22. □

Proof details. The following propositions are proved by simultaneous induction on the typing judgement:

lem $\Gamma \vdash_S A : s \implies [\![\Gamma]\!] \vdash_{S^t} \overline{A} : \overline{s}$.

Proved by the thinning lemma (Barendregt, 1992, Lemma 5.2.12, p. 220). For each A_i, erase from the context $[\![\Gamma]\!]$ the relational variables and j-indexed variables such that $j \neq i$. The legality of the context is ensured by **ind**.

ind $\Gamma \vdash_S A : B \implies [\![\Gamma]\!] \vdash_{S^t} [\![A]\!] : [\![B]\!]\, \overline{A}$.

The proof proceeds by induction on the derivation of $\Gamma \vdash A : B$. We have one case for each typing rule: each type rule translates to a portion of a corresponding relational typing judgement; and we detail them in the rest of the section. The construction of the derivation makes use of the propositions **lem**, **ind** and **ind'** (on smaller judgements).

ind' $\Gamma \vdash_S B : s \implies [\![\Gamma]\!] \vdash_{S^t} [\![B]\!] : \overline{B} \to \lceil s \rceil$

Corollary of **ind**.

We proceed with the case analysis for the proof of **ind**.

axiom $\mathsf{c} : s$ If c is not a sort, the proposition is assumed as an hypothesis. For the remaining case $s : t$ we have

$$\dfrac{\dfrac{\vdots\ \text{Lemma 23}}{\vdash [\![s]\!] : \overline{s} \to \lceil s \rceil'} \qquad \dfrac{\dfrac{\vdots\ \text{Lemma 23}}{\vdash (\lambda \overline{\mathsf{x}{:}t}.\ \overline{\mathsf{x}} \to \lceil t \rceil) : \overline{t} \to \lceil t \rceil'} \qquad \dfrac{}{\vdash s : t}}{\vdash (\lambda \overline{\mathsf{x}{:}t}.\ \overline{\mathsf{x}} \to \lceil t \rceil)\,\overline{s} : \lceil t \rceil'}\ \text{app}}{\vdash [\![s]\!] : (\lambda \overline{\mathsf{x}{:}t}.\ \overline{\mathsf{x}} \to \lceil t \rceil)\,\overline{s}}\ \text{conv},\ \lceil s \rceil' = \lceil t \rceil$$

start

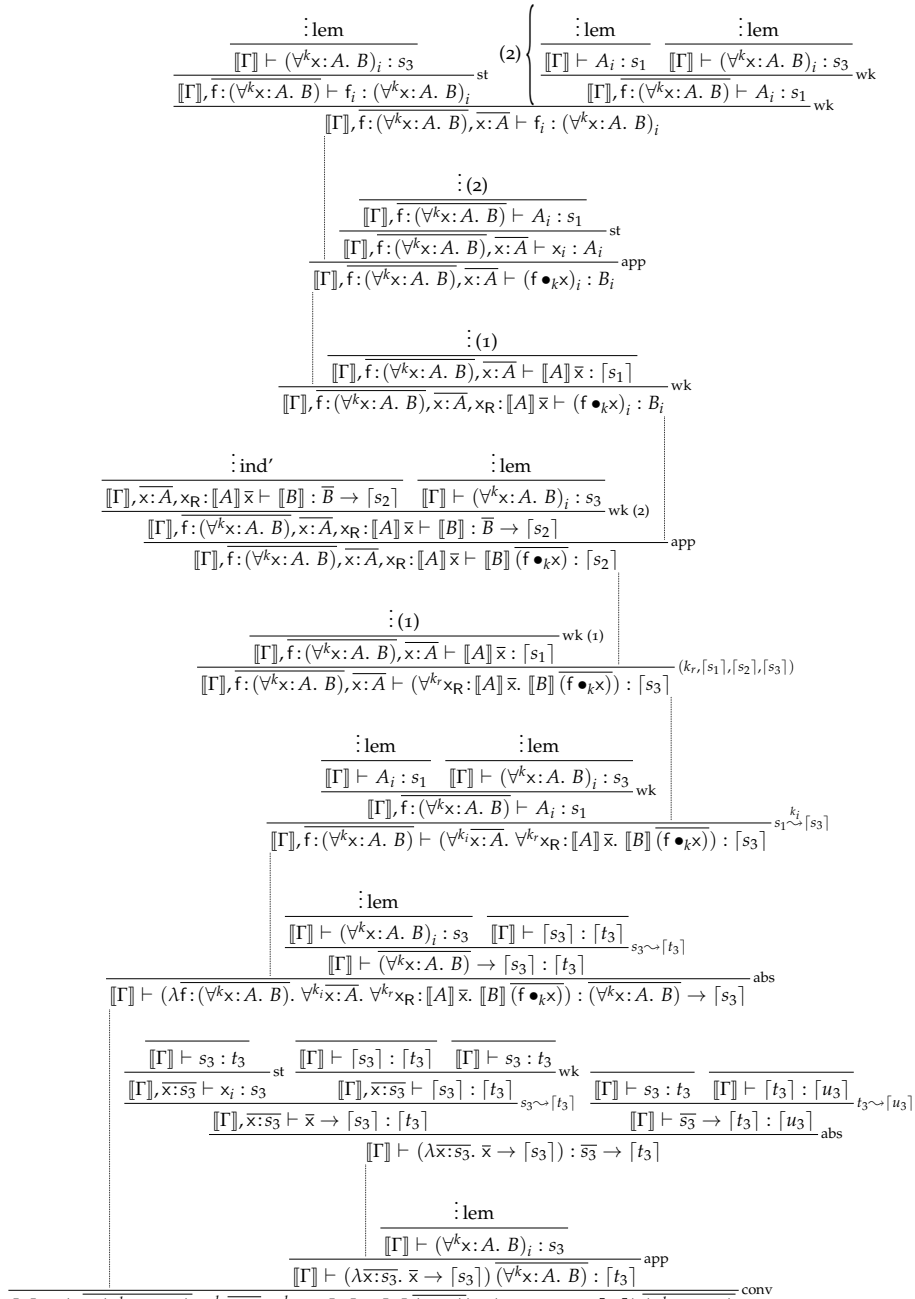
$$\dfrac{\dfrac{\dfrac{\vdots\ \text{ind}'}{[\![\Gamma]\!] \vdash [\![A]\!] : \overline{A} \to \lceil s \rceil} \quad \dfrac{\vdots\ \text{lem}}{[\![\Gamma]\!] \vdash A_i : s}\ \text{wk}}{[\![\Gamma]\!], \overline{\mathsf{x}{:}A} \vdash [\![A]\!] : \overline{A} \to \lceil s \rceil} \quad \dfrac{\dfrac{\vdots\ \text{lem}}{[\![\Gamma]\!] \vdash A_i : s}\ \text{st}}{[\![\Gamma]\!], \overline{\mathsf{x}{:}A} \vdash \mathsf{x}_i : A_i}\ \text{app}}{[\![\Gamma]\!], \overline{\mathsf{x}{:}A} \vdash [\![A]\!]\,\overline{\mathsf{x}} : \lceil s \rceil}\ \text{st}}{[\![\Gamma]\!], \overline{\mathsf{x}{:}A}, \mathsf{x}_R : [\![A]\!]\,\overline{\mathsf{x}} \vdash \mathsf{x}_R : [\![A]\!]\,\overline{\mathsf{x}}}\ \text{st}$$

weakening

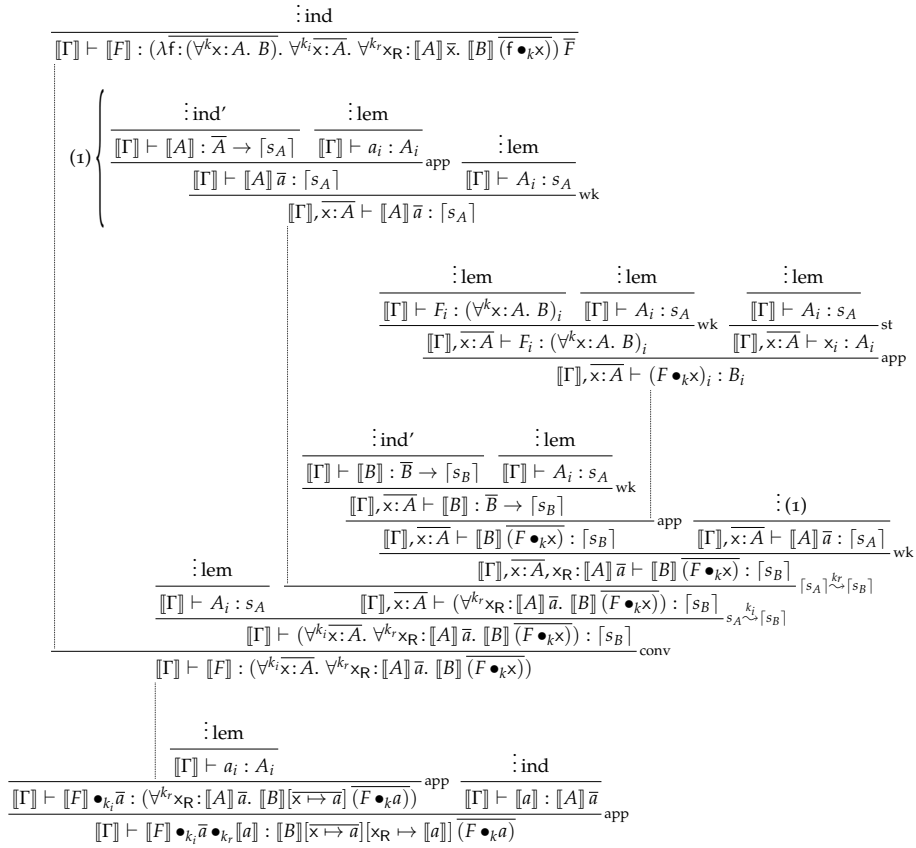
$$\dfrac{\dfrac{\dfrac{\vdots\ \text{ind}}{[\![\Gamma]\!] \vdash [\![A]\!] : [\![B]\!]\,\overline{A}} \quad \dfrac{\vdots\ \text{lem}}{[\![\Gamma]\!] \vdash C_i : s}\ \text{wk}}{[\![\Gamma]\!], \overline{\mathsf{x}{:}C} \vdash [\![A]\!] : [\![B]\!]\,\overline{A}} \quad \dfrac{\dfrac{\dfrac{\vdots\ \text{ind}'}{[\![\Gamma]\!] \vdash [\![C]\!] : \overline{C} \to \lceil s \rceil} \quad \dfrac{\vdots\ \text{lem}}{[\![\Gamma]\!] \vdash C_i : s}\ \text{wk}}{[\![\Gamma]\!], \overline{\mathsf{x}{:}C} \vdash [\![C]\!] : \overline{C} \to \lceil s \rceil} \quad \dfrac{\dfrac{\vdots\ \text{lem}}{[\![\Gamma]\!] \vdash C_i : s}\ \text{st}}{[\![\Gamma]\!], \overline{\mathsf{x}{:}C} \vdash \mathsf{x}_i : C_i}\ \text{app}}{[\![\Gamma]\!], \overline{\mathsf{x}{:}C} \vdash [\![C]\!]\,\overline{\mathsf{x}} : \lceil s \rceil}\ \text{wk}}{[\![\Gamma]\!], \overline{\mathsf{x}{:}C}, \mathsf{x}_R : [\![C]\!]\,\overline{\mathsf{x}} \vdash [\![A]\!] : [\![B]\!]\,\overline{A}}\ \text{wk}$$

product (k, s_1, s_2, s_3)

$$(1)\left\{\dfrac{\dfrac{\dfrac{\vdots\ \text{ind}'}{[\![\Gamma]\!] \vdash [\![A]\!] : \overline{A} \to \lceil s_1 \rceil} \quad \dfrac{\vdots\ \text{lem}}{[\![\Gamma]\!] \vdash A_i : s_1}\ \text{wk}}{[\![\Gamma]\!], \overline{\mathsf{x}{:}A} \vdash [\![A]\!] : \overline{A} \to \lceil s_1 \rceil} \quad \dfrac{\dfrac{\vdots\ \text{lem}}{[\![\Gamma]\!] \vdash A_i : s_1}\ \text{st}}{[\![\Gamma]\!], \overline{\mathsf{x}{:}A} \vdash \mathsf{x}_i : A_i}\ \text{app}}{[\![\Gamma]\!], \overline{\mathsf{x}{:}A} \vdash [\![A]\!]\,\overline{\mathsf{x}} : \lceil s_1 \rceil} \quad \dfrac{\vdots\ \text{lem}}{[\![\Gamma]\!] \vdash (\forall^k \mathsf{x}{:}A.\ B)_i : s_3}\ \text{wk}\ (1)}{[\![\Gamma]\!], \mathsf{f} : (\forall^k \mathsf{x}{:}A.\ B), \overline{\mathsf{x}{:}A} \vdash [\![A]\!]\,\overline{\mathsf{x}} : \lceil s_1 \rceil}\right.$$



application



abstraction We apply the generation lemma (Barendregt, 1992, Theorem 5.2.13, case 3) on $\Gamma \vdash (\forall^k \mathsf{x}\colon A.\ B) : s$. We get: $\exists s_A \rightsquigarrow s_B$ such that

- $\Gamma \vdash A : s_A$
- $\Gamma, \mathsf{x}\colon A \vdash B : s_B$
- $s =_\beta s_B$

Since sorts are irreducible, the last equation becomes $s = s_B$, so we have: $\exists s_A \rightsquigarrow s$ such that

- $\Gamma \vdash A : s_A$
- $\Gamma, \mathsf{x}\colon A \vdash B : s$

Induction on the judgements constructed above is valid, because the generation lemma generates smaller judgements. It yields:

- $[\![\Gamma]\!] \vdash [\![A]\!] : [\![s_A]\!]\,\overline{A}$
- $[\![\Gamma]\!], \overline{\mathsf{x}:A}, \mathsf{x}_\mathsf{R} : [\![A]\!]\,\overline{\mathsf{x}} \vdash [\![B]\!] : [\![s]\!]\,\overline{B}.$

and these judgements will be used in the construction of the target derivation.

First we show that the type is properly sorted:

$$
\dfrac{
\dfrac{
\dfrac{\vdots\,\mathrm{lem}}{[\![\Gamma]\!] \vdash (\forall^k \mathsf{x}{:}A.\ B)_i : s}\,\mathrm{st}
\qquad
\dfrac{\vdots\,(4)}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\ B)} \vdash A_i : s_A}\,\mathrm{wk}
}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\ B)}, \overline{\mathsf{x}:A} \vdash \mathsf{f}_i : (\forall^k \mathsf{x}{:}A.\ B)_i}
\qquad
\dfrac{
\dfrac{\vdots\,(4)}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\ B)} \vdash A_i : s_A}\,\mathrm{st}
}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\ B)}, \overline{\mathsf{x}:A} \vdash \mathsf{x}_i : A_i}
}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\ B)}, \overline{\mathsf{x}:A} \vdash (\mathsf{f} \bullet_k \mathsf{x})_i : B_i}\,\mathrm{app}
$$

$$
\dfrac{
\dfrac{\vdots\,\mathrm{ind}'}{[\![\Gamma]\!] \vdash [\![B]\!] : \overline{B} \to \lceil s \rceil}
\qquad
\dfrac{\vdots\,\mathrm{lem}}{[\![\Gamma]\!] \vdash (\forall^k \mathsf{x}{:}A.\ B)_i : \lceil s \rceil}\,\mathrm{wk}
}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\ B)} \vdash [\![B]\!] : \overline{B} \to \lceil s \rceil}\,\mathrm{wk}
\qquad (4)\left\{
\dfrac{
\dfrac{\vdots\,\mathrm{lem}}{[\![\Gamma]\!] \vdash A_i : s_A}
\qquad
\dfrac{\vdots\,\mathrm{lem}}{[\![\Gamma]\!] \vdash (\forall^k \mathsf{x}{:}A.\ B)_i : s}\,\mathrm{wk}
}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\ B)} \vdash A_i : s_A}\,\mathrm{wk}
\right.
$$

$$
\dfrac{}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\ B)}, \overline{\mathsf{x}:A} \vdash [\![B]\!] : \overline{B} \to \lceil s \rceil}\,\mathrm{app}
$$

$$
\dfrac{}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\ B)}, \overline{\mathsf{x}:A} \vdash [\![B]\!]\,\overline{(\mathsf{f} \bullet_k \mathsf{x})} : \lceil s \rceil}
$$

$$
\dfrac{
\dfrac{\vdots\,(2)}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\ B)}, \overline{\mathsf{x}:A} \vdash [\![A]\!]\,\overline{\mathsf{x}} : \lceil s_A \rceil}\,\mathrm{wk}
}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\ B)}, \overline{\mathsf{x}:A}, \mathsf{x}_\mathsf{R} : [\![A]\!]\,\overline{\mathsf{x}} \vdash [\![B]\!]\,\overline{(\mathsf{f} \bullet_k \mathsf{x})} : \lceil s \rceil}\,\mathrm{wk}
$$

(S)

$$
(2)\left\{
\dfrac{
\dfrac{\vdots\,(1)}{[\![\Gamma]\!], \overline{\mathsf{x}:A} \vdash [\![A]\!]\,\overline{\mathsf{x}} : \lceil s_A \rceil}
\qquad
\dfrac{\vdots\,\mathrm{lem}}{[\![\Gamma]\!] \vdash (\forall^k \mathsf{x}{:}A.\ B)_i : s}\,\mathrm{wk}\ (1)
}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\ B)}, \overline{\mathsf{x}:A} \vdash [\![A]\!]\,\overline{\mathsf{x}} : \lceil s_A \rceil}
\right.
\qquad
\dfrac{}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\ B)}, \overline{\mathsf{x}:A} \vdash (\forall^{k_r} \mathsf{x}_\mathsf{R}{:}[\![A]\!]\,\overline{\mathsf{x}}.\ [\![B]\!]\,\overline{(\mathsf{f} \bullet_k \mathsf{x})}) : \lceil s \rceil}\ {}^{\lceil s_A \rceil}{\sim}^{k_r}_{\lceil s \rceil}
$$

$$
\dfrac{
\dfrac{\vdots\,\mathrm{lem}}{[\![\Gamma]\!] \vdash A_i : s_A}
\qquad
\dfrac{\vdots\,\mathrm{lem}}{[\![\Gamma]\!] \vdash (\forall^k \mathsf{x}{:}A.\ B)_i : s}\,\mathrm{wk}
}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\ B)} \vdash A_i : s_A}\,\mathrm{wk}
\qquad
\dfrac{}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\ B)} \vdash (\forall^{k_i} \overline{\mathsf{x}:A}.\ \forall^{k_r} \mathsf{x}_\mathsf{R}{:}[\![A]\!]\,\overline{\mathsf{x}}.\ [\![B]\!]\,\overline{(\mathsf{f} \bullet_k \mathsf{x})}) : \lceil s \rceil}\ {}^{s_A}_{\lceil s \rceil}{\sim}^{k_i}
$$

$$
\dfrac{
\dfrac{\vdots\,\mathrm{lem}}{[\![\Gamma]\!] \vdash (\forall^k \mathsf{x}{:}A.\ B)_i : s}
\qquad
\dfrac{
\dfrac{}{[\![\Gamma]\!] \vdash \lceil s \rceil : \lceil t \rceil}
\qquad
\dfrac{\vdots\,\mathrm{lem}}{[\![\Gamma]\!] \vdash (\forall^k \mathsf{x}{:}A.\ B)_i : s}\,\mathrm{wk}
}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\ B)} \vdash \lceil s \rceil : \lceil t \rceil}\ {}^{s}_{\lceil t \rceil}{\sim}
}{[\![\Gamma]\!] \vdash (\forall \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\ B)}.\ \lceil s \rceil) : \lceil t \rceil}\,\mathrm{abs}
$$

$$
\dfrac{
\dfrac{\vdots\,\mathrm{lem}}{[\![\Gamma]\!] \vdash (\lambda^k \mathsf{x}{:}A.\ b)_i : (\forall^k \mathsf{x}{:}A.\ B)_i}
}{[\![\Gamma]\!] \vdash (\lambda \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\ B)}.\ \forall^{k_i} \overline{\mathsf{x}:A}.\ \forall^{k_r} \mathsf{x}_\mathsf{R}{:}[\![A]\!]\,\overline{\mathsf{x}}.\ [\![B]\!]\,\overline{(\mathsf{f} \bullet_k \mathsf{x})})\ \overline{(\lambda^k \mathsf{x}{:}A.\ b)} : \lceil s \rceil}\,\mathrm{app}
$$

Then use this in the top-level tree:

$$
\dfrac{
\dfrac{
\dfrac{\vdots\ \text{ind}'}{[\![\Gamma]\!] \vdash [\![B]\!] : \overline{B} \to \lceil s \rceil}\quad
\dfrac{\vdots\ \text{lem}}{[\![\Gamma]\!] \vdash b_i : B_i}\ \text{app}
}{[\![\Gamma]\!] \vdash [\![B]\!]\,\overline{b} : \lceil s \rceil}\quad
\dfrac{\vdots\ \text{lem}}{[\![\Gamma]\!] \vdash A_i : s_A}\ \text{wk}
}{[\![\Gamma]\!], \overline{x : A} \vdash [\![B]\!]\,\overline{b} : \lceil s \rceil}\quad
\dfrac{\vdots\ (1)}{[\![\Gamma]\!], \overline{x : A} \vdash [\![A]\!]\,\overline{x} : \lceil s_A \rceil}\ \text{wk}
}{[\![\Gamma]\!], \overline{x : A}, x_R : [\![A]\!]\,\overline{x} \vdash [\![B]\!]\,\overline{b} : \lceil s \rceil}
\quad (3)
$$

$$
\dfrac{
\dfrac{
\dfrac{\vdots\ \text{ind}'}{[\![\Gamma]\!] \vdash [\![A]\!] : \overline{A} \to \lceil s_A \rceil}\quad
\dfrac{\vdots\ \text{lem}}{[\![\Gamma]\!] \vdash A_i : s_A}\ \text{wk}
}{[\![\Gamma]\!], \overline{x : A} \vdash [\![A]\!] : \overline{A} \to \lceil s_A \rceil}\quad
\dfrac{\vdots\ \text{lem}}{[\![\Gamma]\!] \vdash A_i : s_A}\ \text{st}
}{[\![\Gamma]\!], \overline{x : A} \vdash x_i : A_i}\ \text{app}
}{[\![\Gamma]\!], \overline{x : A} \vdash [\![A]\!]\,\overline{x} : \lceil s_A \rceil}\ \lceil s_A \rceil \overset{k_r}{\sim} \lceil s \rceil
\quad (1)
$$

$$
\dfrac{
\dfrac{
\dfrac{\vdots\ \text{ind}}{[\![\Gamma]\!] \vdash [\![b]\!] : [\![B]\!]\,\overline{b}}\quad
\dfrac{\vdots\ \text{lem}}{[\![\Gamma]\!] \vdash A_i : s_A}\ \text{wk}
}{[\![\Gamma]\!], \overline{x : A} \vdash [\![b]\!] : [\![B]\!]\,\overline{b}}\quad
\dfrac{\vdots\ (1)}{[\![\Gamma]\!], \overline{x : A} \vdash [\![A]\!]\,\overline{x} : \lceil s_A \rceil}\ \text{wk}
}{[\![\Gamma]\!], \overline{x : A}, x_R : [\![A]\!]\,\overline{x} \vdash [\![b]\!] : [\![B]\!]\,\overline{b}}\ \text{abs}
}{[\![\Gamma]\!], \overline{x : A} \vdash (\lambda^{k_r} x_R : [\![A]\!]\,\overline{x}.\ [\![b]\!]) : (\forall^{k_r} x_R : [\![A]\!]\,\overline{x}.\ [\![B]\!]\,\overline{b})}
$$

$$
\dfrac{
\dfrac{
\dfrac{\vdots\ \text{lem}}{[\![\Gamma]\!] \vdash A_i : s_A}\quad
\dfrac{\vdots\ (3)}{[\![\Gamma]\!], \overline{x : A} \vdash (\forall^{k_r} x_R : [\![A]\!]\,\overline{x}.\ [\![B]\!]\,\overline{b}) : \lceil s \rceil}
}{[\![\Gamma]\!] \vdash (\forall^{k_i} \overline{x : A}.\ \forall^{k_r} x_R : [\![A]\!]\,\overline{x}.\ [\![B]\!]\,\overline{b}) : \lceil s \rceil}\ s_A \overset{k_i}{\sim} \lceil s \rceil
}{[\![\Gamma]\!] \vdash (\lambda^{k_i} \overline{x : A}.\ \lambda^{k_r} x_R : [\![A]\!]\,\overline{x}.\ [\![b]\!]) : (\forall^{k_i} \overline{x : A}.\ \forall^{k_r} x_R : [\![A]\!]\,\overline{x}.\ [\![B]\!]\,\overline{b})}\ \text{abs}
$$

$$
\dfrac{
\dfrac{\vdots\ (S)}{[\![\Gamma]\!] \vdash (\lambda \overline{f : (\forall^k x : A.\ B)}.\ \forall^{k_i} \overline{x : A}.\ \forall^{k_r} x_R : [\![A]\!]\,\overline{x}.\ [\![B]\!]\ \overline{(f \bullet_k x)})\ \overline{(\lambda^k x : A.\ b)} : \lceil s \rceil}\ \text{app}
}{[\![\Gamma]\!] \vdash (\lambda^{k_i} \overline{x : A}.\ \lambda^{k_r} x_R : [\![A]\!]\,\overline{x}.\ [\![b]\!]) : (\lambda \overline{f : (\forall^k x : A.\ B)}.\ \forall^{k_i} \overline{x : A}.\ \forall^{k_r} x_R : [\![A]\!]\,\overline{x}.\ [\![B]\!]\ \overline{(f \bullet_k x)})\ \overline{(\lambda^k x : A.\ b)}}\ \text{conv}
$$

conversion

$$
\dfrac{
\dfrac{\vdots\ \text{ind}}{[\![\Gamma]\!] \vdash [\![A]\!] : [\![B]\!]\,\overline{A}}\quad
\dfrac{
\dfrac{\vdots\ \text{ind}'}{[\![\Gamma]\!] \vdash [\![B]\!]' : \overline{B'} \to \lceil s \rceil}\quad
\dfrac{\vdots\ \text{lem}}{[\![\Gamma]\!] \vdash A_i : B'_i}\ \text{app}
}{[\![\Gamma]\!] \vdash [\![B]\!]'\,\overline{A} : \lceil s \rceil}\ \text{conv}
}{[\![\Gamma]\!] \vdash [\![A]\!] : [\![B]\!]'\,\overline{A}}
$$

The β-equality constraint $([\![B]\!]\,\overline{A} =_\beta [\![B']\!]\,\overline{A})$ holds because $[\![\cdot]\!]$ preserves β-equivalence (Lemma 22). □

Paper II

Realizability and Parametricity in Pure Type Systems

The following paper is an extended version of a paper with the same title appearing in the proceedings of FoSSaCS 2011.

Realizability and Parametricity in Pure Type Systems

Jean-Philippe Bernardy Marc Lasson

Abstract

We describe a systematic method to build a logic from any programming language described as a Pure Type System (PTS). The formulas of this logic express properties about programs. We define a parametricity theory about programs and a realizability theory for the logic. The logic is expressive enough to internalize both theories. Thanks to the PTS setting, we abstract most idiosyncrasies specific to particular type theories. This confers generality to the results, and reveals parallels between parametricity and realizability.

1 Introduction

During the past decades, a recurring goal among logicians was to give a computational interpretation of the reasoning behind mathematical proofs. In this paper we adopt the converse approach: we give a systematic way to build a logic from a programming language. The structure of the programming language is replicated at the level of the logic: the expressive power of the logic (e.g. the ability of expressing conjunctions) is directly conditioned by the constructions available in the programming language (e.g. presence of products).

We use the framework of Pure Type Systems (PTS) to represent both the starting programming language and the logic obtained by our construction. A PTS (Barendregt, 1992; Berardi, 1989) is a generalized λ-calculus where the syntax for terms and types are unified. Many systems can be expressed as PTSs, including the simply typed λ-calculus, Girard and Reynolds' polymorphic λ-calculus (System F) and its extension System Fω, Coquand's Calculus of Constructions, as well as some exotic, and even inconsistent systems such as λU (Girard, 1972). PTSs can model the functional core of many modern programming languages (Haskell, Objective Caml) and proof assistants (Coq (The Coq development team, 2010), Agda (Norell, 2007), Epigram (McBride and McKinna, 2004)). This unified framework provides meta-theoretic results such as substitution lemmas, subject reduction and uniqueness of types.

In Section 3, we describe a transformation which maps any PTS P to a PTS P^2. The starting PTS P will be viewed as a programming language in which live *types* and *programs* and P^2 will be viewed as a proof system in which live *proofs* and *formulas*. The logic P^2 is expressive enough to state

properties about the programs. It is therefore a setting of choice to develop a parametricity and a realizability theory.

Parametricity. Reynolds (1983) originally developed the theory of parametricity to capture the meaning of types of his polymorphic λ-calculus (equivalent to Girard's System F). Each closed type can be interpreted as a predicate that all its inhabitants satisfy. Reynolds' approach to parametricity has proven to be a successful tool: applications range from program transformations to speeding up program testing (Wadler, 1989; Gill, Launchbury, and Peyton Jones, 1993; Bernardy, Jansson, and Claessen, 2010).

Parametricity theory can be adapted to other λ-calculi, and for each calculus, parametricity predicates are expressed in a corresponding logic. For example, Abadi, Cardelli, and Curien (1993) remark that the simply-typed lambda calculus corresponds to LCF (Milner, 1972). For System F, predicates can be expressed in second order predicate logic, in one variant or another (Abadi, Cardelli, and Curien, 1993; Mairson, 1991; Wadler, 2007). More recently, Bernardy, Jansson, and Paterson (2010) have shown that parametricity conditions for a reflective PTS can be expressed in the PTS itself.

Realizability. The notion of realizability was first introduced by Kleene (1945) in his seminal paper. The idea of relating programs and formulas, in order to study their constructive content, was then widely used in proof theory. For example, it provides tools for proving that an axiom is not derivable in a system (excluded middle in (Kleene, 1971; Troelstra, 1998)) or that intuitionistic systems satisfy the *existence property*[1] (Harrop, 1956; Troelstra, 1998); see Van Oosten (2002) for a historical account of realizability.

Originally, Kleene represented programs as integers in a theory of recursive functions. Later, this technique has been extended to other notions of programs like combinator algebra (Staples, 1973; Troelstra, 1998) or terms of Gödel's System T (Kreisel, 1959; Troelstra, 1998) in Kreisel's modified realizability. In this article, we generalize the latter approach by using an arbitrary pure type system as the language of programs.

Krivine (1997) and Leivant (1990) have used realizability to prove Girard's representation theorem[2] (Girard, 1972) and to build a general framework for extracting programs from proofs in second-order logic (Krivine and Parigot, 1990). In this paper, we extend Krivine's methodology to languages with dependent types, like Paulin-Mohring (1989*a*); Paulin-Mohring

[1] If $\forall x.\exists y.\varphi(x,y)$ is a theorem, then there exists a program f such that $\forall x.\varphi(x,f(x))$.

[2] Functions definable in System F are exactly those provably total in second-order arithmetic.

(1989*b*) did with the realizability theory behind the program extraction in the Coq proof assistant (The Coq development team, 2010).

Contributions. Viewed as syntactical notions, realizability and parametricity bear a lot of similarities. Our aim was to understand through the generality of PTSs how they are related. Our main contributions are:

- The general construction of a logic from the programming language of its realizers with syntactic definitions of parametricity and realizability (Section 3).
- The proof that this construction is strongly normalizing if the starting programming language is (Theorem 16).
- A characterization of both realizability in terms of parametricity (Theorem 27) and parametricity in terms of realizability (Theorem 22).

2 The first level

In this section, we recall basic definitions and theorems about pure types systems (PTSs). We refer the reader to (Barendregt, 1992) for a comprehensive introduction to PTSs.[3] A PTS is defined by a specification $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ where $\mathcal{S}$ is a set of *sorts*, $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ a set of *axioms* and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ a set of *rules*, which determines the typing of product types. This specification parameterizes both the syntax of term and the rules of the type system.

Definition 1 (Syntax of terms)**.** A PTS is a type system over a λ-calculus with the following syntax:

$$\begin{array}{rcll} \mathcal{T} & = & \mathcal{S} & \text{sort} \\ & | & \mathcal{V} & \text{variable} \\ & | & \mathcal{T}\,\mathcal{T} & \text{application} \\ & | & \lambda \mathcal{V} : \mathcal{T}.\ \mathcal{T} & \text{abstraction} \\ & | & (\mathcal{V} : \mathcal{T}) \to \mathcal{T} & \text{product} \end{array}$$

The product $(x : A) \to B$ may be also written $\forall (x : A).B$, or $A \to B$ when x does not occur free in B.

The rules of the typing judgement (written $\Gamma \vdash A : B$) of the PTS $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ are given in Figure 2.2. The notation $\Gamma \vdash A : B : C$ is a shorthand for having both $\Gamma \vdash A : B$ and $\Gamma \vdash B : C$ simultaneously.

[3] Readers familiar with Paper I of the thesis might wish to skip this section, as it largely overlaps with the corresponding section of Paper I. This present section adds the syntactic notion of sort-annotation to the system, but colours (as introduced in Paper I) could also be used for the same purpose.

$$\frac{}{\vdash s_1 : s_2}\ s_1 : s_2 \in \mathcal{A}\quad \textsc{Axiom} \qquad \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}\quad \textsc{Start} \qquad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}\quad \textsc{Weakening}$$

$$\frac{\Gamma \vdash F : ((x : A) \to B) \qquad \Gamma \vdash a : A}{\Gamma \vdash F\,a : B[x \mapsto a]}\quad \textsc{Application}$$

$$\frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash ((x : A) \to B) : s}{\Gamma \vdash (\lambda x : A.\ b) : ((x : A) \to B)}\quad \textsc{Abstraction}$$

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash ((x : A) \to B) : s_3}\quad \textsc{Product}\ (s_1, s_2, s_3) \in \mathcal{R} \qquad \frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_\beta B'}{\Gamma \vdash A : B'}\quad \textsc{Conversion}$$

Figure 2.1: Typing rules of the PTS $(\mathcal{S}, \mathcal{A}, \mathcal{R})$

$$\frac{}{\vdash s_1 : s_2}\ s_1 : s_2 \in \mathcal{A}\quad \textsc{Axiom} \qquad \frac{\Gamma \vdash A : s}{\Gamma, x^s : A \vdash x : A}\quad \textsc{Start} \qquad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x^s : C \vdash A : B}\quad \textsc{Weakening}$$

$$\frac{\Gamma \vdash F : ((x^s : A) \to B) \qquad \Gamma \vdash a : A}{\Gamma \vdash (F\,a)_s : B[x \mapsto a]}\quad \textsc{Application}$$

$$\frac{\Gamma, x^{s_1} : A \vdash b : B \qquad \Gamma \vdash ((x^{s_1} : A) \to B) : s}{\Gamma \vdash (\lambda x^{s_1} : A.\ b) : ((x^{s_1} : A) \to B)}\quad \textsc{Abstraction}$$

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x^{s_1} : A \vdash B : s_2}{\Gamma \vdash ((x^{s_1} : A) \to B) : s_3}\quad \textsc{Product}\ (s_1, s_2, s_3) \in \mathcal{R} \qquad \frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_\beta B'}{\Gamma \vdash A : B'}\quad \textsc{Conversion}$$

Figure 2.2: Typing rules PTS $(\mathcal{S}, \mathcal{A}, \mathcal{R})$, with sort annotations.

Example 1 (System F)**.** The PTS F has the following specification:

$$\mathcal{S}_{\mathrm{F}} = \{\star, \Box\} \qquad \mathcal{A}_{\mathrm{F}} = \{(\star, \Box)\} \qquad \mathcal{R}_{\mathrm{F}} = \{(\star,\star,\star), (\Box,\star,\star)\} \ .$$

It defines the λ-calculus with polymorphic types known as system F (Girard, 1972). The rule $(\star,\star,\star)$ corresponds to the formation of arrow types (usually written $\sigma \to \tau$) and the rule $(\Box,\star,\star)$ corresponds to quantification over types ($\forall\alpha.\tau$).

Even though we use F as a running example throughout the article to illustrate our general definitions our results apply to any PTS.

Sort annotations. We sometimes decorate terms with *sort annotations*. They function as a syntactic reminder of the first component of the rule used to type a product. We divide the set of variables into disjoint infinite subsets $\mathcal{V} = \bigsqcup\{\mathcal{V}_s | s \in \mathcal{S}\}$ and we write x^s to indicate that a variable x belongs to $\mathcal{V}_s$. We also annotate applications $F\,a$ with the sort of the variable of the product type of F. Using this notation, the product rule and the application rule are written

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x^{s_1} : A \vdash B : s_2}{\Gamma \vdash ((x^{s_1} : A) \to B) : s_3}\ \textsc{Product}\ (s_1,s_2,s_3) \in \mathcal{R} \qquad \frac{\Gamma \vdash F : ((x^s : A) \to B) \qquad \Gamma \vdash a : A}{\Gamma \vdash (F\,a)_s : B[x \mapsto a]}\ \textsc{Application} \ .$$

Since sort annotations can always be recovered by using the type derivation, we do not write them in our examples.

Example 2 (System F terms)**.** In System F, we adopt the following convention: the letters x, y, z, ... range over $\mathcal{V}_\star$, and α, β, γ, ... over $\mathcal{V}_\Box$. For instance, the identity program $\mathrm{Id} \equiv \lambda(\alpha : \star)(x : \alpha).x$ is of type $\mathrm{Unit} \equiv \forall\alpha : \star.\alpha \to \alpha$. The Church numeral $0 \equiv \lambda(\alpha : \star)(f : \alpha \to \alpha)(x : \alpha).x$ has type $\mathbb{N} \equiv \forall\alpha : \star.(\alpha \to \alpha) \to (\alpha \to \alpha)$ and the successor function on Church numerals $\mathrm{Succ} \equiv \lambda(n : \mathbb{N})(\alpha : \star)(f : \alpha \to \alpha)(x : \alpha).f\,(n\,\alpha\,f\,x)$ is a program of type $\mathbb{N} \to \mathbb{N}$.

In any PTS a term A is said to be *strongly normalizing* if there is no infinite β-reducing sequence starting from A. And A is *weakly normalizing* if there is a term A' in normal form (*i.e.* such that there is no B such that $A' \longrightarrow_\beta B$) with $A \longrightarrow_\beta^* A'$. A PTS is *strongly normalizing* (resp. *weakly normalizing*) if all its valid terms are strongly normalizing (resp. weakly normalizing).

Normalization properties are useful for solving the following problems:

- *Type checking problem:* Given a context Γ and two terms A and B, is $\Gamma \vdash A : B$ derivable ?

- *Type synthesis problem:* Given a context Γ and a term A, is there a term B such that $\Gamma \vdash A : B$?

Remark 2. *In (weakly or strongly) normalizing PTSs, the type checking problem and the type synthesis problem are decidable.*

The uniqueness of types is a very convenient property enjoyed by most interesting pure type systems.

Definition 3 (Singly sorted)**.** A PTS $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is *singly sorted* if

1. $(s_1, s_2), (s_1, s_2') \in \mathcal{A} \Rightarrow s_2 = s_2'$,
2. $(s_1, s_2, s_3), (s_1, s_2, s_3') \in \mathcal{R} \Rightarrow s_3 = s_3'$.

Lemma 4 (Uniqueness of types for singly sorted PTSs)**.** *In a singly sorted PTS we have,*

$$\Gamma \vdash A : B_1 \textit{ and } \Gamma \vdash A : B_2 \textit{ implies } B_1 =_\beta B_2.$$

Proof. See (Barendregt, 1992). □

3 The second level

In this section we describe a logic to reason about the programs and types written in an arbitrary PTS P, as well as basic results concerning the consistency of the logic. This logic is also a PTS, which we name P^2. Because we carry out most of our development in P^2, judgments refer to that system unless the symbol $\vdash$ is subscripted with the name of a specific system.

Definition 5 (second-level system)**.** Given a PTS $P = (\mathcal{S}, \mathcal{A}, \mathcal{R})$, we define $P^2 = (\mathcal{S}^2, \mathcal{A}^2, \mathcal{R}^2)$ by

$$\begin{array}{rcl} \mathcal{S}^2 & = & \mathcal{S} \cup \{\lceil s \rceil \mid s \in \mathcal{S}\} \\ \mathcal{A}^2 & = & \mathcal{A} \cup \{(\lceil s_1 \rceil, \lceil s_2 \rceil) \mid (s_1, s_2) \in \mathcal{A}\} \\ \mathcal{R}^2 & = & \mathcal{R} \cup \{(\lceil s_1 \rceil, \lceil s_2 \rceil, \lceil s_3 \rceil), (s_1, \lceil s_3 \rceil, \lceil s_3 \rceil) \mid (s_1, s_2, s_3) \in \mathcal{R}\} \\ & & \quad \cup \{(s_1, \lceil s_2 \rceil, \lceil s_2 \rceil) \mid (s_1, s_2) \in \mathcal{A}\} \end{array}$$

Because we see P as a programming language and P^2 as a logic for reasoning about programs in P, we adopt the following terminology and conventions. We use the metasyntactic variables $s, s_1, s_2, \ldots$ to range over sorts in $\mathcal{S}$ and $t, t_1, t_2, \ldots$ to range over sorts in $\mathcal{S}^2$. We call *type* a term inhabiting a first-level sort in some context (we write $\Gamma \vdash A : s$ for a type A), *programs* are inhabitants of types ($\Gamma \vdash A : B : s$ for a program A of type B), *formulas* denote inhabitants of a lifted sort (written $\Gamma \vdash A : \lceil s \rceil$) and *proofs* are inhabitants of formulas ($\Gamma \vdash A : B : \lceil s \rceil$). We also say that types

and programs are *first-level* terms, and formulas and proofs are *second-level* terms.

If s is a sort of P, then $\lceil s \rceil$ is the sort of formulas expressing properties of types of sort s. For each rule (s_1, s_2, s_3) in $\mathcal{R}$, $(\lceil s_1 \rceil, \lceil s_2 \rceil, \lceil s_3 \rceil)$ lifts constructs of the programming language at the level of the logic, and $(s_1, \lceil s_3 \rceil, \lceil s_3 \rceil)$ allows to form the quantification of programs of sort s_1 in formulas of sort $\lceil s_3 \rceil$.

For each axiom (s_1, s_2) in $\mathcal{A}$, we add the rule $(s_1, \lceil s_2 \rceil, \lceil s_2 \rceil)$ in order to form the type of predicates of sort $\lceil s_2 \rceil$ parameterized by programs of sort s_1.

Example 3. The PTS F^2 has the following specification:

$$\begin{array}{lcll} \mathcal{S}^2_{\mathrm{F}} & = & \{ & \star, \Box, \lceil\star\rceil, \lceil\Box\rceil \quad \} \\ \mathcal{A}^2_{\mathrm{F}} & = & \{ & (\star, \Box), (\lceil\star\rceil, \lceil\Box\rceil) \quad \} \\ \mathcal{R}^2_{\mathrm{F}} & = & \{ & (\star,\star,\star), (\Box,\star,\star), (\lceil\star\rceil, \lceil\star\rceil, \lceil\star\rceil), (\lceil\Box\rceil, \lceil\star\rceil, \lceil\star\rceil) \\ & & & (\star, \lceil\Box\rceil, \lceil\Box\rceil), (\star, \lceil\star\rceil, \lceil\star\rceil), (\Box, \lceil\star\rceil, \lceil\star\rceil) \quad \}. \end{array}$$

We extend our variable-naming convention to $\mathcal{V}_{\lceil\star\rceil}$ and $\mathcal{V}_{\lceil\Box\rceil}$ as follows: the variables h, h_1, h_2, ... range over $\mathcal{V}_{\lceil\star\rceil}$, and the variables X, Y, Z, ... range over $\mathcal{V}_{\lceil\Box\rceil}$. The logic F^2 is a second-order logic with typed individuals (Wadler (2007) gives another presentation of the same system). The sort $\star$ is the type of types and the only inhabitant of $\Box$, while $\lceil\star\rceil$ is the sort of propositions. The sort $\lceil\Box\rceil$ is inhabited by the type of propositions ($\lceil\star\rceil$), the type of predicates ($\tau \to \lceil\star\rceil$), and in general the type of relations ($\tau_1 \to \cdots \to \tau_n \to \lceil\star\rceil$). The product formation rules can be understood as follows:

- $(\lceil\star\rceil, \lceil\star\rceil, \lceil\star\rceil)$ allows to build implication between formulas, written $P \to Q$.[4]
- $(\star, \lceil\star\rceil, \lceil\star\rceil)$ allows to quantify over individuals (as in $\forall x : \tau.P$).
- $(\Box, \lceil\star\rceil, \lceil\star\rceil)$ allows to quantify over types (as in $\forall \alpha : \star.P$).
- $(\star, \lceil\Box\rceil, \lceil\Box\rceil)$ is used to build types of predicates depending on programs (as in $\forall(x : \sigma).\tau \to \lceil\star\rceil$).
- $(\lceil\Box\rceil, \lceil\star\rceil, \lceil\star\rceil)$ allows to quantify over predicates (as in $\forall(X : \tau \to \lceil\star\rceil).P$).

In F^2, truth can be encoded by $\top \equiv \forall X : \lceil\star\rceil.X \to X$ and is proved by Obvious $\equiv \lambda(X : \lceil\star\rceil)(h : X).h$. The formula $x =_\tau y \equiv \forall X : \tau \to \lceil\star\rceil.X\,x \to X\,y$ defines the Leibniz equality at type τ. The term

$$\mathrm{Refl} \equiv \lambda(\alpha : \star)(x : \alpha)(X : \alpha \to \lceil\star\rceil)(h : X\,x).h$$

[4] In this example P and Q stand for formulas (terms of type $\lceil\star\rceil$).

is a proof of the reflexivity of equality $\forall(\alpha : \star)(x : \alpha).x =_\alpha x$. And the induction principle over Church numerals is a formula

$$N \equiv \lambda x : \mathbb{N}.\forall X{:}\mathbb{N} \to \lceil\star\rceil.(\forall y : \mathbb{N}.X\,y \to X\,(\text{Succ}\ y)) \to X\,0 \to X\,x.$$

3.1 Structure of P^2

Remark 6. *P^2 contains two copies of P, one syntactically equal to P and one where sorts have been renamed from s to $\lceil s\rceil$. The only rules where the two copies interact are of the form $(s, \lceil s'\rceil, \lceil s'\rceil)$. We call these rules* interaction rules.

Programs (or types) can never refer to proofs (nor formulas). In other words, a first-level term never contains a second-level term: it is typable in P. Formally:

Theorem 7 (separation)**.** *For $s \in \mathcal{S}$, if $\Gamma \vdash A : B : s$ (resp. $\Gamma \vdash B : s$), then there exists a sub-context Γ' of Γ such that $\Gamma' \vdash_P A : B : s$ (resp. $\Gamma' \vdash_P B : s$).*

Proof. By induction on the structure of terms, and relying on the generation lemma (Barendregt, 1992, p. 5.2.13) and on the form of the rules in $\mathcal{R}^2$: assuming $(t_1, t_2, t_3) \in \mathcal{R}^2$ then $t_3 \in \mathcal{S} \Rightarrow (t_1 \in \mathcal{S} \wedge t_2 \in \mathcal{S})$ and $t_2 \in \mathcal{S} \Rightarrow (t_1 \in \mathcal{S} \wedge t_3 \in \mathcal{S})$. □

Remark 8. *If P is singly sorted, then so is P^2.*

Therefore if P is singly sorted, then type checking and type synthesis are decidable in P^2. We will see in Section 3.2 that the strong normalization of P is also preserved by our construction.

Lifting. The major part of the paper is about transformations and relations between the first and the second level. The first and simplest transformation lifts terms from the first level to the second level, by substituting occurrences of a sort s by $\lceil s\rceil$ everywhere (see Figure 2.3). The function is defined only on first-level terms, and is extended to contexts in the obvious way. In addition to substituting sorts, lifting performs renaming of a variable x in $\mathcal{V}_s$ to $\mathring{x}$ in $\mathcal{V}_{\lceil s\rceil}$.

Example 4. In F^2, the lifting of inhabited types gives rise to logical tautologies. For instance, $\lceil\text{Unit}\rceil = \lceil\forall\alpha : \star.\alpha \to \alpha\rceil = \forall X : \lceil\star\rceil.X \to X = \top$, and $\lceil\mathbb{N}\rceil = \forall X : \lceil\star\rceil.(X \to X) \to (X \to X)$.

Lifting preserves both typing and β-reduction.

Lemma 9 (lifting preserves typing)**.** $\Gamma \vdash A : B : s \Rightarrow \lceil\Gamma\rceil \vdash \lceil A\rceil : \lceil B\rceil : \lceil s\rceil$

Proof. A consequence of P^2 containing a copy of P with every sort s renamed to $\lceil s\rceil$. □

$$
\begin{array}{ll}
\lceil x \rceil & = \mathring{x} \\
\lceil s \rceil & = \lceil s \rceil \\
\lceil (x : A) \to B \rceil & = (\mathring{x} : \lceil A \rceil) \to \lceil B \rceil \\
\lceil \lambda x : A.\ b \rceil & = \lambda \mathring{x} : \lceil A \rceil .\ \lceil b \rceil \\
\lceil A\ B \rceil & = \lceil A \rceil\ \lceil B \rceil \\
\\
\lceil - \rceil & = - \\
\lceil \Gamma, x : A \rceil & = \lceil \Gamma \rceil, \mathring{x} : \lceil A \rceil
\end{array}
\qquad
\begin{array}{ll}
\lfloor x^{\lceil s \rceil} \rfloor & = \dot{x}^s \\
\lfloor \lceil s \rceil \rfloor & = s \\
\lfloor \forall x^s : A.B \rfloor & = \lfloor B \rfloor \\
\lfloor \forall x^{\lceil s \rceil} : A.B \rfloor & = \forall \dot{x}^s : \lfloor A \rfloor . \lfloor B \rfloor \\
\lfloor \lambda x^s : A.B \rfloor & = \lfloor B \rfloor \\
\lfloor \lambda x^{\lceil s \rceil} : A.B \rfloor & = \lambda \dot{x}^s : \lfloor A \rfloor . \lfloor B \rfloor \\
\lfloor (A\ B)_s \rfloor & = \lfloor A \rfloor \\
\lfloor (A\ B)_{\lceil s \rceil} \rfloor & = \lfloor A \rfloor\ \lfloor B \rfloor \\
\\
\lfloor - \rfloor & = - \\
\lfloor \Gamma, x^s : A \rfloor & = \lfloor \Gamma \rfloor \\
\lfloor \Gamma, x^{\lceil s \rceil} : A \rfloor & = \lfloor \Gamma \rfloor, \dot{x}^s : \lfloor A \rfloor
\end{array}
$$

Figure 2.3: lifting from P to P^2 and projection form P^2 to P.

Lemma 10 (lifting preserves β-reduction)**.** $A \longrightarrow_\beta B \Rightarrow \lceil A \rceil \longrightarrow_\beta \lceil B \rceil$

Proof. By induction on the structure of A. □

Projection. We define a projection from second-level terms into first-level terms, which maps second-level constructs into first-level constructs. The first-level subterms are removed, as well as the interactions between the first and second levels. The reader may worry that some variable bindings are removed, potentially leaving some occurrences unbound in the body of the transformed term. However, these variables are first level, and hence their occurrences are removed too (by the application case).

The function is defined only on second-level terms, and behaves differently when facing pure second level or interaction terms. In order to distinguish these cases, the projection takes sort-annotated terms as input. Like the lifting, the projection performs renaming of each variable x in $\mathcal{V}_{\lceil s \rceil}$ to $\dot{x}$ in $\mathcal{V}_s$. We postulate that this renaming cancels that of the lifting: we have $\mathring{\dot{x}} = x$.

Example 5 (projections in F^2)**.**

$$\lfloor \top \rfloor = \text{Unit} \qquad \lfloor \text{Obvious} \rfloor = \text{Id} \qquad \lfloor \forall (\alpha : \star)(x : \alpha).x =_\alpha x \rfloor = \text{Unit}$$

$$\lfloor N\ t \rfloor = \mathbb{N}$$

Lemma 11 (projection is the left inverse of lifting)**.** $\lfloor \lceil A \rceil \rfloor = A$

Proof. By induction on the structure of A. □

As lifting, projection preserves typing.

Lemma 12 (projection preserves typing)**.** $\Gamma \vdash A : B : \lceil s \rceil \Rightarrow \lfloor \Gamma \rfloor \vdash \lfloor A \rfloor : \lfloor B \rfloor : s$

Proof. By induction on the derivation $\Gamma \vdash A : B$. □

In contrast to lifting, which keeps a term intact, projection may remove parts of a term, in particular abstractions at the interaction level. Therefore, β-reduction steps may be removed by projection.

Lemma 13 (projection preserves or removes β-reduction)**.**
If $A \longrightarrow_\beta B$, then either $\lfloor A \rfloor \longrightarrow_\beta \lfloor B \rfloor$ or $\lfloor A \rfloor = \lfloor B \rfloor$.

3.2 Strong normalization

Armed with the basic tools of projection and lifting, we can already prove that P^2 is as consistent as P.

Definition 14 (inconsistent sort)**.** In any PTS Q, we say that a sort s is *inconsistent* if for all B such that $\vdash_Q B : s$, there exists A such that $\vdash_Q A : B$ (in other words, all inhabitants of the sort are inhabited).

Theorem 15. *If a sort $\lceil s \rceil$ is inconsistent, then s is inconsistent.*

Proof. Assume B be such that $\vdash B : s$. By Lemma 9 we have $\vdash \lceil B \rceil : \lceil s \rceil$. Because $\lceil s \rceil$ is inconsistent, we can find A such that $\vdash A : \lceil B \rceil : \lceil s \rceil$ and, by Lemma 12, $\vdash \lfloor A \rfloor : \lfloor \lceil B \rceil \rfloor : s$. We finally apply Lemma 11 ($\lfloor \lceil B \rceil \rfloor = B$) and obtain $\vdash \lfloor A \rfloor : B : s$. □

Example 6. In F^2, the sort $\star$ is consistent since the type $\forall \alpha : \star . \alpha$ is not inhabited. The previous lemma gives us a proof that $\lceil \star \rceil$ is also consistent since a proof of $\bot = \forall X : \lceil \star \rceil . X = \lceil \forall \alpha : \star . \alpha \rceil$ could be projected to a proof of $\forall \alpha : \star . \alpha$.

Theorem 16 (normalization)**.** *If P is strongly normalizing, so is P^2.*

Proof. The proof is based on the observation that, if a term A is typable in P^2 and not normalizable, then at least either:

- one of the first-level subterms of A is not normalizable, or
- the first-level term $\lfloor A \rfloor$ is not normalizable.

And yet $\lfloor A \rfloor$ and the first-level subterms are typable in P (Theorem 7) which would contradict the strong normalization of P.

□

3.3 Parametricity

In this section we develop Reynolds-style (Reynolds, 1983) parametricity for P, in P^2. While parametricity theory is often defined for binary relations, we abstract from the arity and develop the theory for an arbitrary arity n, though we omit the index n when the arity of relations plays no role or is obvious from the context.

The definition of parametricity is done in two parts: first we define what it means for an n-tuple of programs $\overline{z}$ to satisfy the relation generated by a type T ($\overline{z} \in [\![T]\!]_n$[5]); then we define the translation from a program z of type T to a proof $[\![z]\!]_n$ that a tuple $\overline{z}$ satisfies the relation.

The definition below uses $n+1$ renamings: one of them ($^\circ$) coincides with that of lifting, and the others map x respectively to $x_1, \ldots, x_n$. The tuple $\overline{A}$ denotes n terms A_i, where A_i is the term A where each free variable x is replaced by a fresh variable x_i.

Definition 17 (parametricity)**.**

$$
\begin{array}{lcl}
\overline{C} \in [\![s]\!] & = & \overline{C} \to \lceil s \rceil \\
\overline{C} \in [\![(x : A) \to B]\!] & = & (\overline{x : A}) \to (\mathring{x} : \overline{x} \in [\![A]\!]) \to \overline{C\,x} \in [\![B]\!] \\
\overline{C} \in [\![T]\!] & = & [\![T]\!]\,\overline{C} \textit{ otherwise} \\
\\
[\![x]\!] & = & \mathring{x} \\
[\![\lambda x : A.\ B]\!] & = & \lambda \overline{x : A}.\ \lambda \mathring{x} : \overline{x} \in [\![A]\!].\ [\![B]\!] \\
[\![A\ B]\!] & = & [\![A]\!]\,\overline{B}\,[\![B]\!] \\
[\![T]\!] & = & \lambda \overline{z : T}.\ \overline{z} \in [\![T]\!] \textit{ otherwise} \\
\\
[\![-]\!] & = & - \\
[\![\Gamma, x : A]\!] & = & [\![\Gamma]\!], \overline{x : A}, \mathring{x} : \overline{x} \in [\![A]\!]
\end{array}
$$

Because the syntax of values and types are unified in a PTS, each of the definitions $\cdot \in [\![\cdot]\!]$ and $[\![\cdot]\!]$ must handle all constructions. In both cases, this is done by using a catch-all case (the last line) that refers to the other part of the definition.[6]

Remark 18. *For arity* 0*, parametricity specializes to lifting* ($[\![A]\!]_0 = \lceil A \rceil$).

Example 7. For instance, in F^2, we have

$$
\begin{array}{l}
(f, g) \in [\![\forall(\alpha : \star).\alpha \to \forall(\beta : \star).\beta \to \alpha]\!] \equiv \\
\quad \forall(\alpha_1\,\alpha_2 : \star)(X : \alpha_1 \to \alpha_2 \to \lceil\star\rceil)(x_1 : \alpha_1)(x_2 : \alpha_2).X\,x_1\,x_2 \to \\
\quad \forall(\beta_1\,\beta_2 : \star)(Y : \beta_1 \to \beta_2 \to \lceil\star\rceil)(y_1 : \beta_1)(y_2 : \beta_2).Y\,y_1\,y_2 \to \\
\quad X\,(f\,\alpha_1\,\beta_1\,x_1\,y_1)\,(g\,\alpha_2\,\beta_2\,x_2\,y_2)
\end{array}
$$

[5] A note about syntax: the construction $\cdot \in [\![\cdot]\!]$ constructs types and therefore binds tighter that the colon.

[6] Readers familiar with Paper I might wonder why we need a longer definition of the relational interpretation here. The question is addressed in Section 4.1.

We can then state our version of the abstraction theorem:

Theorem 19 (abstraction). *If* $\Gamma \vdash A : B : s$, *then* $[\![\Gamma]\!] \vdash [\![A]\!] : (\overline{A} \in [\![B]\!]) : \lceil s \rceil$

Proof. The result is a consequence of the following lemmas.

- $A \longrightarrow_\beta B \Rightarrow [\![A]\!] \longrightarrow_\beta^* [\![B]\!]$
- $\Gamma \vdash A : B \Rightarrow [\![\Gamma]\!] \vdash \overline{A : B}$
- $\Gamma \vdash B : s \Rightarrow [\![\Gamma]\!], \overline{z : B} \vdash \bar{z} \in [\![B]\!] : \lceil s \rceil$
- $\Gamma \vdash A : B : s \Rightarrow [\![\Gamma]\!] \vdash [\![A]\!] : \overline{A} \in [\![B]\!]$

The proof of the last three lemmas is done by simultaneous induction on the length of the derivations. (Details in appendix.) □

A direct reading of the above result is as a typing judgement about translated terms (as for lemmas 9 and 12): if A has type B, then $[\![A]\!]$ has type $\overline{A} \in [\![B]\!]$. However, it can also be understood as an abstraction theorem for system P: if a program A has type B in Γ, then various interpretations of A ($\overline{A}$) in related environments ($[\![\Gamma]\!]$) are related, by the formula $\overline{A} \in [\![B]\!]$.

The system P^2 is a natural setting to express parametricity conditions for P. Indeed, the interaction rules of the form $(s_1, \lceil s_2 \rceil, \lceil s_2 \rceil)$ coming from axioms (s_1, s_2) in P are needed to make the sort case valid; and the interaction rules $(s_1, \lceil s_3 \rceil, \lceil s_3 \rceil)$ are needed for the quantification over individuals in coming from rules (s_1, s_2, s_3) in the product case.

3.4 Realizability

We develop here a Krivine-style (Krivine, 1997) internalized realizability theory. Realizability bears similarities both to the projection and the parametricity transformations defined above.

Like the projection, the realizability transformation is applied on second-level constructs, and behaves differently depending on whether it treats interaction constructs or pure second-level ones. It is also similar to parametricity, as it is defined in two parts. In the first part we define what it means for a program C to realize a formula F ($C \Vdash F$); then we define the translation from a proof p to a proof $\langle p \rangle$ that the program $\lfloor p \rfloor$ satisfies the realizability predicate.

Definition 20 (realizability)**.**

$$
\begin{array}{lcl}
C \Vdash \lceil s \rceil & = & C \to \lceil s \rceil \\
C \Vdash \forall x^s : A.B & = & \forall x^s : A.\ C \Vdash B \\
C \Vdash \forall x^{\lceil s \rceil} : A.B & = & \forall (\dot{x}^s : \lfloor A \rfloor)(x^{\lceil s \rceil} : \dot{x} \Vdash A).\ C\,\dot{x} \Vdash B \\
C \Vdash F & = & \langle F \rangle\ C \textit{ otherwise}
\end{array}
$$

$$
\begin{array}{lcl}
\langle x^{\lceil s \rceil} \rangle & = & x^{\lceil s \rceil} \\
\langle \lambda x^s : A.B \rangle & = & \lambda x^s : A.\langle B \rangle \\
\langle \lambda x^{\lceil s \rceil} : A.B \rangle & = & \lambda (\dot{x}^s : \lfloor A \rfloor)(x^{\lceil s \rceil} : \dot{x} \Vdash A).\langle B \rangle \\
\langle (A\ B)_s \rangle & = & (\langle A \rangle\ B)_s \\
\langle (A\ B)_{\lceil s \rceil} \rangle & = & ((\langle A \rangle\ \lfloor B \rfloor)_s\ \langle B \rangle)_{\lceil s \rceil} \\
\langle T \rangle & = & \lambda z^s : \lfloor T \rfloor.\ z \Vdash T \textit{ otherwise}
\end{array}
$$

$$
\begin{array}{lcl}
\langle - \rangle & = & - \\
\langle \Gamma, x^s : A \rangle & = & \langle \Gamma \rangle, x^s : A \\
\langle \Gamma, x^{\lceil s \rceil} : A \rangle & = & \langle \Gamma \rangle, \dot{x}^s : \lfloor A \rfloor, x^{\lceil s \rceil} : \dot{x} \Vdash A
\end{array}
$$

Theorem 21 (adequacy)**.** *If* $\Gamma \vdash A : B : \lceil s \rceil$*, then* $\langle \Gamma \rangle \vdash \langle A \rangle : \ \lfloor A \rfloor \Vdash B \ : \lceil s \rceil$

Proof idea. Similar in structure to the proof of the abstraction theorem. □

Example 8. In F^2, the formula $y \Vdash \mathbb{N}\,x$ unfolds to

$$
\begin{array}{c}
\forall (\alpha : \star)(X : \mathbb{N} \to \alpha \to \lceil \star \rceil)(f : \alpha \to \alpha). \\
(\forall (n : \mathbb{N})(y : \alpha).X\,n\,y \to X\,(\mathrm{Succ}\ n)\,(f\,y)) \to \forall (z : \alpha).X\,0\,y \to X\,x\,(y\,\alpha\,f\,z)
\end{array}
$$

In F^2 this formula may be used to prove a representation theorem. We can prove that $\Sigma \vdash \forall x\,y : \mathbb{N}.y \Vdash N\,x \Leftrightarrow x =_{\mathbb{N}} y \wedge N\,x$ where Σ is a set of extensionality axioms ($\wedge$ and $\Leftrightarrow$ are defined by usual second-order encodings). Let π be a proof of $\forall x : \mathbb{N}.N\,x \to N\,(f\,x)$ then $\vdash \lfloor \pi \rfloor : \mathbb{N} \to \mathbb{N}$ and $\vdash \langle \pi \rangle : \lfloor \pi \rfloor \Vdash \forall x : \mathbb{N}.N\,x \to N\,(f\,x)$ which unfold to $\vdash \langle \pi \rangle : \forall x\,y : \mathbb{N}.y \Vdash Nx \to \lfloor \pi \rfloor y \Vdash N(fx)$. Let m be a term in closed normal form such that $\vdash m : \mathbb{N}$, we can prove $N\,m$ and therefore $m \Vdash N\,m$. We now have a proof (under Σ) that $\lfloor \pi \rfloor\ m \Vdash N\,(f\,m)$ and we conclude that $\lfloor \pi \rfloor\ m =_{\mathbb{N}} f\,m$. We have proved that the projection of any proof of $\forall x : \mathbb{N}.N\,x \to N\,(f\,x)$ can be proved extensionally equal to f. See (Wadler, 2007; Krivine, 1997; Leivant, 1990) for more details.

4 The third level

By casting both parametricity and realizability in the mold of PTSs, we are able to discern the connections between them. The connections already surface in the previous sections: the definitions of parametricity and realizability bear some resemblance, and the adequacy and abstraction theo-

rems appear suspiciously similar. In this section we precisely spell out the connection: realizability and parametricity can be defined in terms of each other.

We first remark that realizability increases arity of parametricity.

Theorem 22 (realizability increases arity of parametricity)**.** *For any tuple terms* $(B, \overline{C})$*,*
$\big(B, \overline{C}\big) \in [\![A]\!]_{n+1} = B \Vdash \big(\overline{C} \in [\![A]\!]_n\big)$ *and* $[\![A]\!]_{n+1} = \langle [\![A]\!]_n \rangle$

Proof. By induction on the structure of A. □

As a corollary, n-ary parametricity is the composition of lifting and n realizability steps:

Corollary 23 (from realizability to parametricity)**.** $\overline{C} \in [\![A]\!]_n = C_1 \Vdash C_2 \Vdash \cdots \Vdash C_n \Vdash \lceil A \rceil$ *and* $[\![A]\!]_n = \langle \cdots \langle \lceil A \rceil \rangle \cdots \rangle$ *(assuming right-associativity of* $\Vdash$*)*

Proof. By induction on n. The base case uses $[\![A]\!]_0 = \lceil A \rceil$. □

One may also wonder about the converse: is it possible to define realizability in terms of parametricity? We can answer by the affirmative, but we need a bigger system to do so. Indeed, we need to extend $[\![\cdot]\!]$ to work on second-level terms, and that is possible only if a third level is present in the system. To do so, we can iterate the construction used in Section 3 to build a logic for an arbitrary PTS.

Definition 24 (third-level system)**.** Given a PTS $P = (\mathcal{S}, \mathcal{A}, \mathcal{R})$, we define $P^3 = (P^2)^2$, where the sort-lifting $\lceil \cdot \rceil$ used by both instances of the $\cdot^2$ transformation are the same.

Remark 25. *Because the sort-lifting used by both instances of the* $\cdot^2$ *transformation are the same,* P^3 *contains only three copies of* P *(not four). That is, for each* s*, we unify the sort* $\lceil s \rceil$ *obtained from the first application of* $\cdot^2$ *and* $\lceil s \rceil$ *obtained from the second application. In fact* $P^3 = (\mathcal{S}^3, \mathcal{A}^3, \mathcal{R}^3)$*, where*

$$\begin{array}{rcl}
\mathcal{S}^3 & = & \mathcal{S} \ \cup \lceil \mathcal{S} \rceil \cup \lceil\lceil \mathcal{S} \rceil\rceil \\
\mathcal{A}^3 & = & \mathcal{A} \ \cup \lceil \mathcal{A} \rceil \cup \lceil\lceil \mathcal{A} \rceil\rceil \\
\mathcal{R}^3 & = & \mathcal{R} \ \cup \lceil \mathcal{R} \rceil \cup \lceil\lceil \mathcal{R} \rceil\rceil \\
 & & \quad \cup \{(s_1, \lceil s_3 \rceil, \lceil s_3 \rceil), (\lceil s_1 \rceil, \lceil\lceil s_3 \rceil\rceil, \lceil\lceil s_3 \rceil\rceil) \mid (s_1, s_2, s_3) \in \mathcal{R}\} \\
 & & \quad \cup \{(s_1, \lceil s_2 \rceil, \lceil s_2 \rceil), (\lceil s_1 \rceil, \lceil\lceil s_2 \rceil\rceil, \lceil\lceil s_2 \rceil\rceil) \mid (s_1, s_2) \in \mathcal{A}\}
\end{array}$$

The $[\![\cdot]\!]$ transformation is extended to second-level constructs in P^2, mapping them to third-level ones in P^3. The $\lfloor \cdot \rfloor$ transformation is similarly extended, to map the third level constructs to the second level, in addition to mapping the second to the first one (only the first level is removed).

Given these extensions, we obtain that realizability is the composition of parametricity and projection.

Lemma 26. *If A is a first-level term, then*
$A = \lfloor C \in [\![A]\!]_1 \rfloor$ *and* $A = \lfloor [\![A]\!]_1 \rfloor$

Proof. By induction on the structure of A, using separation (Theorem 7). □

Theorem 27 (from parametricity to realizability)**.** *If A is a second-level term, then*
$C \Vdash A = \lfloor \lceil C \rceil \in [\![A]\!]_1 \rfloor$ *and* $\langle A \rangle = \lfloor [\![A]\!]_1 \rfloor$

Proof. By induction on the structure of A, using the above lemma. □

4.1 Infinite PTSs

In the previous sections, we describe parametricity (or realizability) in two interwoven parts; one that treats types (or formulas) and the other that treats programs (or proofs). This is the schema classically found in the literature. However, handling types and programs separately is somewhat disappointing in the context of PTSs, whose one of the main strengths is the unification between programs and types (or proofs and formulas).

Such a unification can apparently be done by simply unfolding the uses of $\cdot \in [\![\cdot]\!]$ in the definition of $[\![\cdot]\!]$ (or $\cdot \Vdash \cdot$ in the definition of $\langle\cdot\rangle$). We obtain the definition given in Figure 2.4. However, this definition introduces more abstractions in the terms generated by $[\![\cdot]\!]$ or $\langle\cdot\rangle$; and this means that more product rules are needed in the logic.

Furthermore, one would like to also unify terms and types in the abstraction and adequacy theorems. That is, use the shorter notation $[\![\Gamma]\!] \vdash [\![A]\!] : [\![B]\!]\,\overline{A}$ (or $\langle\Gamma\rangle \vdash \langle A\rangle : \langle B\rangle\, A$) instead of $[\![\Gamma]\!] \vdash [\![A]\!] : \overline{A} \in [\![B]\!] : \lceil s \rceil$ (or $\langle\Gamma\rangle \vdash \langle A\rangle : \lfloor A \rfloor \Vdash B : \lceil s \rceil$).

The issue is then that, B can now be a top-sort, and $A \in [\![s]\!]$ is $A \to \lceil s \rceil$, which is not typable in P^2.

Definition 28 (top-sort)**.** A sort s is called a *top sort* if there is no axiom $(s, s') \in \mathcal{R}$.

An obvious solution is to forbid top-sorts, as Bernardy, Jansson, and Paterson (2010).[7] In that case, not only can the theorems be simplified, but the definitions of parametricity and realizability as well (the extra abstractions become typable). Forbidding top-sorts seems like a drastic measure.

[7] In Paper I (which is an extended version of Bernardy, Jansson, and Paterson (2010)), we have refined the result: we do not need an infinite sort hierarchy, but only a few extra sorts and rules to make the terms generated by $[\![\cdot]\!]$ typeable.

Parametricity

$$
\begin{aligned}
[\![x]\!] &= \mathring{x} \\
[\![s]\!] &= \lambda\overline{x : s}.\ \overline{x} \to \lceil s \rceil \\
[\![(x : A) \to B]\!] &= \lambda\overline{f : ((x : A) \to B)}.\ (\overline{x : A}) \to (\mathring{x} : [\![A]\!]\,\overline{x}) \to [\![B]\!]\,(\overline{f\,x}) \\
[\![F\,a]\!] &= [\![F]\!]\,\overline{a}\,[\![a]\!] \\
[\![\lambda x : A.\ b]\!] &= \lambda\overline{x : A}.\ \lambda\mathring{x} : [\![A]\!]\,\overline{x}.\ [\![b]\!]
\end{aligned}
$$

Projection

$$
\begin{aligned}
\lfloor x^{\lceil s \rceil} \rfloor &= \dot{x}^s \\
\lfloor \lceil s \rceil \rfloor &= s \\
\lfloor \forall x^s : A.B \rfloor &= \lfloor B \rfloor \\
\lfloor \forall x^{\lceil s \rceil} : A.B \rfloor &= \forall \dot{x}^s : \lfloor A \rfloor . \lfloor B \rfloor \\
\lfloor \lambda x^s : A.B \rfloor &= \lfloor B \rfloor \\
\lfloor \lambda x^{\lceil s \rceil} : A.B \rfloor &= \lambda \dot{x}^s : \lfloor A \rfloor . \lfloor B \rfloor \\
\lfloor (A\,B)_s \rfloor &= \lfloor A \rfloor \\
\lfloor (A\,B)_{\lceil s \rceil} \rfloor &= \lfloor A \rfloor\ \lfloor B \rfloor
\end{aligned}
$$

Realizability $\leftrightarrow$ Parametricity

$$
\begin{aligned}
\lceil A \rceil &= [\![A]\!]_0 \\
\langle A \rangle &= \lfloor [\![A]\!] \rfloor \\
[\![A]\!]_{n+1} &= \langle [\![A]\!]_n \rangle
\end{aligned}
$$

Theorems

$$
\begin{aligned}
\Gamma \vdash A : B &\implies \lfloor \Gamma \rfloor \vdash \lfloor A \rfloor : \lfloor B \rfloor \\
\Gamma \vdash A : B &\implies \langle \Gamma \rangle \vdash \langle A \rangle : \langle B \rangle\ \lfloor A \rfloor \\
\Gamma \vdash A : B &\implies [\![\Gamma]\!] \vdash [\![A]\!] : [\![B]\!]\,\overline{A}
\end{aligned}
$$

Figure 2.4: Parametricity and realizability for infinite PTSs in a nutshell. (We could also have chosen realizability plus lifting instead of parametricity plus projection as basic constructs)

However, many systems have already been extended with infinite sort hierarchies.

sort hierarchies. For example, the Generalized Calculus of Constructions (Coquand, 1986; Miquel, 2001) extends CC in that way.

Definition 29 (CC_ω)**.** CC_ω is a PTS with this specification:

- $\mathcal{S} = \{\star\} \cup \{\Box_i \mid i \in \mathbb{N}\}$
- $\mathcal{A} = \{\star : \Box_0\} \cup \{\Box_i : \Box_{i+1} \mid i \in \mathbb{N}\}$
- $\mathcal{R} = \{\star \rightsquigarrow \star, \star \rightsquigarrow \Box_i, \Box_i \rightsquigarrow \star \mid i \in \mathbb{N}\} \cup \{(\Box_i, \Box_j, \Box_{\max(i,j)}) \mid i, j \in \mathbb{N}\}$

Many dependently-typed programming languages and proof assistants already support infinite sort hierarchies: Agda (Norell, 2007) and Coq (The Coq development team, 2010) are two well-known examples.

5 Extensions

5.1 Inductive definitions

Even though our development assumes pure type systems, with only axioms of the form (s_1, s_2), the theory easily accommodates the addition of inductive definitions.

For parametricity, the way to extend the theory is exposed by Bernardy, Jansson, and Paterson (2010). In brief: if for every inductive definition in the programming language there is a corresponding inductive definition in the logic, then the abstraction theorem holds. For instance, to the indexed inductive definition I corresponds $[\![I]\!]$, as defined below. (We write only one constructor c_p for concision, but the result applies to any number of constructors.)

$$\begin{array}{l}\textbf{data } I : \forall(x_1 : A_1) \cdots (x_n : A_n).s \textbf{ where} \\ \quad c_p : \forall(x_1 : B_{p,1}) \cdots (x_{n_1} : B_{p,n_1}).I\, a_{p,1} \cdots a_{p,n}\end{array}$$

$$\begin{array}{l}\textbf{data } [\![I]\!] : \overline{I} \in [\![\forall(x_1 : A_1) \cdots (x_n : A_n).s]\!] \textbf{ where} \\ \quad [\![c_p]\!] : \overline{c_p} \in [\![\forall(x_1 : B_{p,1}) \cdots (x_{n_1} : B_{p,n_1}).I\, a_{p,1} \cdots a_{p,n}]\!]\end{array}$$

The result can be transported to realizability by following the correspondence developed in the previous section. By taking the composition of $[\![\cdot]\!]$ and $\lfloor\cdot\rfloor$ for the definition of realizability, and knowing how to extend $[\![\cdot]\!]$ to inductive types, it suffices to extend $\lfloor\cdot\rfloor$ as well (respecting typing: Lemma 12). The corresponding extension to realizability is compatible

with the definition for a pure system (by Theorem 27). Adequacy is proved by the composition of abstraction and Lemma 12. The definition of $\lfloor\cdot\rfloor$ is straightforward: each component of the definition must be transformed by $\lfloor\cdot\rfloor$. That is, for any inductive definition in the logic, there must be another inductive definition in the programming language that realizes it. For instance, given the definition I given below, one must also have $\lfloor I\rfloor$. $\langle I\rangle$ is then given by $\langle I\rangle = \lfloor[\![I]\!]\rfloor$, but can also be expanded as below.

$$\begin{array}{r}\textbf{data}\ I : \forall(x_1 : A_1)\cdots(x_n : A_n).\lceil s\rceil\ \textbf{where}\\ c_p : \forall(x_1 : B_{p,1})\cdots(x_{n_1} : B_{p,n_1}).I\, a_{p,1}\cdots a_{p,n}\end{array}$$

$$\begin{array}{r}\textbf{data}\ \lfloor I\rfloor : \lfloor\forall(x_1 : A_1)\cdots(x_n : A_n).\lceil s\rceil\rfloor\ \textbf{where}\\ \lfloor c_p\rfloor : \lfloor\forall(x_1 : B_{p,1})\cdots(x_{n_1} : B_{p,n_1}).I\, a_{p,1}\cdots a_{p,n}\rfloor\end{array}$$

$$\begin{array}{r}\textbf{data}\ \langle I\rangle : \lfloor I\rfloor \Vdash (\forall(x_1 : A_1)\cdots(x_n : A_n).\lceil s\rceil)\ \textbf{where}\\ \langle c_p\rangle : \lfloor c_p\rfloor \Vdash \left(\forall(x_1 : B_{p,1})\cdots(x_{n_1} : B_{p,n_1}).I\, a_{p,1}\cdots a_{p,n}\right)\end{array}$$

We can use inductive types to encode usual logical connectives, and derive realizability for them.

Example 9 (conjunction)**.** The encoding of conjunction in a sort $\lceil s\rceil$ is as follows:

$$\begin{array}{r}\textbf{data}\ \_\wedge\_ : \lceil s\rceil \to \lceil s\rceil \to \lceil s\rceil\ \textbf{where}\\ \mathsf{conj} : \forall\, P\, Q : \lceil s\rceil.P \to Q \to P \wedge Q\end{array}$$

If we apply the projection operator to the conjunction we obtain the type of its realizers: the cartesian product in s.

$$\begin{array}{r}\textbf{data}\ \_\times\_ : s \to s \to s\ \textbf{where}\\ (\_,\_) : \forall\,\alpha\,\beta : s.\alpha \to \beta \to \alpha\times\beta\end{array}$$

Now we can apply our realizability construction to obtain a predicate telling what it means to realize a conjunction.

$$\begin{array}{r}\textbf{data}\ \langle\wedge\rangle : \forall(\alpha : s).(\alpha \to \lceil s\rceil) \to\\ \forall(\beta : s).(\beta \to \lceil s\rceil) \to\\ \alpha\times\beta \to s\ \textbf{where}\\ \langle\mathsf{conj}\rangle : \forall(\alpha : s)(P : \alpha \to \lceil s\rceil)\\ (\beta : s)(Q : \beta \to \lceil s\rceil)(x : \alpha)(y : \beta).\\ P\,x \to Q\,y \to \langle\wedge\rangle\,\alpha\,P\,\beta\,Q\,(x,y)\end{array}$$

By definition, $t \Vdash P \wedge Q$ means $\langle\wedge\rangle\ \lfloor P\rfloor\ \langle P\rangle\ \lfloor Q\rfloor\ \langle Q\rangle\ t$. We have

$$t \Vdash P \wedge Q \Leftrightarrow (\pi_1\, t) \Vdash P \wedge (\pi_2\, t) \Vdash Q$$

where π_1 and π_2 are projections upon Cartesian product.

We could build the realizers of other logical constructs in the same way:

we would obtain a sum-type for the disjunction, an empty type for falsity, and a box type for the existential quantifier. All the following properties (corresponding to the usual definition of the realizability predicate) would then be satisfied:

- $t \Vdash P \vee Q \Leftrightarrow \mathbf{case}\, t\, \mathbf{with}\, \iota_1\, x \to x \Vdash P \mid \iota_2\, x \to x \Vdash Q$.
- $t \Vdash \bot \Leftrightarrow \bot$ and $t \Vdash \neg P \Leftrightarrow \forall(x : \lfloor P \rfloor).\neg(x \Vdash P)$
- $t \Vdash \exists x : A.P \Leftrightarrow \exists x : A.(unbox\, t) \Vdash P$

where **case** ... **with** ... is the destruction of the sum type, and *unbox* is the destructor of the box type.

5.2 Program extraction and computational irrelevance

An application of the theory developed so far is the extraction of programs from proofs. Indeed, an implication of the adequacy theorem is that the program $\lfloor A \rfloor$, obtained by projection of a proof A of a formula B, corresponds to an implementation of B, viewed as a specification. One says that $\lfloor \cdot \rfloor$ implements program extraction.

For example, applying extraction to an expression involving vectors (as defined in the previous section) yields a program over lists. This means that programs can be justified in the rich system P^2, and realized in the simple system P. Practical benefits include a reduction in memory usage: Brady, McBride, and McKinna (2004) measure an 80% reduction using a technique with similar goals (but using a different technique).

While P^2 is already much more expressive than P, it is possible to further increase the expressive power of the system, while retaining the adequacy theorem, by allowing quantification of first-level terms by second-level terms.

Definition 30 ($P^{2'}$)**.** Let $P = (\mathcal{S}, \mathcal{A}, \mathcal{R})$, we define $P^{2'} = (\mathcal{S}^{2'}, \mathcal{A}^{2'}, \mathcal{R}^{2'})$

$$\begin{aligned}
\mathcal{S}^{2'} &= \mathcal{S} \cup \{\lceil s \rceil \mid s \in \mathcal{S}\} \\
\mathcal{A}^{2'} &= \mathcal{A} \cup \{(\lceil s_1 \rceil, \lceil s_2 \rceil) \mid (s_1, s_2) \in \mathcal{A}\} \\
\mathcal{R}^{2'} &= \mathcal{R} \cup \{(\lceil s_1 \rceil, \lceil s_2 \rceil, \lceil s_3 \rceil), (s_1, \lceil s_3 \rceil, \lceil s_3 \rceil), (\lceil s_1 \rceil, s_3, s_3) \mid (s_1, s_2, s_3) \in \mathcal{R}\} \\
&\quad \cup \{(s_1, \lceil s_2 \rceil, \lceil s_2 \rceil), (\lceil s_1 \rceil, s_2, s_2) \mid (s_1, s_2) \in \mathcal{A}\}
\end{aligned}$$

The result is a symmetric system, with two copies of P. Within either side of the system, one can reason about terms belonging to the other side. Furthermore, either side has a computational interpretation where the terms of the other side are irrelevant. For the second level, this interpretation is given by $\lfloor \cdot \rfloor$.

Even though there is no separation between first and second level in $P^{2'}$, adequacy is preserved: the addition of rules of the form $(\lceil s_1 \rceil, s_2, s_3)$ only adds first level terms, which are removed by projection.

6 Related work and conclusion

This work builds upon realizability in the style of Krivine (1997) and parametricity in the style of Reynolds (1983), which have both spawned large bodies of work.

Logics for parametricity. Study of parametricity is typically semantic, including the seminal work of Reynolds (1983). There, the concern is to capture the polymorphic character of λ-calculi (typically System F) in a model.

Mairson (1991) pioneered a different angle of study, where the expressions of the programming language are (syntactically) translated to formulas describing the program. That style has then been picked by various authors before us, including Abadi, Cardelli, and Curien (1993); Plotkin and Abadi (1993); Bernardy, Jansson, and Paterson (2010).

Plotkin and Abadi (1993) introduce a logic for parametricity, similar to F^2, but with several additions. The most important addition is that of a parametricity axiom. This addition allows to prove the initiality of Church-style encoding of types.

Wadler (2007) defines essentially the same concepts as us, but in the special case of System F. He points out that realizability transforms unary parametricity into binary parametricity, but does not generalize to arbitrary arity. We find the $n = 0$ case particularly interesting, as it shows that parametricity can be constructed purely in terms of realizability and a trivial lifting to the second level. We additionally show that realizability can be obtained by composing parametricity and projection, while Wadler only defines the realizability transformation as a separate construct. Our projection $\lfloor \cdot \rfloor$ corresponds to what Wadler calls Girard's projection.

The parametricity transformation and the abstraction theorem that we expose here are a modified version of (Bernardy, Jansson, and Paterson, 2010). The added benefits of the present version is that we handle finite PTSs, and we allow the target system to be different from the source. The possible separation of source and targets is already implicit in that paper though. The way we handle finite PTSs is by separating the treatment of types and programs.

Realizability. Our realizability construction can be understood as an extension of the work of Paulin-Mohring (1989*a*), providing a realizability in-

terpretation for a variant of the Calculus of Construction. Paulin-Mohring (1989*a*) splits CC in two levels; one where $\star$ becomes Prop and one where it becomes Spec. Perhaps counter-intuitively, Prop lies in what we call the first level; and Spec lies in the second level. Indeed, Prop is removed from the realizers. The system is symmetric, as the one we expose in Section 5.2, in the sense that there is both a rule $(\mathsf{Spec}, \mathsf{Prop}, \mathsf{Prop})$ and $(\mathsf{Prop}, \mathsf{Spec}, \mathsf{Spec})$. In order to see that Paulin-Mohring's construction as a special case of ours, it is necessary to recognize a number of small differences:

1. The sort Spec is transformed into Prop in the realizability transformation, whereas we would keep Spec.

2. The sorts of the original system use a different set of names (*Data* and *Order*). Therefore the sort Spec is transformed into *Data* in the projection, whereas we would use Prop.

3. The types of Spec and Prop inhabit the same sort, namely *Type*.

4. There is elimination from Spec to Prop, breaking the computational irrelevance in that direction.

The first two differences are essentially renamings, and thus unimportant.

Connections. We are unaware of previous work showing the connection between realizability and parametricity, at least as clearly as we do. Wadler (2007) comes close, giving a version of Theorem 22 specialized to System F, but not its converse, Theorem 27. Mairson (1991) mentions that his work on parametricity is directly inspired by that of Leivant (1990) on realizability, but does not formalize the parallels.

Conclusion. We have given an account of parametricity and realizability in the framework of PTSs. The result is very concise: the definitions occupy only a dozen of lines. By recognizing the parallels between the two, we are able to further shrink the number of primitive concepts, as we show in Figure 2.4.

Our work points the way towards the transportation of every parametricity theory into a corresponding realizability theory, and *vice versa*.

Acknowledgements

Thanks to Andreas Abel, Thorsten Altenkirch, Thierry Coquand, Peter Dybjer and Guilhem Moulin for helpful comments and discussions.

A Vectors from Lists

When programming with rich type systems, one often defines multiple variants of a structure, with more or less information captured in the type. For example, one may define a structure for lists, and a variant which records the length of the list in an index:

$$\begin{array}{l}\textbf{data}\ \mathsf{List}\ (\alpha : \star) : \star\ \textbf{where}\\ \quad [\,] : \mathsf{List}\ \alpha\\ \quad \_ :: \_ : \alpha \to \mathsf{List}\ \alpha \to \mathsf{List}\ \alpha\end{array}$$

$$\begin{array}{l}\textbf{data}\ \mathsf{Vec}\ (\alpha : \star) : \mathbb{N} \to \star\ \textbf{where}\\ \quad [\,] : \mathsf{Vec}\ \alpha\ \mathsf{zero}\\ \quad \_ :: \_ : \alpha \to (\mathsf{n} : \mathbb{N}) \to \mathsf{Vec}\ \alpha\ \mathsf{n} \to \mathsf{Vec}\ \alpha\ (\mathsf{succ}\ \mathsf{n})\end{array}$$

It is then sometimes unclear which version of the datatype to use for which purpose. Therefore, anticipating a wide range of applications, the authors of such structures cannot help but duplicate the algorithms in addition of the types, as it is done for example in the Agda standard library Danielsson, 2010.

Further, the above basic blocks are often combined to build complex programs, yielding a combinatorial explosion in the number of variants of types. This proliferation of variants makes dependently-typed programming awkward; and ultimately impedes the use of rich types.

We believe that the transformations that we expose here can help relating the various versions of a type, and therefore alleviate the type-explosion problem.

Consider the following version of the List type[8]:

$$\begin{array}{l}\textbf{data}\ \mathsf{List}\ (\alpha : \star) : \lceil\star\rceil\ \textbf{where}\\ \quad [\,] : \mathsf{List}\ \alpha\\ \quad \_ :: \_ : \alpha \to \mathsf{List}\ \alpha \to \mathsf{List}\ \alpha\end{array}$$

We can apply the projecting transformation on this version of List, to obtain $\lfloor\mathsf{List}\rfloor$ (since α is first-level, all its occurrences are removed):

$$\begin{array}{l}\textbf{data}\ \lfloor\mathsf{List}\rfloor : \star\ \textbf{where}\\ \quad \lfloor [\,] \rfloor : \lfloor\mathsf{List}\rfloor\\ \quad \lfloor :: \rfloor \_ : \lfloor\mathsf{List}\rfloor \to \lfloor\mathsf{List}\rfloor\end{array}$$

which is equal to the usual inductive definition of natural numbers, up to renaming:

[8] One may wonder why it is admissible to change the sort of the argument as we do. The reason is that the actual elements of a list are irrelevant to its structure. Using different sorts is a way to express this fact.

```
data ℕ : ⋆ where
  zero : ℕ
  succ_ : ℕ → ℕ
```

One can also apply the realizability transformation on List, and obtain:

```
data ⟨List⟩ (α : ⋆) : ⌊List⌋ → ⌈∗⌉ where
  ⟨ [] ⟩ : ⟨List⟩ α ⌊ [] ⌋
  _⟨::⟩_ : α → (n : ⌊List⌋) → ⟨List⟩ α n → ⟨List⟩ α (⌊::⌋ n)
```

which is (up to renaming) equal to the definition of vectors shown above. The above development can be summarised in the slogan: vectors show that naturals realise lists.

In itself, the above observation is already useful: it can save a lot of work to users of dependently-typed programming languages. Indeed, from regular types, the language may take advantage of realizability to automatically generate indexed versions.

The benefits do not stop there however, since, from any program involving lists, one can extract its homologue on vectors. Consider for example a function appending two lists:

```
_ ++ _ : List α → List α → List α
[] ++ xs = xs
(x : xs) ++ ys = x : (xs ++ ys)
```

its projection is

```
_⌊++⌋_ : ℕ → ℕ → ℕ
⌊ [] ⌋   ⌊++⌋ xs = xs
(⌊::⌋ xs) ⌊++⌋ ys = ⌊::⌋ (xs ⌊++⌋ ys)
```

which is merely addition of naturals; and its realizability interpretation has type

```
_⟨++⟩_ : (n : ℕ) → Vec α n → (m : ℕ) → Vec α m →
         Vec α (n ⌊++⌋ m)
```

and is vector concatenation.

The correspondence goes all the way: by adequacy, one can also transform formulas and proofs concerning lists into formulas and proofs on vectors.

Using a similar technique, one may also transform all programs and proofs on vectors to programs and proofs on lists. In that case one needs to project away the index.

The connection between lists and vectors has been pointed out before. For example, Mcbride (2010) and Atkey, Johann, and Ghani (2010) do it using an algebraic approach. Attempts to unfiy various list-like structures also exist (Danielsson, 2010, Data.Star module). Still, we believe that the connection is an elegant illustration of the power of the realizability transform.

B Details of proofs

This appendix contains the details of the proofs of normalization and abstraction theorems.

B.1 Normalization

Theorem 31 (normalization)**.** *If P is strongly normalizing, so is P^2.*

Proof. The proof is based on the observation (referred as $(*)$ below) that, if a term A is typable in P^2 and not normalizable, then at least either:

- one of the first-level subterms of A is not normalizable, or
- the first-level term $\lfloor A \rfloor$ is not normalizable.

Then, by separation (Theorem 7), the first-level subterms are typable in P, so they must be normalizable. We conclude that A must be normalizable.

To the above observation $(*)$ we first decompose the reduction relation $\longrightarrow_\beta$ into three disjoint relations $\longrightarrow_\beta = \longrightarrow_1 \cup \longrightarrow_2 \cup \longrightarrow_i$:

1. The relation $\longrightarrow_1$ reduces abstractions typable with the rules already in $\mathcal{R}$.
2. The relation $\longrightarrow_2$ reduces abstractions typable with rules of the form $(\lceil s_1 \rceil, \lceil s_2 \rceil, \lceil s_3 \rceil)$ for $(s_1, s_2, s_3) \in \mathcal{R}$.
3. The relation $\longrightarrow_i$ reduces abstractions typable with the other rules (corresponding to interaction reductions).

We then remark the following facts:

1. If $A \longrightarrow_2 A'$, then A is a second-level term and $\lfloor A \rfloor \longrightarrow_\beta \lfloor A' \rfloor$; because the projection does not erase redexes reduced by $\longrightarrow_2$.
2. If A is a second-level term, then
$$A(\longrightarrow_1 \cup \longrightarrow_i)A' \text{ implies } \lfloor A \rfloor = \lfloor A' \rfloor$$
because the projection erases all redexes reduced by $\longrightarrow_1$ and by $\longrightarrow_i$.
3. If $A \longrightarrow_i A'$, then the number of interaction redexes in A has been decreased by one in A'.
Indeed, an interaction redex is always a second-level term and it always involves an abstraction whose argument is a first-level term. Therefore, the argument does not contain any interaction redex and cannot be an abstraction that would create an interaction redex. This is why $\longrightarrow_i$ does not create nor duplicate interaction redexes.

4. The number of interaction redexes is invariant by $\longrightarrow_1$ because interaction redexes are second-level terms.

Let $A \longrightarrow_\beta A_1 \longrightarrow_\beta A_2 \longrightarrow_\beta ... \longrightarrow_\beta A_n \longrightarrow_\beta ...$ be an infinite sequence of terms. [9] Then we are in one of these situations:

- either we can extract a sub-sequence $(A_{n_i})_{i\in\mathbb{N}}$ such that $A_{n_i}(\longrightarrow_1 \cup \longrightarrow_i)^* \cdot \longrightarrow_2 A_{n_{i+1}}$ for all $i \in \mathbb{N}$;
- or there exists a N such that for all $n \geqslant N$, $A_n(\longrightarrow_1 \cup \longrightarrow_i)A_{n+1}$ or more prosaically $\longrightarrow_2$ is not used in the chain starting from N.

In the former case, because $A(\longrightarrow_1 \cup \longrightarrow_i)^* \cdot \longrightarrow_2 A'$ implies $\lfloor A \rfloor \longrightarrow_\beta \lfloor A' \rfloor$, we can build an infinite sequence $(\lfloor A_{n_i} \rfloor)_{i\in\mathbb{N}}$ decreasing for $\longrightarrow_1$.

In the latter case, because $\longrightarrow_i$ strictly decreases the number of insignificant redexes and the reduction $\longrightarrow_1$ does not change this number, there exists an integer $M \geqslant N$, such that for all $n \geqslant M$, $A_n \longrightarrow_1 A_{n+1}$. We can write A_M as $B[x_1 \mapsto t_1, ..., x_k \mapsto t_k]$ where all subterms of B that are types or programs are variables among $\{x_1, ..., x_k\}$. Now, if $B[x_1 \mapsto t_1, ..., x_k \mapsto t_k] \longrightarrow_\beta A_{N+1}$ it means there exists t'_i such that $A_{N+1} = B[x_1 \mapsto t_1, ..., x_i \mapsto t'_i, ..., x_k \mapsto t^k]$ and $t_i \longrightarrow_\beta t'_i$. By iterating this, we can build an infinite decreasing sequence starting from t_i for some $1 \leqslant i \leqslant k$. □

B.2 Abstraction

Lemma 32 ($[\![\cdot]\!]$ and substitution)**.**

$$[\![t[\mathrm{x} \mapsto e]]\!] = [\![t]\!][\overline{\mathrm{x}} \mapsto \overline{e}][\mathring{\mathrm{x}} \mapsto [\![e]\!]]$$

Proof. Recall that if x is free in t, then x_i and $\mathring{x}$ are free in $[\![t]\!]$. The free variable $\mathring{x}$ is introduced by the rule $[\![x]\!] = \mathring{x}$, therefore if x is substituted by e, $\mathring{x}$ must be substituted by $[\![e]\!]$. Similarly, each of the x_i must be substituted by e_i (renaming must be applied to the substituted expression). □

Lemma 33. $A \longrightarrow_\beta B \implies [\![A]\!] \longrightarrow_\beta^* [\![B]\!]$

Proof. By induction on the derivation. All cases are congruences, except for the interesting base case, where β-reduction happens.

In that case, we want to show that if

$$(\lambda x : T.\ t)\, e \longrightarrow_\beta t[\mathrm{x} \mapsto e]$$

[9] The proof may also be carried out constructively: the idea is to reuse the normalization procedure of terms in P to normalize terms in P^2. More precisely, given a well-typed A, one can use the normalization procedure of $\lfloor A \rfloor$ to normalize the 2nd level structure, and normalize the 1st level subterms independently. The separation properties guarantee that the interactions between first and second level structure only add a finite number of β-reductions.

then

$$[\![(\lambda x : T.\ t)\ e]\!] \longrightarrow_\beta^* [\![t[\mathsf{x} \mapsto e]]\!].$$

By definition:

$$\begin{aligned}[\![(\lambda x : T.\ t)\ e]\!] &= [\![\lambda x : T.\ t]\!]\ \overline{e}\ [\![e]\!] \\ &= (\lambda \overline{x : T}.\ \lambda \mathring{x} : [\![T]\!]\ \overline{x}.\ [\![t]\!])\ \overline{e}\ [\![e]\!]\end{aligned}$$

And by Lemma 32, we are left with showing that

$$(\lambda \overline{x : T}.\ \lambda \mathring{x} : [\![T]\!]\ \overline{x}.\ [\![t]\!])\ \overline{e}\ [\![e]\!] \longrightarrow_\beta^* [\![t]\!][\overline{\mathsf{x}} \mapsto \overline{e}][\mathring{\mathsf{x}} \mapsto [\![e]\!]]$$

which one can identify as $n+1$ instances of β-reduction. □

Corollary 34 ($[\![\cdot]\!]$ preserves reduction)**.**

$$A \longrightarrow_\beta^* B \implies [\![A]\!] \longrightarrow_\beta^* [\![B]\!]$$

Furthermore, the number of reductions in the target is $n+1$ times the number of reductions in the source.

Corollary 35 ($[\![\cdot]\!]$ preserves β-equivalence)**.** $A =_\beta B \implies [\![A]\!] =_\beta [\![B]\!]$

The following lemmas (36, 37 and 38) are proved by construction of a derivation tree in P^2 from a derivation tree in P. The three corresponding functions are denoted as follows:

1. $|\cdot|$ for $\Gamma \vdash A : B \Rightarrow [\![\Gamma]\!] \vdash \overline{A : B}$
2. $\{\cdot\}$ for $\Gamma \vdash B : s \Rightarrow [\![\Gamma]\!], \overline{z : B} \vdash \overline{z} \in [\![B]\!] : \lceil s \rceil$
3. $[\![\cdot]\!]$ for $\Gamma \vdash A : B : s \Rightarrow [\![\Gamma]\!] \vdash [\![A]\!] : \overline{A} \in [\![B]\!]$

Even though the constructions are interdependent, it is not difficult to see that recursive calls are made only on strictly smaller trees.

Lemma 36 ($|\cdot|$)**.** $\Gamma \vdash A : B \Rightarrow [\![\Gamma]\!] \vdash \overline{A : B}$

Proof. By the thinning lemma. For each A_i, erase from the context $[\![\Gamma]\!]$ the relational variables and j-indexed variables such that $j \neq i$. The legality of the context is ensured by Lemma 37 and Lemma 38.

□

The following two lemmas proceed by case analysis on the derivation tree. The presentation uses the following conventions:

- Each case is presented separately: first the input tree is recalled, then the transformed tree is shown. The symbol $\Rightarrow$ is used to separate input and output trees.

- The constructions may make use of the other lemmas, and usages are marked by $|\cdot|$, $\{\cdot\}$ or $[\![\cdot]\!]$.
- Usage of the generation lemma (Barendregt, 1992, Theorem 5.2.13) is indicated in the input tree. Consequences of generation are placed higher up.
- For the sake of concision, some usage of the weakening rule are omitted.
- For concision again, mundane parts of the construction are omitted (squiggly lines indicate missing parts).

Lemma 37 ($\{\cdot\}$). $\Gamma \vdash B : s \Rightarrow [\![\Gamma]\!], \overline{z : B} \vdash \overline{z} \in [\![B]\!] : \lceil s \rceil$

Proof. By case analysis on the derivation of $\Gamma \vdash A : B$.

Axiom

$$\frac{}{\vdash s : s'}\,\mathrm{ax} \;\Rightarrow\; \frac{\dfrac{\dfrac{}{\vdash s : s'}\,\mathrm{ax}}{\overline{z : s} \vdash \overline{z : s}}\,\mathrm{st} \quad \dfrac{}{\vdash \lceil s \rceil : \lceil s' \rceil}\,\mathrm{ax}}{\dfrac{\overline{z : s} \vdash \overline{z} \to \lceil s \rceil : \lceil s' \rceil}{\overline{z : s} \vdash \overline{z} \in [\![s]\!] : \lceil s' \rceil}\,\mathrm{def}}\,(s, \lceil s' \rceil, \lceil s' \rceil)$$

Start

$$\frac{\Gamma \vdash s : s'}{\Gamma, x : s \vdash x : s}\,\mathrm{st}$$

$$\Rightarrow$$

$$\frac{\dfrac{\vdots}{[\![\Gamma]\!], \overline{x : s}, \hat{x} : \overline{x} \in [\![s]\!], \overline{z : x} \vdash \hat{x} : \overline{x} \to \lceil s \rceil} \quad \dfrac{\dfrac{\vdots}{[\![\Gamma]\!], \overline{x : s} \vdash \overline{x : s}}}{[\![\Gamma]\!], \overline{x : s}, \hat{x} : \overline{x} \in [\![s]\!], \overline{z : x} \vdash \overline{z : x}}\,\mathrm{st}}{\dfrac{[\![\Gamma]\!], \overline{x : s}, \hat{x} : \overline{x} \in [\![s]\!], \overline{z : x} \vdash \hat{x}\,\overline{z} : \lceil s \rceil}{[\![\Gamma]\!], \overline{x : s}, \hat{x} : \overline{x} \in [\![s]\!], \overline{z : x} \vdash \overline{z} \in [\![x]\!] : \lceil s \rceil}\,\mathrm{def}}\,\mathrm{app}$$

Weakening

$$\frac{\Gamma \vdash A : s \quad \Gamma \vdash C : s'}{\Gamma, x : C \vdash A : s}\,\mathrm{wk}$$

$$\Rightarrow$$

$$\frac{\dfrac{\dfrac{\vdots}{[\![\Gamma]\!], \overline{z : A} \vdash \overline{z} \in [\![A]\!] : \lceil s \rceil} \quad \dfrac{\vdots}{[\![\Gamma]\!] \vdash \overline{C : s'}}}{[\![\Gamma]\!], \overline{x : C}, \overline{z : A} \vdash \overline{z} \in [\![A]\!] : \lceil s \rceil}\,\mathrm{wk} \quad \dfrac{\vdots}{[\![\Gamma]\!], \overline{x : C} \vdash \overline{x} \in [\![C]\!] : \lceil s' \rceil}\,\mathrm{wk}}{[\![\Gamma]\!], \overline{x : C}, \hat{x} : \overline{x} \in [\![C]\!], \overline{z : A} \vdash \overline{z} \in [\![A]\!] : \lceil s \rceil}$$

$\{\Gamma \vdash A : s\}$ $\quad$ $|\Gamma \vdash C : s'|$ $\quad$ $\{\Gamma \vdash C : s'\}$

Abstraction impossible: no type is a lambda abstraction.

Application

$$\dfrac{\Gamma \vdash A : s_1 \quad \Gamma \vdash A \to s : s_3}{\dfrac{\dfrac{\textit{generation}}{\Gamma \vdash F : A \to s} \quad \Gamma \vdash a : A}{\Gamma \vdash F\,a : s}\,\text{app}}$$

$\Rightarrow$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots\ [\![\Gamma \vdash F : A \to s : s_3]\!]}{[\![\Gamma]\!] \vdash [\![F]\!] : \overline{F} \in [\![A \to s]\!]}\,\text{def} \qquad \dfrac{\vdots\ |\Gamma \vdash a : A|}{[\![\Gamma]\!] \vdash \overline{a : A}}\,\text{app}}{[\![\Gamma]\!] \vdash [\![F]\!] : (x : A) \to \overline{x} \in [\![A]\!] \to \overline{F\,x} \in [\![s]\!]} \qquad \dfrac{\vdots\ [\![\Gamma \vdash a : A : s_1]\!]}{[\![\Gamma]\!] \vdash [\![a]\!] : \overline{a} \in [\![A]\!]}\,\text{app}}{[\![\Gamma]\!] \vdash [\![F]\!]\,\overline{a} : \overline{a} \in [\![A]\!] \to \overline{F\,a} \in [\![s]\!]}\,\text{def}}{[\![\Gamma]\!] \vdash [\![F]\!]\,\overline{a}\,[\![a]\!] : \overline{F\,a} \in [\![s]\!]}\,\text{def}}{[\![\Gamma]\!] \vdash [\![F]\!]\,\overline{a}\,[\![a]\!] : \overline{F\,a} \to \lceil s \rceil}$$

$$\dfrac{\dfrac{\dfrac{\vdots\ |\Gamma \vdash F\,a : s|}{[\![\Gamma]\!] \vdash \overline{F\,a : s}}\,\text{st}}{[\![\Gamma]\!], \overline{z : F\,a} \vdash \overline{z : F\,a}}\,\text{app}}{[\![\Gamma]\!], \overline{z : F\,a} \vdash [\![F]\!]\,\overline{a}\,[\![a]\!]\,\overline{z} : \lceil s \rceil}\,\text{def}}{[\![\Gamma]\!], \overline{z : F\,a} \vdash \overline{z} \in [\![F\,a]\!] : \lceil s \rceil}$$

Product

$$\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (x : A) \to B : s}\,(s_1, s_2, s)$$

$\Rightarrow$

$$\dfrac{\dfrac{\dfrac{\vdots}{[\![\Gamma]\!], \overline{z : ((x : A) \to B)}, \overline{x : A} \vdash \overline{z\,x : B}}\,\text{app} \qquad \dfrac{\vdots\ \{\Gamma, x : A \vdash B : s_2\}}{[\![\Gamma]\!], \overline{x : A}, \mathring{x} : \overline{x} \in [\![A]\!], \overline{y : B} \vdash \overline{y} \in [\![B]\!] : \lceil s_2 \rceil}\,\text{substitution}}{[\![\Gamma]\!], \overline{z : ((x : A) \to B)}, \overline{x : A}, \mathring{x} : \overline{x} \in [\![A]\!] \vdash \overline{z\,x} \in [\![B]\!] : \lceil s_2 \rceil}$$

$$\dfrac{\dfrac{\dfrac{\vdots\ |\Gamma \vdash A : s_1|}{[\![\Gamma]\!] \vdash \overline{A : s_1}} \qquad \dfrac{\dfrac{\vdots\ \{\Gamma \vdash A : s_1\}}{[\![\Gamma]\!], \overline{x : A} \vdash \overline{x} \in [\![A]\!] : \lceil s_1 \rceil}}{[\![\Gamma]\!], \overline{z : ((x : A) \to B)}, \overline{x : A} \vdash (\mathring{x} : \overline{x} \in [\![A]\!]) \to \overline{z\,x} \in [\![B]\!] : \lceil s \rceil}\,(\lceil s_1 \rceil, \lceil s_2 \rceil, \lceil s \rceil)}{[\![\Gamma]\!], \overline{z} : ((x : A) \to B) \vdash (\overline{x : A}) \to (\mathring{x} : \overline{x} \in [\![A]\!]) \to \overline{z\,x} \in [\![B]\!] : \lceil s \rceil}\,(s_1, \lceil s \rceil, \lceil s \rceil)\,\text{def}}{[\![\Gamma]\!], \overline{z : ((x : A) \to B)} \vdash \overline{z} \in [\![(x : A) \to B]\!] : \lceil s \rceil}$$

Conversion

$$\dfrac{\Gamma \vdash B : s' \quad s =_\beta s' \quad \Gamma \vdash s : s''}{\Gamma \vdash B : s}\,\text{conv}$$

$\Rightarrow$

$$\dfrac{\dfrac{\vdots\ \{\Gamma \vdash B : s'\}}{[\![\Gamma]\!], \overline{z : B} \vdash \overline{z} \in [\![B]\!] : \lceil s' \rceil} \quad \lceil s \rceil =_\beta \lceil s' \rceil \quad \dfrac{\vdots\ |\Gamma \vdash s : s''|}{[\![\Gamma]\!] \vdash s : s''}\,\text{conv}}{[\![\Gamma]\!], \overline{z : B} \vdash \overline{z} \in [\![B]\!] : \lceil s \rceil}$$

□

Lemma 38 ($[\![\cdot]\!]$). $\Gamma \vdash A : B : s \Rightarrow [\![\Gamma]\!] \vdash [\![A]\!] : \overline{A} \in [\![B]\!]$

Proof. By case analysis on the derivation of $\Gamma \vdash B : s$.

Product, Axiom $\Gamma \vdash T : s : s'$

$\Rightarrow$

$$\dfrac{\dfrac{\dfrac{\vdots\ \{\Gamma \vdash T : s\}}{[\![\Gamma]\!], \overline{z : T} \vdash \overline{z} \in [\![T]\!] : \lceil s \rceil}\quad \dfrac{\dfrac{\vdots\ |\Gamma \vdash T : s|}{[\![\Gamma]\!] \vdash \overline{T : s}}\quad \dfrac{\vdots\ \{\Gamma \vdash s : s'\}}{[\![\Gamma]\!], \overline{z : s} \vdash \overline{z} \in [\![s]\!] : \lceil s' \rceil}\ \text{substitution}}{[\![\Gamma]\!] \vdash \overline{T} \in [\![s]\!] : \lceil s' \rceil}\ \text{def}}{[\![\Gamma]\!] \vdash \overline{T} \to \lceil s \rceil : \lceil s' \rceil}\ \text{abs}}{[\![\Gamma]\!] \vdash \lambda \overline{z : T}.\ \overline{z} \in [\![T]\!] : \overline{T} \to \lceil s \rceil}\ \text{def}}{[\![\Gamma]\!] \vdash [\![T]\!] : \overline{T} \in [\![s]\!]}$$

Start

$$\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}\ \text{st} \quad \Rightarrow \quad \dfrac{\vdots\ \{\Gamma \vdash A : s\}}{[\![\Gamma]\!], \overline{x : A} \vdash \overline{x} \in [\![A]\!] : \lceil s \rceil}\ \text{st}}{[\![\Gamma]\!], \overline{x : A}, \mathring{x} : \overline{x} \in [\![A]\!] \vdash \mathring{x} : \overline{x} \in [\![A]\!]}$$

Weakening

$$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}\ \text{wk}$$

$\Rightarrow$

$$\dfrac{\dfrac{\vdots\ [\![\Gamma \vdash A : B : s]\!]}{[\![\Gamma]\!] \vdash [\![A]\!] : \overline{A} \in [\![B]\!]}\quad \dfrac{\vdots\ |\Gamma \vdash C : s|}{[\![\Gamma]\!] \vdash \overline{C : s}}\ \text{wk}}{[\![\Gamma]\!], \overline{x : C} \vdash [\![A]\!] : \overline{A} \in [\![B]\!]}\quad \dfrac{\vdots\ \{\Gamma \vdash C : s\}}{[\![\Gamma]\!], \overline{x : C} \vdash \overline{x} \in [\![C]\!] : \lceil s \rceil}\ \text{wk}}{[\![\Gamma]\!], \overline{x : C}, \mathring{x} : \overline{x} \in [\![C]\!] \vdash [\![A]\!] : \overline{A} \in [\![B]\!]}\ .$$

The construction also uses that $\Gamma \vdash A : B \,\&\, \Gamma, x : C \vdash B : s \Rightarrow \Gamma \vdash B : s$

Abstraction

$$\dfrac{\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\dfrac{}{\Gamma \vdash ((x : A) \to B) : s}\ \textit{generation}}}{\Gamma \vdash (\lambda x : A.\ b) : ((x : A) \to B)}\ \text{abs}$$

$\Rightarrow$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots\ [\![\Gamma, x : A \vdash b : B : s_2]\!]}{[\![\Gamma]\!], \overline{x : A}, \mathring{x} : \overline{x} \in [\![A]\!] \vdash [\![b]\!] : \overline{b} \in [\![B]\!]}}{[\![\Gamma]\!], \overline{x : A} \vdash (\lambda \mathring{x} : \overline{x} \in [\![A]\!].\ [\![b]\!]) : ((\mathring{x} : \overline{x} \in [\![A]\!]) \to \overline{b} \in [\![B]\!])}\ \text{abs}}{[\![\Gamma]\!] \vdash (\lambda \overline{x : A}.\ \lambda \mathring{x} : \overline{x} \in [\![A]\!].\ [\![b]\!]) : ((\overline{x : A}) \to (\mathring{x} : \overline{x} \in [\![A]\!]) \to \overline{b} \in [\![B]\!])}\ \text{abs}}{[\![\Gamma]\!] \vdash (\lambda \overline{x : A}.\ \lambda \mathring{x} : \overline{x} \in [\![A]\!].\ [\![b]\!]) : ((\overline{x : A}) \to (\mathring{x} : \overline{x} \in [\![A]\!]) \to \overline{(\lambda x : A.\ b)\, x} \in [\![B]\!])}\ \text{conv}$$

Continuations of the tree (squiggly lines) are similar to derivations found in $\{\Gamma \vdash (\Pi x : A.B) : s\}$.

Application

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash A : s_1}{\Gamma \vdash ((x : A) \to B) : s_3}\ (s_1, s_2, s_3)}{\dfrac{}{\Gamma \vdash F : ((x : A) \to B)}\ \textit{generation}} \quad \Gamma \vdash a : A}{\Gamma \vdash F\, a : B[x \mapsto a]}\ \text{app}$$

$\Rightarrow$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots\ [\![\Gamma \vdash F : ((x : A) \to B) : s_3]\!]}{[\![\Gamma]\!] \vdash [\![F]\!] : \overline{F} \in [\![(x : A) \to B]\!] : s_3}\,\text{def} \qquad \dfrac{\vdots\ |\Gamma \vdash a : A|}{[\![\Gamma]\!] \vdash \overline{a : A}}\,\text{app}}{[\![\Gamma]\!] \vdash [\![F]\!] : (\overline{x : A}) \to (\mathring{x} : \overline{x} \in [\![A]\!]) \to \overline{F\,x} \in [\![B]\!]}\,\text{def} \qquad \vdots\ [\![\Gamma \vdash a : A : s_1]\!]}{[\![\Gamma]\!] \vdash [\![F]\!]\,\overline{a} : (\mathring{x} : \overline{a} \in [\![A]\!]) \to (\overline{F\,a} \in [\![B]\!])[\overline{x \mapsto a}]}\,\text{app} \qquad [\![\Gamma]\!] \vdash [\![a]\!] : \overline{a} \in [\![A]\!]}{[\![\Gamma]\!] \vdash [\![F]\!]\,\overline{a}\,[\![a]\!] : (\overline{F\,a} \in [\![B]\!])[\overline{x \mapsto a}][\mathring{x} \mapsto [\![a]\!]]}\,\text{app}}{[\![\Gamma]\!] \vdash [\![F\,a]\!] : \overline{F\,a} \in [\![B[x \mapsto a]]\!]}\,\text{def}$$

Conversion

$$\dfrac{\Gamma \vdash A : B' \quad B =_\beta B' \quad \Gamma \vdash B : s}{\Gamma \vdash A : B}\,\text{conv}$$

$\Rightarrow$

$$\dfrac{\vdots\ [\![\Gamma \vdash A : B' : s']\!] \qquad \dfrac{\vdots\ |\Gamma \vdash A : B| \qquad \vdots\ \{\Gamma \vdash B : s\}}{[\![\Gamma]\!] \vdash \overline{A : B} \qquad [\![\Gamma]\!], \overline{z : B} \vdash \overline{z} \in [\![B]\!] : \lceil s \rceil}\,\text{subs}}{[\![\Gamma]\!] \vdash [\![A]\!] : \overline{A} \in [\![B']\!] \quad \overline{A} \in [\![B]\!] =_\beta \overline{A} \in [\![B']\!] \qquad [\![\Gamma]\!] \vdash \overline{A} \in [\![B]\!] : \lceil s \rceil}\,\text{conv}$$

$\Gamma \vdash B' : s'$ is a consequence of $\Gamma \vdash B : s$ and $B =_\beta B'$ (Corollary 35).

□

Theorem 39 (abstraction). *If* $\Gamma \vdash A : B : s$, *then* $[\![\Gamma]\!] \vdash [\![A]\!] : \overline{A} \in [\![B]\!] : \lceil s \rceil$

Proof. Combine Lemma 37 and Lemma 38. □

Paper III

Testing Polymorphic Properties

The following paper was originally published in the European Symposium of Programming (ESOP) 2010. The version given here includes the appendices, where details of proofs can be found, as well as small corrections. Additionally, the body of the text has been edited for typography, and to match the notation used in the previous papers.

Testing Polymorphic Properties

Jean-Philippe Bernardy, Patrik Jansson,
Koen Claessen

Abstract

This paper is concerned with testing properties of polymorphic functions. The problem is that testing can only be performed on specific monomorphic instances, whereas parametrically polymorphic functions are expected to work for any type. We present a schema for constructing a monomorphic instance for a polymorphic property, such that correctness of that single instance implies correctness for all other instances. We also give a formal definition of the class of polymorphic properties the schema can be used for. Compared with the standard method of testing such properties, our schema leads to a significant reduction of necessary test cases.

1 Introduction

How should one test a polymorphic function?

A modern and convenient approach to testing is to write specifications as properties, and let a tool generate test cases. Such tools have been implemented for many programming languages, such as Ada, C++, Curry, Erlang, Haskell, Java, .NET and Scala (Hoffman, Nair, and Strooper, 1998; Bagge, David, and Haveraaen, 2008; Christiansen and Fischer, 2008; Arts et al., 2006; Claessen and Hughes, 2000; Saff, 2007; Tillmann and Schulte, 2005; Nilsson, 2009). But how should one generate test cases for polymorphic functions? Parametrically polymorphic functions, by their very nature, work uniformly on values of any type, whereas in order to run a concrete test, one must pick values from a specific monomorphic type.

As an example, suppose we have two different implementations of the standard function reverse that reverses a list:

$$\mathsf{reverse1}, \mathsf{reverse2} : \forall\, \mathsf{a}.\, \mathsf{List\ a} \;\to\; \mathsf{List\ a}$$

In order to test that they do the same thing, what monomorphic type should we pick for the type variable a? Standard praxis, as for example used by QuickCheck (Claessen and Hughes, 2000), suggests to simply use a type with a large enough domain, such as natural numbers, resulting in the following property:

$$\forall \mathsf{xs} : \mathsf{List}\,\mathbb{N}\,.\;\; \mathsf{reverse1\ xs} \;==\; \mathsf{reverse2\ xs}$$

Intuitively, testing the functions only on the type $\mathbb{N}$ is "enough"; if the original polymorphic property has a counter example (in this case a monomorphic type T and a concrete list xs : List T), there also exists a counter example to the monomorphic property (in this case a concrete list xs′ : List $\mathbb{N}$).

However, how do we *know* this is enough? And, can we do better than this? This paper aims to provide an answer to these questions for a large class of properties of polymorphic functions. We give a systematic way of computing the monomorphic type that a polymorphic property should be tested on. Perhaps surprisingly, we do this by only inspecting the type of the functions that are being tested, not their definition. Moreover, our method significantly improves on the standard testing praxis by making the monomorphic domains over which we quantify even more precise. For example, to check that reverse1 and reverse2 implement the same function, it turns out to be enough to test:

$$\forall n : \mathbb{N} .\ \mathsf{reverse1}\,[1\,..\,n] == \mathsf{reverse2}\,[1\,..\,n]$$

In other words, we only need to quantify over the *length* of the argument list, and not its elements! This is a big improvement over the previous property; for each list length n, only *one* test suffices, whereas previously, we had an unbounded number of lists to test for each length. This significantly increases test efficiency.

Related Work There are a few cases in the literature where it has been shown that, for a specific polymorphic function, testing it on a particular monomorphic type is enough. For example, Knuth's classical result that verifying a sorting network only has to be done on booleans (Knuth, 1998, sec. 5.3.4), can be cast into a question about polymorphic testing (Day, Launchbury, and Lewis, 1999). The network can be represented as a polymorphic function parametrised over a comparator (a 2-element sorter):

$$\mathsf{sort} : \forall\, a.\, (a \times a \to a \times a) \to \mathsf{List}\, a \to \mathsf{List}\, a$$

Knuth has shown that, in order to check whether such a function really sorts, it is enough to show that it works for booleans; in other words checking if the following function is a sorting function:

$$\begin{aligned} &\mathsf{sort\_Bool} : \mathsf{List\,Bool} \to \mathsf{List\,Bool} \\ &\mathsf{sort\_Bool} = \mathsf{sort}\,(\lambda\,(x, y) \to (x \wedge y, x \vee y)) \end{aligned}$$

Another example is a result by Voigtländer (2008), which says that in order to check that a given function is a scan function, it is enough to check it for all possible combinations on a domain of three elements.

The result we present in this paper has the same motivation as these earlier results, but the concrete details are not exactly the same. In section 4, we compare our general result with Knuth's and Voigtländer's specific results.

Contributions and outlook Our main contribution is a schema for testing polymorphic properties effectively and efficiently. We explain the schema both from a theoretical and practical point of view. Our examples are aimed at giving practitioners a good intuition of the method (section 2) and demonstrate some of its applications (section 4). A more formal exposition is provided in section 3. We cover related and future work in sections 5 and 6 and we conclude in section 7.

2 Examples

In this section, we discuss a number of examples illustrating the idea behind our method in preparation for the more formal treatment in the next section. We are using Haskell-style notation[1] and QuickCheck-style properties here, but our result can be used in the context of other languages and other property-testing frameworks.

Example 1. Let us first compare two implementations of the function filter:

$$\mathsf{filter1}, \mathsf{filter2} : \forall \mathsf{a}.\ (\mathsf{a} \to \mathsf{Bool}) \to \mathsf{List\ a} \to \mathsf{List\ a}$$

A parametric polymorphic function knows nothing about the type it is being used on. So, the only way an element of type a can appear in the result, is if it was produced somehow by the argument of the function. We can analyse the type of the arguments of the functions under test, in order to see in what way the arguments can be used to produce an element of type a. The concrete type A we are going to construct to test the functions on will represent all such ways in which the arguments can be used to produce an a.

In the case of filter, the only way we can produce elements of type a, is by using an element from its argument list (the predicate $(\mathsf{a} \to \mathsf{Bool})$ can only inspect elements). So, a natural choice for A is to be the index of the element from the argument list it used:

$$\begin{array}{l}\textbf{data}\ \mathsf{A} : \star\ \textbf{where}\\ \quad \mathsf{X} : \mathbb{N} \to \mathsf{A}\end{array}$$

In other words, $\mathsf{X\,i}$ stands for the i^{th} element (of type a) from the input list. Now, we have not only fixed a type to use for a, but also decided which elements the list xs should be filled with, once we know the length. Thus, the final monomorphic property becomes:

$$\begin{array}{l}\forall \mathsf{n} : \mathbb{N}\ .\ \mathsf{p} : \mathsf{A} \to \mathsf{Bool}.\ \textbf{let}\ \mathsf{xs} = [\mathsf{X\,1}\ ..\ \mathsf{X\,n}]\\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathsf{filter1\ p\ xs} == \mathsf{filter2\ p\ xs}\end{array}$$

[1] In this version of the paper, the notation has been adapted to improve the coherence with other chapters of the thesis; however we sometimes omit here the type (or kind) of universally quantified variables, as in Haskell. For example $\forall\, \mathsf{a} : \star.\, \mathsf{X}$ is often just written $\forall\, \mathsf{a}.\, \mathsf{X}$

Note that we still need to quantify over the predicate p of type A → Bool. The construction we apply here can be seen as a kind of *symbolic simulation*: we feed the function with symbolic variables (here represented by naturals), and examine the output. This becomes more clear in the next example.

Example 2. Let us take a look at a typical polymorphic property, relating the functions reverse and append (++)

```
∀a : ⋆. ∀xs, ys : List a. reverse (xs ++ ys) == reverse ys ++ reverse xs
```

We can view the left- and right-hand sides of the property as two different polymorphic functions that are supposed to deliver the same result. Where can elements in the result list come from? Either from the list xs, or the list ys. Thus, the monomorphic type A becomes:

```
data A : ⋆ where
  X : ℕ → A
  Y : ℕ → A
```

And in the property, we not only instantiate the type, but also the elements of the lists:

```
∀n, m : ℕ . let xs = [X 1 .. X n]
                ys = [Y 1 .. Y m]
             in reverse (xs ++ ys) == reverse ys ++ reverse xs
```

Example 3. Let us take a closer look at the reverse example again, where we compare two different implementations of the function reverse:

```
reverse1, reverse2 : ∀a. List a → List a
```

The type analysis works in the same way as for reverse. The only way the function can construct elements in the result list is by taking them from the argument list; the function argument can only inspect the elements, not create them. So, the monomorphic datatype A becomes:

```
data A : ⋆ where
  X : ℕ → A
```

And the property:

```
∀n : ℕ . let xs = [X 1 .. X n] in reverse1 xs == reverse2 xs
```

Arguments of higher-order functions can not only be used to *inspect* elements of type a, but also used to *construct* such elements, as the next example shows.

Example 4. Let us now compare two implementations of the function iterate:

$$\mathsf{iterate1}, \mathsf{iterate2} : \forall\, \mathsf{a}.\ (\mathsf{a} \to \mathsf{a}) \to \mathsf{a} \to \mathsf{List\ a}$$

The expression $\mathsf{iterate\ f\ x}$ generates the infinite list $[\mathsf{x}, \mathsf{f\ x}, \mathsf{f\ (f\ x)}, \ldots]$. The standard way of testing equality between infinite lists is to compare finite prefixes:

$$\begin{array}{l}\forall \mathsf{a} : \star.\ \forall \mathsf{k} : \mathbb{N}.\ \ \forall \mathsf{f} : \mathsf{a} \to \mathsf{a}.\ \forall \mathsf{x} : \mathsf{a}. \\ \quad \mathsf{take\ k\ (iterate1\ f\ x)} \ == \ \mathsf{take\ k\ (iterate2\ f\ x)}\end{array}$$

In order to calculate a suitable monotype A to test these functions on, we again analyse the sources of as. The two possible sources are: the second argument x, and the first argument f. The first argument needs another a in order to produce an a. The resulting monotype A thus becomes *recursive*:

$$\begin{array}{l}\textbf{data}\ \mathsf{A} : \star\ \textbf{where} \\ \quad \mathsf{X} : \mathsf{A} \\ \quad \mathsf{F} : \mathsf{A} \to \mathsf{A}\end{array}$$

And the monomorphic property becomes:

$$\forall \mathsf{k} : \mathsf{Nat}.\ \mathsf{take\ k\ (iterate1\ F\ X)} \ == \ \mathsf{take\ k\ (iterate2\ F\ X)}$$

The interesting thing is that we do not even need to quantify over the arguments of the functions anymore!

Finally, an example of a property that does not hold.

Example 5. Take a look at the following property which claims that map and filter commute (which is incorrect as formulated).

$$\begin{array}{l}\forall \mathsf{a} : \star.\ \forall \mathsf{p} : \mathsf{a} \to \mathsf{Bool}.\ \forall \mathsf{f} : \mathsf{a} \to \mathsf{a}.\ \forall \mathsf{xs} : \mathsf{List\ a}. \\ \quad \mathsf{map\ f\ (filter\ p\ xs)} \ == \ \mathsf{filter\ p\ (map\ f\ xs)}\end{array}$$

A typical QuickCheck user may pick a to be $\mathbb{N}$, and running QuickCheck might produce the following counterexample[2]:

$$\begin{array}{lll}\mathsf{p} & = & \{1 \to \mathsf{True}, \_ \to \mathsf{False}\} \\ \mathsf{f} & = & \{\_ \to 1\} \\ \mathsf{xs} & = & [3]\end{array}$$

In other words, if p is a predicate that holds only for 1, and f is the constant function 1, and if we start with a list $[3]$, the property does not hold.

Investigating the left- and right-hand sides as functions from p, f, and xs to lists, we see that an element of type a may either directly come from the

[2]Using a recent QuickCheck extension to show functions.

list xs, or be the result of applying f. Expressing this in terms of a datatype, we get:

```
data A : ⋆ where
  X : ℕ → A
  F : A → A
```

And the property turns into:

```
∀p : A → Bool. ∀n : ℕ . let xs = [X 1 .. X n]
                        in map F (filter p xs) == filter p (map F xs)
```

The only arguments we need to quantify over are the predicate p and the length of the list xs: the function f is fixed to the constructor F. But there is one more advantage; the counterexample that is produced is more descriptive:

```
p = {F (X 1) → True, _ → False}
f = F
xs = [X 1]
```

We clearly see that p holds only for the result of applying f to the (only) element in the list xs.

3 Generalisation

In this section we present a systematic formulation of our schema to test polymorphic functions. Additionally we expose the main theoretical results that back up the method and argue for their correctness. We assume familiarity with basic notions of category theory, notably the interpretation of data types as initial algebras (Bird and de Moor, 1997, ch. 2).

3.1 Notation

We use a notation close to that of Bird and de Moor (1997), but borrow the names of functions from the Haskell prelude. Other notable idiosyncrasies are the following:

- If F denotes a functor, then the action on morphisms is also written F. (In Haskell it would be the fmap instance for type constructor F).
- An (F-)algebra is a pair of a type a and function of type F a → a, but we often omit the type component. If an initial algebra exists, we call the type component the least fixed point of F, and write it μF.
- The catamorphism (also known as fold) of the algebra p : F a → a is denoted by ([p]) : μF → a.

- The letters σ, τ, denote type expressions. The brackets in $\sigma[a]$ indicate that a may appear in the expression σ.
- The operators $+$ and $\times$ denote sum- and product-types, respectively.
- We often use explicit $\forall$ in type schemes.

3.2 Revisiting reverse

We start by going through all the necessary steps for one particular concrete example, namely testing two implementations of reverse against each other:

$$\mathsf{reverse1}, \mathsf{reverse2} : \forall\, \mathsf{a}.\, \mathsf{List\ a} \;\to\; \mathsf{List\ a}$$

The method we use makes a clear distinction between *arguments* (values that are passed to the function) and *results* (values which are delivered by the function, and should be compared with other results). Furthermore, the arguments are divided up into two kinds: arguments that can be used by the function to *construct* elements of type a, and arguments that can only be used to *observe* arguments of type a.

The first step we take in order to compute the monomorphic instance is to transform the function under test into a function that makes these three parts of the function type explicit. The final type we are looking for is of the form:

$$\mathsf{Canonic} \;=\; \forall\, \mathsf{a}.\, (\mathsf{F\ a} \to \mathsf{a}) \;\times\; (\mathsf{G\ a} \to \mathsf{X}) \to \mathsf{H\ a}$$

for functors $\mathsf{F}, \mathsf{G}, \mathsf{H}$ and a monomorphic type X. The argument of type $\mathsf{F\ a} \to \mathsf{a}$ can be used to construct elements of type a, the argument of type $\mathsf{G\ a} \to \mathsf{X}$ can be used to observe arguments of type a (by transforming them into a known type X), and $\mathsf{H\ a}$ is the result of the function. We call the type above the *canonical testing type*; all polymorphic functions of the above type can be tested using our method, if there exists an initial F-algebra.

How do we transform functions like reverse into functions with a canonical testing type? We start by "dissecting" arguments that can produce as into functions that produce exactly one a. For reverse, the one argument that contains as is of type $\mathsf{List\ a}$. We now make use of the fact that all lists can be represented by a pair of its length and its indexing function, and we thus replace the list argument with an argument of type $\mathbb{N} \times (\mathbb{N} \to \mathsf{a})$ (we will say more about this transformation in section 3.6). After re-ordering the arguments the new type is

$$\forall\, \mathsf{a}.\, (\mathbb{N} \to \mathsf{a}) \;\times\; \mathbb{N} \to \mathsf{List\ a}$$

which fits the requirement, with $\mathsf{F\ a} = \mathbb{N}$, $\mathsf{G\ a} = ()$[3], $\mathsf{X} = \mathbb{N}$, and $\mathsf{H\ a} = \mathsf{List\ a}$.

[3] taking advantage of the isomorphism between $() \to \mathbb{N}$ and $\mathbb{N}$.

For the original function reverse1 (and similarly for reverse2), we can define a corresponding function with a canonical testing type as follows:

$$\begin{aligned}&\mathsf{reverse1'} : \forall\, \mathsf{a}.\, (\mathbb{N} \to \mathsf{a}) \times \mathbb{N} \to \mathsf{List\ a}\\ &\mathsf{reverse1'} = \mathsf{reverse1} \circ \mathsf{project}\end{aligned}$$

This uses an auxiliary function to project the arguments of the new function to the initial arguments:

$$\begin{aligned}&\mathsf{project} : (\mathbb{N} \to \mathsf{a}) \times \mathbb{N} \to \mathsf{List\ a}\\ &\mathsf{project}\,(\mathsf{f}, \mathsf{obs}) = \mathsf{map\ f}\,[1 \,..\, \mathsf{obs}]\end{aligned}$$

Observe that if the new arguments properly cover the domain $(\mathbb{N} \to \mathsf{a}) \times \mathbb{N}$, then the original arguments also properly cover the domain $\mathsf{List\ a}$. It means that the transformations that we have performed to fit the canonical testing type do not weaken the verification procedure.

What have we gained by this rewriting? Our main result says: to test whether two polymorphic functions with a canonical testing type are equal, it is enough to test for equality on the monomorphic type A, where A is the least fixpoint of the functor F, and to use the initial algebra $\alpha : \mathsf{F\ A} \to \mathsf{A}$ as the first argument.

For the reverse example, the least fixpoint of F is simply $\mathbb{N}$ and the initial algebra is the identity function. Thus, to check if reverse1′ and reverse2′ are equal, we merely have to check

$$\forall\, \mathsf{obs} : \mathbb{N}\,.\ \mathsf{reverse1'}\,(\mathsf{id}, \mathsf{obs}) \mathrel{==} \mathsf{reverse2'}\,(\mathsf{id}, \mathsf{obs})$$

Writing the transformation explicitly is cumbersome, and indeed we can avoid it, by picking arguments directly from the image of the partially applied projection function, that is, from the set $\{\,\mathsf{project}\,(\mathsf{id}, \mathsf{obs}) \mid \mathsf{obs} \in \mathbb{N}\,\}$. By doing so, we obtain the property given in the introduction.

$$\forall\, \mathsf{n} : \mathbb{N}\,.\ \mathsf{reverse1}\,[1 \,..\, \mathsf{n}] \mathrel{==} \mathsf{reverse2}\,[1 \,..\, \mathsf{n}]$$

3.3 Overview

In general, given a function of type $\forall\, \mathsf{a}.\, \sigma[\mathsf{a}] \to \mathsf{H\ a}$, the objective is to construct a type A, and identify a set of arguments of type $\sigma[\mathsf{a} := A]$ to test it against. To do so, we proceed with the following three steps.

1. Transform the function to test, whose type is $\forall\, \mathsf{a}.\, \sigma[\mathsf{a}] \to \mathsf{H\ a}$, into a function whose type is in the canonical form

   $$\forall\, \mathsf{a}.\, (\mathsf{F\ a} \to \mathsf{a}) \times (\mathsf{G\ a} \to \mathsf{X}) \to \mathsf{H\ a}$$

   where F, G, H are functors. This must be done through an embedding-projection pair $((\mathsf{e}, \mathsf{p}) : \sigma[\mathsf{a}] \subseteq (\mathsf{F\ a} \to \mathsf{a}) \times (\mathsf{G\ a} \to \mathsf{X}))$. The purpose is

to identify all the ways (for the function) to construct values of type a, and express them as an algebra of a functor F. (Sect. 3.6).

2. Calculate the initial algebra $(\mu F, \alpha)$ of the functor F. Parametricity and initiality implies that fixing the algebra to α and a to μF still covers all cases. Note that the quantification over the type argument has now been removed. (Sect. 3.4)

3. Re-interpret the fixing of the algebra to α in step 2 in the context of the original type, using the projection produced in step 1. The arguments to test the function on are picked in the set $\{ p\,(\alpha, s) \mid s \in G\,(\mu F) \to X \}$. (Sect. 3.5)

After these steps the type argument is gone, and testing can proceed as usual. We detail the procedure and argue for its validity in the following sections.

3.4 The initial view

In this section we expose and justify the crucial step of our approach: the removal of polymorphism itself. We begin with showing that applications of (some) polymorphic functions can be expressed in terms of a monomorphic case.

Suppose that the polymorphic function has type

$$\forall a.\,(F\,a \to a) \times (G\,a \to X) \to H\,a$$

that is, its only way to construct values of type a are given by an algebra of functor F, (X is a constant type where a cannot appear). Then, instead of passing a given algebra to a polymorphic function, one can pass the initial algebra, and use the catamorphism of the algebra to translate the results. If the function can also observe the values of the polymorphic parameter, then the observation functions passed as argument must be composed with the catamorphism.

By passing the initial algebra, the type parameter is fixed to μF. The applications of the catamorphism handle the polymorphism, effectively hiding it from the function under test. The following theorem expresses the idea formally. Our proof relies on parametricity (Wadler, 1989) and properties of initial algebras (Bird and de Moor, 1997, ch. 2)

Theorem 1. *Let*

- F, G, H *be functors and*
- $f : (\forall a : \star.\,(F\,a \to a) \times (G\,a \to X) \to H\,a)$.

If there is an initial F*-algebra* $(\mu\mathsf{F}, \alpha)$*, then*

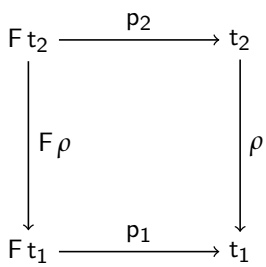
$$\begin{aligned}&\forall\, \mathsf{t} : \star, \mathsf{p} : \mathsf{F}\,\mathsf{t} \to \mathsf{t}, \mathsf{r} : \mathsf{G}\,\mathsf{t} \to \mathsf{X}.\\ &\quad \mathsf{f}_{\mathsf{t}}\,(\mathsf{p}, \mathsf{r}) \;=\; \mathsf{H}\,([\![\mathsf{p}]\!])\,(\mathsf{f}_{\mu\mathsf{F}}\,(\alpha, \mathsf{r} \circ \mathsf{G}\,([\![\mathsf{p}]\!])))\end{aligned}$$

Proof. We apply the parametricity theorem (restricted to functions) on the type of f, following mechanically the rules given by Fegaras and Sheard (1996, theorem 1). After simplification we obtain:

$$\begin{aligned}&\forall\, \mathsf{f} : (\forall\, \mathsf{a} : \star.\, (\mathsf{F}\,\mathsf{a} \to \mathsf{a}) \;\times\; (\mathsf{G}\,\mathsf{a} \to \mathsf{X}) \to \mathsf{H}\,\mathsf{a}),\\ &\quad \mathsf{t}_1, \mathsf{t}_2 : \star,\;\; \rho : \mathsf{t}_2 \to \mathsf{t}_1,\\ &\quad \mathsf{p}_1 : \mathsf{F}\,\mathsf{t}_1 \to \mathsf{t}_1, \mathsf{p}_2 : \mathsf{F}\,\mathsf{t}_2 \to \mathsf{t}_2,\;\; \mathsf{r} : \mathsf{G}\,\mathsf{t}_1 \to \mathsf{X}.\\ &\mathsf{p}_1 \circ \mathsf{F}\,\rho \;=\; \rho \circ \mathsf{p}_2 \;\Rightarrow\; \mathsf{f}_{\mathsf{t}_1}\,(\mathsf{p}_1, \mathsf{r}) = \mathsf{H}\,\rho\,(\mathsf{f}_{\mathsf{t}_2}\,(\mathsf{p}_2, \mathsf{r} \circ \mathsf{G}\,\rho))\end{aligned}$$

This equation expresses a general case $(\mathsf{f}_{\mathsf{t}_1}\,(\mathsf{p}_1, \mathsf{r}))$ in terms of a specific case $(\mathsf{H}\,\rho\,(\mathsf{f}_{\mathsf{t}_2}\,(\mathsf{p}_2, \mathsf{r} \circ \mathsf{G}\,\rho)))$, under the assumption $\mathsf{p}_1 \circ \mathsf{F}\,\rho \;=\; \rho \circ \mathsf{p}_2$. Here, we hope to find specific values for t_2, q and ρ which verify the assumption, and obtain a characterisation of the polymorphic case in terms of a monomorphic case.

Satisfying the assumption $(\mathsf{p}_1 \circ \mathsf{F}\,\rho \;=\; \rho \circ \mathsf{p}_2)$ is equivalent to making the diagram on the right commute.



Let us pick the following values for t_2, p_2 and ρ:

- $\mathsf{t}_2 \;=\; \mu\mathsf{F}$, the least fixpoint of F;
- $\mathsf{p}_2 \;=\; \alpha$, the initial F-algebra;
- $\rho \;=\; ([\![\mathsf{p}_1]\!])$, the catamorphism of p_1.

We know from properties of initial algebras and catamorphisms that these choices make the diagram commute. Thus, the assumption is verified, and the proof is complete.

□

Remark 2. *The above theorem is a generalisation of (the inverse of) Church encodings.*

The purpose of Church encodings is to encode data types in the pure λ*-calculus. Church encodings can also target the polymorphic* λ*-calculus (Böhm and Berarducci, 1985), and the resulting types are polymorphic. In essence, a data-type which is the fixpoint of a functor* $(\mu\,\mathsf{F})$ *is encoded to the type* $\forall\,\mathsf{a}.\,(\mathsf{F}\,\mathsf{a} \;\to\; \mathsf{a}) \;\to\; \mathsf{a}$*. Conversely, the concrete type* $\mu\,\mathsf{F}$ *is a proper representation of* $\forall\,\mathsf{a}.\,(\mathsf{F}\,\mathsf{a} \;\to\; \mathsf{a}) \;\to\; \mathsf{a}$*.*

Therefore, for the special case of $\mathsf{G} \;=\; ()$ *and* $\mathsf{H}\,\mathsf{a} \;=\; \mathsf{a}$*, the above theorem is a direct consequence of the correctness of Church encodings.*

$$
\begin{array}{ll}
 & \forall s : G\,(\mu\,F) \to X, \alpha : F\,(\mu\,F) \to (\mu\,F)\,. \qquad f_{\mu F}\,\alpha\,s = g_{\mu F}\,\alpha\,s \\
\Rightarrow & \quad \{\text{by } r \circ G\,([p]) \text{ being a special case of } s\} \\
 & \forall p, r. \qquad f_{\mu F}\,\alpha\,(r \circ G\,([p])) = g_{\mu F}\,\alpha\,(r \circ G\,([p])) \\
\Rightarrow & \quad \{\text{by (cata p) being a function}\} \\
 & \forall p, r. \qquad H\,([p])\,(f_{\mu F}\,\alpha\,(r \circ G\,([p]))) = H\,([p])\,(g_{\mu F}\,\alpha\,(r \circ G\,([p])))\\
\Rightarrow & \quad \{\text{by theorem 1}\} \\
 & \forall p, r. \qquad f_a\,p\,r = g_a\,p\,r
\end{array}
$$

Figure 3.1: Long proof for theorem 3. The universally quantified variables have the following types, when omitted: $a : \star, p : F\,a \to a, r : G\,a \to X$.

Theorem 1 shows that we can express a polymorphic function in terms of a particular monomorphic instance, but the expressions still involve applying (polymorphic) catamorphisms. In the case where we have a function to test (f) and a model (g) to compare against, we can apply theorem 1 to both sides and simplify away the catamorphisms.

Theorem 3. *Let* F, G, H *be functors, let* $f, g : \forall\, a : \star.\,(F\,a \to a) \times (G\,a \to X) \to H\,a$. *If there is an initial* F*-algebra* $(\mu F, \alpha)$*, then*

$$
\begin{array}{rl}
\forall\, s : G\,(\mu F) \to X. & f_{\mu F}\,(\alpha, s) = g_{\mu F}\,(\alpha, s) \\
\Rightarrow \forall\, a : \star, p : F\,a \to a, r : G\,a \to X. & f_a\,(p, r) = g_a\,(p, r)
\end{array}
$$

Proof. If $f_{\mu F}\,(\alpha, s) = g_{\mu F}\,(\alpha, s)$ holds for any s, then in particular the equality $f_{\mu F}\,(\alpha, r \circ G\,([p])) = g_{\mu F}\,(\alpha, r \circ G\,([p]))$ holds. Applying $H\,([p])$ to both sides of the equality preserves it, and then we can use theorem 1 to transform both sides and obtain that $f_a\,(p, r) = g_a\,(p, r)$ holds for any choice of a, p and r. The steps are detailed formally in figure 3.1. □

3.5 General form of arguments

The results of the previous section apply only to functions of the canonical type $(\forall\, a.\,(F\,a \to a) \times (G\,a \to X) \to H\,a)$. In this section we show that we can extend these results to any argument types which can be *embedded* in $(F\,a \to a) \times (G\,a \to X)$.

Definition 4. An embedding-projection pair (an EP) is a pair of functions $e : A \to B$, $p : B \to A$ satisfying $p \circ e = \mathrm{id}$. Because it constitutes evidence

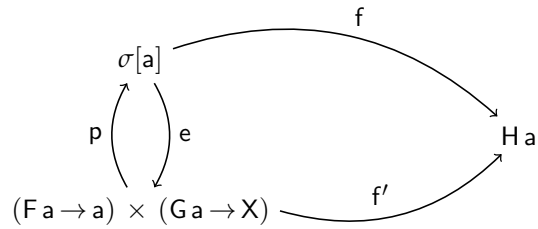


Figure 3.2: Algebra isolation

that covering B is enough to cover A, we write $(e, p) : A \subseteq B$ to denote such a pair.

Given an EP[4] $(e, p) : \sigma[a] \subseteq (F\,a \to a) \times (G\,a \to X)$, one can transform the arguments calculated in the previous section (α paired with any function of type $G\,(\mu F) \to X$) into $\sigma[a]$ by using the projection component, p. The existence of the embedding guarantees that the domain of the original function is properly covered. This idea is expressed formally in the following theorem. The type information is summarised diagrammatically in figure 3.2.

Theorem 5. *Let* F, G, H *be functors and let* $f, g : \forall\, a.\, \sigma[a] \to H\,a$. *If there is an initial* F*-algebra* $(\mu F, \alpha)$ *and an EP* $(e, p) : \sigma[a] \subseteq (F\,a \to a) \times (G\,a \to X)$, *then*

$$\begin{aligned} &\forall\, s : G\,(\mu F) \to X. && f_{\mu F}\,(p\,(\alpha, s)) = g_{\mu F}\,(p\,(\alpha, s)) \\ \Rightarrow\ &\forall\, a : \star, l : \sigma[a]. && f_a\,l = g_a\,l \end{aligned}$$

[4]Strictly speaking, this is a polymorphic EP — one EP for each type a.

Proof. Apply theorem 3 to $f' = f \circ p$ and $g' = g \circ p$ as follows:

$$
\begin{array}{ll}
 & \forall s : G(\mu F) \to X. \qquad f_{\mu F}(p(\alpha, s)) = g_{\mu F}(p(\alpha, s)) \\
\Leftrightarrow & \quad \{\text{by definition of } \circ\} \\
 & \forall s : G(\mu F) \to X. \qquad (f_{\mu F} \circ p)(\alpha, s) = (g_{\mu F} \circ p)(\alpha, s) \\
\Leftrightarrow & \quad \{\text{by definition of } f' \text{ and } g'\} \\
 & \forall s : G(\mu F) \to X. \qquad f'_{\mu F}(\alpha, s) = g'_{\mu F}(\alpha, s) \\
\Rightarrow & \quad \{\text{by theorem 3}\} \\
 & \forall a : \star, q : (F\,a \to a) \times (G\,a \to X). \qquad f'_a\,q = g'_a\,q \\
\Rightarrow & \quad \{\text{by } e\,l \text{ being a special case of } q\} \\
 & \forall a : \star, l : \sigma[a]. \qquad f'_a(e\,l) = g'_a(e\,l) \\
\Leftrightarrow & \quad \{\text{by definition of } f' \text{ and } g'\} \\
 & \forall a : \star, l : \sigma[a]. \qquad (f_a \circ p)(e\,l) = (g_a \circ p)(e\,l) \\
\Leftrightarrow & \quad \{\text{by definition of } \circ\} \\
 & \forall a : \star, l : \sigma[a]. \qquad f_a((p \circ e)\,l) = g_a((p \circ e)\,l) \\
\Leftrightarrow & \quad \{\text{by the EP law: } p \circ e \equiv \mathsf{id}\} \\
 & \forall a : \star, l : \sigma[a]. \qquad f_a\,l = g_a\,l
\end{array}
$$

□

Properties used for testing are not always expressed in terms of a model, but very often directly as a predicate: they are merely Boolean-valued functions. We can specialise the above result to that case: given a polymorphic predicate, it is enough to verify it for the initial algebra.

Theorem 6. *Let* F, G *be functors, let* $f : \forall a.\, \sigma[a] \to \mathsf{Bool}$. *If there is an EP* $(e, p) : \sigma[a] \subseteq (F\,a \to a) \times (G\,a \to X)$ *and an initial* F*-algebra* $(\mu F, \alpha)$, *then*

$$
\begin{array}{ll}
 & \forall s : G(\mu F) \to X. \qquad f_{\mu F}(p(\alpha, s)) \\
\Rightarrow & \forall a : \star, l : \sigma[a]. \qquad f_a\,l
\end{array}
$$

Proof. Substitute $\mathsf{const}\,\mathsf{True}$ for g in theorem 5. □

One might think that theorem 5, about models, follows from theorem 6, about properties, using $f(p, r) = \mathsf{test}(p, r) == \mathsf{model}(p, r)$. This is in fact invalid in general, because one cannot assume that equality ($==$) is available for arbitrary types. Indeed, our usage of parametricity in the proof assumes the opposite.

The above results show that it is enough to test on arguments picked from the set $I = \{p(\alpha, s) \mid s : G(\mu F) \to X\}$. This could be done by picking

elements s in $G\,(\mu F) \to X$ and testing on $p\,(\alpha, s)$. However, for the efficiency of testing, it is important *not* to proceed as such, because it can cause redundant tests to be performed. This is because the projection can map different inputs into a single element in I. A better way to proceed is to generate elements of I directly.

3.6 Embedding construction

The previous section shows that our technique can handle arguments that can be embedded in $(F\,a \to a) \times (G\,a \to X)$. In this section we show that all first-order polymorphic arguments can be embedded. Our proof is constructive: it is also a method to build the EP. It is important to construct the embedding because it is used in computing the set of arguments to test the property on.

The general form of a first order argument is a function of type $C\,a \to D\,a$, where C and D are functors and D is polynomial[5]. Note that non-functional values can be represented by adding a dummy argument. Similarly, the above form includes n-ary functions, as long as they are written in an uncurried form. We structure the proof as a series of embedding steps between the most general form and the canonical form. EPs for each step are composed into the final EP. The overall plan is to split all complex arguments into observations or constructors, then group each class together. Lemmas detailing these important steps are given after the top-level proof outline.

Theorem 7. *Let C_i and D_i be functors. If D_i are constructed by sum, products and fixpoints $(0, 1, +, \times, \mu)$, and none of the $C_i\,a$ are empty, then there exist functors F, G and an EP $(e, p) : \forall\, a : \star.\ \bigtimes_i (C_i\,a \to D_i\,a) \subseteq (F\,a \to a) \times (G\,a \to X)$.*

[5]Contructed only from $+$ and $\times$.

Proof.
$$
\begin{aligned}
& \bigtimes_i (C_i\,\mathsf{a} \to D_i\,\mathsf{a}) \\
\subseteq \quad & \{\text{by lemma 9}\} \\
& \bigtimes_i (C_i\,\mathsf{a} \to (S_i \times (P_i \to \mathsf{a}))) \\
\equiv \quad & \{\text{by distributing } \to \text{ over } \times\} \\
& \bigtimes_i (C_i\,\mathsf{a} \to S_i) \times (C_i\,\mathsf{a} \times P_i \to \mathsf{a}) \\
\equiv \quad & \{\text{by letting } F_i\,\mathsf{a} = G_i\,\mathsf{a} \times P_i\} \\
& \bigtimes_i (C_i\,\mathsf{a} \to S_i) \times (F_i\,\mathsf{a} \to \mathsf{a}) \\
\equiv \quad & \{\text{by commutativity and associativity of } \times\} \\
& \bigtimes_i (C_i\,\mathsf{a} \to S_i) \times \bigtimes_i (F_i\,\mathsf{a} \to \mathsf{a}) \\
\subseteq \quad & \{\text{by lemma 8}\} \\
& (\mathsf{G}\,\mathsf{a} \to \mathsf{X}) \times \bigtimes_i (F_i\,\mathsf{a} \to \mathsf{a}) \\
\equiv \quad & \{\text{by } (\tau_1 \to a) \times (\tau_2 \to a) \equiv (\tau_1 + \tau_2) \to a\} \\
& (\mathsf{G}\,\mathsf{a} \to \mathsf{X}) \times (\mathsf{F}\,\mathsf{a} \to \mathsf{a})
\end{aligned}
$$
where $\mathsf{G}\,\mathsf{a} = \bigtimes_i (C_i\,\mathsf{a})$; $\mathsf{F}\,\mathsf{a} = +_i (F_i\,\mathsf{a})$ and X is given by the following lemma from C_i and S_i. □

Lemma 8. *For all types σ_1, σ_2 and non-empty types τ_1, τ_2 ($\mathsf{witness}_1 : \tau_1$ and $\mathsf{witness}_2 : \tau_2$) then there exists $(\mathsf{e}, \mathsf{p}) : (\tau_1 \to \sigma_1) \times (\tau_2 \to \sigma_2) \subseteq \tau_1 \times \tau_2 \to \sigma_1 \times \sigma_2$.*

Proof. The embedding applies the embedded functions pair-wise.

$$\mathsf{e}\,(\mathsf{f}_1, \mathsf{f}_2) = \lambda\,(\mathsf{t}_1, \mathsf{t}_2) \to (\mathsf{f}_1\,\mathsf{t}_1, \mathsf{f}_2\,\mathsf{t}_2)$$

The projection can be constructed by providing dummy arguments ($\mathsf{witness}$) to missing parts of the pair. It is safe to do so, because that part of the pair is ignored by the embedding e anyway. That is, $\mathsf{p} \circ \mathsf{e} = \mathsf{id}$ regardless of the choice of witnesses.

$$
\begin{aligned}
\mathsf{p}\,\mathsf{h} = (&\lambda\,\mathsf{t}_1 \to \mathsf{fst}\ (\mathsf{h}\,(\mathsf{t}_1 \quad\quad\ , \mathsf{witness}_2)), \\
&\lambda\,\mathsf{t}_2 \to \mathsf{snd}\,(\mathsf{h}\,(\mathsf{witness}_1, \mathsf{t}_2 \quad\quad\ )))
\end{aligned}
$$

□

Lemma 9. *Let D be a functor constructed by sum, products and fixpoints. Then there exist types S, P and $(\mathsf{e}, \mathsf{p}) : \mathsf{D}\,\mathsf{a} \subseteq \mathsf{S} \times (\mathsf{P} \to \mathsf{a})$*

Proof. D represents a data structure, which can be decomposed into a shape (S) and a function from positions inside that shape to elements ($\mathsf{P} \to \mathsf{a}$). (See appendix B for a detailed discussion). The shape can be obtained by using trivial elements ($\mathsf{S} = \mathsf{D}\,1$). For testing purposes, only

structures with a finite number of elements can be generated, and therefore one can use natural numbers for positions ($P = \mathbb{N}$). The projection can traverse the data structure in pre-order and use the second component of the pair ($\mathbb{N} \to \mathsf{a}$) to look up the element to put at each position (as done by Voigtländer (2009*a*)). The corresponding embedding is easy to build.

□

3.7 Correctness in practice

We have reasoned in a fast-and-loose fashion: our proofs rely on the strongest version of parametricity, which holds only in the polymorphic lambda-calculus.

Applying them to languages with non-termination (like Haskell) is merely "morally correct" (Danielsson et al., 2006). In general, we assume that the functions under test are well-behaved with respect to parametricity: they should not make use of side-effects, infinite data structures, bottoms, etc. In the context of random or exhaustive testing, these assumptions are generally valid. Therefore, our results are readily applicable in practice with a very high level of confidence.

Still, we could extend the result by using a more precise version of parametricity, as for example Johann and Voigtländer (2006) expose it.

4 More examples

In this section, we will deal with some more complicated examples.

4.1 Multiple type parameters

While the theoretical development assumes a single type parameter, we can apply our schema to functions with multiple type parameters. The basic idea is to treat parameters one at a time, assuming the others constant. We do not justify this formally, but merely show how to proceed on a couple of examples.

Example 6 (map)**.** Consider the ubiquitous function map, which applies a function to each element in a list.

$$\mathsf{map} : \forall\, \mathsf{a}\, \mathsf{b}.\, (\mathsf{a} \to \mathsf{b}) \to \mathsf{List}\, \mathsf{a} \to \mathsf{List}\, \mathsf{b}$$

As usual, we are interested in testing a candidate map function against a known-working model.

We first aim to remove the type parameter a. To do so, we isolate the constructors for a by embedding the list argument into a shape (the length

of the list) and a function giving the element at each position (see lemma 9). We obtain the type $\forall\, a\, b.\, (a \to b) \to \mathbb{N} \to (\mathbb{N} \to a) \to \mathsf{List}\, b$. We see from the type that the only constructor is an algebra of the functor $F\, a = \mathbb{N}$. The initial F-algebra is

$$\begin{array}{l}\textbf{data}\, A : \star\, \textbf{where}\\ \quad X : \mathbb{N} \to A\end{array}$$

After substitution, we have the type $\forall\, b.\, (A \to b) \to \mathbb{N} \to (\mathbb{N} \to A) \to \mathsf{List}\, b$, and we know that the third argument is fixed to X.

We can proceed and remove the type parameter b. There is only one constructor for b, which is already isolated, so the initial algebra is easy to compute:

$$\begin{array}{l}\textbf{data}\, B : \star\, \textbf{where}\\ \quad F : A \to B\end{array}$$

After substitution, we have the type $(A \to B) \to \mathbb{N} \to (\mathbb{N} \to A) \to \mathsf{List}\, B$, and we know that the first argument is fixed to F. The second and third arguments can be projected back into a list, so we get the final property:

$$\begin{array}{l}\forall\, n : \mathbb{N}.\ \textbf{let}\, xs = [X\, 1\, ..\, X\, n]\\ \qquad\quad \textbf{in}\ \mathsf{map}_1\, F\, xs == \mathsf{map}_2\, F\, xs\end{array}$$

Note that the function to pass to map is fixed: again, only testing for various lengths is enough!

Example 7 (prefix)**.** In Haskell, the standard function isPrefixOf tests whether its first argument is a prefix list of its second argument. isPrefixOf normally uses the overloaded equality $((==) : a \to a \to \mathsf{Bool})$ to compare elements in the first list to elements in the second one. Instead we consider a more general version that explicitly takes a comparison function as parameter. In that case, the types of elements in input lists do not have to match. This generalisation is captured in a type as follows:

$$\mathsf{isPrefixOf} : \forall\, a\, b.\, (a \to b \to \mathsf{Bool}) \to \mathsf{List}\, a \to \mathsf{List}\, b \to \mathsf{Bool}$$

In this example, the type arguments are completely independent, so we can remove both at once. Both lists can be embedded into a shape $(\mathbb{N})$ and a function from positions $(\mathbb{N} \to a)$ in the familiar way. We get the type: $\forall\, a\, b.\, (a \to b \to \mathsf{Bool}) \to \mathbb{N} \to (\mathbb{N} \to a) \to \mathbb{N} \to (\mathbb{N} \to b) \to \mathsf{Bool}$.

Computing the initial algebras offers no surprise. We obtain:

$$\begin{array}{l}\textbf{data}\, A : \star\, \textbf{where}\\ \quad X : \mathbb{N} \to A\\ \textbf{data}\, B : \star\, \textbf{where}\\ \quad Y : \mathbb{N} \to B\end{array}$$

We have to test functions of type $(A \to B \to \mathsf{Bool}) \to \mathbb{N} \to (\mathbb{N} \to A) \to \mathbb{N} \to (\mathbb{N} \to B) \to \mathsf{Bool}$, with the third argument fixed to X and the fifth fixed to Y. Again, by using the projection, we know that we can instead generate lists of $X\,i$ and $Y\,j$ to pass directly to the polymorphic function.

Thus, a property to check that two implementations of isPrefixOf have the same behaviour is written as follows:

$$
\begin{array}{l}
\forall\, \mathsf{eq} : A \to B \to \mathsf{Bool}, \mathsf{m} : \mathbb{N}, \mathsf{n} : \mathbb{N}\,. \\
\quad \mathbf{let}\, \mathsf{xs} = [X\,1\,..\,X\,\mathsf{m}] \\
\qquad\; \mathsf{ys} = [Y\,1\,..\,Y\,\mathsf{n}] \\
\quad \mathbf{in}\ \mathsf{isPrefixOf1}\ \mathsf{eq}\ \mathsf{xs}\ \mathsf{ys} \;==\; \mathsf{isPrefixOf2}\ \mathsf{eq}\ \mathsf{xs}\ \mathsf{ys}
\end{array}
$$

What if we had used the type $\forall\, a.\, (a \to a \to \mathsf{Bool}) \to \mathsf{List}\, a \to \mathsf{List}\, a \to \mathsf{Bool}$[6], which is not the most precise type that can be given to the function? In that case, the initial algebra would be

$$
\begin{array}{l}
\mathbf{data}\, A : \star\, \mathbf{where} \\
\quad X : \mathbb{N} \to A \\
\quad Y : \mathbb{N} \to A
\end{array}
$$

and the property would look exactly the same. The difference is that the function eq would be quantified over a larger set. It would only be passed values of the form $X\,i$ for the first argument, and $Y\,i$ for the second argument, but the generator of random values does not "know" it, because the type we gave is too imprecise. Therefore, it might generate redundant test cases, where eq only differs in its results for argument-pairs that are not in the form $X\,i$, $Y\,i$. As we have seen in the above example, this redundancy is avoided by using the most general type. This is another example where more polymorphism makes testing more efficient.

4.2 Assumptions on arguments

It can be quite challenging to write properties for functions whose arguments must satisfy non-trivial properties. For example, generating associative functions or total orders is not obvious. A naïve solution is to generate unrestricted arguments and then condition the final property on the arguments being well behaved. This can be highly inefficient if the probability to generate a well-behaved argument is small. Since our technique fixes some parameters, it is sometimes easier to find (or more efficient to generate) arguments with a complex structure. We give examples in the following sections.

[6] This type is isomorphic to the type of the function isPrefixOf from the standard Haskell libraries, $\forall\, a.\, \mathsf{Eq}\, a \Rightarrow \mathsf{List}\, a \to \mathsf{List}\, a \to \mathsf{Bool}$

Example 8 (Parallel Prefix)**.** A parallel-prefix computation computes the list $[x_1, x_1 \oplus x_2, \ldots, x_1 \oplus \ldots \oplus x_n]$, given an associative operation $\oplus$ and a list of inputs $x_1, \ldots, x_n$. How can we test that two given parallel-prefix computations have equivalent outputs?

We start with the type $\forall\, \mathsf{a}.\, (\mathsf{a} \to \mathsf{a} \to \mathsf{a}) \to \mathsf{List}\,\mathsf{a} \to \mathsf{List}\,\mathsf{a}$. To isolate the constructors, we rewrite the list type as usual and get

$$\forall\, \mathsf{a}.\, (\mathsf{a} \to \mathsf{a} \to \mathsf{a}) \to \mathbb{N} \to (\mathbb{N} \to \mathsf{a}) \to \mathsf{List}\,\mathsf{a}$$

We can group the constructors to make the algebra more apparent:

$\forall\, \mathsf{a}.\, ((\mathsf{a} \times \mathsf{a} + \mathbb{N}) \to \mathsf{a}) \to \mathbb{N} \to \mathsf{List}\,\mathsf{a}$. The next step is to pick the initial algebra.

One might be tempted to use the following datatype and its constructors for the initial algebra.

```
data A : ⋆ where
  OPlus : A → A → A
  X : ℕ → A
```

However, we must take into account that the operation passed to the prefix computation must be associative. The OPlus constructor retains too much information: one can recover how the applications of $\oplus$ were associated by examining the structure of A. In order to reflect associativity, a "flat" structure is required. Thus, one should work with lists[7], as follows:

```
A = List ℕ
x n = [n]
oplus = (++)
```

The final property is therefore:

```
∀ n : ℕ . let xs = map x [1 .. n]
          in prefix1 oplus xs == prefix2 oplus xs
```

The problem of testing parallel prefix networks has been studied before, notably by Sheeran, who has presented a preliminary version of our result in an invited talk in Hardware Design and Functional Languages (Sheeran, 2007). Voigtländer (2008) presents another monomorphic instance: he shows that it is enough to test over a 3-value type (**3**). At first sight, it might seem that testing over **3** is better than over $\mathbb{N}$. However, merely substituting the type-variable with **3** still requires testing all combinations of the other arguments, yielding 113×3^n tests[8] to cover the lists of length n, while by

[7] Ideally a sequence with efficient concatenation should be used, such as finger trees.

[8] Voigtländer (2008) shows that only some combinations are relevant, but the number of tests is still quadratic in the length of the input list. 113 is the number of associative functions in $\mathbf{3} \to \mathbf{3} \to \mathbf{3}$.

our method a single test is enough for a given length. Again, the efficiency of our method comes from the fixing of more arguments than the type variable.

The above explanation to deal with associativity relies very much on intuition, but it can be generalised. One must always take in account the laws restricting the input when computing the initial algebra: that is, one must find the initial object of the category of algebras that respect those laws. We direct the interested reader to Fokkinga (1996) for details.

Example 9 (Insertion in sorted list)**.** Consider testing an insertion function which assumes that its input list is strictly ascending. That is, its type is $\forall\, a.\, (a \to a \to \mathsf{Bool}) \to a \to \mathsf{List}\, a \to \mathsf{List}\, a$, but the list argument is restricted to lists that are strictly ascending according to the first argument, which in turn must be a strict total order. After breaking down the list as usual one must handle the type $\forall\, a.\, (a \to a \to \mathsf{Bool}) \to a \to \mathbb{N} \to (\mathbb{N} \to a) \to \mathsf{List}\, a$.

Forcing the list to be sorted can be tricky to encode as a property of an algebra. So, instead of constraining the lists, we put all the burden on the first argument (an observation): it must be a strict total order that also makes the list ascending. This change of perspective lets us calculate the initial algebra without limitation. We obtain

$$\begin{array}{l}\textbf{data}\ \mathsf{A} : \star\ \textbf{where}\\ \quad \mathsf{Y} : \mathsf{A}\\ \quad \mathsf{X} : \mathbb{N} \to \mathsf{A}\end{array}$$

The element to insert is Y, and as in many preceding examples, the function receives lists of the form $[\mathsf{X}\,1\,..\,\mathsf{X}\,n]$. This makes generating suitable orders $(\mathsf{A} \to \mathsf{A} \to \mathsf{Bool})$ easy. Indeed, for such an order (ord) to respect the order of the list, it must satisfy the equation:

$$\mathsf{ord}\,(\mathsf{X}\,i)\,(\mathsf{X}\,j) \;=\; i < j$$

Therefore, we only need to decide on how to order Y with respect to $\mathsf{X}\,i$. That is, decide where to position Y in the list. For an input list of length n, there are exactly $n + 1$ possible positions to insert an element. The final property shows how to define the order given a position k for Y.

$$\begin{array}{l}\forall\, n : \mathbb{N}, k : \{0\,..\,n\}.\ \textbf{let}\, xs \;=\; [\mathsf{X}\,1\,..\,\mathsf{X}\,n]\\ \qquad\qquad\qquad\qquad\ \ \textbf{in}\ \mathsf{insert1}\,(\mathsf{ord}\,k)\,\mathsf{Y}\,xs \;==\; \mathsf{insert2}\,(\mathsf{ord}\,k)\,\mathsf{Y}\,xs\\ \textbf{where}\ \mathsf{ord}\,k\,(\mathsf{X}\,i)\,(\mathsf{X}\,j) \;=\; i < j\\ \qquad \mathsf{ord}\,k\,\mathsf{Y}\quad\ \mathsf{Y} \;=\; \mathsf{False}\\ \qquad \mathsf{ord}\,k\,(\mathsf{X}\,i)\,\mathsf{Y} \;=\; i \leqslant k\\ \qquad \mathsf{ord}\,k\,\mathsf{Y}\quad\ (\mathsf{X}\,j) \;=\; k < j\end{array}$$

Example 10 (Sorting network)**.** A generator of sorting networks can be represented as a polymorphic function of type $\forall\, \mathsf{a}.\, (\mathsf{a} \times \mathsf{a} \to \mathsf{a} \times \mathsf{a}) \to \mathsf{List}\, \mathsf{a} \to \mathsf{List}\, \mathsf{a}$. The first argument is a two-element comparator. Note that, by parametricity, the function cannot check whether the comparator swaps its inputs or not. It is restricted to merely compose instances of the comparator.

Let us apply our schema on the above type. We use the isomorphism $\tau \to a \times b \cong (\tau \to a) \times (\tau \to b)$ to split the first argument, and handle the list as usual. We obtain the following type.

$$\forall\, \mathsf{a}.\, (\mathsf{a} \times \mathsf{a} \to \mathsf{a}) \to (\mathsf{a} \times \mathsf{a} \to \mathsf{a}) \to \mathbb{N} \to (\mathbb{N} \to \mathsf{a}) \to \mathsf{List}\, \mathsf{a}$$

If we overlook the restrictions on the constructors, the initial algebra is

$$\begin{array}{l}\textbf{data}\ \mathsf{A} : \star\ \textbf{where}\\ \quad \mathsf{Min} : \mathsf{A} \to \mathsf{A} \to \mathsf{A}\\ \quad \mathsf{Max} : \mathsf{A} \to \mathsf{A} \to \mathsf{A}\\ \quad \mathsf{X} : \mathsf{Int} \to \mathsf{A}\end{array}$$

As usual, the sorting function is to be run on $[\mathsf{X}\,1\,..\,\mathsf{X}\,n]$. The comparator is built out of Min and Max. Therefore, to fully test the sorting function, it suffices to test the following function.

$$\begin{array}{l}\mathsf{sort\_Lat} : \mathbb{N} \to \mathsf{List}\, \mathsf{a}\\ \mathsf{sort\_Lat}\, n = \mathsf{sort}\, (\lambda\, (x, y) \to (\mathsf{Min}\, x\, y, \mathsf{Max}\, x\, y))\, [\mathsf{X}\,1\,..\,\mathsf{X}\,n]\end{array}$$

The output is a list where each element is a comparison tree: a description of how to compute the element by taking minimums and maximums of some elements of the input. In order to verify that the function works, we are left with checking that the output trees are those of a correct sorting function.

Note that this must be checked modulo the laws which restrict our initial algebra. Min and Max must faithfully represent 2-element comparators which can be passed to the polymorphic function. Therefore, the type A must be understood as a free distributive lattice (Davey and Priestley, 2002) where Min and Max are meet ($\wedge$) and join ($\vee$) and $\mathsf{X}\,i$ are generators. Note that every term of a free distributive lattice can be transformed to the *normal form* given in appendix C.

The correctness of the function can then be expressed as checking each element of the output (o_k) against the output of a known sorting function. Formally:

$$o_k = \bigvee_{M \subseteq \{1 \ldots n\}, \#M = n-k} \left(\bigwedge_{i \in M} \mathsf{X}\,i \right)$$

There are (at least) two possible approaches to proceed with the verification.

1. Verify the equivalence symbolically, using the laws of the distributive lattice. This is known as the word problem for distributive lattices. One way to do this is to test for syntactical equivalence after transformation to normal form.

2. Check the equivalence for all possible assignments of booleans to the variables $X\,i$, meet and join being interpreted as Boolean conjunction and disjunction. This is valid because truth tables are a complete interpretation of free distributive lattices. (See appendix C). In effect, proceeding as such is equivalent to testing the sorting function on all lists of booleans.

   This second way to test equivalence shows that our technique is essentially (at least) as efficient as that of Knuth (1998), provided that properties of the distributive lattice structure are cleverly exploited.

5 Related work

Universe-bound polymorphism Jansson, Jeuring, and students of the Utrecht University Generic Programming class (2007) have studied the testing of datatype-generic functions: polymorphic functions where the type parameter is bound to a given universe. This restriction allows them to proceed by case analysis on the shape of the type. In contrast, our method makes the assumption that type parameters are *universally* quantified, taking advantage of parametricity. Since universal quantification and shape analysis are mutually exclusive, Jansson's method and ours complement each other very well.

Shortcut Fusion Shortcut deforestation (Gill, Launchbury, and Peyton Jones, 1993) is a technique to remove intermediate lists. A pre-requisite to shortcut deforestation is that producers of lists are written on the form $g\ (:)\ [\,]$, or essentially, $g\ \alpha$ where α is the initial algebra of the list functor. In general, functions that are normally written in terms of the initial algebra must be parametrised over any algebra, thereby adding a level of polymorphism. This is the exact opposite of the transformation we perform.

Similarity with our work does not stop here, as the correctness argument for shortcut deforestation also relies heavily on polymorphism and parametricity.

Defunctionalisation Reynolds describes defunctionalisation: a transformation technique to remove higher-order functions (Reynolds, 1998). Each λ-abstraction is replaced by a distinctive constructor, whose argument holds the free variables. Applications are implemented via case-analysis: the tag of the constructor tells which abstraction is entered.

criterion	traditional	new
type	$\mathbb{N}$	μF
constructors	$F\,\mathbb{N} \to \mathbb{N}$	$\{\alpha\}$
observations	$G\,\mathbb{N} \to X$	$G\,(\mu F) \to X$

Table 3.1: Comparison of the traditional QuickCheck praxis to the new method.

Danvy and Nielsen (2001) have shown that defunctionalisation works as an inverse to Church encoding. Thus, theorem 1 can be seen as a special case of defunctionalisation, targeted at the constructors of a polymorphic type. However, our main focus is not the removal of function parameters, but of type parameters. Indeed, our embedding step, which *introduces* function parameters, is often crucial for the removal of polymorphism. Note also that we do not transform the function under test. In fact, only the arguments passed to the function are defunctionalised. The constructing functions are transformed to constructors of a datatype, and the observations have to perform case-analysis on this datatype.

Concretisation Pottier and Gauthier (2006) introduce *concretisation*: a generalisation of defunctionalisation that can target any source language construct by translating its introduction form into an injection, and its elimination form into case analysis. They apply concretisation to Rémy-style polymorphic records and Haskell type classes, but not removal of polymorphism altogether.

QuickCheck As explained in the introduction, the standard way to test polymorphic functions in QuickCheck (Claessen and Hughes, 2000) is to substitute $\mathbb{N}$ for polymorphic parameters. In the first runs, QuickCheck assigns only small values to parameters of this type, effectively testing small subsets of $\mathbb{N}$. As testing progresses, the size is increased. This strategy is already very difficult to beat! Indeed, we observe that, thanks to parametricity, if one verifies correctness for a type of size n, the function works for all types of size n or less. Additionally, because of the inherent nature of testing, it is only possible to run a finite number of test cases. Therefore, the standard QuickCheck strategy of type instantiation is already very good. We can do better because, in addition to fixing the type, we also fix some (components of) parameters passed to the function. In effect, meaningless tests (tests that are isomorphic to other already run tests, or tests that are unnecessarily specific) are avoided.

The situation is summarised in table 3.1. By fixing the constructors, a whole dimension is removed from the testing space. Even though the space of observations is enlarged when the type μF is bigger than $\mathbb{N}$ (from

$G\,\mathbb{N} \to X$ to $G\,(\mu F) \to X$), the trade-off is still beneficial in most cases. We argue informally as follows: if $\mu F > \mathbb{N}$, then F is a "big" functor, such as $F\,a = 1 + a \times a$. This means that the set $F\,\mathbb{N} \to \mathbb{N}$ is big, and as we replace that by a singleton set, this gain dwarfs the ratio between $G\,(\mu F) \to X$ and $G\,\mathbb{N} \to X$, for any polynomial functor G.

Besides efficiency, another benefit to the new method is that the generated counter examples are more informative, as seen on an example in section 2.

In Haskell, there is another pitfall to substituting the polymorphic parameter by $\mathbb{N}$: type classes. Imagine for example that the type parameter is constrained to be an instance of the `Eq` typeclass. Because $\mathbb{N}$ is such an instance, it *is* possible to use it for the type parameter, but this badly skews the distribution of inputs. Indeed, on average, the probability that `a == b`, for generated `a` and `b` tends to be very small. A better strategy would be to have a different instance of `Eq` for each run, each with a probability of equality close to $1/2$. Our method does not suffer from this problem: we insist that the methods of classes are explicitly taken into account when identifying the constructors and the observations.

Exhaustive Checking We argue in the previous section that using $\mathbb{N}$ for type parameters is a sensible approach for random testing. However, as Runciman, Naylor, and Lindblad (2008) remark, this does not work as well for depth-bound exhaustive testing: the dimension of the test space for constructors $F\,\mathbb{N} \to \mathbb{N}$) grows exponentially as the depth of the search increases. They suggest to use smaller types to test on (such as the unit or Boolean), but the user of the library is left to guess which size is suitable. Our method kills two birds with one stone: we conjure up a suitable type parameter to use, and prevent the exponential explosion of the search for constructors by fixing them. Therefore, we believe that our method is an essential improvement for exhaustive testing of polymorphic functions.

Typeful programming We make essential use of types. Many property-based testing tools use type information to generate suitable parameters to test functions automatically. Here we further the exercise and show that polymorphism and calculations at the type level can produce type-level and (more precise) value-level arguments for polymorphic functions.

Symbolic execution Tillmann and Schulte (2005) generate test cases by symbolic execution of the property to check. As we have mentioned in section 2, our technique can be understood as symbolic execution, therefore, generating test cases by symbolic execution potentially subsumes our method. The advantage of our approach is that it is purely type-based: the monomorphic instance is independent of the actual definition of the property. Therefore, it can work with an underlying black-box tester for

monomorphic code.

6 Future work

While the scope of this paper is the testing of polymorphic functions, our technique to remove polymorphism is not specific to testing: any kind of verification technique can be applied on the produced monomorphic instance. This suggests that it may have applications outside the domain of testing, maybe in automated theorem proving. This remains to be investigated.

Automated test-case generation libraries typically address the problem of generating random values for monomorphic arguments. We have addressed the problem of calculating values for type arguments. A natural development would be to unify both approaches in the framework of a dependently-typed programming language. A first step towards this goal would be to give a detailed account of parametricity in presence of dependent types.[9]

With the exception of computing initial algebras with laws, the technique described here is completely algorithmic. Therefore, one can assume that it is easy to automate it and build a QuickCheck-like library to test polymorphic properties. However, such a tool would need to analyse the type structure of the functions it is given, and languages based on the polymorphic lambda calculus typically lack such a feature. Moreover, this very feature would invalidate the parametricity theorem, since it relies on universally quantified types being opaque, thereby invalidating our "monomorphisation" transformation. A long term area of research would be to design a programming language where parametricity and type-analysis can be specified on a case-by-case basis. As a short-term goal, we propose to mechanise the technique as an external tool rather than a library, or require the programmer to explicitly inform the polymorphic test generator about the type structure.

We have shown how to get rid of polymorphism using the "initial view" of the type parameters. As there exists a dual to shortcut fusion (Svenningsson, 2002), we conjecture that there exists a dual to our method, using the "final view". That is, the function should be transformed to isolate a co-algebra and fix it to the final element of the category. It is unclear at this point what would be the outcome of this dual in terms of testing behaviour.

The technique that we present requires a specific form for the type of the function to test. While our examples show that this form covers a wide range of polymorphic functions that are commonly tested, one can still

[9] This sentence reflects the status at the time of submission of the paper to ESOP. The previous chapters of the thesis constitute an attempt to tackle the problem.

aspire for a larger applicability. We hope to improve this aspect, either by showing that more types can be embedded, or by amending the core theory. In particular, we address only rank-1 polymorphism: extending to rank-n would be useful. Also, the restriction that F must be a functor in $(\mathsf{F}\,\mathsf{a} \to \mathsf{a}) \times (\mathsf{G}\,\mathsf{a} \to \mathsf{X})$ seems too specific. Indeed, Church-encoding some types may lead to F being a type-function that is not a functor, and there is *a-priori* no reason that the encoding cannot be reverted. An example is given by Washburn and Weirich (2003): $\mathbf{data}\,\mathsf{T} = \mathsf{Lam}\,(\mathsf{T} \to \mathsf{T}) \mid \mathsf{App}\,\mathsf{T}\,\mathsf{T}$ is encoded as $\forall\,\mathsf{a}.\,((\mathsf{a} \to \mathsf{a}) \to \mathsf{a}) \to (\mathsf{a} \to \mathsf{a} \to \mathsf{a}) \to \mathsf{a}$, and $\mathsf{F}\,\mathsf{a} = (\mathsf{a} \to \mathsf{a}) + (\mathsf{a} \times \mathsf{a})$, which is not a functor. We hope to achieve this by fully explaining our technique in a defunctionalisation setting.

7 Conclusion

We have presented a schema for efficient testing of polymorphic properties. The idea is to substitute polymorphic values by a faithful symbolic representation. This symbolic representation is obtained by type analysis, in two steps:

1. isolation of the constructors (yielding a functor F); and
2. restriction to the initial F-algebra.

We suspect that neither of these steps is original, but we could not find them spelt out as such, and therefore we believe that bringing them to the attention of the programming languages community is worthwhile. Furthermore, the testing of polymorphic properties is a novel application for these theoretical ideas.

We have shown on numerous examples, and informally argued that applying our technique not only *enables* testing polymorphic properties by removing polymorphism, but yields good efficiency compared to the standard praxis of substituting $\mathbb{N}$ for the polymorphic argument. In some cases, this improvement is so dramatic that it makes the difference between testing being useful or not. As another evidence of the value of the method, we have applied it to classical problems and have recovered or improved on the corresponding specific results.

Giving a more polymorphic type to a given function enlarges its domain, so one might think that this can increase the amount of testing necessary to verify properties about that function. If our technique is applied, the opposite is true.

You love polymorphism, but you were afraid that it would complicate testing? Fear no more! On the contrary, polymorphism can facilitate testing if approached from the right angle.

Acknowledgments.

We would like to give special thanks to Marcin Zalewski, whose repeated interest for the topic and early results gave us the motivation to pursue the ideas presented in this paper. Peter Dybjer gave useful references about Church encodings. Anonymous reviewers and Jasmin Blanchette gave useful comments and helped improve the presentation of the paper. This work is partially funded by the Swedish Research Council.

A Applying parametricity

In this we section show the details of applying the parametricity theorem to our canonical testing type: $\forall\, a.\, (F\, a \to a) \times (G\, a \to X) \to H\, a$. The idea behind parametricity is that types can be interpreted as relations, and all values in a closed types are related to themselves. Using double brackets to denote the relation corresponding to a type, the parametricity theorem applied to $f : \forall\, a.\, (F\, a \to a) \times (G\, a \to X) \to H\, a$ is

$$[\![\, \forall\, a.\, (F\, a \to a) \to (G\, a \to X) \to H\, a \,]\!]\, f\, f$$

Our task is to evaluate the double brackets to obtain a theorem about f expressed in higher order logic. To do so we could use the definition given in previous papers (Definition 9, page 22). However, it is more convenient to use a version where the relations corresponding to type variables are specialized to functions. Hence, we follow the rules given by Fegaras and Sheard (1996). The first step is to handle polymorphism, quantifying over any function (ρ) between any types.

$$\forall\, t_1, t_2 : \star, \rho : t_2 \to t_1.\, [\![\, (F\, a \to a) \to (G\, a \to X) \to H\, a \,]\!]_{a \mapsto \rho}\, f_{t_1}\, f_{t_2}$$

The index $[\![\ldots]\!]_{a \mapsto \rho}$ is a reminder that the type variable a is to be interpreted as the function ρ inside the double brackets. In the interest of conciseness, this index is left implicit if there is no ambiguity as to the interpretation of a type variable.

The rest of the proof is a mostly straightforward interpretation of the type as relation, and is detailed in figure 3.3. The most tricky case involves applications of functors, so we detail it in the following lemma.

Lemma 10. *Let* $F : \star \to \star$ *be a functor,* $t_1, t_2 : \star, \rho : t_2 \to t_1, x_1 : F\, t_1, x_2 : F\, t_2$. *Then*

$$[\![F\, a]\!]_{a \mapsto \rho}\, x_1\, x_2 \Rightarrow x_1 \;=\; F\, \rho\, x_2$$

Proof.

$$
\begin{array}{ll}
 & [\![\mathsf{F}\,\mathsf{a}]\!]\,x_1\,x_2 \\
\equiv & \quad \{\text{by interpretation of functor application}\} \\
 & \forall\, u : t_2 \to t_1. \qquad (\forall\, z : t_2.\, [\![\mathsf{a}]\!]\, z\,(u\,z)) \Rightarrow x_1 \;=\; \mathsf{F}\,u\,x_2 \\
\equiv & \quad \{\text{by interpretation of type variable}\} \\
 & \forall\, u : t_2 \to t_1. \qquad (\forall\, z : t_2, u\,z \;=\; \rho\,z) \Rightarrow x_1 \;=\; \mathsf{F}\,u\,x_2 \\
\equiv & \quad \{\text{by extensionality}\} \\
 & \forall\, u : t_2 \to t_1. \qquad\qquad\qquad u \;=\; \rho \Rightarrow x_1 \;=\; \mathsf{F}\,u\,x_2 \\
\equiv & \quad \{\text{by elimination of } u\} \\
 & \qquad\qquad\qquad\qquad\qquad\qquad x_1 \;=\; \mathsf{F}\,\rho\,x_2
\end{array}
$$

□

B Embedding containers

In section 3.6 we have given a very restricted view of containers. Abbott, Altenkirch, and Ghani (2003); Morris and Altenkirch (2009) provide the full picture. In this section we refine lemma 9 by using a more precise definition of containers.

Abbott et al. observe that data types can be expressed as dependent pairs of a *shape* $(\mathsf{s} : \mathsf{S})$ and a function from *positions* $(\mathsf{P}\,\mathsf{s})$ inside that shape to the type of the element. (We only deal with the 1-parameter case here.) Figure 3.4 gives the definitions for containers and expresses the list type as a container.

The transformation of types into their container form yields a dependent pair $((\mathsf{s} : \mathsf{S}) \times (\mathsf{P}\,\mathsf{s} \to \mathsf{a}))$, where a is the polymorphic parameter we attempt to eliminate. As its name indicates, a dependent pair does not quite achieve independence of the two components. Therefore, transformation into containers does not quite achieve the separation between constructors and observations. It is unclear how to compute the initial algebra in this setting. Therefore, we choose to embed the container types in a (non-dependent) pair: $(\mathsf{s} : \mathsf{S}) \times (\mathsf{P}\,\mathsf{s} \to \mathsf{a})$. This pair type can be properly broken down into an observation (S) and a constructor $(\mathsf{s} : \mathsf{S}) \times (\mathsf{P}\,\mathsf{s} \to \mathsf{a})$. For lists the observation (the shape) is just the length and the constructor is a function from list index to list elements. It bears repeating that the observation will be generated randomly, and the constructor fixed by taking the initial view. (They will be recombined using the projection.) Therefore, to fully cover all test cases for a container, one needs only a single test per shape.

$\forall\, t_1, t_2 : \star, \rho : t_2 \to t_1.$ $\quad [\![(F\,a \to a) \to (G\,a \to X) \to H\,a]\!]_{a \mapsto \rho}\, f_{t_1}\, f_{t_2}$

$\Rightarrow$ {by definition of $[\![\ldots]\!]$ on function type}

$[\![F\,a \to a]\!]\, p_1\, p_2$ $\Rightarrow [\![G\,a \to X]\!]\, r_1\, r_2$ $\Rightarrow [\![H\,a]\!]\,(f_{t_1}\, p_1\, r_1)\,(f_{t_2}\, p_2\, s)$

$\Rightarrow$ {by definition of $[\![\ldots]\!]$ on function type}

$(\forall\, x : F\,t_1, y : F\,t_2.\, [\![F\,a]\!]\, x\, y \Rightarrow [\![a]\!]\,(p_1\, x)\,(p_2\, y))$ $\Rightarrow (\forall\, x : G\,t_1, y : G\,t_2.\, [\![G\,a]\!]\, x\, y \Rightarrow [\![X]\!]\,(r_1\, x)\,(r_2\, y))$ $\Rightarrow [\![H\,a]\!]\,(f_{t_1}\, p_1\, r_1)\,(f_{t_2}\, p_2\, s)$

$\Rightarrow$ {by applying 3 times Lemma 10 and $[\![X]\!]$ is equality}

$(\forall\, x : F\,t_1, y : F\,t_2.\, x = F\,\rho\, y \Rightarrow p_1\, x = \rho\,(p_2\, y))$ $\Rightarrow (\forall\, x : G\,t_1, y : G\,t_2.\, x = G\,\rho\, y \Rightarrow r_1\, x = r_2\, y)$ $\Rightarrow f_{t_1}\, p_1\, r_1 = H\,\rho\,(f_{\_}\,\{t_2\}\, p_2\, s)$

$\Rightarrow$ {by substitutions}

$(\forall\, y : F\,t_2.\, p_1\,(F\,\rho\, y) = \rho\,(p_2\, y))$ $\Rightarrow (\forall\, y : G\,t_2.\, r_1\,(G\,\rho\, y) = r_2\, y)$ $\Rightarrow f_{t_1}\, p_1\, r_1 = H\,\rho\,(f_{t_2}\, p_2\, s)$

$\Rightarrow$ {by definition of function composition and extensional equality on functions}

$p_1 \circ F\,\rho = \rho \circ p_2$ $\Rightarrow r_1 \circ G\,\rho = r_2$ $\Rightarrow f_{t_1}\, p_1\, r_1 = H\,\rho\,(f_{t_2}\, p_2\, s)$

$\Rightarrow$ {by elimination of s}

$p_1 \circ F\,\rho = \rho \circ p_2$ $\Rightarrow f_{t_1}\, p_1\, r_1 = H\,\rho\,(f_{t_2}\, p_2\,(r_1 \circ G\,\rho))$

Figure 3.3: Applying parametricity, part 2. In the above we omit the binders for the variables p, q, r and s, which are implicitly universally quantified as follows: $p_1 : F\,t_1 \to t_1, p_2 : F\,t_2 \to t_2, r_1 : G\,t_1 \to X, s : G\,t_2 \to X$

```
Cont : (S : ⋆) → (S → ⋆) → ⋆ → ⋆
Cont S P X = (s : S) × (P s → X)
Cont′ : (S : ⋆) → (S → ⋆) → ⋆ → ⋆
Cont′ S P X = S × ((s : S) × P s → X)
List = Cont ℕ Fin   -- Example
```

where Fin n denotes a set with n elements.

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc : {n : ℕ} (i : Fin n) → Fin (suc n)
```

Figure 3.4: The definition of containers (Cont) has a top-level dependent pair. It can be embedded in a non-dependent version (Cont′).

There remains to show that such an embedding exists.

Theorem 11. $\forall S, P, X$, *given an arbitrary* $x : X$, *there exists an EP*

$$(e, p) : (s : S) \times (P\,s \to X) \subseteq S \times ((s : S) \times P\,s \to X)\,.$$

Proof. Note that the first component of each pair is the same: a shape. The second component is a function returning X in both cases, but the embedded type has a "smaller" domain. The idea is to use the same shape on both sides, embed the input function in the output function by returning the same elements when the shape coincides with that of the input. A default element is returned otherwise. This yields the definitions given in figure 3.5. Note that the embedding requires a decidable equality relation between shapes.

Showing that the projection is the left-inverse of the embedding is trivial, provided extensionality. □

Note that our proof provides a construction of the projection, so we have all the elements to generate arguments for testing.

B.1 Breaking containers, in practice

While the procedure described above is very general (it works for any container), it has two disadvantages:

- dependent pairs are introduced in the monomorphic instance (and only a few languages support dependent pairs);
- it can be complicated if one wishes to perform it by hand.

```
projection : ∀ {S P X} →
             Cont′ S P X → Cont S P X
projection (s, f) = (s, λ p → f (s, p))
getX : ∀ {S P X} → (def : X) → {s : S} →
       (f : (P s → X)) → (sp : (s′ : S) × P s′) →
       Dec (s ≡ fst sp) → X
getX def f (_, p) (yes refl) = f p
getX def f _      (no _)     = def
embedding : ∀ {S P X} → (def : X) →
            Cont S P X → Cont′ S P X
embedding def (s, f) =
  (s, λ sp → getX def f sp (s ≟ fst sp))
```

Figure 3.5: Embedding containers in a non-dependent pair. The Agda (Norell, 2007) code uses a type for decidable equality Dec with constructors yes and no, the equality proof type $\equiv$ with constructor refl and assumes S has a decidable equality $\stackrel{?}{=}$.

The first disadvantage can be overcome by using a position type which works for any shape (even if it is "too big"). This can be done by erasing the type index. In the case of lists, we have seen in section 3.2 that we can use $\mathbb{N}$ instead of $(n : \mathbb{N}) \times (\mathsf{Fin}\, n)$.

For the purpose of testing, one can often overcome the second disadvantage by short-circuiting the procedure entirely, as we have proposed in section 3.6.

We know that our method can treat arguments types as long as they can be embedded in $(F\, a \to a) \times (G\, a \to X)$. We do not have a complete syntactic characterisation of the types which possess such an embedding. However, we can formulate subsets which do possess such an embedding. Such a subset is any product of $\times_i (C_i\, a \to D_i\, a)$, where D_i is a container type with decidable equality on shapes and C_i is a functor. This subset is a refinement of that given in section 3.6.

To give some insight to the limitation of the method, we can also give types for which we failed to find a proper embedding. Notably, an argument of type $(a \to X) \to X$ is problematic: while it seems to fit the pattern $G\, a \to X$ with $G\, a = (a \to X)$, G is not a functor. Types of the form $(a \to X) \to X$ are created by continuation-passing style (CPS) transformations. We suspect that in general the method does not apply to functions expressed in CPS.

C Auxiliary results about free distributive lattices

We assume free distributive lattices with a finite number or generators.

Theorem 12. *Every term of a free distributive lattice can be transformed to the following normal form:*

$$\bigvee_{M_i \in J} \left(\bigwedge_{x \in M_i} x \right)$$

where there is no $i \neq j$ such that $M_i \subseteq M_j$.

Proof (sketch). Inner joins can be eliminated by the distributive law. Redundant meets can be removed by the absorption law. □

Definition 13. The *truth table* of term x, denoted $[\![x]\!]$, is a mapping of sets of variables to Boolean values. It can be computed by substituting each variable by 1 if it is in the set, 0 otherwise, meet and joins being interpreted as Boolean conjunction and disjunction.

$$\begin{aligned} [\![Xi]\!](S) &= i \in S \\ [\![x \wedge y]\!](S) &= [\![x]\!](S) \wedge [\![y]\!](S) \\ [\![x \vee y]\!](S) &= [\![x]\!](S) \vee [\![y]\!](S) \end{aligned}$$

Remark 14. *If*

$$e = \bigvee_{M \in J} \left(\bigwedge_{x \in M} x \right)$$

then $\forall M \in J. [\![e]\!]M = 1$.

Theorem 15. *If $[\![a]\!] = [\![b]\!]$, then $a = b$*

Proof. Let us prove the contrapositive, namely $a \neq b \Rightarrow [\![a]\!] \neq [\![b]\!]$. Assume (without loss of generality) that a and b are written in normal form, and let J_a and J_b be the respective sets of meets of a and b. If $a \neq b$ then either

- there exists $M_a \in J_a$ such that $M_a \notin J_b$, or
- there exists $M_b \in J_b$ such that $M_b \notin J_a$.

Let use examine the first alternative, knowing that the second can be handled symmetrically. We know that $[\![a]\!](M_a) = 1$. If $[\![b]\!](M_a) = 0$, we have a discrepancy in the truth tables. Assume then that $[\![b]\!](M_a) = 1$. Then,

there must be an $M_b \in J_b$ such that $M_b \subset M_a$. By definition of the normal form, $M_b \notin J_a$. Again, either there must either be a discrepancy in the truth tables, or we can repeat the argument with strictly smaller sets. Eventually, that option becomes unavailable: since there is a finite number of variables, the empty set is eventually reached. We conclude that there must be a set M such that $[\![a]\!](M) \neq [\![b]\!](M)$ □

References

Abadi, Martín, Luca Cardelli, and Pierre-Louis Curien (1993). "Formal parametric polymorphism". In: *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Charleston, South Carolina, United States: ACM, pp. 157–170. ISBN: 0-89791-560-7. DOI: 10.1145/158511.158622. See pp. 62, 80.

Abbott, Michael, Thorsten Altenkirch, and Neil Ghani (2003). "Categories of Containers". In: *Foundations of Software Science and Computation Structures*. Vol. 2620. Lecture Notes in Computer Science. Springer, Heidelberg, pp. 23–38. ISBN: 0302-9743. DOI: 10.1007/3-540-36576-1_2. See p. 120.

Arts, Thomas, John Hughes, Joakim Johansson, and Ulf Wiger (2006). "Testing telecoms software with quviq QuickCheck". In: *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*. Portland, Oregon, USA: ACM, pp. 2–10. ISBN: 1-59593-490-1. DOI: 10.1145/1159789.1159792. See p. 93.

Atkey, Robert, Patricia Johann, and Neil Ghani (2010). "When is a Type Refinement an Inductive Type?" In: *Foundations Of Software Science And Computational Structures*. Ed. by Martin Hofmann. Vol. 6604. Lecture Notes in Computer Science. Springer, pp. 72–87. See p. 83.

Bagge, Anya Helene, Valentin David, and Magne Haveraaen (2008). "Axiom-based testing for C++". In: *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. Nashville, TN, USA: ACM, pp. 721–722. ISBN: 978-1-60558-220-7. DOI: 10.1145/1449814.1449829. See p. 93.

Barendregt, Hendrik Pieter (1992). "Lambda calculi with types". In: *Handbook of logic in computer science* 2, 117–309. DOI: 10.1.1.26.4391. See pp. 6, 13–15, 53, 56, 61, 63, 66, 68, 87.

Berardi, Stepfano (1989). "Type Dependence and Constructive Mathematics". PhD thesis. Dipartimento di Informatica, Torino. See p. 61.

Bernardy, Jean-Philippe (2010). *Lightweight Free Theorems: Agda Library*. http://wiki.portal.chalmers.se/agda/agda.php?n=Libraries.LightweightFreeTheorems. See p. 41.

Bernardy, Jean-Philippe, Patrik Jansson, and Koen Claessen (2010). "Testing Polymorphic Properties". In: *European Symposium on Programming*. Ed. by Andrew Gordon. Vol. 6012. Lecture Notes in Computer Science. Springer, pp. 125–144. See pp. v, 62.

Bernardy, Jean-Philippe, Patrik Jansson, and Ross Paterson (2010). "Parametricity and Dependent Types". In: *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*. Baltimore, Maryland: ACM, pp. 345–356. See pp. v, 14, 49, 62, 75, 77, 80.

Bernardy, Jean-Philippe and Marc Lasson (2011). "Realizability and Parametricity in Pure Type Systems". In: *Foundations Of Software Science And Computational Structures*. Ed. by Martin Hofmann. Vol. 6604. Lecture Notes in Computer Science. Springer, pp. 108–122. See pp. v, 49, 50.

Bird, Richard and Oege de Moor (1997). *Algebra of programming*. Prentice-Hall, Inc. ISBN: 013507245X. See pp. 98, 101.

Brady, Edwin, Conor McBride, and James McKinna (2004). "Inductive Families Need Not Store Their Indices". In: *Types for Proofs and Programs*. Ed. by Stefano Berardi, Mario Coppo, and Ferruccio Damiani. Vol. 3085. Lecture Notes in Computer Science. Springer, pp. 115–129. DOI: 10.1007/978-3-540-24849-1_8. See p. 79.

Böhm, Corrado and Alessandro Berarducci (1985). "Automatic synthesis of typed Lambda-programs on term algebras". In: *Theoretical Computer Science* 39.2-3, pp. 135–154. See pp. 26, 102.

Böhme, Sascha (2007). "Free theorems for sublanguages of Haskell". Tool currently available (2010) at http://www-ps.iai.uni-bonn.de/cgi-bin/free-theorems-webui.cgi. Master's Thesis. Technische Universität Dresden. See p. 41.

Cardelli, Luca and Peter Wegner (1985). "On understanding types, data abstraction, and polymorphism". In: *ACM Computing Surveys* 17.4 (Dec. 1985), pp. 471–523. ISSN: 0360-0300. DOI: 10.1145/6041.6042. See p. 5.

Chlipala, Adam et al. (2009). "Effective interactive proofs for higher-order imperative programs". In: *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*. ICFP '09. Edinburgh, Scotland: ACM, pp. 79–90. ISBN: 978-1-60558-332-7. DOI: http://doi.acm.org/10.1145/1596550.1596565. See p. 6.

Christiansen, Jan and Sebastian Fischer (2008). "EasyCheck — Test Data for Free". In: *Functional and Logic Programming*. Vol. 4989. Lecture Notes in Computer Science. Springer, pp. 322–336. See p. 93.

Church, Alonzo (1940). "A formulation of the simple theory of types". In: *Journal of symbolic logic* 5.2, pp. 56–68. ISSN: 0022-4812. See p. 13.

Claessen, Koen and John Hughes (2000). "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. ACM, pp. 268–279. ISBN: 1-58113-202-6. DOI: 10.1145/351240.351266. See pp. 93, 115.

Coquand, Thierry (1986). "An Analysis of Girard's Paradox". In: *Logic in computer science*. IEEE Computer Society Press, pp. 227–236. See pp. 16, 77.

— (1992). "Pattern Matching with Dependent Types". In: *Proceedings of the Workshop on Types for Proofs and Programs*, pp. 66–79. See p. 29.

Coquand, Thierry and Gérard Huet (1986). "The calculus of constructions". PhD thesis. INRIA. See p. 6.

Danielsson, Nils Anders (2010). *The Agda standard library*. See pp. 82, 83.

Danielsson, Nils Anders, John Hughes, Patrik Jansson, and Jeremy Gibbons (2006). "Fast and loose reasoning is morally correct". In: *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 206–217. DOI: 10.1145/1111320.1111056. See p. 108.

Danvy, Olivier and Lasse R. Nielsen (2001). "Defunctionalization at work". In: *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*. Florence, Italy: ACM, pp. 162–174. ISBN: 1-58113-388-X. DOI: 10.1145/773184.773202. See p. 115.

Davey, Brian A. and Hilary A. Priestley (2002). *Introduction to lattices and order*. Cambridge University Press. ISBN: 0521784514, 9780521784511. See p. 113.

Day, Nancy A., John Launchbury, and Jeff Lewis (1999). "Logical abstractions in Haskell". In: *In Proceedings of the 1999 Haskell Workshop*. DOI: 10.1.1.37.2140. See p. 94.

Dybjer, Peter (1994). "Inductive families". In: *Formal Aspects of Computing* 6.4, pp. 440–465. DOI: 10.1007/BF01211308. See pp. 16, 27.

Fegaras, Leonidas and Tim Sheard (1996). "Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space)". In:

Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. St. Petersburg Beach, Florida, United States: ACM, pp. 284–294. ISBN: 0-89791-769-3. DOI: 10.1145/237721.237792. See pp. 102, 119.

Fokkinga, Maarten M. (1996). "Datatype Laws Without Signatures". In: *Mathematical Structures in Computer Science* 6.01, pp. 1–32. DOI: 10.1017/S0960129500000852. See p. 112.

Gibbons, Jeremy and Ross Paterson (2009). "Parametric datatype-genericity". In: *Proceedings of the 2009 ACM SIGPLAN workshop on Generic programming*. Edinburgh, Scotland: ACM, pp. 85–93. ISBN: 978-1-60558-510-9. DOI: 10.1145/1596614.1596626. See p. 46.

Gill, Andrew, John Launchbury, and Simon Peyton Jones (1993). "A short cut to deforestation". In: *Proceedings of the conference on Functional programming languages and computer architecture*. Copenhagen, Denmark: ACM, pp. 223–232. ISBN: 0-89791-595-X. DOI: 10.1145/165180.165214. See pp. 62, 114.

Girard, Jean-Yves (1972). "Interprétation fonctionnelle et elimination des coupures de l'arithmétique d'ordre supérieur". Thèse d'état. Université de Paris 7. See pp. 5, 7, 50, 61, 62, 65.

Harrop, Ronald (1956). "On disjunctions and existential statements in intuitionistic systems of logic". In: *Mathematische Annalen* 132.4, pp. 347–361. See p. 62.

Hoffman, Daniel, Jayakrishnan Nair, and Paul Strooper (1998). "Testing generic Ada packages with APE". In: *Ada Letters* XVIII.6, pp. 255–262. DOI: 10.1145/301687.289640. See p. 93.

Hofmann, Martin and Thomas Streicher (1996). "The Groupoid Interpretation of Type Theory". In: *Venice Festschrift*. Oxford University Press, pp. 83–111. See p. 48.

Hughes, John (2007). "QuickCheck Testing for Fun and Profit". In: *Practical Aspects of Declarative Languages*. Springer, pp. 1–32. See p. 8.

Jansson, Patrik, Johan Jeuring, and students of the Utrecht University Generic Programming class (2007). "Testing properties of generic functions". In: *Proceedings of IFL 2006*. Ed. by Zoltan Horvath. Vol. 4449. Lecture Notes in Computer Science. Springer, pp. 217–234. See p. 114.

Johann, Patricia and Janis Voigtländer (2006). "The Impact of seq on Free Theorems-based Program Transformations". In: *Fundamenta Informaticae* 69.1-2, pp. 63–102. See pp. 13, 49, 108.

Kleene, Stephen Cole (1945). "On the interpretation of intuitionistic number theory". In: *Journal of Symbolic Logic* 10.4, pp. 109–124. See p. 62.

— (1971). *Introduction to metamathematics*. Wolters-Noordhoff. See p. 62.

Knuth, Donald E. (1997). *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley Professional. ISBN: 0201896834. See p. 1.

— (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. 2nd ed. Addison-Wesley Professional. ISBN: 0201896850. See pp. 94, 114.

Kreisel, Georg (1959). "Interpretation of analysis by means of constructive functionals of finite types". In: *Constructivity in mathematics*. Ed. by A. Heyting. North-Holland, Amsterdam, pp. 101–128. See p. 62.

Krivine, Jean-Louis (1997). *Lambda-calcul – types et modèles*. Dunod. ISBN: 2225820910. See pp. 62, 72, 73, 80.

Krivine, Jean-Louis and Michel Parigot (1990). "Programming with proofs". In: *Journal of Information Processing and Cybernetics* 26.3, pp. 149–167. ISSN: 0863-0593. See p. 62.

Leivant, Daniel (1990). "Contracting proofs to programs". In: *Logic and Computer Science*, pp. 279–327. See pp. 62, 73, 81.

Leroy, Xavier (2009). "Formal verification of a realistic compiler". In: *Communications of the ACM* 52.7, pp. 107–115. See p. 6.

Mairson, Harry (1991). "Outline of a proof theory of parametricity". In: *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*. Vol. 523. Lecture Notes in Computer Science. Springer-Verlag, pp. 313–327. DOI: 10.1007/3540543961_15. See pp. 62, 80, 81.

Marlow, Simon (2010). "Haskell 2010 Language Report". http://haskell.org/definition/haskell2010.pdf. See p. 3.

Martin-Löf, Per (1984). *Intuitionistic type theory*. Bibliopolis. See pp. 2, 6.

Mcbride, Conor (2010). "Ornamental Algebras, Algebraic Ornaments". Manuscript available online. See p. 83.

McBride, Conor and James McKinna (2004). "The view from the left". In: *Journal of Functional Programming* 14.01, 69–111. See pp. 16, 61.

Metcalf, Michael and John Reid (1990). *Fortran 90 explained*. Oxford University Press. ISBN: 0198537727. See p. 3.

Milner, Robert, Mads Tofte, and Robert Harper (1990). *The definition of Standard ML*. MIT press. ISBN: 0262631296. See p. 3.

Milner, Robin (1972). "Logic for Computable Functions: description of a machine implementation." In: *Artificial Intelligence*. See p. 62.

Miquel, Alexandre (2001). "Le Calcul des Constructions implicite: syntaxe et sémantique". Thèse de doctorat. Université Paris 7. See pp. 16, 19, 77.

Monnier, Stefan and David Haguenauer (2010). "Singleton types here, singleton types there, singleton types everywhere". In: *Proceedings of the 4th ACM SIGPLAN workshop on Programming languages meets program verification*. Madrid, Spain: ACM, pp. 1–8. ISBN: 978-1-60558-890-2. DOI: 10.1145/1707790.1707792. See p. 50.

Morris, Peter and Thorsten Altenkirch (2009). "Indexed Containers". In: *Twenty-Fourth IEEE Symposium on Logic in Computer Science*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 277–285. DOI: http://doi.ieeecomputersociety.org/10.1109/LICS.2009.33. See p. 120.

Neis, Georg, Derek Dreyer, and Andreas Rossberg (2009). "Non-parametric parametricity". In: *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*. Edinburgh, Scotland: ACM, pp. 135–148. ISBN: 978-1-60558-332-7. DOI: 10.1145/1596550.1596572. See p. 13.

Nilsson, Rickard (2009). *ScalaCheck*. http://code.google.com/p/scalacheck/. July 2009. See p. 93.

Norell, Ulf (2007). "Towards a practical programming language based on dependent type theory". PhD Thesis. Chalmers Tekniska Högskola. See pp. 3, 6, 14, 16, 61, 77, 123.

Oury, Nicolas and Wouter Swierstra (2008). "The power of Pi". In: *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*. Victoria, BC, Canada: ACM, pp. 39–50. ISBN: 978-1-59593-919-7. DOI: 10.1145/1411204.1411213. See pp. 3, 51.

Paulin-Mohring, Christine (1989*a*). "Extracting Fω's programs from proofs in the calculus of constructions". In: *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Austin, Texas, United States: ACM, pp. 89–104. ISBN: 0-89791-294-2. DOI: http://doi.acm.org/10.1145/75277.75285. See pp. 62, 80, 81.

— (1989*b*). "Extraction de programmes dans le Calcul des Constructions". PhD thesis. Université Paris 7. See p. 62.

Paulin-Mohring, Christine (1993). "Inductive definitions in the system Coq – rules and properties". In: *Typed Lambda Calculi and Applications*. Springer, pp. 328–345. See pp. 16, 27.

Pierce, Benjamin C. (2002). *Types and Programming Languages*. 1st ed. The MIT Press. ISBN: 0-262-16209-1. See pp. 2, 4.

Plotkin, Gordon and Martín Abadi (1993). "A logic for parametric polymorphism". In: *Proceedings of the International Conference on Typed Lambda Calculi and Applications*. Vol. 664. Lecture Notes in Computer Science. Springer, 361–375. See pp. 48, 80.

Pottier, Francois and Nadji Gauthier (2006). "Polymorphic typed defunctionalization and concretization". In: *Higher-Order Symbol. Comput.* 19.1, pp. 125–162. See p. 115.

Reynolds, John C. (1974). "Towards a theory of type structure". In: *Colloque sur la Programmation*. Springer, pp. 408–425. See p. 5.

— (1983). "Types, abstraction and parametric polymorphism". In: *Information processing* 83.1, pp. 513–523. See pp. 5, 13, 19, 22, 26, 62, 71, 80.

— (1998). "Definitional Interpreters for Higher-Order Programming Languages". In: *Higher-Order and Symbolic Computation* 11.4, pp. 363–397. See p. 114.

Runciman, Colin, Matthew Naylor, and Fredrik Lindblad (2008). "Smallcheck and lazy smallcheck: automatic exhaustive testing for small values". In: *Proceedings of the first ACM SIGPLAN symposium on Haskell*. Victoria, BC, Canada: ACM, pp. 37–48. ISBN: 978-1-60558-064-7. DOI: 10.1145/1411286.1411292. See pp. 8, 116.

Saff, David (2007). "Theory-infected: or how i learned to stop worrying and love universal quantification". In: *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. Montreal, Quebec, Canada: ACM, pp. 846–847. ISBN: 978-1-59593-865-7. DOI: 10.1145/1297846.1297919. See p. 93.

Sheeran, Mary (2007). *Hardware Design and Functional Programming: a Perfect Match*. Talk at Hardware Design and Functional Languages. See p. 111.

Staples, John (1973). "Combinator realizability of constructive finite type analysis". In: *Cambridge Summer School in Mathematical Logic*, pp. 253–273. See p. 62.

Sun Tzu (2003). *The Art of War*. Penguin Classics. ISBN: 0140439196. See p. 1.

Svenningsson, Josef (2002). "Shortcut fusion for accumulating parameters & zip-like functions". In: *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*. Pittsburg PA, USA: ACM, pp. 124–132. DOI: 10.1145/583852.581491. See p. 117.

Takeuti, Izumi (2004). "The Theory of Parametricity in Lambda Cube". Manuscript. See pp. 13, 48.

The Coq development team (2010). *The Coq proof assistant*. See pp. 16, 61, 63, 77.

Tillmann, Nikolai and Wolfram Schulte (2005). "Parameterized unit tests". In: *SIGSOFT Software Engineering Notes* 30.5, pp. 253–262. DOI: 10.1145/1095430.1081749. See pp. 93, 116.

Troelstra, Anne Sjerp (1998). "Handbook of proof theory". In: ed. by Samuel R. Buss. Elsevier. Chap. Realizability. See p. 62.

Van Oosten, Jaap (2002). "Realizability: a historical essay". In: *Mathematical Structures in Computer Science* 12.03, pp. 239–263. See p. 62.

Voigtländer, Janis (2008). "Much ado about two (pearl): a pearl on parallel prefix computation". In: *SIGPLAN Not.* 43.1, pp. 29–35. DOI: 10.1145/1328897.1328445. See pp. 94, 111.

— (2009*a*). "Bidirectionalization for free! (Pearl)". In: *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Savannah, GA, USA: ACM, pp. 165–176. ISBN: 978-1-60558-379-2. DOI: 10.1145/1480881.1480904. See p. 108.

— (2009*b*). "Free theorems involving type constructor classes: Functional pearl". In: *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*. Edinburgh, Scotland: ACM, pp. 173–184. ISBN: 978-1-60558-332-7. DOI: 10.1145/1631687.1596577. See pp. 24, 45.

Vytiniotis, Dimitrios and Stephanie Weirich (2010). "Parametricity, Type Equality, and Higher-Order Polymorphism". In: *Journal of Functional Programming* 20.02, pp. 175–210. DOI: 10.1017/S0956796810000079. See pp. 13, 48.

Wadler, Philip (1989). "Theorems for free!" In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture*. Imperial College, London, United Kingdom: ACM, pp. 347–359.

ISBN: 0-89791-328-0. DOI: 10.1145/99370.99404. See pp. 5, 13, 26, 30, 45, 62, 101.

— (2007). "The Girard–Reynolds isomorphism". In: *Theoretical Computer Science* 375.1–3, 201–226. See pp. 7, 26, 49, 50, 62, 67, 73, 80, 81.

Wadler, Philip and Stephen Blott (1989). "How to make ad-hoc polymorphism less ad hoc". In: *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, pp. 60–76. ISBN: 0897912942. DOI: 10.1145/75277.75283. See p. 45.

Washburn, Geoffrey and Stephanie Weirich (2003). "Boxes go bananas: encoding higher-order abstract syntax with parametric polymorphism". In: *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*. Uppsala, Sweden: ACM, pp. 249–262. ISBN: 1-58113-756-7. DOI: 10.1145/944705.944728. See p. 118.

Werner, Benjamin (1994). "Une théorie des constructions inductives". PhD Thesis. Université de Paris 7. See p. 6.