Abstraction Barriers and Refinement in the Polymorphic Lambda Calculus

Abstraction Barriers and Refinement in the Polymorphic Lambda Calculus

Jo Erskine Hannay

Doctor of Philosophy
2001

Laboratory for Foundations of Computer Science
Division of Informatics
University of Edinburgh
Scotland, United Kingdom.

This thesis is written in $\LaTeX 2_\varepsilon$ using the `report` class together with elements from the Edinburgh University `csthesis` class. The font is Computer Modern 12pt, but scaled down in the booklet format of this thesis.

To Muffin and Tufa

This is a thesis for the degree of Doctor of Philosophy (PhD) in the discipline of Theoretical Computer Science, done at the Laboratory for Foundations of Computer Science within the Division of Informatics at the University of Edinburgh. I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text. The material mentioned in Sect. 2.8, and several of the results in Ch. 4, Ch. 5, and Ch. 6 have been published in earlier formulations by the author in, respectively, (Hannay, 1998, 1999a, 2000, 1999b). The work in Sect. 6.2 and Sect. 6.3 is joint with Martin Hofmann.

Edinburgh, June 2000 Jo Hannay

Synopsis

The use of safety-critical software relies crucially on the certified verification relative to an unambiguous specification that the software will perform correctly, even on the first occasion it is put to use. One way of achieving unambiguity and assurance is to use mathematical or formal methods in the development process of such software, so that the result is software that is mathematically proven correct relative to a formal specification.

The concept of algebraic specification refinement embodies a framework in which to develop software formally. In this thesis, this concept is expressed in a different, and in many respects, richer mathematical framework, namely that of type theory, polymorphic lambda-calculus, and an intuitionistic logic extended to assert relational parametricity, as well as key features of specification refinement. This translation into a different framework enables important extensions to the concept of algebraic specification refinement that enhance the proving power, the applicability, and hence the usefulness of the concept.

Abstract

This thesis examines specification refinement in the setting of polymorphic type theory and a complementary logic for relational parametricity.

The starting point is the specification of abstract data types as done in the discipline of algebraic specification. Here, algebras are seen to match the standard notion of data type, *i.e.,* a data representation together with operations on that data representation. An abstract data type is then a collection of data types sharing some well-defined abstract properties. In algebraic specification, these properties are specified algebraically by axioms in some suitable logic. Specification refinement then encompasses the idea that high-level specifications may be stepwise refined to executable programs that satisfy the initial specification; all in the framework of formal language and logic. This makes certain aspects of program development amenable to formal, computer-aided proofs of correctness.

On the other hand, the discipline of type theory, lambda calculus, and its semantics is the prime field for research on programming languages. This framework is capable of characterising essentially any existing sequential programming-language feature, also advanced features such as recursive types, polymorphism and class-based object orientation. Furthermore, type theory provides a powerful framework for mechanised reasoning.

This thesis is a contribution to lifting the idea of algebraic specification refinement into the more powerful domain of type theory and lambda calculus, thus giving the opportunity to expand in a sensible way a traditionally first order and functional framework to a wider range of programming aspects.

We take a particular account of specification refinement and express it in a type-theoretic setting consisting of the polymorphic lambda calculus and a logic for relational parametricity. Key elements of algebraic specification are internalised in the syntax, *e.g.,* data types *viz.* algebras are inhabitants of existential type, the latter providing essential data abstraction. For data types with only first-order operations, this setting automatically resolves certain issues of specification refinement, such as observational equivalence, stability and input sorts.

After establishing a correspondence at first order, thus implanting the idea of algebraic specification refinement into the type-theoretic setting, the scene is set for lifting the idea of algebraic specification refinement to any number of programming features. In this thesis we focus on the generalisations to higher-order functions and to polymorphism.

A simulation relation between two data types is a relation between their data representations that is preserved by their respective sets of operations. Using sim-

ulation relations is a classical way of explaining data refinement and observational equivalence. This combines with specification refinement to form specification refinement up to observational equivalence. With higher-order operations, however, we encounter in the logic a phenomenon related to what happens on the semantic level, *i.e.,* the standard notion of refinement relation in the form of logical relations does not compose and the correspondence with observational equivalence is lost. In the logic it turns out that the standard notion of simulation relation fails to take into account a certain aspect of the abstraction barrier provided by existential types. We remedy this by proposing an alternative notion of simulation relation that observes this abstraction barrier more closely. We do this in two related ways; one relates to syntactic models while the other relates to a non-syntactic PER-model more apt for interpretive investigations.

In algebraic specification, there is a universal proof method for specification refinement up to observational equivalence. This method can be imported soundly into the type-theoretic setting by asserting certain axioms. At first order, showing soundness for these axioms is straight-forward w.r.t. the standard parametric PER model for the logic. At higher order there are two problems. First, these axioms seemingly do not hold in the standard model. Secondly, the axioms speak in terms of simulation relations. At higher order, it is pertinent to have versions of the axioms featuring the abstraction barrier-observing simulation relations above, and to prove soundness for these poses an additional challenge. We show that the pure higher-order aspect of this problem can be solved by giving a setoid-based semantics. For the remaining task, we continue working from the observation that standard definitions do not observe abstraction barriers closely enough. Hence, we propose an alternative interpretation into the PER-model for data types that captures the abstraction barrier provided by existential types.

The main contribution of this thesis is thus in generalising a prominent account of specification refinement to higher order and polymorphism via type theory incorporating relational parametricity. We also shed light on short-comings in the logic, as well as in the standard semantics, regarding the abstraction barrier provided by existential types. Two central contributions, namely abstraction barrier-observing simulation relations and abstraction barrier-observing semantics for data types, are the result of observing these short-comings. Finally, the work in this thesis also lays a foundation on which to adapt specification refinement to an object-oriented setting, because the theoretical concepts underlying object orientation can be seen as extensions of those for abstract data types.

Acknowledgments

I thank my supervisors Don Sannella and Martin Hofmann who have always been ready to give of their phenomenal theoretical understanding, academic knowledge, and experience. Their ideas and comments on especially later drafts of this thesis greatly improved the end result. If I have one regret, it is that I did not make nearly enough use of their expertise as I could have. I thank my supervisors also for their moral support and help in administrative matters.

Other people have also brought encouragement and have helped with theoretical matters. John Longley clarified a detail concerning the abstraction barrier-observing PER semantics. I thank Furio Honsell for his encouragement and for his interest in the abstraction barrier-observing simulation relation presented in this thesis. Thanks to David Aspinall, Yoshiki Kinoshita, Gordon Plotkin, Erik Poll, John Power, Uday Reddy, John Reynolds, Perdita Stevens, and Martin Wehr for encouragement and guidance. Another group of people that deserve credit, are the anonymous referees for the four papers that give rise to many parts of this thesis. Their thorough comments were very helpful.

I am grateful to Michał Walicki and Sigurd Meldal whose integrity and advice helped me come to Edinburgh, although I was originally meant to work for them in Norway. Together with Olav Lysne and Ole-Johan Dahl at Universitetet i Oslo, they provided excellent references.

This work was funded for three years by Norges Forskningsråd grant 110904/41. A five-month extension came off Don's EPSRC grant GR/K63795, and additional funding was kindly given by Simula Research Laboratory with the help of Dag Sjøberg and Morten Dæhlen.

I met many inspiring people while doing the work presented in this thesis. My office mates Patricia and Carsten, have both been inspirational, helpful and have helped keep a human face on a potentially rather dire form of research. Thanks to Isabel for taking me in as a sleepless tenant, and to Svein Olav who rescued my social life in Edinburgh. Many others remained my friends and also helped me out via email with questions I had. A heartfelt thanks to Marit.

A special thanks to Britta who endured much moose-facedness during the final stages. Also, a very special thanks to my parents Joanne and Alastair for their support in every way.

Contents

It's all very well in practice, but it will never work in theory.
French Proverb

Chapter 1

Introduction

A computer lets you make more mistakes faster than any invention in human history - with the possible exceptions of handguns and tequila.
MITCH RATCLIFFE, *Technology Review (1992)*

1.1 Critical Software

The majority of software that one utilises daily is likely to work quite satisfactorily and within bounds of most people's tolerance and patience. Certainly, software of any substantial format is guaranteed to have bugs and ill-designed features that make a system somewhat unreliable and occasionally crash. Mostly though, such "hidden features" are merely annoying and are simply taken as a fact of life.

There is on the other hand a class of software that may under no circumstance malfunction, because failure may result in death, injury or material loss, see *e.g.,* (Leveson and Turner, 1993; SIAM, 1996; Risks Digest, 2000). Let us call such software *critical*, in contrast to the non-critical *utility* software described above. We are of course relying increasingly on critical software, because many critical tasks cannot viably be done without computers, and because the trend is to let software take over ever more tasks traditionally handled by mechanical devices, on the grounds that this gives better economy and flexibility.

Software development practices that generally underlie the production of utility software, do not suffice for developing critical software. This is understandable, because both the end-product and the development goals for the two types of software have completely different priorities attached to them.

On a more general level, software development has for a long time demonstrated serious short-comings in a different respect as well. Although the consequences are not directly life-threatening, large software development projects famously over-run budgets and time limits massively, and may ultimately turn out to be not what the contractor really wanted, see *e.g.,* (Gibbs, 1994). Moreover, the structure of both development process and product are often such that evolutionary changes are next to impossible, resulting in the system quickly becoming obsolete, perhaps even before a working version is up and running.

1.2 A Development Framework

Much work has gone into resolving the issue of producing safe critical software. Some years ago it could seem that formal methods would eventually be able to prove all programs correct. This of course has not happened yet, and there has been a certain disappointment over formal methods not delivering according to the conceived promises. This was probably for a large part due to over-selling the potential of formal methods and miscommunication between various communities, a destiny shared by countless other ideas and paradigms in computer science and computer technology. But as is also common, once the dust clears, a more realistic picture emerges of how to use an idea or paradigm.

Formal methods now enjoy renewed interest and applicability in industry. This is due to the continued development of better formal methods and tools, but also to the realisation that the development of correct software will not be solved by one hot all-encompassing programming paradigm, powerful programming language, magical CASE-tool, or theoretically brilliant formal method alone. Instead a prevailing view is that the main task at hand is to build a framework of acknowledged methods with sound theoretical foundations. Project development and management is not resolved by the underlying mathematical theory alone, nor by methodologies and practices alone, but by an interaction of the two, and by the existence of a variety of methods and ways of recognising which methods make sense in which situations together with tools for applying those methods.

A particular view on this that is shared by many, but disputed in part by others, is that the art of software development should be an *engineering discipline*, *i.e.,* software development should have a sound formal mathematical basis, and rigorously proven or tested methodologies, modelled on other established engineering disciplines. While the direct analogy is compelling, it is perhaps more than anything else suited to highlight the *deficiencies* in early and still prevailing software development practices that lack any significant scientific basis.

That software development should and can be supported by formal technology pertains not only to critical software, but also to software development in general. The range of formal techniques is large, and different technologies can be applied for different development purposes as suited. The formal-methods web site `http://www.comlab.ox.ac.uk/archive/formal-methods.html` gives comprehensive references. The Common Framework Initiative (CoFI) seeks to unify trends in algebraic specification techniques, thus working towards presenting a coherent framework to industry, see `http://www.brics.dk/Projects/CoFI`.

—

The difference between theory and practice is bigger in practice than it is in theory. Unknown

1.3 Abstract Data Types, Object Orientation, and Components

Object orientation and abstract data type-oriented programming were two of the programming paradigms that were supposed to resolve the software crisis. Again this did not happen, but it is now widely accepted that object-orientation and abstract data types implement crucial prerequisites for successful development.

Classes, objects and modules were originally programming language concepts that have lately not only been adopted as design concepts in the large, but also integrated as software development standards, by *e.g.,* the Object Management Group (OMG), and integrated in modelling languages such as the *Unified Modelling Language* (UML), now part of OMGs standard. Links to OMG and UML web-pages are `http://www.omg.org` and `http://www.omg.org/uml`. See also *e.g.,* (Pooley and Stevens, 1999) for an introduction to using UML, and also *e.g.,* (Precise UML, 2000) for further developments on UML itself.

A significant feature of UML is that it provides hooks to formal methods. For example, one can specify method behaviour with pre-, and post-conditions. On lower levels of development, this thereby employs the triples of Hoare-logic

(Hoare, 1969, 1972). This is facilitated using the special-purpose *Object Constraint language* (OCL) designed to interface with UML. Another example is the feature to specify class invariants. This employs formal methods developed years ago and described comprehensively in (Dahl, 1992). It is here also worth mentioning the formal description language *Oslo University Notation* (OUN) (Owe and Ryl, 1999; Traore et al., 1999).

Both object orientation and abstract data types promote a module concept that provides a notion of interface for users of a module, giving the crucial notion of *information hiding*. This is a somewhat diffuse notion, but the concept is summarised nicely in (Pooley and Stevens, 1999):

> *Information hiding* consists of 'abstraction' and 'encapsulation'.
>
> *Abstraction* is when a client of a module doesn't need to know more than what is in the interface.
>
> *Encapsulation* is when a client of a module isn't able to know more than what is in the interface.

An interface provides information hiding by raising an appropriate *abstraction barrier*, the nature of which varies according to the level of development one is considering. Abstraction barriers constitute the key notion in this thesis.

Abstraction gives *high cohesion*, *i.e.,* unnecessary details of implementation are hidden from the user, leaving an uncluttered view of *what* a module does rather than *how* it does it. Encapsulation gives *low coupling*, *i.e.,* low interdependence of modules so that changes to one module are less likely to propagate the necessity for changes elsewhere in the system. High cohesion and low coupling are generally regarded as prerequisites for a good system. High cohesion and low coupling enables *component-based development* (CBD) or *component-oriented programming* (COP), which advocates development and programming with pluggable components, a component being a unit of reuse and replacement.

What exactly constitutes a component, not to mention a *standard* component, is under intense discussion at the moment. This highly relevant issue of standardisation is one of the problems haunting the re-use of software in general, and component-based critical software development in particular.

The problem we are concerned with however, pertains to developing guaranteed components, whatever the notion of 'component' might be. Component-based development is greatly facilitated by having components carrying a precise specification of what they do, and some sort of guarantee that they actually fulfil this specification. For critical software this is absolutely crucial, and the sought-after scenario is to build critical systems with off-the-shelf components that are certified to appropriate safety standards. Re-use of *guaranteed* components is essential for critical software, because this minimises risk and enhances maintainability. Ironically, at the moment there is very little re-use in this particular application domain that would seem to need it the most. This is due to the difficulty of substantiating the guarantees one would like to attach to critical software components, as well as the standardisation issues above.

Guaranteed component-based design also makes sense economically. A company selling a safety-critical component several times over, will be able to justify the added overhead of developing the component to a required safety standard. Note that a much pushed argument for using formal methods is that formal methods allow semi-mechanised proof, thereby facilitating the added work of certifying components and as a result cutting the cost. Although this argument has again been over-sold, it remains true on a reasonable level. Indeed, newly developed technology is making computer-aided reasoning increasingly more feasible.

—

If knowledge can create problems, it is not through ignorance that we can solve them. ISAAC ASIMOV

1.4 The Theoretical Foundations

This thesis works on theoretical foundations relevant to developing guaranteed components according to a specification. Specifically, we work towards making relevant parts of abstract data type development applicable to more powerful proof methods and tools. In this thesis, we do not address object-oriented technology directly. Nonetheless, object orientation can be seen as based on formalisms for abstract data types (Reddy, 1999).

1.4.1 Specification Refinement

A data type essentially consists of a data representation together with operations on that data representation. An abstract data type is a collection of data types. An interface provides information hiding, and two data types are instances of the same abstract data type if they provide the same interface, *i.e.,* they are considered equivalent up to their abstract properties. Data types, are hence just like algebras from universal algebra, and then abstract data types are classes of algebras determined by given abstract properties. This is the basic observation for the field of *algebraic specification*. The primary interface is provided by a signature guaranteeing the existence of, and giving certain syntactic information about the operations which all algebras of that signature must have. Additionally, one has *specifications* providing an additional secondary interface giving more information about what the operations do. Specifications use various logics to give axioms describing the operations. There can be many essentially different algebras satisfying the axioms, and so the information provided by the specification is abstract, or the axioms may fully specify the behaviour of the operations, in which case there is in essence only one model for the specification. On top of this, one usually declares *behavioural equivalence* as a further abstraction mechanism. This is the idea that if the behaviours of two data types are indistinguishable for users of the data types, then they indeed share the same abstract properties, regardless of differences in internal implementation that may imply that they do not fulfil the same axioms. This sensibly extends the concept of abstract property. An important instance of behavioural equivalence is *observational equivalence*, where equal behaviour is defined on the basis of *observable computations* on the algebra.

A framework in algebraic specification and other specification paradigms that has had a certain amount of success and enjoys particular applicability is that of *stepwise specification refinement*, *e.g.,* (Sannella and Tarlecki, 1997, 1988b, 1987), (Back and Wright, 1998), (Morgan, 1994), (Hoffmann and Krieg-Brückner, 1993). The idea is that a program is the end-product of a stepwise refinement process starting from an abstract high-level specification. At each refinement step, some design decisions and implementation issues are resolved, and if each refinement step is proven correct, the resulting program is guaranteed to satisfy the initial specification. A stepwise approach is more manageable than one massive development step, and also facilitates iterative development. This formal methodology for software development is supported *e.g.,* by the specification language Extended ML (EML) for Standard ML (SML) (Paulson, 1996) result programs.

Specifications and specification refinement belong on the implementation level of software development. With specification refinement one can develop specification-correct abstract data types where properties that are expressible in the relevant logic are guaranteed. This method can at present only help certify certain highly specific parts of the guarantee one would like to associate with a component, but it still provides substantial benefits to the overall verification process.

Formidable research has been done in the field of algebraic specification, see (Cerioli et al., 1997). In addition, the spirit of specification refinement has been adopted or kindled by disciplines other than algebraic specification, see (Engelhardt and de Roever, 1998). In many instances this lifts the initial ideas to a more powerful and hence more usable technology. This thesis follows this trend.

1.4.2 Computer-aided System Development

Software engineering has a characteristic that makes it unique amongst engineering disciplines: Since programming languages are formal languages, the product delivered, namely a piece of software, belongs technically in the same realm as the discipline's theoretical foundations, namely formal mathematics of one sort or another. This means that in theory, the formal description and model of the planned system, can be more or less seamlessly transformed into the planned system itself. Wide-spectrum specification languages such as Extended ML (Kahrs et al., 1997, 1994; Sannella, 1991), the language of the Refinement Calculus, (Morgan, 1994; Back and Wright, 1998), ABEL (Dahl and Owe, 1991, 1995; Dahl and Kristoffersen, 1995), the languages in the PROSPECTRA project (Hoffmann and Krieg-Brückner, 1993; Krieg-Brückner et al., 1991; Krieg-Brückner, 1990), and CIP-L (Bauer et al., 1981; CIP-L, 1985), have the target programming language embedded in them. Then through stepwise refinement, the abstract specification gets transformed into a concrete executable program.

The relatedness of theoretical foundation and end-product facilitates mechanical reasoning, *i.e.,* computer-aided development on the certification aspect. Mechanical aid is an absolute necessity, in order to certify critical software components of any substantial size. What can be mechanically proven is bounded by what can be formalised, so there is no promise yet of automatic systems development. However, critical properties of specialised components may benefit substantially from mechanised reasoning. It is also the case that state-of-the-art reasoning tools are getting ever more powerful and user-friendly, now incorporating an arsenal of heuristics as well.

1.4.3 Lambda Calculus and Type Theory

For the above scenario of specification refinement to work, certain criteria must be fulfilled, at least to some degree. Ideally,

1. the programming languages used should have a formal semantics
2. the specification formalism should be expressive enough
3. there should be a well-defined refinement relation
4. there should be a well-defined notion of abstraction
5. there should exist computer-aided tools for formal reasoning

As of yet, (1) is fulfilled by almost no commercial programming languages. Important exceptions are Standard ML (Milner et al., 1997), and ongoing efforts on Java (Nipkow et al., 2000; Oheimb and Nipkow, 1999; Coglio et al., 2000; Qian, 1999; Drossopoulou et al., 1999; Attali et al., 1998, 2000; Java Semantics, 2000). The other points are fulfilled for various frameworks. However, (1) is fulfilled for a wide range of lambda-calculi, thus almost all features found in commercial sequential programming languages are given a precise semantics via (typed) lambda calculus (Landin, 1965, 1966). Lambda calculus and type theory thus play the rôle of the most important research object on which to investigate features of "real" programming languages, imperative, functional or object oriented. Moreover, through type theory one gets access to a range of highly-developed proof assistants, *e.g.,* Coq (Coq, 2000), Isabelle (Isabelle, 2000) and LEGO (LEGO, 2000), see also (Proof General, 2000), for formal semi-automated reasoning.

1.4.4 Specification Refinement Generalised

The notions and the main body of research in algebraic specification refinement have been intrinsically *first-order*. There is a desire to transfer the successful concept of specification refinement and its theoretical rigour to a wider class of language principles, and to go beyond the first-order boundaries inherent in the universal algebra approach. There have been several good attempts at amending the formalisms of algebraic specification to deal with higher-order functions (Meinke, 1992; Kirchner and Mosses, 1998), and many other features, see (Cerioli et al., 1997). But the resulting formalisms are often difficult. In comparison, type-theoretic formalisms express higher-order functions and more advanced features of programming languages with ease and uniformity.

Our main motivation for the work in this thesis lies in the obvious opportunity to lift the idea of specification refinement to a wide range of programming features through the power of type theory and lambda calculus.

1.4.5 Polymorphism and Relational Parametricity

More specifically, we will express the particular account of algebraic specification refinement due to (Sannella and Tarlecki, 1997, 1988b,a, 1987; Sannella et al., 1992; Sannella and Wirsing, 1983) in a type-theoretic setting consisting of the *polymorphic lambda calculus* (Girard, 1971; Reynolds, 1974) and a logic for *relational parametricity* (Plotkin and Abadi, 1993). In this setting we will specifically deal with higher-order functions, and also *polymorphic* functions, *i.e.,* functions that capture a uniform behaviour across all types.

Polymorphism turns out to be extremely powerful. One can encode various semantic notions in syntax by using polymorphism (Mitchell and Plotkin, 1988), (Böhm and Berarducci, 1985), (Pierce et al., 1989). One gets syntactic representations for abstract data types, algebras and specification refinement. With the assumption of relational parametricity, (Reynolds, 1983; Ma and Reynolds, 1991), these constructs become universal constructs in the sense of category theory, *i.e.,* the encodings mirror exactly their intended meanings. Internalising semantic concepts into type theory is highly beneficial, because as mentioned above, one then has a multitude of semi-automated reasoning tools available.

1.5 The Main Goals

The overall goal of the research herein is to contribute to the formal underpinnings of component-based software development, by lifting the concept of algebraic specification refinement to higher-order and polymorphism via type theory. We now provide a more specific overview. It will be helpful to consult fig. 1.1.

1.5.1 A Simplification at First Order

There are several ways of expressing specifications and specification refinement in type theory. First of all, we choose the formulation of data types as inhabitants of existential types (Mitchell and Plotkin, 1988). Existential types provide information hiding in an elegant way. Then we straightforwardly form specifications and express specification refinement in terms of existential types. The way of writing specifications has a slight resemblance to a form suggested in (Luo, 1993), although the type theory in question is a different one. An important feature in our setting is that for data types with only first-order operations, the additional assumption of relational parametricity lifts the straightforward forms for specification and specification refinement to observational equivalence. Thus virtually without doing anything, we automatically get observational specifications and specification refinement up to observational equivalence. This is because the assumption of relational parametricity entails that data types are equal exactly when they are observationally equivalent. This also means that constructors, or parameterised programs in the sense of (Goguen, 1984; Schoett, 1986; Sannella and Tarlecki, 1997) are inherently *stable*, *i.e.,* preserve observational equivalence. Thus, toilsome issues in algebraic specification become trivialities in this type-theoretic setting. An additional example is that the information-hiding abstraction barrier provided by existential types also automatically singles out a sensible choice of *input types* corresponding to *input sorts* in algebraic specification. Input sorts determine parameters to observable computations, which again define observational equivalence.

1.5.2 A Correspondence at First Order

It seems then, that the type-theoretic setting incorporating relational parametricity has something to offer algebraic specification. Prior to further development, however, the two notions of specification refinement, *i.e.,* the type-theoretic notion and the originating one from algebraic specification, are shown to coincide in basic terms, for data types with only first-order operations. Thus the type-theoretic account of specification refinement has algebraic specification refinement embedded. This lays the foundation for lifting specification refinement to new concepts and features.

1.5.3 Simulation Relations

Above we said that one must have a well-defined notion of abstraction. That is, one must make precise when one module can replace another in a program context. This directly relates to component-based development (CBD) mentioned earlier. Currently the favoured view is that two modules are interchangeable if they are behaviourally equivalent. As mentioned previously, one way to define this technically is by observational equivalence, *i.e.,* by defining a set of designated observable computations (returning printable values, say), and defining two modules to be behaviourally equivalent, if all observable computations return equal results regardless of which module one plugs into the computation.

In general it is hard to show observational equivalence. For development purposes, it is often better to use some notion of *refinement relation*. In particular, a *simulation relation* is a relation between the data representations of two data types, such that their corresponding operations preserve the relation. At first order, one can define simulation relations that correspond to observational equivalence, but for data types with higher-order functions this becomes difficult and not in general possible. We will develop an alternative notion of simulation relation in the logic that works at arbitrary order. The main ingredient is a weakened arrow-type relation, that is motivated by the abstraction barrier inherent in existential types. On the semantic level, alternative notions of refinement relation have recently been studied in (Honsell et al., 2000; Honsell and Sannella, 1999; Kinoshita and Power, 1999; Kinoshita et al., 1997; Plotkin et al., 2000), and our work provides in spirit, a syntactic counterpart to these and on-going studies.

However, there is a conceptual difference in our approach. The semantic alternative notions of refinement relation incorporate lambda definability in some way or another. We do that as well, but instead of absolute definability, we assert a looser definability w.r.t. virtual operations in abstract data types. A more thorough comparison between semantic notions and our syntactic notion of alternative simulation relation is left as future research.

1.5.4 A Proof Method for Observational Refinement

There is a universal proof method for showing specification refinement up to observational equivalence that has been formalised in (Bidoit et al., 1997; Bidoit and Hennicker, 1996; Bidoit et al., 1995) in the context of algebraic specification. This proof method cannot be expressed in the type-theoretic setting and logic of (Plotkin and Abadi, 1993). The method can however be imported by soundly asserting certain axioms postulating the existence of quotients and subobjects. This proof method is also instrumental in showing the above correspondence of refinement in type theory to the notion in algebraic specification.

At first order, soundness for these axioms for quotients and subobjects can be verified w.r.t. the parametric PER-model (Bainbridge et al., 1990), one of the more interesting models for Plotkin and Abadi's logic. At higher order, we are here able to show soundness for the axioms w.r.t. a parametric setoid semantics.

It is also necessary to consider versions of these axioms incorporating the alternative notion of simulation relation. By observing that the abstraction barrier present in the type theory and lambda calculus through existential types, is again not adequately reflected, this time in the semantics, one can define an interpretation specifically for data types that does incorporate the abstraction barrier. This semantics then shows soundness for the axioms incorporating the alternative notion of simulation relation. In this, the parametric PER-model is kept as structure, but the interpretation map is modified.

1.5.5 Related Work

The work in this thesis relates to other work in several ways. First of all, we build on the work of others, *e.g.,* we use the account of algebraic specification mainly due to Sannella and Tarlecki, and map it into the type-theoretic setting consisting of Girard and Reynold's System F and Plotkin and Abadi's logic for parametric polymorphism. We also import a generic proof strategy formalised by Bidoit, Hennicker and Wirsing into the type-theoretic setting.

Secondly, the idea of expressing principles of algebraic specification in type theory is not new; we give references to other work later on. The novelty in this thesis is to formally map specification refinement from algebraic specification into the indicated specific type-theoretic setting, and then to generalise to higher-order and polymorphism. The work closest related to ours is probably that of (Poll and Zwanenburg, 1999; Zwanenburg, 1999) which demonstrates a principle of data refinement for first-order signatures in the same type-theoretical setting that we consider. It is from here we have the idea of extending the logic with axioms postulating the existence of quotients and subobjects, in order to mirror Bidoit *et al*'s proof strategy in the type theory. Our work differs in that our discussion is based on well-established concepts and strategies from the field of algebraic specification, that we do a formal translation into the type-theoretic setting, and that we use this setting to generalise to higher order and polymorphic signatures. The axiom schema for subobjects appeared in (Hannay, 1999a), and then independently in (Zwanenburg, 1999) in a similar version. The setoid semantics in this thesis settles the question posed in (Zwanenburg, 1999) regarding the soundness of asserting higher-order functions over subobjects and quotients.

Thirdly, our discussion concerning the alternative notion of simulation relation in the logic relates to ongoing work on the semantic level, in particular perhaps, to the pre-logical relations of Honsell and Sannella. Finally, all related work known to us is indicated at relevant places in the main discussion.

—

Foolproof systems do not take into account the ingenuity of fools.
GENE BROWN

1.6 Structure

We have aimed for this thesis to be somewhat self-contained, but compromises have been made. We hope the result is reasonably understandable.

Chapter 2 reviews the basics of algebraic specification, highlighting the concepts of constructor implementation, observational equivalence and stability. The originating notions in this chapter serve as a stepping stone and motivation for the development in type theory in chapters to come.

Chapter 3 gives a fuller account of typed lambda calculi. Starting from the simply-typed lambda calculus, we go via inductive type definition and on to the polymorphic lambda calculus. We review how abstract types are expressed as existential types, and highlight a crucial aspect of the information-hiding abstraction barrier provided by existential types. This crucial aspect is the cornerstone of the later discussion and development.

Chapter 4 presents notions of specification and specification refinement in the type theory and logic. We demonstrate how certain issues in algebraic specification are trivially resolved in the type-theoretic setting at first order. We then define a translation mapping algebraic specification refinement to the type-theoretic notion, and we show an exact correspondence between the two notions for basic specifications. The universal method formalised by Bidoit, Hennicker and Wirsing for proving observational refinements is imported soundly into the type-theoretic setting, by including axioms asserting the existence of quotients and subobjects. Soundness for the axioms is easily shown w.r.t. the parametric PER-model. It is indicated how to do this in Ch. 6 in the validation of more general versions of these axioms.

Chapter 5 takes on the problems arising when introducing higher-order operations in data types. The standard notion of simulation relation is seen to ignore the crucial aspect of the existential-type abstraction barrier. Thus at higher order, the composability of simulation relations is lost, and the coincidence of observational equivalence with the existence of a simulation relation is no longer robust. Both problems are solved by developing an alternative notion of simulation relation that respect the crucial aspect of the existential-type abstraction barrier. We also show the stability of System F constructors independently of any connection between observational equivalence and equality at existential type.

Chapter 6 generalises the universal method for proving observational refinement, to higher order. New versions of the axioms asserting the existence of quotients and subobjects are used that utilise the alternative notion of simulation relation developed earlier. However, the higher-order versions of these axioms do seemingly not hold in the parametric PER-model, so we must look for alternatives for showing the soundness of the logic augmented by the higher-order versions of these axioms. An interesting solution is to modify the standard interpretation into the parametric PER-model. The basis for this modification is again that the standard formalism does not adequately reflect all aspects of the existential-type abstraction barrier. In order to know when to apply the resulting special *data type semantics*, we introduce a method of annotating types for specific use for data type operations. We also consider a *setoid-semantics* based on work in (Hofmann, 1995a). The resulting model proves the soundness at higher order of the original axioms in Ch. 4. Although this model cannot validate the above-mentioned versions of the axioms featuring the alternative notion of simulation relation, the result is still useful, and also resolves an interesting open problem.

Chapter 7 takes up the issue of polymorphism inside data types to a fuller extent. It is in fact necessary to use a polymorphic lambda calculus of higher order than System F, in order to express that the data representation of data types depends on element types. However, the notions of observational equivalence and simulation relation can still be expressed in System F terms. This means that the results developed in earlier chapters lift easily to this scenario.

Chapter 8 summarises the thesis and presents further directions of research. Finally, there is an appendix containing material and proofs omitted from the main text. There is also an author index and a subject index giving the primary occurrences of central notions.

—

There are three protagonists in our setting. First there is the *user*. The user is either a programmer or a client program using a program module. The user is interested in program module interfaces, and will want to be spared from knowing the finicky details of a module in use, and from violating an employed module's internal functioning. Both these aspects are taken care of by the information hiding principle mentioned earlier.

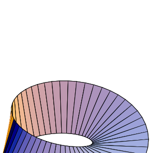
Then there is the *specifier*. The specifier is involved in developing a system and is probably a person. The specifier will be interested in good utilities and formalisms to aid in the specification of abstract modules which will ultimately yield program modules with well-defined interfaces and information hiding.

Finally there is *we*. We are the writer, or the writer and reader of this thesis, and sometimes other people developing formalisms and theory in computer science. In this thesis, we are attempting to aid the specifier, by supplying underlying theory that the specifier may use through some sort of interface. The interface between the theory and specifier is a syntax or a set of formalisms for specifying abstract modules that the specifier may use without knowing all the underlying theoretical details.

Finally, we, the writer, wish the reader a fruitful read.

—

Producing 25 kg of computer generates about 22 kg of toxins and 63 kg of other rubbish. EINAR FLYDAL — TELENOR



—

Fig. 1.1 gives a conceptual overview of the thesis: Concepts of algebraic specification refinement are expressed in a type-theoretical setting. At first order, the existence of simulation relations coincides with observational equivalence, and simulation relations compose. The proof strategy for observational refinement is expressed by adding axioms for quotients and subobjects to the logic. Soundness of the logic extended with these axioms is shown w.r.t. the parametric PER model. The axioms use simulation relations. At higher order, the coincidence between simulation relations and observational equivalence breaks down, as does the composability of simulation relations. One solution is to devise an alternative notion of simulation relation that observes the abstraction barrier inherent in existential types. These abstraction barrier-observing simulation relation give rise to new axioms for the proof strategy. Soundness for these axioms is proved by giving an abstraction barrier-observing semantics for data types. The axioms for the proof method without the alternative notion of simulation relation are justified w.r.t. a parametric setoid model. The main body of work emphasises the development for data types with higher-order operators. Data types with polymorphic operators are treated by pointwise extensions of the achieved results for higher-order operations. The work in this thesis lays a foundation for data types utilising further programming constructs.

—

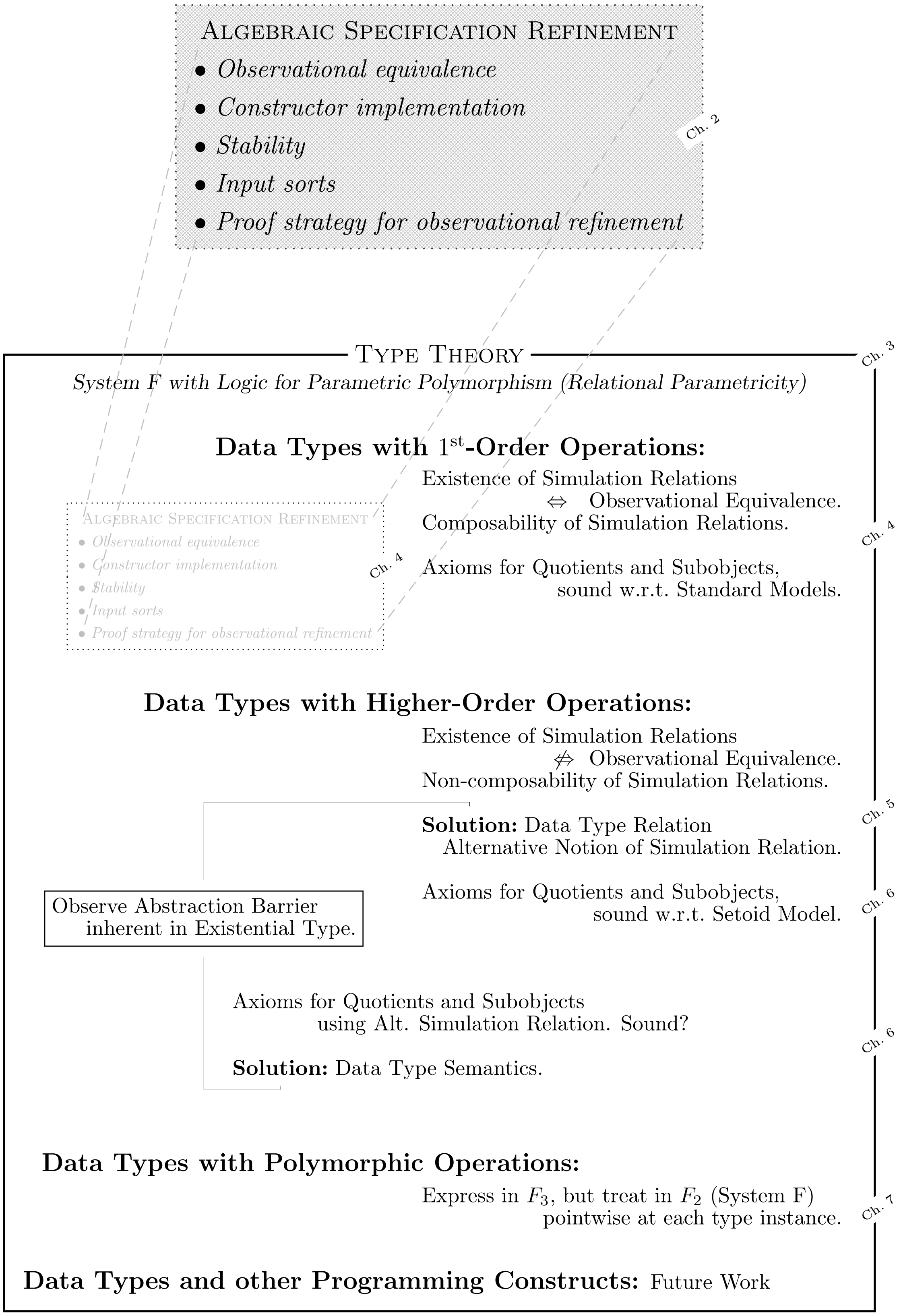


Figure 1.1: Conceptual Overview of thesis.

Fig. 1.2 gives an overview of the most central technical results of the thesis. The lines indicate relatedness of issues. The result concerning data types with polymorphic operations are not included, since they are derived directly from the core results.

Expressing Specification Refinement in Type Theory

Def. 4.4: Observational Equivalence   Th. 4.5: Tight Connection with Components

Def. 3.3: Simulation Relations

Def. 4.6, 4.9, 4.12, 4.15: Specification of Abstract Data Types & Constructors

Def. 4.14: Specification Refinement up to Observational Equivalence

Proof Method for Observational Refinement

Simulation Relations for Data Types with First-Order Operations

Th. 4.17, 4.18: Correspondence with Observational Equivalence.

Th. 4.20: Tight Connection with Components

Th. 4.22: Composability of Simulation Relations

Specification Refinement for Data Types with First-Order Operations

Def. 4.23, 4.24: Axioms for Implementing Proof Method   Th. 4.25: Soundness w.r.t. Parametric PER model

Def. 4.28, 4.30: Translation from Alg. Spec.   Th. 4.31, 4.32, 4.33: Correspondence with Alg. Spec.

Failures of Simulation Relations for Data Types with Higher-Order Operations:

Th. 5.1: Failure of Correspondence with Observational Equivalence.

Ex. 5.7: Counter Example   Th. 5.5: Failure of Tight Connection with Components.

Th. 5.2, 5.3: Failure of Composability and Transitivity.

Solution: Abstraction Barrier-Observing (*abo*-)Simulation Relations.

Def. 5.8 & 5.27: Weak Arrow-Type Relation giving
Def. 5.9 & 5.28: Alternative Notion of Simulation Relation, namely
(*abo*-)Simulation Relations.

Th. 5.14, 5.15, & 5.33, 5.34: Correspondence with Observational Equivalence.

Th. 5.18 & 5.37: Tight Connection with Components

Th. 5.17 & 5.36: Composability of *abo*-Simulation Relations

Specification Refinement for Data Types with Higher-Order Operations

Def. 4.23, 4.24: Axioms for Implementing Proof Method   Th. 6.5, 6.6: Soundness w.r.t. Parametric Setoid model

Def. 6.14, 6.15: Axioms for Implementing Proof Method   Th. 6.16: Soundness w.r.t. Data Type Semantics

Figure 1.2: Technical Core of Thesis.

Chapter 2

Algebraic Specification

In this chapter we give a brief background of algebraic specification refinement. We begin with some basics of universal algebra and algebraic specification. Then specification refinement is introduced, and then we focus on three essential ingredients of refinement, namely constructor implementations, observational equivalence and stability. There are several styles of algebraic specification and of algebraic specification refinement. The concept of algebraic specification originated in the pioneering work of *e.g.,* (Guttag, 1975; Goguen et al., 1978; Liskov and Zilles, 1974). The particular accounts adhered to here are due mainly to (Sannella and Tarlecki, 1997, 1988b,a, 1987; Sannella et al., 1992; Sannella and Wirsing, 1983), where (Sannella and Tarlecki, 1997) gives a descriptive overview that is particularly readable.

—

2.1 Basics

Let $\Sigma = \langle S, \Omega \rangle$ be a *signature*, consisting of a set S of *sorts*, and an $S^* \times S$-sorted set Ω of *operation names*. We write *function profiles* $f : s_1 \times \cdots \times s_n \to s \in \Omega$, meaning $f \in \Omega_{s_1,\ldots,s_n,s}$. If $n = 0$, we write $f : s$, in which case f is a *constant*. Semantically, we consider total algebras with non-empty carriers. A Σ-*algebra* $A = \langle (A)_{s\in S}, F \rangle$ therefore consists of an S-sorted set $(A)_{s\in S}$ of non-empty carriers together with a set F containing a total function $f^A \in (A_{s_1} \times \cdots \times A_{s_n} \to A_s)$ for every $f : s_1 \times \cdots \times s_n \to s \in \Omega$. The class of Σ-algebras is denoted by $\Sigma\mathbf{Alg}$, and the class of Σ-homomorphisms from A to B is denoted by $\Sigma\mathbf{Alg}(A,\ B)$. We also write $\phi : A \to B$ to indicate that ϕ is a homomorphism from A to B.

Given a countable S-sorted set X of variables, the *free* Σ-*algebra over* X is denoted $T_\Sigma(X)$, and for $s \in S$ the carrier $T_\Sigma(X)_s$ contains the *terms of sort* s. Usually, $T_\Sigma(\emptyset)$, the Σ-*ground-term algebra*, is written G_Σ.

Let $\Sigma' = \langle S', \Omega' \rangle$ be a signature contained within Σ, and let A' be a Σ'-algebra. The class of Σ-algebras *containing* A', *i.e.,* whose S'-sorted carriers are A'_s for $s \in S'$, and whose interpretation of every $f \in \Omega'$ is $f^{A'}$, is denoted $\Sigma\mathbf{Alg}(A')$.

Reflecting the situation in most programming languages, we will in general assume certain *built-in* data types. These are represented by designated built-in sorts and operations that have fixed interpretations. For example, we assume natural numbers and booleans and their usual operations to be available anywhere. We therefore assume a designated signature Σ_{Lib} which may be assumed to be contained in signatures if necessary. The contents of Σ_{Lib} will in each situation contain the sorts and operations of the utilised built-in data types. The predetermined interpretation of Σ_{Lib} is the Σ_{Lib}-algebra *Lib* representing the built-in data types. For a signature Σ containing Σ_{Lib}, we will thus be interested in algebras in $\Sigma\mathbf{Alg}(Lib)$. A typical example is of course

$$\Sigma_{Lib} \stackrel{def}{=} \langle \{\mathsf{Nat}, \mathsf{Bool}\}, \mathsf{0} : \mathsf{Nat}, \mathsf{succ} : \mathsf{Nat} \to \mathsf{Nat}, \mathsf{true} : \mathsf{Bool}, \mathsf{false} : \mathsf{Bool} \rangle$$

where

$$Lib \stackrel{def}{=} \langle \{\mathbb{N}, \mathbb{B}\}, \{0, succ, true, false\} \rangle$$

where $\mathbb{N}$ denotes the set of natural numbers, $\mathbb{B}$ is the set of booleans, *succ* is the successor function, and *true* and *false* are the boolean truth values, but in future we may assume other built-in data types and also various extensions to the natural numbers and booleans with the usual operations.

We consider sorted logics. For now we consider first-order logic where equality is the only predicate. A formula φ is a Σ-*formula* if all terms in φ are of sorts in S. The satisfaction of formulae by a Σ-algebra A is defined as usual. Let

$\nu = (\nu_s : X_s \to A_s)_{s \in S}$ be an S-sorted family of functions. Then ν extends uniquely to a homomorphism $\phi_\nu : T_\Sigma(X) \to A$. In this context ν is usually called a *valuation*, and ϕ_ν is called an *interpretation* since the latter can be seen as giving a meaning in A to the terms in $T_\Sigma(X)$. A Σ-equation $u = v$ is satisfied by A if for all $\nu : X \to A$, $\phi_\nu(u) = \phi_\nu(v)$. The semantics of other formulae are then defined as usual on the basis of the semantics of equations. In the world of algebraic specification, this is done classically, rather than constructively.

Now let Ax be a set of closed Σ-formulae. Then $SP = \langle \Sigma, Ax \rangle$ is a *basic algebraic specification*, and its *semantics* $[\![SP]\!]$ is the class of algebras in $\Sigma\mathbf{Alg}(Lib)$ that are models of Ax, *i.e.,* those algebras in $\Sigma\mathbf{Alg}(Lib)$ that satisfy every formula in Ax. Given any basic specification SP, we indicate its components by Σ_{SP} and Ax_{SP}. We will in Sect. 2.6 introduce specification-building operators for building complex specifications from basic specification. Everything introduced in the following applies also to complex specifications, unless indicated otherwise.

It is customary to display specifications in a programming/specification-language style syntax as illustrated in the following example. From now on, we leave built-in sorts and operations implicit.

Example 2.1 The following specification specifies stacks of natural numbers.

$$
\begin{array}{l}
\textbf{spec}\ \mathsf{STACK}\ \textbf{is} \\
\quad \textbf{sorts}\ \mathsf{Stack} \\
\quad \textbf{operations}\ \mathsf{empty} : \mathsf{Stack},\ \mathsf{push} : \mathsf{Nat} \times \mathsf{Stack} \to \mathsf{Stack}, \\
\qquad\qquad\qquad \mathsf{pop} : \mathsf{Stack} \to \mathsf{Stack},\ \mathsf{top} : \mathsf{Stack} \to \mathsf{Nat} \\
\quad \textbf{axioms}\ Ax_{\mathsf{STACK}} : \ \forall x{:}\mathsf{Nat}, s{:}\mathsf{Stack}\ .\ \mathsf{pop}(\mathsf{push}(x, s)) = s \\
\qquad\qquad\qquad\qquad\quad \forall x{:}\mathsf{Nat}, s{:}\mathsf{Stack}\ .\ \mathsf{top}(\mathsf{push}(x, s)) = x
\end{array}
$$

$\circ$

In programming, an entity consisting of a data representation together with operations on that data representation, is sometimes called a *data type*. Hence data types can be seen as algebras in the above sense of universal algebra; indeed Mitchell and Plotkin call data types *data algebras* in (Mitchell and Plotkin, 1988). This is the basic observation for the field of algebraic specification.

An *abstract data type* (ADT) essentially consists of a family of data types sharing some abstract properties, usually given by an *interface* of some sort. This is a loose definition and there are many ways of making this more precise. Algebras in universal algebra have already a primary interface provided by the signature. This guarantees the existence of operations which all algebras of that signature must have, as well as giving certain syntactic information about those operations. In algebraic specification one furthermore has specifications that provide more information about what the operations do. A specification can be more or less *abstract*. For basic specifications, the more abstract they are, the more non-isomorphic algebras they have as models. If the axioms fully specify the behaviour of the operations, there is only one model up to isomorphism for the specification, in which case the specification is *concrete*. Later, observational abstraction will add a new level of abstraction. In any case, it is clear that the type of specification we are dealing with here is

the specification of abstract data types

and this is the topic with which this thesis is primarily concerned.

2.2 Specification Refinement

For any one abstract data type there can exist many specifications characterising that abstract data type. *Stepwise specification refinement* is the constructive version of this statement. Given an abstract specification of an abstract data type, one seeks to construct a replacement specification for the same abstract data type. In addition one adds direction to this process in that the replacement specification is less abstract. This constitutes a *refinement step*, and this process is repeated if possible until a concrete specification emerges. This then constitutes a *refinement process*, where the goal is to develop in a sound methodical way a concrete specification from an abstract specification.

Wide-spectrum specification languages allow specifications and programs to be written in one uniform language, so that specifications are abstract descriptions of abstract data types, while program modules are concrete executable descriptions of the same. The resulting concrete specification of a refinement process is in this case an actual executable program.

Algebras satisfying a specification are often referred to as *realisations* of a specification. The final concrete specification in a refinement process is called a *full refinement* of the abstract specification. In the case where the concrete specification happens to be a program, this then also refers to the unique realisation up to isomorphism of the final specification.

There are various notions of specification refinement, *e.g.,* (Sannella and Tarlecki, 1997, 1988b, 1987), (Back and Wright, 1998), (Morgan, 1994), (Hoffmann and Krieg-Brückner, 1993). Important wide-spectrum specification languages are Extended ML (Kahrs et al., 1997, 1994; Sannella, 1991), ABEL (Dahl and Owe, 1991, 1995; Dahl and Kristoffersen, 1995), CIP-L (Bauer et al., 1981; CIP-L, 1985), languages in the PROSPECTRA project (Hoffmann and Krieg-Brückner, 1993; Krieg-Brückner et al., 1991; Krieg-Brückner, 1990), and the language of the Refinement Calculus, see (Morgan, 1994) and (Back and Wright, 1998).

The basic definition of refinement we adopt here is given by the following refinement relation $\leadsto$ on specifications of the same signature (Sannella and Tarlecki, 1988b; Sannella and Wirsing, 1983):

$$SP \leadsto SP' \quad \overset{def}{\Leftrightarrow} \quad [\![SP]\!] \supseteq [\![SP']\!]$$

In this case we say that the specification SP *refines* to the specification SP', and that SP' is a *refinement* of SP.

There are two indispensable refinements as it were, of the refinement relation. One introduces constructors, the other involves behavioural abstraction.

2.3 Constructor Implementation

A refinement process involves making decisions about design and implementation detail. At some point, a particular function or module may become completely determined and remain unchanged throughout the remainder of the refinement process. It is convenient to lay aside the fully refined parts and continue development on the remaining unresolved parts only. Let κ be a *parameterised program* (Goguen, 1984) with input interface SP' and output interface SP. Given a program P that is a full refinement of SP', the instantiation $\kappa(P)$ is then a full refinement of SP. The semantics of a parameterised program is a function $[\![\kappa]\!] \in (\Sigma_{SP'}\mathbf{Alg} \to \Sigma_{SP}\mathbf{Alg})$ called a *constructor*. *Constructor implementation* is then defined (Sannella and Tarlecki, 1988b) as

$$SP \underset{\kappa}{\leadsto} SP' \quad \overset{def}{\Leftrightarrow} \quad [\![SP]\!] \supseteq [\![\kappa]\!]([\![SP']\!])$$

where $[\![\kappa]\!]([\![SP']\!])$ denotes the image of $[\![SP']\!]$ under $[\![\kappa]\!]$. The parameterised program κ is the fully refined part of the system which is set aside, and SP' specifies the remaining unresolved part that needs further refinement. One then gets the refinement scenario

$$SP_0 \underset{\kappa_1}{\leadsto} SP_1 \underset{\kappa_2}{\leadsto} \cdots \underset{\kappa_n}{\leadsto} SP_n$$

where, as i increases, the SP_i specify the ever smaller unresolved part of the system, whereby SP_n is the empty specification. The result program of the refinement process is then obtained by composing the sedimentary κ_i precipitated out during the process:

$$\kappa_0(\kappa_1(\cdots(\kappa_n(\emptyset)\cdots)))$$

Here $\emptyset$ is the unique empty model of the empty specification. Parameterised programs correspond in spirit to SML *functors* (Paulson, 1996), to SIMULA *class-parameterised functions* and *class-parameterised classes* (Kirkerud, 1989; Pooley, 1987; Dahl and Nygaard, 1981; Dahl et al., 1970; Dahl and Nygaard, 1966), to Modula-3 *generic modules* (Nelson, 1991), and to Java *object-parameterised objects* (Borror, 1999; Flanagan and Ferguson, 1999).

2.4 Observational Equivalence

A major point in algebraic specification is that an abstract specification really is abstract enough to give freedom of implementation. The notion of *behavioural abstraction* captures the concept that two programs are considered equivalent if their observable behaviours are equivalent. Algebraically, one assumes a designated set $Obs \subseteq S$ of observable sorts, and a designated set $In \subseteq S$ of input sorts. *Observable computations* are represented by terms in $T_\Sigma(X^{In})_s$, for $s \in Obs$ and where $X^{In}_s = X_s$ for $s \in In$ and $\emptyset$ otherwise. The signature $\Sigma = \langle S, \Omega\rangle$ is assumed *sensible* w.r.t. $In \subseteq S$, that is, we assume that $T_\Sigma(X^{In})$, the free Σ-algebra generated by X^{In}, is non-empty in every sort, meaning there exists a term of every sort built from function symbols in Ω possibly using variables from X^{In}.

Two Σ-algebras A and B are *observationally equivalent w.r.t.* Obs, In, written $A \equiv^{Obs,In} B$, if they satisfy the same equations at observable sorts, *i.e.,* if there exist surjective valuations $\nu^A : X^{In} \to A$ and $\nu^B : X^{In} \to B$ such that for all $u, v \in T_\Sigma(X^{In})_s$, $s \in Obs$, we have $\phi_{\nu^A}(u) = \phi_{\nu^A}(v) \;\Leftrightarrow\; \phi_{\nu^B}(u) = \phi_{\nu^B}(v)$. This encompasses notions due to *e.g.,* (Reichel, 1987; Nivela and Orejas, 1988; Sannella and Tarlecki, 1987, 1988b; Schoett, 1986), see (Hennicker, 1997).

Suppose now that all observable and input sorts are built in, *i.e.,* have predetermined semantics in the built-in data type algebra *Lib*. In this case, the above definition of observational equivalence reduces to A and B being observationally equivalent if every observable computation computes to the same value in A and B, *i.e.,* if for all $\nu^A : X^{In} \to A$ and $\nu^B : X^{In} \to B$ that agree on X^{In}, we have $\phi_{\nu^A}(u) = \phi_{\nu^B}(u)$ for all $u \in T_\Sigma(X^{In})_s$, $s \in Obs$. We shall return to this later in Ch. 4 when we look at specification up to observational equivalence in type theory. The type theory will have built-in observable data types.

Now, the semantics $[\![SP]\!]$ is not always closed under observational equivalence. For example, the stack-with-pointer implementation of stacks of natural numbers does not satisfy $\mathsf{pop}(\mathsf{push}(x, s)) = s$, because there might be junk values above the top-pointer, and is therefore not in $[\![\mathsf{Stack}]\!]$, but is nevertheless observationally equivalent w.r.t. $Obs = In = \{\mathsf{Nat}\}$ to an algebra that is. To capture this idea of abstraction, one defines the *closure under observational equivalence* of a class $\mathcal{C}$ of algebras as $\mathit{Abstract}(\mathcal{C}) \stackrel{\mathit{def}}{=} \{B \mid \exists A \in \mathcal{C} \,.\, B \equiv^{\mathit{Obs},\mathit{In}} A\}$. One then introduces *observational specifications* by writing

$$\textbf{abstract } \mathit{SP} \textbf{ wrt } \equiv^{\mathit{Obs},\mathit{In}}$$

whose semantics is given by

$$[\![\textbf{abstract } \mathit{SP} \textbf{ wrt } \equiv^{\mathit{Obs},\mathit{In}}]\!] \stackrel{\mathit{def}}{=} \mathit{Abstract}([\![\mathit{SP}]\!])$$

The previous notions of refinement, then give *refinement up to observational equivalence* (Sannella and Tarlecki, 1988b), if the specifications involved are observational specifications.

Why do we want designated input sorts? One extremal view is to say that observable computations should be ground terms, proclaiming $\mathit{In} = \emptyset$. But that would be too strict in a refinement situation where a data type depends on another as yet undeveloped data type. On the other hand, letting all sorts be input sorts would disallow intuitively feasible observational refinements as illustrated in the following example from (Hennicker, 1997).

Example 2.2 Consider the following specification of sets of natural numbers.

$$
\begin{array}{l}
\textbf{spec } \mathsf{SET} \textbf{ is} \\
\quad \textbf{sorts } \mathsf{Set} \\
\quad \textbf{operations } \mathsf{empty} : \mathsf{Set},\ \mathsf{add} : \mathsf{Nat} \times \mathsf{Set} \to \mathsf{Set} \\
\quad \qquad\qquad\qquad \mathsf{in} : \mathsf{Nat} \times \mathsf{Set} \to \mathsf{Bool},\ \mathsf{remove} : \mathsf{Nat} \times \mathsf{Set} \to \mathsf{Set} \\
\quad \textbf{axioms } \mathit{Ax}_{\mathsf{SET}} : \ \forall x{:}\mathsf{Nat}, s{:}\mathsf{Set} \,.\, \mathsf{add}(x, \mathsf{add}(x, s)) = \mathsf{add}(x, s) \\
\qquad\qquad\qquad\qquad \forall x, y{:}\mathsf{Nat}, s{:}\mathsf{Set} \,.\, \mathsf{add}(x, \mathsf{add}(y, s)) = \mathsf{add}(y, \mathsf{add}(x, s)) \\
\qquad\qquad\qquad\qquad \forall x{:}\mathsf{Nat} \,.\, \mathsf{in}(x, \mathsf{empty}) = \mathsf{false} \\
\qquad\qquad\qquad\qquad \forall x, y{:}\mathsf{Nat}, s{:}\mathsf{Set} \,.\, \mathsf{in}(x, \mathsf{add}(y, s)) = \mathsf{if}\ x =_{\mathsf{Nat}} y\ \mathsf{then\ true} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else\ in}(x, s) \\
\qquad\qquad\qquad\qquad \forall x{:}\mathsf{Nat}, s{:}\mathsf{Set} \,.\, \mathsf{in}(x, \mathsf{remove}(x, s)) = \mathsf{false} \\
\qquad\qquad\qquad\qquad \forall x, y{:}\mathsf{Nat}, s{:}\mathsf{Set} \,.\, x \neq_{\mathsf{Nat}} y \ \Rightarrow\ \mathsf{in}(x, \mathsf{remove}(y, s)) = \mathsf{in}(x, s)
\end{array}
$$

Consider the Σ_{SET}-algebra *ListImpl* (LI) whose carrier $\mathit{LI}_{\mathsf{Set}}$ is the set of finite lists over the natural numbers; $\mathsf{empty}^{\mathit{LI}}$ gives the empty list, $\mathsf{add}^{\mathit{LI}}$ appends a given element to the end of a list only if the element does not occur already, $\mathsf{in}^{\mathit{LI}}$ is the occurrence function, and $\mathsf{remove}^{\mathit{LI}}$ removes the first occurrence of a given element. Being a Σ_{SET}-algebra, LI allows users only to build lists using $\mathsf{empty}^{\mathit{LI}}$ and $\mathsf{add}^{\mathit{LI}}$, and on such lists the efficient $\mathsf{remove}^{\mathit{LI}}$ gives the intended result. However, $\mathit{LI} \notin [\![\textbf{abstract } \mathsf{SET} \textbf{ wrt } \equiv^{\mathit{Obs},\mathit{In}}]\!]$, for $\mathit{Obs} = \{\mathsf{Bool}, \mathsf{Nat}\}$

and $In = \{\mathsf{Set}, \mathsf{Bool}, \mathsf{Nat}\}$, because the observable computation $\mathsf{in}(x, \mathsf{remove}(x, s))$ might give true, since s ranges over all lists, not only the canonical ones generated by empty^{LI} and add^{LI}. On the other hand, $LI \in [\![\mathbf{abstract}\ \mathsf{SET}\ \ \mathbf{wrt}\ \equiv^{Obs,In}]\!]$ for $In = Obs = \{\mathsf{Bool}, \mathsf{Nat}\}$, since now the use of Set-variables in observable computations is prohibited. ○

Restricting input sorts means restricting the range of observations one may make, because observations may only observe certain parts of the data representation. For the user observing from outside the data type, this effectively restricts the data representation. For example, although LI in the above example uses the set of all lists as data representation, users may only use the parts of this data representation that can be expressed in terms of empty^{LI} and add^{LI}. It would make no difference to the user if LI instead used a more complicated data representation consisting of lists in which elements occur only once.

There has been debate about how to choose the set of input sorts. In Example 2.2, the correct choice was $In = Obs$. Many hold that $In = Obs$ is practically always a sensible choice. It turns out that when we later enter the realm of type theory, this choice is in a sense inherent. Choosing $In = Obs$ is in any case a simplifying assumption.

2.5 Stability

Observational refinement steps are in general hard to verify. A helpful concept is that of *stability* (Schoett, 1986). A constructor $[\![\kappa]\!]$ is stable if

$$A \equiv^{Obs',In'} B \ \Rightarrow\ [\![\kappa]\!](A) \equiv^{Obs,In} [\![\kappa]\!](B)$$

Under stability, it suffices for proving

$$(\mathbf{abstract}\ SP\ \mathbf{wrt}\ \equiv^{Obs,In}) \underset{\kappa}{\leadsto} (\mathbf{abstract}\ SP'\ \mathbf{wrt}\ \equiv^{Obs',In'})$$

to show $[\![\mathbf{abstract}\ SP\ \mathbf{wrt}\ \equiv^{Obs,In}]\!] \supseteq [\![\kappa]\!]([\![SP']\!])$, *i.e.,* it suffices to show

$$(\mathbf{abstract}\ SP\ \mathbf{wrt}\ \equiv^{Obs,In}) \underset{\kappa}{\leadsto} SP'$$

This simplification is hugely beneficial, since one can now consider algebras only in $[\![SP']\!]$. For SP' a basic specifications, this in particular means that we can assume literal satisfaction of the axioms of SP'.

To justify this, assume $[\![\mathbf{abstract}\ SP\ \mathbf{wrt}\ \equiv^{Obs,In}]\!] \supseteq [\![\kappa]\!]([\![SP']\!])$. We want to show $[\![\mathbf{abstract}\ SP\ \mathbf{wrt}\ \equiv^{Obs,In}]\!] \supseteq [\![\kappa]\!]([\![(\mathbf{abstract}\ SP'\ \mathbf{wrt}\ \equiv^{Obs',In'})]\!])$.

Let $A' \in [\![(\textbf{abstract } SP' \textbf{ wrt } \equiv^{Obs',In'})]\!]$, *i.e.,* there exists a $B' \in [\![SP']\!]$, such that $A' \equiv^{Obs',In'} B'$. We must show the existence of an $A \in [\![SP]\!]$ such that $[\![\kappa]\!](A') \equiv^{Obs,In} A$. By assumption we have the existence of a $B \in [\![SP]\!]$, such that $[\![\kappa]\!](B') \equiv^{Obs,In} B$. Stability then gives $[\![\kappa]\!](A') \equiv^{Obs,In} [\![\kappa]\!](B') \equiv^{Obs,In} B$.

The other direction is trivial, since $[\![SP']\!] \subseteq [\![(\textbf{abstract } SP' \textbf{ wrt } \equiv^{Obs',In'})]\!]$, hence the simplified proof criterion is also the necessary condition.

The following contrived but short example from (Sannella and Tarlecki, 1997) illustrates the point. See *e.g.,* (Schoett, 1986) for a more realistic example.

Example 2.3 (Sannella and Tarlecki, 1997) Consider the specification

$$
\begin{array}{l}
\textbf{spec } \mathsf{TRIV} \textbf{ is} \\
\quad \textbf{operations } \mathsf{id} : \mathsf{Nat} \times \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat} \\
\quad \textbf{axioms } Ax_{\mathsf{TRIV}} : \ \forall x, n, z : \mathsf{Nat} \ . \ \mathsf{id}(x, n, z) = x
\end{array}
$$

Now we define the constructor $Tr \in (\Sigma_{\mathsf{STACK}}\textbf{Alg} \to \Sigma_{\mathsf{TRIV}}\textbf{Alg})$ as follows. For any algebra $A \in \Sigma_{\mathsf{STACK}}\textbf{Alg}$, define $multipush_A \in (\mathbb{N} \times \mathbb{N} \times A_{\mathsf{Stack}} \to A_{\mathsf{Stack}})$ and $multipop_A \in (\mathbb{N} \times A_{\mathsf{Stack}} \to A_{\mathsf{Stack}})$ by

$$
multipush_A(n, z, a) = \begin{cases} a, & n = 0 \\ \mathsf{push}^A(z, multipush_A(n', z, a)), & n = succ(n') \end{cases}
$$

$$
multipop_A(n, a) = \begin{cases} a, & n = 0 \\ multipop_A(n', \mathsf{pop}^A(a)), & n = succ(n') \end{cases}
$$

Then $Tr(A)$ is the TRIV-algebra whose single operation is given by

$$
id(x, n, z) = \mathsf{top}^A(multipop_A(n, multipush_A(n, z, \mathsf{push}^A(x, \mathsf{empty}^A)))).
$$

We have $[\![\textbf{abstract } \mathsf{TRIV} \textbf{ wrt } \{\mathsf{Nat}\}, In]\!] \supseteq Tr([\![\textbf{abstract } \mathsf{STACK} \textbf{ wrt } \{\mathsf{Nat}\}, In]\!])$, but to prove this only assuming membership in $[\![\textbf{abstract } \mathsf{STACK} \textbf{ wrt } \{\mathsf{Nat}\}, In]\!]$ is not straight-forward. However, Tr is in fact stable, so it suffices to show $[\![\textbf{abstract } \mathsf{TRIV} \textbf{ wrt } \{\mathsf{Nat}\}, In]\!] \supseteq Tr([\![\mathsf{STACK}]\!])$, and the proof of this goes by easy induction (Sannella and Tarlecki, 1997). In particular, one may now hold $\forall x : \mathsf{Nat}, s : \mathsf{Stack} \ . \ \mathsf{pop}(\mathsf{push}(x, s)) = s$ among ones assumptions, although this formula does not hold in $[\![\textbf{abstract } \mathsf{STACK} \textbf{ wrt } \{\mathsf{Nat}\}, In]\!]$. ○

One still has to prove the stability of constructors. However, since constructors are given by concrete parameterised programs, this can be done in advance for the language as a whole. A key observation is that stability is intimately related to the effectiveness of encapsulation mechanisms in the language. There is much to say about this, but here we only indicate the issue with another slightly contrived but short example.

Example 2.4 (Sannella and Tarlecki, 1997) Let $Tr' \in (\Sigma_{\mathsf{STACK}}\mathbf{Alg} \to \Sigma_{\mathsf{TRIV}}\mathbf{Alg})$ be the constructor such that $Tr'(A)$ is the TRIV-algebra whose operation is

$$id(x,n,z) = \begin{cases} x, & \mathsf{pop}^A(\mathsf{push}^A(z, \mathsf{empty}^A)) = \mathsf{empty}^A \\ z, & \text{otherwise} \end{cases}$$

Now $[\![\mathbf{abstract}\ \mathsf{TRIV}\ \mathbf{wrt}\ \{\mathsf{Nat}\}, In]\!] \not\supseteq Tr'([\![\mathbf{abstract}\ \mathsf{STACK}\ \mathbf{wrt}\ \{\mathsf{Nat}\}, In]\!])$, because if we now consider A the array-with-pointer algebra, we shall get $Tr'(A) \notin [\![\mathbf{abstract}\ \mathsf{TRIV}\ \mathbf{wrt}\ \{\mathsf{Nat}\}, In]\!]$. Here Tr' is not stable; in fact Tr' breaches the abstraction barrier by checking equality on the underlying implementation. ○

2.6 Specification-Building Operators

Algebraic specifications may be complex, built from basic specifications using *specification-building operators*. Various specification languages provide various specification-building operators, but there are certain canonical ones, in terms of which most others may be expressed, see *e.g.,* (Wirsing, 1993; Sannella and Wirsing, 1983, 1999).

A central specification-building operator is the **derive** operator. It is defined in terms of reducts as follows. For signatures $\Sigma = \langle S, \Omega \rangle$ and $\Sigma' = \langle S', \Omega' \rangle$, a *signature morphism* $\sigma : \Sigma' \to \Sigma$ maps the sorts and operator symbols of Σ' to those of Σ such that sorts are preserved, *i.e.,* if $t\!:\!s$ where $s \in S'$, then $\sigma(t)\!:\!\sigma(s)$ where $\sigma(s) \in S$. For a Σ-algebra A, the *σ-reduct* $A|_\sigma$ of A is the Σ'-algebra with carriers $(A|_\sigma)_s = A_{\sigma(s)}$ for each sort $s \in S'$ and $f^{A|_\sigma} = \sigma(f)^A$ for each $f \in \Omega'$. If σ is not surjective, the effect is that of *hiding*, *i.e.,* removing, those carriers and operators of A which are not interpretations of symbols in $\sigma(\Sigma')$. Now we have

$$[\![\,\mathbf{derive}\ SP\ \mathbf{by}\ \sigma\,]\!] \stackrel{\mathit{def}}{=} \{A|_\sigma \mid A \in [\![SP]\!]\}$$

for $\Sigma_{SP} = \Sigma$. We have $\Sigma_{\mathbf{derive}\ SP\ \mathbf{by}\ \sigma} \stackrel{\mathit{def}}{=} \Sigma'$. If the signature morphism is an inclusion $\iota\!:\!\Sigma' \to \Sigma$, one usually writes reducts simply as $A|_{\Sigma'}$, and uses the **hide** keyword instead, *i.e.,* for $S'' = S \setminus S'$ and $\Omega'' = \Omega \setminus \Omega'$,

$$[\![\,\mathbf{hide\ sorts}\ S''\ \ \mathbf{operations}\ \Omega''\ \mathbf{in}\ SP\,]\!] \stackrel{\mathit{def}}{=} \{A|_{\Sigma'} \mid A \in [\![SP]\!]\}$$

The **sum** operator is defined by $\Sigma_{SP\ \mathbf{sum}\ SP'} \stackrel{\mathit{def}}{=} \Sigma_{SP} \cup \Sigma_{SP'}$, and

$$[\![\,SP\ \mathbf{sum}\ SP'\,]\!] \stackrel{\mathit{def}}{=} \{A \in \Sigma\mathbf{Alg}_{\Sigma_{SP\ \mathbf{sum}\ SP'}} \mid A|_{\Sigma_{SP}} \in [\![SP]\!] \ \ \wedge \ \ A|_{\Sigma_{SP'}} \in [\![SP']\!]\}$$

We omit the definition of **translate**.

Any first-order specification built from a basic specification by applying the canonical specification building operators **sum**, **derive**, and **translate** (Sannella and Wirsing, 1999) can be algorithmically normalised to a basic specification, possibly with a **hide** operator outermost (Wirsing, 1993; Farrés-Casals, 1992; Cengarle, 1995).

A common specification-building operator which is expressible by **sum** is **enrich**, with semantics

$$[\![\,\textbf{enrich}\ SP\ \textbf{by sorts}\ S'\ \textbf{ops}\ \Omega'\ \textbf{axioms}\ Ax'\,]\!] \stackrel{def}{=} \{A \in \Sigma'\mathbf{Alg} \mid A|_\Sigma \in [\![SP]\!] \quad\wedge\quad A \models Ax'\}$$

where $\Sigma_{SP} = \Sigma = \langle S, \Omega\rangle$ and $\Sigma_{\textbf{enrich}\ SP\ \textbf{by sorts}\ S'\ \textbf{ops}\ \Omega'} \stackrel{def}{=} \Sigma' = \langle S \cup S', \Omega \cup \Omega'\rangle$.

The other relevant operators are **abstract** and **behaviour**, and also **quotient** and **restrict**. The semantics of the first two operators are related. First we have

$$[\![\,\textbf{abstract}\ SP\ \textbf{wrt}\ \equiv]\!] \stackrel{def}{=} \{B \mid \exists A \in [\![SP]\!]\ .\ B \equiv A\}$$

for any equivalence relation $\equiv$ on $\Sigma_{SP}\mathbf{Alg}$. The equivalence $\equiv$ usually goes under the name *behavioural equivalence*. The special case $\equiv^{Obs,In}$ that we have seen before in Sect. 2.4, is called an *observational equivalence*.

For a signature $\Sigma = \langle S, \Omega\rangle$, a *partial* Σ*-congruence* $\approx_A = (\approx_{As})_{s\in S}$ on a Σ-algebra A is a family of partial equivalence relations on A that are *compatible* with Σ, *i.e.,*

$$a_i \approx_{As_i} b_i \ \Rightarrow\ f(a_1, \ldots, a_n) \approx_{As} f(b_1, \ldots, b_n)$$

for every $f : s_1 \times \cdots \times s_n \to s \in \Omega$. The *domain* $Dom(\approx_A)$ of $\approx_A$ is given by $Dom(\approx_A)_s \stackrel{def}{=} \{a \in A_s \mid a \approx_{As} a\}$ for all $s \in S$. A *total* Σ-congruence $\approx_A$ is such that $Dom(\approx_A) = A$. For any Σ-algebra A and total Σ-congruence $\approx_A$, the quotient algebra $A/\!\approx_A$ is formed by $(A/\!\approx_A)_s \stackrel{def}{=} A_s/\!\approx_{As}$, for every $s \in S$, and

$$f^{A/\approx_A}([a_1]_{A_{s_1}/\approx_{A_{s_1}}}, \ldots, [a_n]_{A_{sn}/\approx_{A_{sn}}}) \stackrel{def}{=} [f^A(a_1, \ldots\ a_n)]_{A_s/\approx_{A_s}}$$

for every $f : s_1 \times \cdots \times s_n \to s \in \Omega$. For any Σ-algebra A and a set of Σ-equations E, the *congruence* $\sim_E^A$ *induced by* E is defined as the least Σ-congruence containing $\{\langle \phi(l), \phi(r)\rangle \mid \langle l, r\rangle \in E\}$, for all Σ-homomorphisms $\phi : T_\Sigma(X) \to A$. One usually writes A/E for the quotient w.r.t. this total congruence.

A *subalgebra* of a Σ-algebra A, is a Σ-algebra B such that $B_s \subseteq A_s$ for all $s \in S$, and f^B is the restriction of f^A to B, for all $f \in \Omega$.

The restriction of $\approx_A$ to the subalgebra $Dom(\approx_A)$ is total. We now get

$$[\![\,\textbf{behaviour}\ SP\ \textbf{wrt}\ \approx]\!] \stackrel{def}{=} \{A \in \Sigma_{SP}\mathbf{Alg} \mid Dom(\approx_A)/\!\approx_A \in [\![SP]\!]\}$$

where $\approx\ = (\approx_A)_{A\in\Sigma_{SP}\mathbf{Alg}}$ is a family of partial Σ_{SP}-congruences. This family is usually called a *behavioural congruence*.

Semantically, **abstract** and **behaviour** fulfil the same task in practice. Here is an outline why, see (Bidoit et al., 1995) for details. The reason for considering the **behaviour** approach is that it is the basis for a refinement calculus.

Let $\equiv$ be any equivalence on $\Sigma\mathbf{Alg}$, and let $\approx$ be any family of Σ-congruences. Then $\equiv$ is *factorisable* by $\approx$, if for all $A, B \in \Sigma\mathbf{Alg}$,

$$A \equiv B \;\Leftrightarrow\; Dom(\approx_A)/\!\approx_A \,\cong\, Dom(\approx_B)/\!\approx_B$$

Factorisability means that we can express the equivalence between algebras by a congruence on each algebra. For observational equivalences this is always the case, and this is instrumental for proving observational refinements. The reason we need partial congruences is in case the input sorts do not include all sorts of the signature. For example, if we were to refine the specification SET from Example 2.2 using a specification of lists, the appropriate observational congruence on lists would be defined only on lists generated by empty and add. The domain of this congruence would then include only those values representing such lists.

Now, given such a family of congruences $\approx$, which might be an *observational congruence* $\approx^{Obs,In}$ factorising an observational equivalence, a Σ-specification SP is said to be *behaviourally closed* (or *behaviourally consistent*) *w.r.t.* $\approx$, if

$$\{Dom(\approx_A)/\!\approx_A \mid A \in [\![SP]\!]\} \quad \subseteq \quad [\![SP]\!]$$

This is methodologically an obvious requirement for observational specification (Bidoit et al., 1997, 1995). Another notion is that of $\approx$ being *weakly regular*, (Bidoit et al., 1995). We omit the definition; every observational congruence is weakly regular. It is then shown in (Bidoit et al., 1995) that under factorisability, behavioural closedness and weak regularity,

$$[\![\,\mathbf{abstract}\ SP\ \mathbf{wrt}\ \equiv\,]\!] \quad = \quad [\![\,\mathbf{behaviour}\ SP\ \mathbf{wrt}\ \approx\,]\!]$$

We return to these facts in the next section.

The semantics of **quotient**, where $\approx$ is family of total Σ_{SP}-congruences, goes as follows.

$$[\![\,\mathbf{quotient}\ SP\ \mathbf{by}\ \approx\,]\!] \stackrel{def}{=} \{A/\!\approx_A \mid A \in [\![SP]\!]\}$$

The semantics of **restrict** are more complicated. Let $\Sigma = \langle S, \Omega\rangle$ and let $S' \subseteq S$. We now write $In = S \setminus S'$. This is no coincidence, because we will look at **restrict** in connection with observational specification in a moment. A Σ-algebra A is *reachable on* S' if there is no proper Σ-subalgebra whose In-sorted carriers are the same as those of A. Equivalently, let $X^{In} \subseteq X$ denote the In-sorted variables of X. Then A is reachable on S' if and only if for every $a \in A$ there is a term $t \in T_\Sigma(X^{In})$ such that $\phi(t) = a$ for some homomorphism $\phi : T_\Sigma(X^{In}) \to A$. Any Σ-algebra A has a unique Σ-subalgebra which is reachable on S', denoted $R_{S'}(A)$. We then get

$$[\![\, \textbf{restrict } SP \textbf{ on } S' \,]\!] \stackrel{def}{=} \{R_{S'}(A) \mid A \in [\![SP]\!]\}$$

There is an obvious connection between observational specification and the quotient and restrict operators, because we can factorise an observational equivalence through a congruence. We can thus look at a certain **quotient** specification. If the congruence is partial, one has to use **restrict** prior to quotienting. We will come back to this later, but suffice it to say that this relates observational specification to the *forget-restrict-identify* (FRI) implementation strategy of algebraic specification (Wirsing, 1990). The FRI structure goes as follows.

$$\textbf{quotient} \;\; (\textbf{restrict} \;\; (\textbf{derive } SP \textbf{ by } \sigma : \Sigma^{export} \to \Sigma_{SP}) \;\; \textbf{on } S') \;\; \textbf{by} \;\; \approx$$

for $\Sigma^{export} = \langle S^{export}, \Omega^{export} \rangle$, $S' \subseteq S^{export}$, and $\approx$ a family of total Σ^{export}-congruences. Its semantics is $\{R_{S'}(A|_\sigma)/\!\approx \;\mid A \in [\![SP]\!]\}$. Note that the input sorts *In* are now $S^{export} \setminus S'$. The idea behind the *forget* step is to hide internal implementation. There is a range of model classes according to the choice of S'. The case $S' = \emptyset$ gives $R_\emptyset(A|_\sigma)/\!\approx \;= A|_\sigma/\!\approx$, and corresponds to the special case *forget-identify* (FI). The case $S' = S^{export}$ is of course ground term denotability.

Let us return to the issue of the normalisability of specifications. The normalisability result mentioned earlier applies only to specifications built from basic specifications using **sum**, **derive**, and **translate**, and not to the other non-derivable specification building operators we have mentioned. However, in a refinement context it can be argued that **abstract** and **behaviour** should be seen as meta-operators and should only be applied outermost (Bidoit et al., 1996). A similar argument can be made for **quotient** and **restrict**, although specifications with the **restrict** operator are normalisable, albeit with infinitary axioms.

This all means that theoretically, it is fine to only consider observational specifications of the form **abstract** SP **wrt** $\equiv^{Obs,In}$ where SP is in normal form. When we later translate specifications into type theory and show an exact correspondence between type-theoretic refinement and the algebraic specification notion of refinement, we thereby do this for complex specifications as well.

This does by no means render specification-building operators superfluous. The specifier will of course still want to use specification-building operators in order to modularise and structure his specification.

2.7 Proving Observational Refinement

Specifications using **abstract** are based on the intuitive semantic notion of observational equivalence. On the other hand, the approach represented by **behaviour** provides a better basis for devising proof methods, because behavioural congruences may be axiomatised and thereby expressed in the logic.

There exists a sound and complete calculus $\vdash_{\Pi_{\leadsto}}$ for **behaviour** specification refinement, based on a calculus $\vdash_{\Pi_S}$ for structured specifications. By the equivalence of **abstract** and **behaviour** under factorisability and behavioural closedness, this calculus deals with **abstract** specifications as well, see (Bidoit et al., 1997). The caveat is of course that completeness is here modulo the underlying predicate logic. Also, the calculus $\vdash_{\Pi_{\leadsto}}$ imposes certain reasonable restrictions on specifications, *e.g.,* in the context of **hide**. We will come back to this in Sect. 4.6.

The refinement proof strategy for **behaviour** specifications, and thus for **abstract** specifications, is based on a Galois correspondence between so-called behaviour classes and behavioural quotients, entailing that in order to prove that

$$(\textbf{behaviour}\ SP\ \ \textbf{wrt}\ \approx) \ \leadsto\ SP'$$

it suffices to prove

$$SP \ \leadsto\ SP'/\!\approx$$

where $[\![\ SP'/\!\approx\ ]\!] \stackrel{def}{=} \{Dom(\approx_A)/\!\approx_A \mid A \in [\![SP']\!]\}$. If the partiality of $\approx$ is expressible by **restrict**, we get the sufficient condition of

$$SP \ \leadsto\ \textbf{quotient}\ (\textbf{restrict}\ SP'\ \textbf{on}\ S')\ \ \textbf{wrt}\ \approx$$

for some appropriate S'. The strategy now seeks to axiomatise the partial congruence, giving $Ax(\sim)$, where $\sim$ is a new symbol representing $\approx$. Then $Ax(\sim)$ is added for proving *relativised* versions of the SP-axioms, *i.e.,* versions where, roughly, equality is replaced by $\sim$ (Bidoit and Hennicker, 1996). The strategy is illustrated in Example 2.5 below. A comprehensive up-to-date account of work on observational specification and on proving observational refinement is (Hennicker, 1997), which also includes the results of (Bidoit et al., 1997; Bidoit and Hennicker, 1996; Bidoit et al., 1995).

2.8 Simplified Equational Proofs

There is an opening for simplification in the case of basic equational specifications in observational refinement. Suppose now the behavioural congruence can be expressed by equations. It would then be proof-technically convenient if these latter equations could be used in an equational calculus directly in conjunction with other equations specifying the data type, without having to introduce a new symbol $\sim$ *etc.* This would also preserve the equational character of the refinement, meaning that efficient equational reasoning tools can still be employed.

In (Hannay, 1998) it is shown that this is indeed possible, provided one incorporates the appropriate *abstraction barrier* in the equational calculus. The resulting calculi $\vdash^{\mathrm{FI}}$ and $\vdash^{\mathrm{FRI}}$ appear in Sect. A.1 in Appendix A. It suffices to restrict the congruence (monotonicity) axiom of the equational calculus to contexts without designated hidden symbols, *i.e.,* imposing *referential opacity*, see (Qian and Goldberg, 1993; Søndergaard and Sestoft, 1990) for other uses of referential opacity. Without such an abstraction barrier, the resulting set of equations may be inconsistent since (the axioms for) hidden operators might not respect the intended equality predicate.

Example 2.5 In this adaptation of a classic illustrative example *e.g.,* (Sannella and Tarlecki, 1988b), the specification SET is refined by using BAG and specification building operators. We have the following simple specification of sets.

spec SET **is**
sorts Set
operations empty : Set, add : Nat × Set → Set
in : Nat × Set → Bool
axioms Ax_{SET} : $\forall x{:}\mathsf{Nat}, s{:}\mathsf{Set} \,.\, \mathsf{add}(x, \mathsf{add}(x, s)) = \mathsf{add}(x, s)$
$\forall x, y{:}\mathsf{Nat}, s{:}\mathsf{Set} \,.\, \mathsf{add}(x, \mathsf{add}(y, s)) = \mathsf{add}(y, \mathsf{add}(x, s))$
$\forall x{:}\mathsf{Nat} \,.\, \mathsf{in}(x, \mathsf{empty}) = \mathsf{false}$
$\forall x, y{:}\mathsf{Nat}, s{:}\mathsf{Set} \,.\, \mathsf{in}(x, \mathsf{add}(y, s)) = \mathsf{if}\ x =_{\mathsf{Nat}} y\ \mathsf{then\ true}$
$\mathsf{else\ in}(x, s)$

We then specify bags, *i.e.,* multisets as follows.

spec BAG **is**
sorts Bag
operations empty : Bag, add : Nat × Bag → Bag
count : Nat × Bag → Nat
axioms Ax_{BAG} : $\forall x, y{:}\mathsf{Nat}, b{:}\mathsf{Bag} \,.\, \mathsf{add}(x, \mathsf{add}(y, b)) = \mathsf{add}(y, \mathsf{add}(x, b))$
$\forall x{:}\mathsf{Nat} \,.\, \mathsf{count}(x, \mathsf{empty}) = 0$
$\forall x, y{:}\mathsf{Nat}, s{:}\mathsf{Set} \,.\, \mathsf{count}(x, \mathsf{add}(y, b)) = \mathsf{if}\ x =_{\mathsf{Nat}} y\ \mathsf{then}$
$\mathsf{succ}(\mathsf{count}(x, b))\ \mathsf{else\ count}(x, b)$

The idea is to put an appropriate interface on bags as specified by BAG, so that they look like sets as specified by SET. This may be done by adding in as an interface operator, then hiding its implementation in terms of count. This results in the following specification.

spec $\mathsf{BlackBAG}$ **is**
derive
  enrich BAG **by**
    operations $\mathsf{in} : \mathsf{Nat} \times \mathsf{Bag} \rightarrow \mathsf{Bool}$
    axioms $\forall x{:}\mathsf{Nat}, b{:}\mathsf{Bag} \ .\ \mathsf{in}(x,b) = \mathsf{count}(x,b) > 0$
 by $\sigma = \iota[\mathsf{Set} \mapsto \mathsf{Bag}]$

where σ is the signature morphism from Σ_{SET} to $\Sigma_{\mathsf{BlackBAG}}$ which is the identity on everything except the sort Set which is renamed to Bag. The morphism is not surjective thus hiding count. We want, for $Obs = In = \{\mathsf{Bool}, \mathsf{Nat}\}$,

$$(\textbf{abstract } \mathsf{SET} \textbf{ wrt } \equiv_{Obs,In}) \rightsquigarrow (\textbf{abstract } \mathsf{BlackBAG} \textbf{ wrt } \equiv_{Obs,In})$$

The implicit identity constructor is stable, so it suffices to show

$$(\textbf{abstract } \mathsf{SET} \textbf{ wrt } \equiv_{Obs,In}) \rightsquigarrow \mathsf{BlackBAG}$$

or by the equivalence to **behaviour**,

$$(\textbf{behaviour } \mathsf{SET} \textbf{ wrt } \approx^{Obs,In}) \rightsquigarrow \mathsf{BlackBAG}$$

where $\approx^{Obs,In}$ identifies bags that represent the same set. Then by the Galois connection, it suffices to prove

$$\mathsf{SET} \rightsquigarrow \mathsf{BlackBAG}/\!\approx^{Obs,In}$$

Here, $\approx^{Obs,In}$ is easily axiomatisable by a new symbol $\sim$ using in, *e.g.,*

$$Ax(\sim) \stackrel{def}{=} \forall b, b'{:}\mathsf{Bag} \ .\ b \sim b' \ \Leftrightarrow\ \forall x{:}\mathsf{Nat} \ .\ \mathsf{in}(x,b) = \mathsf{in}(x,b')$$

and then we can prove the refinement by proving relativised SET-axioms, *i.e.,* the axioms where equality at sort Set is replaced by $\sim$.

However it is also possible to simply add the missing characteristic set axiom. For this we first look at

$$\mathsf{SET} \rightsquigarrow \mathsf{SETbyBAG}$$

where $\mathsf{SETbyBAG}$ is the FI-structure

spec $\mathsf{SETbyBAG}$ **is**
quotient $\mathsf{BlackBAG}$ **by** $E' : \{\mathsf{add}(x, \mathsf{add}(x,s)) = \mathsf{add}(x,s)\}$

There is an essential abstraction barrier in the FI-structure. Crucially, the hiding **derive** step is done before quotienting, since quotienting in the presence of hidden operators might give inconsistency. Thus in $\mathsf{SETbyBAG}$, count is hidden *before* quotienting. The specification would otherwise be inconsistent w.r.t. the intended semantics on Nat, since any model $B = A/E'$ would then have to satisfy *e.g.,*

$$2 = \mathsf{count}^B(x, \mathsf{add}^B(x, \mathsf{add}^B(x, \mathsf{empty}^B))) = \mathsf{count}^B(x, \mathsf{add}^B(x, \mathsf{empty}^B)) = 1$$

To sum up, the desired congruence is a congruence on $\Sigma_{\mathsf{BlackBAG}}$-algebras, but not on Σ_{BAG}-algebras, and the FI-structure ensures the appropriate abstraction barrier in syntax and semantics.

Now, in the refinement proof, if we are to simply add the quotienting

$$\mathsf{add}(x, \mathsf{add}(x, s)) = \mathsf{add}(x, s)$$

to the equations, the abstraction barrier in the specificational structure has to be mirrored somehow in the logic. Note that although count is ultimately hidden both in the specification and in the semantics, this hidden symbol and its axioms are needed for reasoning, since in is implemented in terms of it.

In (Hannay, 1998), the calculus $\vdash^{\mathrm{FI}}$, which is sound and complete w.r.t. FI-specification structures, can be used in verifying this refinement. The calculus ensures the safe interaction between the set E of equations associated with $\mathsf{BlackBAG}$ and the set E' of equations introduced in the quotienting step forming $\mathsf{SetbyBag}$. For instance, although we would have

$$\vdash^{\mathrm{FI}} \mathsf{add}(x, \mathsf{add}(x, \mathsf{empty})) = \mathsf{add}(x, \mathsf{empty})$$

referential opacity would prevent the inference

$$\vdash^{\mathrm{FI}} \mathsf{count}(x, \mathsf{add}(x, \mathsf{add}(x, \mathsf{empty}))) = \mathsf{count}(x, \mathsf{add}(x, \mathsf{empty}))$$

which would have given $\vdash^{\mathrm{FI}} 2 = 1$. The inference is illegal because count is a hidden operator symbol. Referential opacity thereby ensures soundness and is an appropriate abstraction barrier in the calculus. There is also a calculus $\vdash^{\mathrm{FRI}}$ which is sound and complete for FRI-specification structures.

In closing this example, we mention that in an executable implementation of $\mathsf{SetbyBag}$, the derive operator might be implemented by an encapsulation mechanism hindering outside access to count, and the quotient operator might be implemented by an equality predicate. ○

We used this simple example to illustrate proof strategies, but it should be pointed out that the example is too simple to really warrant the use of any of these proof methods. More realistic examples abound in the literature.

2.9 Specification of Constructors

We close this chapter by very briefly showing how one can specify constructors. We have seen how to specify programs or algebras via abstract data types. It is natural to try to specify parameterised programs or constructors as well. The simplest form of *constructor specification* is

$$SP' \to SP$$

where $[\![SP' \to SP]\!] \stackrel{def}{=} \{F : \Sigma_{SP'}\mathbf{Alg} \to \Sigma_{SP}\mathbf{Alg} \mid A \in [\![SP']\!] \;\Rightarrow\; F(A) \in [\![SP]\!]\}$. Constructors satisfying specifications of the form above have no obligation to produce algebras that depend in any way on the input algebras. If such a dependency is desired, one can use a *dependent constructor specification* of the form

$$\Pi S{:}SP'.SP[S]$$

where $SP[S]$ is a *data-type parameterised specification*.

Example 2.6 A specification of Example 2.3's Tr can be given by the dependent constructor specification

$$\Pi S{:}\textbf{abstract}\ \mathsf{STACK}\ \textbf{wrt}\ \{\mathsf{Nat}\}, In\ .\ \textbf{abstract}\ \mathsf{TRIVe}[S]\ \textbf{wrt}\ \{\mathsf{Nat}\}, In$$

where $\mathsf{TRIVe}[S]$ is

$$
\begin{array}{l}
\textbf{hide operations}\ \mathsf{multipush}, \mathsf{multipop}\ \textbf{in}\\
\quad \textbf{operations}\ \mathsf{multipush}{:}\mathsf{Nat} \times \mathsf{Nat} \times S.\mathsf{Stack} \to S.\mathsf{Stack},\\
\qquad \mathsf{multipop}{:}\mathsf{Nat} \times S.\mathsf{Stack} \to S.\mathsf{Stack}, \quad \mathsf{id}{:}\mathsf{Nat} \times \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}\\
\quad \textbf{axioms}\ Ax_{Tr} : \forall n, z{:}\mathsf{Nat}.\forall s{:}S.\mathsf{Stack}\ .\ \mathsf{multipop}(n, \mathsf{multipush}(n, z, s)) = s\\
\qquad \forall x, n, z{:}\mathsf{Nat}\ .\\
\qquad\qquad \mathsf{id}(x, n, z) = S.\mathsf{top}(\mathsf{multipop}(n, \mathsf{multipush}(n, z, S.\mathsf{push}(x, S.\mathsf{empty}))))
\end{array}
$$

○

We refrain from defining and giving the semantics of data-type parameterised specifications $SP[S]$ formally. This would be defined by induction on the structure of $SP[S]$, and would for each instance algebra S involve $\Sigma'\mathbf{Alg}(S)$, *i.e.,* models containing an algebra S, *cf.* Sect. 2.1. Here Σ' is the signature of some subspecification of $SP[S]$.

Given the semantics of data-type parameterised specifications, and provided $\Sigma_{SP[S]}$ does not depend on S, we can easily give the semantics of $\Pi S{:}SP'.SP[S]$ as $[\![\Pi S{:}SP'.SP[S]]\!] \stackrel{def}{=} \{F{:}\Sigma_{SP'}\mathbf{Alg} \to \Sigma_{SP}\mathbf{Alg} \mid A \in [\![SP']\!] \;\Rightarrow\; F(A) \in [\![SP[A]]\!]\}$.

Chapter 3

Polymorphism and Relational Parametricity

In this chapter we present the type-theoretic setting consisting of the second-order polymorphic lambda calculus System F and the logic for relational parametric polymorphism due to Plotkin and Abadi. Although our technical discussion will mainly be done w.r.t. System F, it will be useful to view System F in a wider context. We start by reviewing the simply-typed lambda calculus. Then inductive types are explained. After that, the general polymorphic lambda calculus F_ω is introduced, which we then restrict to F_3 and F_2, the latter being what is known as System F. We will use F_3 in Ch. 7. We then look at the building blocks of abstract data types in System F. The logic for relational parametric polymorphism is presented, and at the end we briefly present the PER semantics for the logic due to Bainbridge *et al*, and also syntactic models due to Hasegawa.

3.1 Introduction

Ch. 2 outlined an account of algebraic specification perfected by Sannella and Tarlecki. Our goal is now twofold. First we would like to extend the established concepts of specification refinement to deal with higher-order signatures and polymorphism. Secondly, we want to internalise the semantic notions of refinement into syntactic formalisms in the hope that the process of refinement will become more amenable to mechanical proof aids. The polymorphic lambda calculus is a good choice in which to do this, and indeed this is not a new idea. In (Mitchell and Plotkin, 1988) the notion of algebra is internalised into a version (SOL) of the second-order polymorphic lambda-calculus. In (Luo, 1993) a notion of specification refinement is expressed in the Extended Calculus of Constructions. There is lots of other work linking algebraic specification and type theory, some of which we will get back to later. There are also non-type theoretic approaches to higher-order operations using higher-order universal algebra (Meinke, 1992), and various other set-theoretic models (Kirchner and Mosses, 1998).

3.2 The Lambda Calculus

The main motivation behind lambda calculus and combinatory logic is the view that functions are foundational, rather than derived notions of Zermelo-Fraenkel set theory. The pioneers of lambda-calculus (Church) and combinatory logic (Schönfinkel and Curry) had grand visions of developing a functional foundation for logic and parts of mathematics. This vision has yet to be fulfilled, but the main motivation remains and has of course found extensive theoretic and pragmatic justification in the present-day proliferation of computer technology.

In conjunction with type theory, there now exist numerous lambda calculi in various classifications, see *e.g.,* (Barendregt, 1992) for the famous λ-cube, and (Jacobs, 1996) for an alternative to the cube. Present-day lambda calculi are all in spirit based on the ideas in Church's original formulation (Church, 1940, 1941). It is now common to take the *simply-typed lambda calculus* as a basis and define other lambda calculi as extensions to this basis. The extensions are roughly speaking defined according to what manner of dependencies are allowed for types, *i.e.,* to what degree and combination one allows types to depend on types and types to depend on terms. Our discussion uses the polymorphic lambda calculus without dependent types, *i.e.,* types may depend on types, but not on terms.

We will now survey the notions necessary for our discourse. For fuller accounts, see *e.g.,* (Barendregt, 1992; Mitchell, 1996; Girard et al., 1990; Pierce et al., 1989). Our account is a pragmatic one. It would be theoretically and aesthetically more pleasing to describe the various lambda calculi in the framework of *e.g.,* Pure Type Systems, but the overhead is not quite justifiable for our purposes.

—

3.2.1 The Simply-Typed Lambda Calculus

The simply-typed lambda calculus is in principle close to (Church, 1940). For TyC a collection of *type constants*, TeC a collection of *term constants*, and TeV a collection of *term variables*, the abstract syntax for *types* and *terms* of the calculus is given by the following grammars.

$$\begin{array}{llcl} \text{(types)} & T & ::= & C \mid (T \to T) \\ \text{(terms)} & t & ::= & c \mid x \mid (\lambda x\!:\!T.t) \mid (tt) \end{array}$$

for all $C \in TyC$, $c \in TeC$, and $x \in TeV$. Parentheses are omitted from the syntax whenever no ambiguity arises, with the usual conventions that $\to$ associates to the right and application associates to the left, *i.e.,* $T_1 \to T_2 \to \cdots \to T_{n-1} \to T_n$ stands for $(T_1 \to (T_2 \to \cdots \to (T_{n-1} \to T_n) \cdots))$ and $t_1 t_2 \cdots t_{n-1} t_n$ stands for $((\cdots (t_1 t_2) \cdots t_{n-1}) t_n)$. We also sometimes write $\lambda x_1 : T_1, x_2 : T_2, \cdots, x_n : T_n.t$ for $(\lambda x_1 \!:\! T_1.(\lambda x_2 \!:\! T_2.(\cdots (\lambda x_n \!:\! T_n.t) \cdots)))$. We might in later chapters also use other obvious abbreviations. The collection $FTeV(t)$ of *free term variables* of a term t is defined inductively as follows.

$$\begin{array}{rcl} FTeV(x) & = & x \\ FTeV(tt') & = & FTeV(t) \cup FTeV(t') \\ FTeV(\lambda x\!:\!T.t) & = & FTeV(t) \setminus \{x\} \end{array}$$

A term variable that is not free is said to be *bound*. A term with no free term variables is *closed*.

Two terms which differ only in the names of bound variables are said to be α*-equivalent*. Terms that are α-equivalent are regarded as identical. However, one still talks about α*-conversion*, *i.e.,* renaming bound variables in a term. One is thus actually talking about equivalence classes of terms. For our purposes we adopt the usual custom of ignoring such details.

The *substitution* $t[u/x]$ of u for the free occurrences of term variable x in t is defined as follows for $x \neq y$.

$$\begin{array}{rcl} x[u/x] & = & u \\ y[u/x] & = & y \\ (tt')[u/x] & = & (t[u/x])(t'[u/x]) \\ (\lambda y\!:\!T.t)[u/x] & = & \lambda y\!:\!T.(t[u/x]) \\ (\lambda x\!:\!T.t)[u/x] & = & \lambda x\!:\!T.t \end{array}$$

This definition of substitution only works if we obey *Barendregt's variable convention*, namely that bound variables are always chosen to differ from free variables in a term. This is justified by α-equivalence and greatly simplifies the presentation. Adopting this convention also means sweeping certain issues under a thick carpet (as does adopting α-equivalence itself), but see (Vestergaard and Brotherston, 2001b,a). Again we follow custom and claim that the simplified view works and is correct for our purposes. We may write $t[u]$ instead of $t[u/x]$ if it is clear what is meant. We also write $t[x]$ to indicate a possible occurrence of variable x in term t.

The *well-formed* expressions of the language are now given by term formation rules. A *term context* $\Gamma = \{x_1 \!:\! T_1, \ldots x_k \!:\! T_k\}$ is a set where any term variable x_i occurs at most once. We write $\Gamma, x \!:\! T$ meaning $\Gamma \cup x \!:\! T$ where x does not occur in Γ. A *typing judgement* has the form $\Gamma \rhd t \!:\! T$, for a context Γ, term t and type T. Term formation is then given by the following inference rules.

$$\textit{te-var}: \qquad x \!:\! T \rhd x \!:\! T$$

$$\textit{te-weak}: \qquad \frac{\Gamma \rhd t \!:\! T'}{\Gamma, x \!:\! T \rhd t \!:\! T'}$$

$$\textit{te-abstr}: \qquad \frac{\Gamma, x \!:\! T' \rhd t \!:\! T}{\Gamma \rhd \lambda x \!:\! T'.t : T' \to T}$$

$$\textit{te-app}: \qquad \frac{\Gamma \rhd t \!:\! T' \to T, \quad \Gamma \rhd t' \!:\! T'}{\Gamma \rhd tt' \!:\! T}$$

We use typing judgements also for meta-level statements. Hence we might write $\Gamma \rhd t \!:\! T$ or perhaps just $t \!:\! T$ meaning 't has type T (in context Γ)'; more precisely this really means '$\Gamma \rhd t \!:\! T$ is derivable by the term formation inference rules'.

Finally for operationality in the calculus, there is the *β-conversion* schema:

$$\beta : \quad (\lambda x \!:\! T'.t[x])t' \ \rightsquigarrow_\beta \ t[t']$$

One also internalises a meta statement concerning β-conversion by including the *η-conversion* schema:

$$\eta : \quad (\lambda x \!:\! T'.tx) \ \rightsquigarrow_\eta \ t$$

for $x \notin FTeV(t)$. The relation $\rightsquigarrow_{\beta\eta}$ relates t and t' if t can be converted to t' by either the β rule or the η rule. The reflexive-transitive closure of $\rightsquigarrow_{\beta\eta}$ is written as $\rightsquigarrow^*_{\beta\eta}$. We have that $\rightsquigarrow^*_{\beta\eta}$ is strongly normalising. The reflexive-transitive closure $\Re^*$ of a non-reflexive relation $\Re$ is *strongly normalising* if

$$\forall t. \exists ! u. (t \Re^* u \ \wedge \ \neg \exists v. u \Re v)$$

This comprises a schematic simply-typed lambda calculus. Viewing $\rightsquigarrow^*_{\beta\eta}$ as computation, this schema is only capable of terminating computations. Depending on the collections TyC and TeC and additional contraction rules, we get different instances and extensions of this schema giving particular, possibly non-terminating simply-typed lambda calculi. For example, with $TyC = \{\mathsf{Nat},\ \mathsf{Bool}\}$ and $TeC = \{\mathsf{0} : \mathsf{Nat},\ \mathsf{succ} : \mathsf{Nat} \rightarrow \mathsf{Nat},\ \mathsf{true} : \mathsf{Bool},\ \mathsf{false} : \mathsf{Bool}\} \cup Comb$, where $Comb$ contains $\mathsf{R}_T : (T \rightarrow (T \rightarrow \mathsf{Nat} \rightarrow T) \rightarrow \mathsf{Nat} \rightarrow T)$ and $\mathsf{D}_T : (T \rightarrow T \rightarrow \mathsf{Bool} \rightarrow T)$ for every type T, and additional conversion rules

$$\begin{array}{rcl} \mathsf{R}_T tu\mathsf{0} & \rightsquigarrow & t \\ \mathsf{R}_T tu(\mathsf{succ} x) & \rightsquigarrow & u(\mathsf{R}_T tux)x \end{array}$$

$$\begin{array}{rcl} \mathsf{D}_T tt'\mathsf{true} & \rightsquigarrow & t \\ \mathsf{D}_T tt'\mathsf{false} & \rightsquigarrow & t' \end{array}$$

we get Gödel's System T. This simply-typed lambda calculus is still strongly normalising. The schema for primitive recursion is given by $\mathsf{R}_{\mathsf{Nat}}$, so this system can express every primitive recursive function. However, since we have 'primitive recursion' at every type, we can represent much more than the primitive recursive functions on $\mathbb{N}$, but all represented functions must be terminating, *i.e., total*. The functions on $\mathbb{N}$ that are representable in System T are exactly those that are provably total in *first-order* Peano arithmetic.

A variant is obtained by replacing R_T and its conversion rules with the *iterator* $\mathsf{It}_T : (T \rightarrow (T \rightarrow T) \rightarrow \mathsf{Nat} \rightarrow T)$ for every type T, with conversion rules

$$\begin{array}{rcl} \mathsf{It}_T tu\mathsf{0} & \rightsquigarrow & t \\ \mathsf{It}_T tu(\mathsf{succ} x) & \rightsquigarrow & u(\mathsf{It}_T tux) \end{array}$$

In this calculus one can define Ackermann's function which grows quicker than any primitive recursive function. The iterators It_T can be defined from the primitive recursors R_T. If one has pairing, then the primitive recursors can conversely be defined from the iterators.

Finally, if one instead adds the *fixed-point operator* $\mathsf{Y}_T : ((T \rightarrow T) \rightarrow T)$ for every type T with conversion rule

$$\mathsf{Y}_T f \ \rightsquigarrow \ f(\mathsf{Y}_T f)$$

one gets the full power of general partial recursion, which is of course not in general terminating. This simply-typed lambda calculus corresponds loosely to PCF (Plotkin, 1977).

A lambda calculus can provide the non-logical syntax for a logic. Thus one may reason formally about programs. We are indeed interested in devising such a logic, but we will defer this until Sect. 3.3.

3.2.2 The Simply-Typed Lambda Calculus with Inductive Types

To cater for user-defined types one can enhance the simply-typed lambda calculus with a means for defining types inductively.

The syntax schema for an *inductive type* definition is

$$\begin{array}{rl}
\textbf{IndType} & \alpha \ \ \textbf{generated by} \\
& g_1 : T_{11} \to \cdots \to T_{1n_1} \to \alpha \\
\textbf{and} & g_2 : T_{21} \to \cdots \to T_{2n_2} \to \alpha \\
\vdots & \\
\textbf{and} & g_m : T_{m1} \to \cdots \to T_{mn_m} \to \alpha
\end{array}$$

This comes with an iteration schema

$$\begin{array}{rl}
\mathsf{It}\alpha_T : \alpha & \to (T_{11} \to \cdots \to T_{1n_1} \to \alpha)[T/\alpha] \\
& \to (T_{21} \to \cdots \to T_{2n_2} \to \alpha)[T/\alpha] \\
& \vdots \\
& \to (T_{m1} \to \cdots \to T_{mn_m} \to \alpha)[T/\alpha] \\
& \to T
\end{array}$$

where $T'[T/\alpha]$ denotes T' with all occurrences of α replaced by T. There are m conversion rules for $\mathsf{It}\alpha_T$. They have the form

$$\mathsf{It}\alpha_T(g_i t_{i1} \cdots t_{in_i}) f_1 \cdots f_m \quad \rightsquigarrow \quad f_i(t_{i1} \cdots t_{in_i})[(\mathsf{It}\alpha_T t f_1 \cdots f_m)/t]$$

where $t'[(\mathsf{It}\alpha_T t f_1 \cdots f_m)/t]$ denotes a multiple term replacement, resulting in t' with every subterm $t : \alpha$ replaced by $\mathsf{It}\alpha_T t f_1 \cdots f_m$.

It is assumed that α, when viewed as a type variable, only occurs positively in any T_{ij} above. A type variable occurs *positively* in a type T if it is in the set $TPos(T)$ defined by

$$\begin{array}{rcll}
TPos(X) & = & \{X\} & \text{for } X \text{ a type variable} \\
TPos(T' \to T) & = & TNeg(T') \cup TPos(T) &
\end{array}$$

where the set of *negatively* occurring variables $TNeg(T)$ is defined by

$$\begin{array}{rcll}
TNeg(X) & = & \emptyset & \text{for } X \text{ a type variable} \\
TNeg(T' \to T) & = & TPos(T') \cup TNeg(T) &
\end{array}$$

We have not yet introduced type variables, but the idea should be clear. The insistence of positivity here is related to the situation for complete lattices, where the condition of *covariance* for any function on the lattice ensures the existence of least and greatest fixed points, by Tarski's fixed point theorem.

Any first-order algebraic signature Σ as described in Ch. 2 can be translated into an inductive type (Böhm and Berarducci, 1985).

Here is an example of all this.

$$\begin{array}{rl} \textbf{IndType} & \mathit{list}\mathsf{Nat}\ \ \textbf{generated by} \\ & \mathsf{nil} : \mathit{list}\mathsf{Nat} \\ \textbf{and} & \mathsf{cons} : \mathsf{Nat} \rightarrow \mathit{list}\mathsf{Nat} \rightarrow \mathit{list}\mathsf{Nat} \end{array}$$

The iteration schema is

$$\begin{array}{rl} \mathsf{It}\mathit{list}\mathsf{Nat}_T : \mathit{list}\mathsf{Nat} & \rightarrow T \\ & \rightarrow \mathsf{Nat} \rightarrow T \rightarrow T \end{array}$$

and the conversion rules are

$$\begin{array}{rcl} \mathsf{It}\mathit{list}\mathsf{Nat}_T \mathsf{nil} f_{\mathsf{nil}} f_{\mathsf{cons}} & \rightsquigarrow & f_{\mathsf{nil}} \\ \mathsf{It}\mathit{list}\mathsf{Nat}_T (\mathsf{cons} n l) f_{\mathsf{nil}} f_{\mathsf{cons}} & \rightsquigarrow & f_{\mathsf{cons}} n (\mathsf{It}\mathit{list}\mathsf{Nat}_T l f_{\mathsf{nil}} f_{\mathsf{cons}}) \end{array}$$

We have delayed presenting the abstract syntax. It goes as follows. Let TyN be a collection of *type names* and TeG be a collection of *constructor names*. The abstract syntax of the simply-typed lambda calculus with inductive types is given by the following grammars.

$$\begin{array}{lccl} \text{(programs)} & P & ::= & t \mid I\ P \\ \text{(inductive type def)} & I & ::= & \textbf{IndType}\ \alpha \mid \textbf{IndType}\ \alpha\ \textbf{generated by}\ G \\ \text{(constructors)} & G & ::= & g\!:\!T \mid g\!:\!T\ \textbf{and}\ G \\ \text{(types)} & T & ::= & \alpha \mid C \mid (T \rightarrow T) \\ \text{(terms)} & t & ::= & g \mid c \mid x \mid (\lambda x\!:\!T.t) \mid (tt) \end{array}$$

for all $\alpha \in TyN$, $g \in TeG$, $C \in TyC$, $c \in TeC$, and $x \in TeV$.

Theoretically one can justify rendering $TyC = TeC = \emptyset$, since one can define the usual built-in types inductively. For example

$$\begin{array}{rl} \textbf{IndType} & \mathit{nat}\ \ \textbf{generated by} \\ & \mathsf{0} : \mathit{nat} \\ \textbf{and} & \mathsf{succ} : \mathit{nat} \rightarrow \mathit{nat} \end{array}$$

and

$$\begin{array}{rl} \textbf{IndType} & \mathit{bool}\ \ \textbf{generated by} \\ & \mathsf{true} : \mathit{bool} \\ \textbf{and} & \mathsf{false} : \mathit{bool} \end{array}$$

with corresponding iterators and conversion rules given by the above schemata. One can furthermore define products and sums or coproducts inductively with constructors pair, and inl, inr respectively. The unit type with one inhabitant and the empty type with no inhabitants can also be defined inductively, although degenerately. Destructors such as pred for *nat*, car and cdr for *list*, fst and snd for products and case for sums, are defined using the iteration schemes, although some of these need pairs.

3.2.3 The Polymorphic Lambda Calculus

Through the Curry-Howard-Feys correspondence, the lambda calculus gets both a logical and a computational motivation. In this respect, the polymorphic lambda calculus originated in logic with (Girard, 1971), and originated computationally with (Reynolds, 1974). We will continue to focus on the computational aspect.

The polymorphic lambda calculus is an extension of the simply-typed lambda calculus. This is clearer in the framework of PTSs, but in essence the extension consists in the ability to abstract over type variables and in being able to define types of higher *kinds*. One may think of this being done by including a lambda calculus over types. For TyV a collection of *type variables*, the *polymorphic lambda calculus* has the following abstract syntax.

$$\begin{array}{lll} \text{(kinds)} & K & ::= \; * \mid (K \to K) \\ \text{(types)} & T & ::= \; X \mid (T \to T) \mid (\forall X : K.T) \mid (\lambda X : K.T) \mid (TT) \\ \text{(terms)} & t & ::= \; x \mid (\lambda x : T.t) \mid (tt) \mid (\Lambda X : K.t) \mid (tT) \end{array}$$

for all $X \in TyV$ and $x \in TeV$.

Conventions about parentheses and association are analogous to the ones for the simply typed calculus, with obvious additions. For example, we may write $\Lambda X_1 : K_1, X_2 : K_2, \ldots, X_n : K_n.t \;:\; \forall X_1 : K_1, X_2 : K_2, \ldots, X_n : K_n.T$ instead of $(\Lambda X_1 : K_1.(\Lambda X_2 : K_2.(\ldots\Lambda X_n : K_n.t)\ldots)) : (\forall X_1 : K_1.(\forall X_2 : K_2.(\ldots\forall X_n : K_n.T)\ldots))$.

The only variable binder for the simply-typed lambda calculus is λ, and this binder determines the definition of free term variables, α-equivalence, and ultimately substitution. Variable binders now include Λ, $\forall$, and λ for types. The definition of the collection $FTeV(t)$ of free term variables of a term t is extended in the obvious manner, and the collections $FTyV(t)$ and $FTyV(T)$ of *free type variables* in term t and type T are defined analogously to $FTeV$. Again, a variable that is not free is said to be *bound*. A *closed term* has no free term or type variables. A *closed* type has no free type variables. Types have no free term variables because we do not have (term-)dependent types.

The definition of *α-equivalence* extends to the new variable binders in the obvious way, as does the definition of substitution, again under the assumption of Barendregt's variable convention; the latter now for term and type variables.

To define the well-formed expressions, we must now employ *kinding judgements* and type formation rules as well as typing judgements and term formation rules. A *type context* $\Delta = \{X_1 : K_1, \dots X_k : K_k\}$ is a set where any type variable X_i occurs at most once. We write $\Delta, X : K$ meaning $\Gamma \cup X : K$ where X does not occur in Δ. Type formation is given by the following inference rules.

$$\textit{ty-var}: \qquad X : K \rhd X : K$$

$$\textit{ty-weak}: \qquad \frac{\Delta \rhd T : K'}{\Delta, X : K \rhd T : K'}$$

$$\textit{ty-abstr}: \qquad \frac{\Delta, X : K' \rhd T : K}{\Delta \rhd \lambda X : K'.T : K' \to K}$$

$$\textit{ty-app}: \qquad \frac{\Delta \rhd T : K' \to K, \quad \Delta \rhd T' : K'}{\Delta \rhd TT' : K}$$

$$\textit{ty-}{\to}: \qquad \frac{\Delta \rhd T : *, \quad \Delta \rhd T' : *}{\Delta \rhd T \to T' : *}$$

$$\textit{ty-}\forall: \qquad \frac{\Delta, X : K \rhd T : *}{\Delta \rhd \forall X : K.T : *}$$

Again we will use kinding and typing judgements also for meta-level statements. There is *βη-conversion* for types:

$$\beta_T: \quad (\lambda X : K'.T[X])T' \leadsto_{\beta_T} T[T']$$

$$\eta_T: \quad (\lambda X : K.TX) \leadsto_{\eta_T} T$$

for $X \notin FTyV(T)$ for the latter.

Typing judgements for term formation must now include kinding judgements for the types involved in the typing judgement. For example, the typing judgement $f : X \to X \rhd f : X \to X$ is only valid when $X : * \rhd X \to X : *$. Thus a typing judgement will have the form $\Delta \mid \Gamma \rhd t : T$, where each type U occurring in Γ is such that $\Delta \rhd U : K$ for some kind K, and $\Delta \rhd T : *$. Terms can only be of types of kind $*$. As before, a context $\Gamma = \{x_1 : T_1, \dots x_k : T_k\}$ is a set where any x_i occurs at most once, but now with the above proviso on the T_is. The term formation rules go like this:

$$\textit{te-var}: \qquad \Delta \mid x{:}T \rhd x{:}T$$

$$\textit{te-weak}: \qquad \frac{\Delta \mid \Gamma \rhd t{:}T'}{\Delta \cup \Delta' \mid \Gamma, x{:}T \rhd t{:}T'} \qquad \text{if } \Delta \cup \Delta' \rhd T{:}*$$

$$\textit{te-abstr}: \qquad \frac{\Delta \mid \Gamma, x{:}T' \rhd t{:}T}{\Delta \mid \Gamma \rhd \lambda x{:}T'.t : T' \to T}$$

$$\textit{te-app}: \qquad \frac{\Delta \mid \Gamma \rhd t{:}T' \to T, \quad \Delta \mid \Gamma \rhd t'{:}T'}{\Delta \mid \Gamma \rhd tt'{:}T}$$

$$\textit{te-poly-abstr}: \qquad \frac{\Delta, X{:}K \mid \Gamma \rhd t{:}T}{\Delta \mid \Gamma \rhd \Lambda X{:}K.t : \forall X{:}K.T} \qquad X \textit{ not free in } \Gamma$$

$$\textit{te-poly-app}: \qquad \frac{\Delta \mid \Gamma \rhd t : \forall X{:}K'.T, \quad \Delta \rhd T'{:}K'}{\Delta \mid \Gamma \rhd tT'{:}T[T'/X]}$$

$$\textit{te-type-eq}: \qquad \frac{\Delta \mid \Gamma \rhd t{:}T, \quad T \leadsto_{\beta_T\eta_T} T', \quad \Delta \rhd T'{:}*}{\Delta \mid \Gamma \rhd t{:}T'}$$

We have now the following extended $\beta\eta$-conversion schemata:

$$\begin{array}{rrcl} \beta: & (\lambda x{:}T'.t[x])t' & \leadsto_\beta & t[t'] \\ & (\Lambda X{:}K.t[X])T & \leadsto_\beta & t[T] \end{array}$$

$$\begin{array}{rrcl} \eta: & (\lambda x{:}T'.tx) & \leadsto_\eta & t \\ & (\Lambda X{:}K.tX) & \leadsto_\eta & t \end{array}$$

where $x \notin FTeV(t)$ and $X \notin FTyV(t)$ for the latter. We still have that $\leadsto^*_{\beta\eta}$ is strongly normalising. And, $\leadsto_{\beta_T\eta_T}$ is strongly normalising too.

The calculus F_ω and its language are extremely expressive, and contains every subcalculus and sublanguage F_i, $1 \leq i \leq \omega$, where F_i is defined as follows. First, define the order $KOrd(K)$ of kind K as

$$KOrd(*) = 1$$

$$KOrd(K' \to K) = max(KOrd(K') + 1, KOrd(K))$$

Then the n^{th}-order polymorphic lambda calculus F_n is built from those terms from F_ω whose types can be derived without mentioning kinds of order n or greater. The calculus F_1 is the simply-typed lambda calculus with $TyC = TeC = \emptyset$.

We will mainly be interested in F_2, and later on F_3. The calculus F_2 has the abstract syntax given by

$$\begin{array}{llll} \text{(kinds)} & K & ::= & * \\ \text{(types)} & T & ::= & X \mid (T \to T) \mid (\forall X{:}*.T) \\ \text{(terms)} & t & ::= & x \mid (\lambda x{:}T.t) \mid (tt) \mid (\Lambda X{:}*.t) \mid (tT) \end{array}$$

for all $X \in TyV$ and $x \in TeV$. For F_2 it is customary to omit the kind $*$. The calculus F_2 is commonly known as System F. Calculus F_3 has abstract syntax

$$\begin{array}{llll} \text{(kinds)} & K & ::= & * \mid * \to K \\ \text{(types)} & T & ::= & X \mid (T \to T) \mid (\forall X\!:\!K.T) \mid (\lambda X\!:\!K.T) \mid (TT) \\ \text{(terms)} & t & ::= & x \mid (\lambda x\!:\!T.t) \mid (tt) \mid (\Lambda X\!:\!K.t) \mid (tT) \end{array}$$

for all $X \in TyV$ and $x \in TeV$.

3.2.4 Inductive Types

We now delve a bit deeper into the second-order polymorphic lambda calculus. The abstract syntax and well-formed expressions are given in the previous section.

A main motivation for type abstraction is of course the economical desire to define polymorphic functionals that capture a behaviour that happens to be uniform across all types. Then, such a functional can be instantiated at any type to give the particular version of the functional at that type. For example, with polymorphism one can define a general function-composition operator by

$$\mathit{comp} \stackrel{\mathit{def}}{=} \Lambda X, Y, Z.\lambda f\!:\!X \to Y.\lambda g\!:\!Y \to Z.\lambda x\!:\!X.g(fx)$$

For types A, B and C, the instance $\mathit{comp}ABC$ is the functional that receives two functions of types $A \to B$ and $B \to C$ as arguments and returns their composite.

There is however another interesting aspect to polymorphism, and that is that one can express inductive types as given in 3.2.2 directly. The general translation is found in (Böhm and Berarducci, 1985). Here we give some examples.

Recall that natural numbers were represented by the inductive type

$$\begin{array}{rl} \textbf{IndType} & \mathit{nat}\ \ \textbf{generated by} \\ & \mathsf{0} : \mathit{nat} \\ \textbf{and} & \mathsf{succ} : \mathit{nat} \to \mathit{nat} \end{array}$$

The encoding in System F is now

$$\mathsf{Nat} \stackrel{\mathit{def}}{=} \forall X.X \to (X \to X) \to X$$

Here the universally quantified X stands for the name nat in the inductive definition. This name is abstract; any other name will of course do, since it is the inductive structure that determines the type. This abstractness is here reflected by polymorphism. The generated inhabitants are now closed terms of the form

$$\bar{n} \stackrel{\mathit{def}}{=} \Lambda X.\lambda z\!:\!X.\lambda s\!:\!X \to X.s^n z$$

where $s^n z$ stands for n applications of s to z. These polymorphic Church-numerals are the only closed terms of type Nat. The constructors are now

$$\mathsf{0} \stackrel{def}{=} \Lambda X.\lambda z\!:\!X.\lambda s\!:\!X \to X.z$$

$$\mathsf{succ} \stackrel{def}{=} \lambda n\!:\!\mathsf{Nat}.\Lambda X.\lambda z\!:\!X.\lambda s\!:\!X \to X.(s(nXzs))$$

The iterators $\mathsf{It}nat_T$ for every type T can now be replaced by a single polymorphic iterator $\mathsf{ItNat}\!:\!\forall Y.\mathsf{Nat} \to Y \to (Y \to Y) \to Y$ defined in System F by

$$\mathsf{ItNat} \stackrel{def}{=} \Lambda Y.\lambda x\!:\!\mathsf{Nat}.\lambda z\!:\!Y.\lambda s\!:\!Y \to Y.xYzs$$

The iterator conversion rules are just given by β-conversion. In actual fact, iterators are not necessary, because we have $\mathsf{ItNat}Txzs \rightsquigarrow_\beta xTzs$, which means we can replace any occurrence of $\mathsf{ItNat}Tx$ by xT. In other words, any $x\!:\!\mathsf{Nat}$ has iteration already built in. This principle holds for every inductive type. For example, using the iterator, we can define

$$\mathsf{ifzero} \stackrel{def}{=} \lambda n\!:\!\mathsf{Nat}\ .\ \mathsf{ItNat}(\mathsf{Bool})(n)(\mathsf{true})(\lambda b\!:\!\mathsf{Bool}.\mathsf{false})$$

But we can also go straight ahead and define

$$\mathsf{ifzero} \stackrel{def}{=} \lambda n\!:\!\mathsf{Nat}\ .\ n(\mathsf{Bool})(\mathsf{true})(\lambda b\!:\!\mathsf{Bool}.\mathsf{false})$$

Booleans are now represented by

$$\mathsf{Bool} \stackrel{def}{=} \forall X.X \to X \to X$$

with constructors

$$\mathsf{true} \stackrel{def}{=} \Lambda X.\lambda t\!:\!X.\lambda f\!:\!X.t$$

$$\mathsf{false} \stackrel{def}{=} \Lambda X.\lambda t\!:\!X.\lambda f\!:\!X.f$$

The boolean iterator is the conditional, or if-then-else construct. Since iteration is built in, we can simply define

$$\mathsf{cond} \stackrel{def}{=} \Lambda Y.\lambda b\!:\!\mathsf{Bool}.bY \ :\ \forall Y.\mathsf{Bool} \to Y \to Y \to Y$$

We can also give a lifted boolean type

$$\mathsf{Bool}_\perp \stackrel{def}{=} \forall X.(X \to X \to X \to X)$$

with constructors

$$\mathsf{true} \stackrel{def}{=} \Lambda X.\lambda t\!:\!X.\lambda f\!:\!X.\lambda b\!:\!X\ .\ t \qquad \mathsf{false} \stackrel{def}{=} \Lambda X.\lambda t\!:\!X.\lambda f\!:\!X.\lambda b\!:\!X\ .\ f$$

$$\mathsf{bot} \stackrel{def}{=} \Lambda X.\lambda t{:}X.\lambda f{:}X.\lambda b{:}X\ .\ b$$

Lists of items of type U are given by

$$\mathsf{List}_U \stackrel{def}{=} \forall X.X \to (U \to X \to X) \to X$$

where $X \notin FTyV(U)$, with constructors

$$\mathsf{nil}_U \stackrel{def}{=} \Lambda X.\lambda n{:}X.\lambda c{:}U \to X \to X.n$$

$$\mathsf{cons}_U \stackrel{def}{=} \lambda u{:}U.\lambda l{:}\mathsf{List}_U.\Lambda X.\lambda n{:}X.\lambda c{:}U \to X \to X.(cu(lXnc))$$

Binary products are encoded in System F as inductive types by

$$U \times V \stackrel{def}{=} \forall X.((U \to V \to X) \to X)$$

where $X \notin FTyV(U) \cup FTyV(V)$, with constructor $\mathsf{pair}_{U,V} : U \to V \to U \times V$ defined by

$$\mathsf{pair}_{U,V} uv \stackrel{def}{=} \Lambda X.\lambda f{:}U \to V \to X.fuv$$

We also have destructors $\mathsf{fst}_{U,V} : U \times V \to U$ and $\mathsf{snd}_{U,V} : U \times V \to V$, defined by

$$\mathsf{fst}_{U,V}(x) = xU(\lambda x{:}U.\lambda y{:}V.x)$$

$$\mathsf{snd}_{U,V}(x) = xV(\lambda x{:}U.\lambda y{:}V.y)$$

The empty product 1, or *Unit* in other programming languages, is defined as

$$1 \stackrel{def}{=} \forall X.X \to X$$

with single constructor

$$\star \stackrel{def}{=} \Lambda X.\lambda x{:}X.x$$

Binary sums or coproducts are encoded as

$$U + V \stackrel{def}{=} \forall X.((U \to X) \to (V \to X) \to X)$$

where $X \notin FTyV(U) \cup FTyV(V)$, with constructors $\mathsf{inl}_{U,V} : U \to U + V$ and $\mathsf{inr}_{U,V} : V \to U + V$, defined by

$$\mathsf{inl}_{U,V}(u) = \Lambda X.\lambda f{:}U \to X.\lambda g{:}V \to X.f(u)$$

$$\mathsf{inr}_{U,V}(v) = \Lambda X.\lambda f{:}U \to X.\lambda g{:}V \to X.g(v)$$

and destructor $\mathsf{case}_{U,V} : \forall X.(U \to X) \to (V \to X) \to (U + V) \to X$ defined by

$$\mathsf{case}_{U,V} Wfgx = xWfg$$

The empty sum 0, or *Void* in other programming languages, is defined by

$$0 \stackrel{def}{=} \forall X.X$$

There are no closed inhabitants of this type.

Binary products generalise to n-ary products with constructor $\mathsf{tuple}_{\boldsymbol{U}}$ and destructors $\mathsf{proj}_{\boldsymbol{U}i}$. Binary sums generalise to n-ary coproducts with constructors $\mathsf{inj}_{\boldsymbol{U}i}$ and destructor $\mathsf{case}_{\boldsymbol{U}}$.

We can define destructors pred for Nat, and car_U and cdr_U for List_U. However, pred and cdr_U do not in fact lend themselves to simple definitions in terms of iteration. These destructors actually need primitive recursion. Primitive recursion schemata can be defined in terms of iteration and pairing (Church, 1941), see *e.g.,* (Pierce et al., 1989).

We can be precise about the class of functions on $\mathbb{N}$ that are representable in System F. Because System F is strongly normalising, all functions represented must be total. The functions on $\mathbb{N}$ representable in System F are exactly those provably total in *second-order* Peano arithmetic.

3.2.5 Abstract Data Types in System F

As stated earlier, we use the term 'algebra' in a broad sense meaning an entity consisting of a data representation together with operations on that data representation. An abstract data type (ADT) is then a collection of algebras sharing some given common characteristics, with each individual algebra being an instance of the ADT. The idea is that a program may rely only on the abstract characteristics associated with the ADT, rather than the characteristics of any individual implementation.

In the context of algebraic specification, we said that an ADT is a class of Σ-algebras satisfying an algebraic specification SP.

In our present type-theoretical context, we will eventually mimic this. In System F it is possible to internalise the semantic notions of algebra and specification refinement into syntax. Algebras will be internalised as terms of existential type. This is due to (Mitchell and Plotkin, 1988), where internalised algebras are called *data algebras*. It is then tempting to call existential types abstract data types, and we could indeed do that. Abstract data types should provide information hiding in the sense outlined in Ch. 1. We will see that existential types provide some very useful information hiding. However, we will later look again at specifications which will add a further level of information to interfaces, and we reserve the term 'abstract data type' for the analogous notion from algebraic specification, namely the collection of realisations for a specification. Existential types will instead be called *abstract types* (Mitchell and Plotkin, 1988).

Existential types and pack and unpack combinators are encoded as follows.

$$\exists X.T[X] \stackrel{def}{=} \forall Y.(\forall X.(T[X] \to Y) \to Y)$$
$$\text{where } Y \text{ does not occur in } T[X]$$

$$\mathsf{pack}_{T[X]} : \forall X.(T[X] \to \exists X.T[X])$$
$$\mathsf{pack}_{T[X]}(A)(opns) \stackrel{def}{=} \Lambda Y.\lambda f : \forall X.(T[X] \to Y).f(A)(opns)$$

$$\mathsf{unpack}_{T[X]} : (\exists X.T[X]) \to \forall Y.(\forall X.(T[X] \to Y) \to Y)$$
$$\mathsf{unpack}_{T[X]}(package)(B)(client) \stackrel{def}{=} package(B)(client)$$

We omit subscripts to pack and unpack as much as possible. Operationally, pack packages a data representation and an implementation of operations on that data representation to give a data algebra of the existential type. The resulting package is a polymorphic functional that given a client computation and its result domain, instantiates the client with the particular elements of the package. The unpack combinator is the application operator for pack.

Example 3.1 An abstract type for stacks could be

$$\exists X.(X \times (Z \to X \to X) \times (X \to X) \times (X \to Z \to Z))$$

A data type of this type is for example ($\mathsf{pack}\,\mathsf{List}_Z\, l$), where

$$\begin{array}{ll} empty: & (\mathsf{proj}_1\, l) = \mathsf{nil} \\ push: & (\mathsf{proj}_2\, l) = \mathsf{cons} \\ pop: & (\mathsf{proj}_3\, l) = \lambda y : \mathsf{List}_Z.(\mathsf{cond}\,\mathsf{List}_Z\,(\mathsf{isnil}\, y)\,\mathsf{nil}\,(\mathsf{cdr}\, y)) \\ top: & (\mathsf{proj}_4\, l) = \lambda y : \mathsf{List}_Z.\lambda z : Z.(\mathsf{cond}\, Z\,(\mathsf{isnil}\, y)\, z\,(\mathsf{car}\, y)) \end{array}$$

where the parameter z for *top* is a default value in case of an empty stack, and

$$\mathsf{isnil} \stackrel{def}{=} \lambda y : \mathsf{List}_Z\ .\ y \mathsf{Bool}\,\mathsf{true}\,(\lambda b : \mathsf{Bool}.\mathsf{false})$$

returns true when y is nil, and false otherwise. ○

3.2.6 Abstraction Barriers

We now arrive at an essential observation concerning central points in this thesis. Existential types together with the pack and unpack combinators embody a crucial abstraction barrier. First, any client computation $f : \forall X.(T[X] \to Y)$ is

η-equivalent to a term of the form $\Lambda X.\lambda x\!:\!T[X].t[X,x]$. Borrowing terminology from SIMULA (Dahl and Nygaard, 1966), we here refer to X as a *virtual data representation*, x as a collection of *virtual operations*, and the whole computation as a *virtual computation*. Any instance $(f\,A\,a)$ of the computation with an *actual data representation* A, and *actual operations* a then gives an *actual computation*.

A crucial observation is now embodied in the following obvious statement.

***Abs-Bar1*:** A virtual client computation of the form $\Lambda X.\lambda x\!:\!T[X].t[X,x]$ cannot have free variables of types involving the virtual data representation X.

The direct reason for ***Abs-Bar1*** is that the type variable X is bound and hence cannot occur in the context in order to justify any free variables. For example, if $X, x\!:\!T[X] \;\triangleright\; (\mathsf{proj}_i\,x)\!:\!X \to \mathsf{Nat}$, then by Barendregt's variable convention,

$$y\!:\!X \;\triangleright\; \Lambda X.\lambda x\!:\!T[X].(\mathsf{proj}_i\,x)(y) : \forall X.(T[X] \to \mathsf{Nat})$$

is ill-formed. The only way a client computation may compute over types containing the virtual data representation X is by accessing virtual operations in the supplied collection x of operations. For example, if $X, x\!:\!T[X] \triangleright (\mathsf{proj}_j\,x)\!:\!X$ then

$$\triangleright\; \Lambda X.\lambda x\!:\!T[X].(\mathsf{proj}_i\,x)(\mathsf{proj}_j\,x) : \forall X.(T[X] \to \mathsf{Nat})$$

is indeed a virtual client computation. Furthermore, due to ***Abs-Bar3*** below, the only way a package can be used is via client computations adhering to the above.

This in turn determines how packages may be used in actual computations. Users of a package $(\mathsf{pack} A a) : \exists X.T[X, \boldsymbol{Z}]$ may only apply operations from a to arguments according to the structure of virtual computations $f\!:\!\forall X.T[X,\boldsymbol{Z}] \to U$.

First, this means that up to $\beta\eta$-normal form, variables of types involving the actual data representation, cannot be arguments.

Example 3.1 (continued) The operation *pop*, *i.e.,* $(\mathsf{proj}_3\,l)$, can in $\beta\eta$-normal form never be applied to, say, $(push\;z\;y)$ for variable $y\!:\!\mathsf{List}_Z$, because there is no way we can form a normalised virtual computation giving rise to this application. Note that for $g \stackrel{def}{=} \lambda z\!:\!Z.\lambda y\!:\!X.(\mathsf{proj}_2 x\;z\;y))$, we can write

$$f \stackrel{def}{=} \Lambda X.\lambda x\!:\!(X \times (Z \to X \to X) \times (X \to X) \times (X \to Z \to Z)).(\mathsf{proj}_3 x\,(g\;z\;\mathsf{proj}_1 x))$$

which, in the actual computation $(f\mathsf{List}_Z l)$ yields the application of a package operator to a variable of the actual data representation List_Z in $(\mathsf{proj}_2 l\;z\;y) = (push\;z\;y)$. However, this computation is not in normal form.

In fact, up to $\beta\eta$-normal form, any argument of type List_Z to operations in l must be formed by *empty*, *push*, *pop*, *top*, and variables of type Z. For example, *pop* can be applied to (*push* z *empty*) via the virtual computation

$$\Lambda X.\lambda x\!:\!(X \times (Z \to X \to X) \times (X \to X) \times (X \to Z \to Z)).(\mathsf{proj}_3 x\,(\mathsf{proj}_2 x\; z\;\; \mathsf{proj}_1 x))$$

○

Packages can only be used by applying them via virtual computations. Therefore, up to $\beta\eta$-normal form, users may only apply package operations from a package $(\mathsf{pack} A a)$ to arguments that are definable from package operations, *i.e.,* to terms $t[A, a]$ such that $t[X, x]$ has no free variables of types involving X. Thus any actual computation $(f\,A\,a)$ cannot directly invoke arbitrary operations on types over A, but only those operations that are in a sense expressible using the supplied operations in a and term formation. This is the *definability* aspect of ***Abs-Bar1***. This aspect is essential in restricting the data representation to valid values, for example, when implementing sets by a package whose data representation is List_Z, but where the supplied operations assume set constructors that build sorted lists. The definability aspect ensures that users of the package cannot directly supply unsorted lists to any operation.

Note however, that a supplied package operation might itself make calls involving arguments of types over the actual data representation, which are not expressed in terms of package operations. In this case, it is not true that package operations will only be applied to definable arguments, but crucially, this is at the discretion of the package implementor, and not due to direct user application. We will see an example of this later on in Example 5.7 (*p.* 117) in Ch. 5.

There are two other aspects of the abstraction barrier inherent in the encoding of existential types. The next one is the already stated condition in the definition of the encoding of existential types.

***Abs-Bar2*:** The type Y does not occur in T in the encoding
$$\exists X.T[X] \stackrel{def}{=} \forall Y.(\forall X.(T[X] \to Y) \to Y).$$

This ensures that data types do not depend on their environment of actual usage, other than via explicitly given parameters if $\exists X.T[X]$ has free types. ***Abs-Bar2*** is a prerequisite for plugability.

The third aspect arises from the fact that when using a data type, *i.e.,* a package of existential type, the user must provide the result type of the client computation. This entails the following.

***Abs-Bar3*:** Client computations $f : \forall X.(T[X] \to Y)$ cannot have a result type containing the virtual data representation, *i.e.,* the bound type variable X.

For example, if $X, x\!:\!T[X] \rhd (\mathsf{proj}_j\, x)\!:\!X$ then

$$f \stackrel{def}{=} \Lambda X.\lambda x\!:\!T[X].(\mathsf{proj}_j\, x) : \forall X.(T[X]\!\rightarrow\! X)$$

is in fact not a possible client computation, since in order to use a package $(\mathsf{pack} Aa)$ in f, we must do $(\mathsf{pack} Aa)(C)(f)$, where C is the result type of f, which in this case is the inaccessible virtual data representation type X.

This also means that one cannot use data-type operations anywhere else than within client computations of the form discussed in conjunction with *Abs-Bar1*. For example, if $X, x : T[X] \rhd (\mathsf{proj}_k\, x) : X \rightarrow \mathsf{Nat}$, one might think that given $(\mathsf{pack} Aa)$, one could use $(\mathsf{proj}_k\, a)$ on an arbitrary inhabitant v of A, by doing *e.g.,* $((\mathsf{unpack}(\mathsf{pack} Aa)(A \rightarrow \mathsf{Nat})(\Lambda X.\lambda x : T[X].(\mathsf{proj}_k\, x)))\ v)$, but this term is ill-formed, because the result type of the client computation is $X \rightarrow \mathsf{Nat}$, not $A\!\rightarrow\!\mathsf{Nat}$, and $X\!\rightarrow\!\mathsf{Nat}$ cannot be the result type of a client computation.

Abs-Bar3 entails that the data representation of a data type is completely hidden from the outside. It is not even accessible via data-type supplied operations and is only observable indirectly via operations of types not involving the data representation. This restriction is arguably too strong in some circumstances. If this restriction were dropped, the data representation itself would not be fully hidden, but merely protected. One would have access to the data representation through data type operations, which would still ensure the safe manipulation of the data representation according to *Abs-Bar1*. See (MacQueen, 1985, 1986) for discussions about this in relation to Standard ML.

Variants of these three aspects of the existential type abstraction barrier are to be found in (Mitchell and Plotkin, 1988) for SOL. In this thesis, it is *Abs-Bar1*, backed by *Abs-Bar3*, that will be most instrumental, and that we emphasise here. We will later on refer to *Abs-Bar1*, *Abs-Bar2*, and *Abs-Bar3* jointly as *Abs-Bar*.

Finally, notice how uniformity is enforced across all actual computations arising from a virtual computation, *i.e.,* all actual computations inherit the form of the virtual computation from which they stem. The only information we have for package operations at computation formation time, is the information provided by the abstract type. This is the important *uniformity* aspect of *Abs-Bar* which we will use later on in Ch. 5.

—

3.2.7 Existential Types with Several Bound Variables

Our main discussion will use existential types with a single existentially bound type variable. It is nonetheless straight-forward to deal with multiple existentially bound type variables. The question is however, what we mean by existential types with several existentially bound variables. For example, does $\exists X_1, X_2.T[X_1, X_2]$ mean the System F encoding of $\exists X_1.\exists X_2.T[X_1, X_2]$, *i.e.*,

$$\forall Y.(\forall X_1\ .\ (\exists X_2.T[X_1, X_2])\ \rightarrow Y)\rightarrow Y$$

or does it mean

$$(\forall Y.(\forall X_1, X_2.(T[X_1, X_2]\rightarrow Y)\rightarrow Y)$$

In fact, these forms are equivalent. Consider the two packages

$$(\mathsf{pack}A_1(\mathsf{pack}A_2a)) \qquad \text{and} \qquad (\mathsf{pack}A_1A_2a)$$

Let $f\!:\!\forall X_1, X_2.(T[X_1, X_2]\rightarrow Y)$ be an arbitrary computation. Then for

$$f' \stackrel{def}{=} \Lambda X_1.\lambda u\!:\!\exists X_2.T[X_1, X_2]\ .\ (\mathsf{unpack}(u)C(fX_1))$$

we get

$$\begin{aligned}(\mathsf{pack}A_1(\mathsf{pack}A_2a))Cf' &= f'A_1(\mathsf{pack}A_2a)\\ &= \mathsf{unpack}(\mathsf{pack}A_2a)C(fA_1)\\ &= fA_1A_2a\\ &= (\mathsf{pack}A_1A_2a)Cf\end{aligned}$$

Conversely, let $g\!:\!\forall X_1\ .\ (\exists X_2.T[X_1, X_2])\ \rightarrow Y$ be arbitrary. Then for

$$g' \stackrel{def}{=} \Lambda X_1, X_2.\lambda x\!:\!T[X_1, X_2]\ .\ (gX_1(\mathsf{pack}X_2x))$$

we get

$$\begin{aligned}(\mathsf{pack}A_1A_2a)Cg' &= (g'A_1A_2a)\\ &= gA_1(\mathsf{pack}A_2a)\\ &= (\mathsf{pack}A_1(\mathsf{pack}A_2a))Cg\end{aligned}$$

Thus, $(\mathsf{pack}A_1(\mathsf{pack}A_2a))$ and $(\mathsf{pack}A_1A_2a)$ simulate each other. This extends to variables of the respective forms, through Theorem 3.6.

3.2.8 Algebras and Coalgebras

In System F one can also encode initial algebras and final coalgebras. If $T[X]$ is a type where the variable X occurs only positively, the *initial* $T[X]$-*algebra* and combinators fold and in are encoded by

$$\mu X.T[X] \stackrel{def}{=} \forall X.((T[X] \to X) \to X)$$

$$\mathsf{fold}_{T[X]} : \forall X.((T[X] \to X) \to ((\mu X.T[X]) \to X))$$
$$\mathsf{fold}_{T[X]}(X)(f) \stackrel{def}{=} \lambda x : \mu X.T[X] \ . \ x(X)(f)$$

$$\mathsf{in}_{T[X]} : T[\mu X.T[X]] \to \mu X.T[X]$$
$$\mathsf{in}_{T[X]}(x) \stackrel{def}{=} \Lambda X.\lambda f : T[X] \to X \ . \ f(T[\mathsf{fold}_{T[X]} X f]x)$$

Initial $T[X]$-algebras are an alternative to inductive types. We will return to this issue later, after introducing the logic.

Although we will not use *final coalgebras*, the encodings in System F go as follows. For $T[X]$ where the variable X occurs only positively,

$$\nu X.T[X] \stackrel{def}{=} \exists X.((X \to T[X]) \times X)$$

$$\mathsf{unfold}_{T[X]} : \forall X.((X \to T[X]) \to (X \to (\nu X.T[X])))$$
$$\mathsf{unfold}_{T[X]}(X)(f)(x) \stackrel{def}{=} \mathsf{pack}(X)(\mathsf{pair}(f)(z))$$

$$\mathsf{out}_{T[X]} : \nu X.T[X] \to T[\nu X.T[X]]$$
$$\mathsf{out}_{T[X]}(u) \stackrel{def}{=} \mathsf{unpack}(u)(T[\nu X.T[X]])$$
$$(\Lambda X.\lambda w : ((X \to T[X]) \times X) \ . \ T[\mathsf{unfold}_{T[X]} X (\mathsf{fst} w)]((\mathsf{fst} w)(\mathsf{snd} w)))$$

3.3 The Logic for Parametric Polymorphism

We now have a language for defining programs and data algebras with abstraction barriers. But ultimately we want to be able to define specifications and specification refinement, and to reason formally about properties pertaining to specification refinement. For this we need a logic. Our language is already very powerful since we can internalise traditionally semantic notions in it. The expressiveness now becomes even greater with the introduction of logic. The logic we shall use is the logic for parametric polymorphism due to (Plotkin and Abadi, 1993). This logic is essentially a second-order logic augmented with relation symbols and a syntax for relation definition. The main point of the logic is the assertion of relational parametricity as an axiom schema. This internalises in a formal logic the semantic notion of relational parametricity. Preliminary studies in this direction were done in (Mairson, 1991). Plotkin and Abadi's logic is a logic based on and around System F terms. It is also possible to extend System F itself in order to internalise relational parametricity. This is done in System $\mathcal{R}$ due to (Abadi et al., 1993).

The logic for parametric polymorphism has formulae built using the standard connectives, but now basic predicates are not only equations, but also relation membership statements. The abstract syntax is given by the grammar for System F and the following grammar for formulae:

$$\begin{array}{lll} \text{(formulae)} & \phi & ::= & (t =_A u) \mid R(t,u) \mid \\ & & & (\phi \;\Rightarrow\; \phi) \mid (\forall x\!:\!T.\phi) \mid (\forall X.\phi) \mid (\forall R\!\subset\! A\!\times\! B.\phi) \\ & & & (\exists x\!:\!T.\phi) \mid (\exists X.\phi) \mid (\exists R\!\subset\! A\!\times\! B.\phi) \\ & & & \bot \mid (\phi \;\wedge\; \phi) \mid (\phi \vee \phi) \end{array}$$

where R ranges over relation variables. The last two lines in the abstract syntax for formulae are not strictly necessary, because the connectives introduced there can be defined in terms of the connectives on the second line, but showing these definitions is of less importance for us.

Parentheses will be omitted whenever we can get away with it. New variable binders are now the logical quantifiers $\forall$ and $\exists$. The definitions of the collections of free term and type variables are extended in the obvious manner to formulae, *i.e.,* $FTeV(\phi)$ and $FTyV(\phi)$. The collection $FRelV(\phi)$ of *free relation variables* is also defined analogously. Variables that are not free are said to be *bound*. A *closed formula* has no free term-, type- or relation variables.

The definition of substitution extends to formulae in the obvious way. We write $\alpha[R, X, x]$ to indicate possible occurrences of R, X and x in type, term or formula α, and may write $\alpha[\rho, A, t]$ for the result of substitution $\alpha[\rho/R, A/X, t/x]$, following the appropriate rules concerning capture, or following Barendregt's variable convention. We also write $t\ R\ u$ in place of $R(t, u)$.

Judgements for formula formation now involve relation symbols, so judgements will have the form $\Delta \mid \Upsilon \mid \Gamma \rhd \phi$, where Δ is a type context, Γ is a term context depending on Δ as usual, and Υ is a *relation context*, *viz.* a set $\{R_1 \subset A_1 \times B_1, \ldots, R_k \subset A_k \times B_k\}$ where each R_i occurs at most once, and where $\Delta \rhd A_i, B_i$. To avoid clutter, and because the level of discussion permits it, we will from now on write contexts as amalgamated into one set Γ, *e.g.,* $X, x\!:\!X \rhd x\!:\!X$ and $X, Y, R \subset X \times Y, x : X, y : Y \rhd x\ R\ y$. The actual judgements for formula formation involve contexts justifying the terms in the formulae and are as one would expect, and we omit them here. What we do explicitly need is relation definition, accommodated by the following syntax.

$$\frac{\Gamma, x\!:\!A, y\!:\!B \rhd \phi}{\Gamma \rhd (x\!:\!A, y\!:\!B)\ .\ \phi \quad \subset A\!\times\! B}$$

where ϕ is a formula. For example $\mathsf{eq}_A \stackrel{def}{=} (x\!:\!A, y\!:\!A).(x =_A y)$.

We now build complex relations using type formers. We get the *arrow-type relation* $\rho \to \rho' \subset (A \to A') \times (B \to B')$ from $\rho \subset A \times B$ and $\rho' \subset A' \times B'$ by

$$(\rho \to \rho') \stackrel{def}{=} (f\!:\!A \to A', g\!:\!B \to B') \;.\; (\forall x\!:\!A.\forall y\!:\!B \;.\; (x\rho y \;\Rightarrow\; (fx)\rho'(gy)))$$

i.e., two f and g are related if they map ρ-related arguments to ρ'-related results. The *universal-type relation* $\forall(Y, Z, R \subset Y \times Z)\rho[R] \;\subset\; (\forall Y.A[Y]) \times (\forall Z.B[Z])$ is defined from $\rho[R] \subset A[Y] \times B[Z]$, where Y, Z and $R \subset Y \times Z$ are free, by

$$\begin{aligned}&\forall(Y, Z, R \subset Y \times Z)\rho[R] \stackrel{def}{=} \\ &\qquad (y\!:\!\forall Y.A[Y], z\!:\!\forall Z.B[Z]) \;.\; (\forall Y.\forall Z.\forall R \subset Y \times Z \;.\; ((yY)\rho[R](zZ)))\end{aligned}$$

i.e., two y and z are related at universal type if all instances yY and zZ are related in $\rho[R]$, whenever R relates Y and Z.

Using this, one can define the *action* of types on relations, by substituting relations for type variables in types. For $\boldsymbol{X} = X_1, \ldots, X_n$, $\boldsymbol{B} = B_1, \ldots, B_n$, $\boldsymbol{C} = C_1, \ldots, C_n$ and $\boldsymbol{\rho} = \rho_1, \ldots, \rho_n$, where $\rho_i \subset B_i \times C_i$, we get $T[\boldsymbol{\rho}] \subset T[\boldsymbol{B}] \times T[\boldsymbol{C}]$, the action of $T[\boldsymbol{X}]$ on $\boldsymbol{\rho}$, defined by cases on $T[\boldsymbol{X}]$ as follows:

$$\begin{array}{ll} T[\boldsymbol{X}] = X_i : & T[\boldsymbol{\rho}] = \rho_i \\ T[\boldsymbol{X}] = T'[\boldsymbol{X}] \to T''[\boldsymbol{X}] : & T[\boldsymbol{\rho}] = T'[\boldsymbol{\rho}] \to T''[\boldsymbol{\rho}] \\ T[\boldsymbol{X}] = \forall X'.T'[\boldsymbol{X}, X'] : & T[\boldsymbol{\rho}] = \forall(Y, Z, R \subset Y \times Z)T'[\boldsymbol{\rho}, R] \end{array}$$

The proof system is natural deduction, intuitionistic style, over formulae now involving relation symbols, and is augmented with inference rules for relation symbols, for example we have for Φ a finite set of formulae:

$$\frac{\Phi \vdash_{\Gamma, R \subset A \times B} \phi[R]}{\Phi \vdash_\Gamma \forall R \subset A \times B \;.\; \phi[R]}, R \text{ not free in } \Phi \qquad \frac{\Phi \vdash_\Gamma \forall R \subset A \times B.\phi[R] \quad \Gamma \rhd \rho \subset A \times B}{\Phi \vdash_\Gamma \phi[\rho]}$$

One has the usual axioms for equational reasoning and $\beta\eta$ equalities. Of these, we just give the axiom schemata for substitution and congruence here:

$$\begin{aligned}&\forall X.\forall Y.\forall R \subset X \times Y.\forall x\!:\!X.\forall x'\!:\!X.\forall y\!:\!Y.\forall y'\!:\!Y \;. \\ &\qquad\qquad R(x, y) \;\wedge\; x =_X x' \;\wedge\; y =_Y y' \;\Rightarrow\; R(x', y')\end{aligned}$$

$$(\forall x\!:\!X \;.\; t[x] =_Y u[x]) \;\Rightarrow\; (\lambda x\!:\!X.t) =_{X \to Y} (\lambda x\!:\!X.u)$$

$$(\forall X \;.\; t[X] =_Y u[X]) \;\Rightarrow\; (\Lambda X.t) =_{\forall X.Y} (\Lambda X.u)$$

Note that the schemata for congruence imply extensional equality for arrow types and universal types. Sect. A.2 in Appendix A contains the full inference system.

3.3.1 Relational Parametricity

The notion of *parametric polymorphism* is originally a semantic notion pertaining to functionals interpreting polymorphic terms. The idea is that although polymorphic functionals can be instantiated at any domain to give the particular version of the functional at that domain, all the instances should still somehow exhibit a uniform behaviour, where what constitutes the uniform behaviour should be definable. Thus a polymorphic functional that, when given two integers returns their product, when given two reals terminates the program, and when given two matrices parks your hard-drive, *etc.* would presumably not be parametric.

There are various notions of parametric polymorphism. The informal one due to Strachey (Strachey, 1967) can be explained in PER models by saying that a polymorphic functional is parametric if all its instances share the same realiser. There is a categorical notion of parametricity in terms of dinatural transformations (Bainbridge et al., 1990). The notion of parametricity we adopt for our discussion is that of *relational parametricity* originally due to Reynolds (Reynolds, 1983; Ma and Reynolds, 1991). Uniformity is then defined by saying that if a polymorphic functional is instantiated at two related domains, then the resulting instances should be related as well. In Plotkin and Abadi's logic, this is asserted in the following axiom schema:

$$\text{PARAM}: \ \forall Y_1, \ldots, \forall Y_n \forall u\colon(\forall X.T[X, Y_1, \ldots, Y_n]) \ . \ u(\forall X.T[X, \mathsf{eq}_{Y_1}, \ldots, \mathsf{eq}_{Y_n}])u$$

To understand, it helps to ignore the parameters Y_i and expand the definition to get $\forall u\colon(\forall X.T[X]) \ .\forall Y.\forall Z.\forall R\subset Y\times Z \ . \ u(Y) \ T[R] \ u(Z)$, *i.e.,* if one instantiates a polymorphic inhabitant at two related types then the results are also related.

This logic is sound w.r.t. the parametric PER-model of (Bainbridge et al., 1990) and also w.r.t. the syntactic parametric models of (Hasegawa, 1991).

Under the assumption that the above axiom schema does capture parametricity in Reynolds' sense, (Plotkin and Abadi, 1993) argues that parametricity in Reynolds' sense does not imply parametricity in Strachey's sense. However the converse is conjectured. Also, parametricity in Reynolds' sense is shown to imply parametricity in the dinatural sense of (Bainbridge et al., 1990).

The assumption of relational parametricity gives us a number of very interesting results. To start with, the following link to equality is fundamental.

Theorem 3.2 (Identity Extension Lemma (Plotkin and Abadi, 1993)**)** *For any $T[\boldsymbol{Z}]$, the following sequent is derivable using* PARAM.

$$\forall \boldsymbol{Z}.\forall u, v\colon T[\boldsymbol{Z}] \ . \ (u \ T[\mathbf{eq}_{\boldsymbol{Z}}] \ v \ \Leftrightarrow \ (u =_{T[\boldsymbol{Z}]} v))$$

3.3.2 Simulation Relations

The notion of *data refinement* has many roots. A significant contribution is the work of Hoare (Hoare, 1972) where concrete programs are verified w.r.t. more abstract programs by using a *representation invariant* together with an *abstraction function* mapping concrete values satisfying the representation invariant, to abstract values, see (Dahl, 1992) for examples and a full account of this. The more general method of using relations to describe data refinement is proposed already in (Milner, 1971). This idea is taken up in (Schoett, 1986, 1990) where the relation used is called a *correspondence*. In the context of lambda calculus, (Reynolds, 1974, 1983) discusses *representation independence* in a polymorphic setting using relations, see also (Reynolds, 1981, 1998). Mitchell promotes the use of *logical relations* in proving data refinement and representation independence in purer forms of lambda calculus (Mitchell, 1991, 1990, 1996), and Tennent and O'Hearn use logical relations in Algol-like extensions of lambda calculus to the same ends (Tennent, 1997; O'Hearn and Tennent, 1993).

The above notions describing data refinement are all on the semantic level. In the relational logic of (Plotkin and Abadi, 1993) one can use the action of types on relations to define a syntactic mirror of the above ideas, namely a notion of *simulation relation*:

Definition 3.3 (Simulation Relation (SimRel) (Plotkin and Abadi, 1993)**)** *Relatedness by* simulation relation *w.r.t.* $T[X, \boldsymbol{Z}]$ *is expressed in the logic by*

$$\begin{aligned}\mathsf{SimRel}_{T[X,\boldsymbol{\rho}]} &\stackrel{def}{=} (u{:}\exists X.T[X,\boldsymbol{U}], v{:}\exists X.T[X,\boldsymbol{V}])\ .\\ &\quad(\exists A, B.\exists a{:}T[A,\boldsymbol{U}], b{:}T[B,\boldsymbol{V}]\ .\ u = (\mathsf{pack}Aa)\ \wedge\ v = (\mathsf{pack}Bb)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \exists R{\subset}A{\times}B\ .\ a(T[R,\boldsymbol{\rho}])b\end{aligned}$$

where $\boldsymbol{Z}$ are the free type variables in $T[X,\boldsymbol{Z}]$ other than X, and $\boldsymbol{\rho}{\subset}\boldsymbol{U}{\times}\boldsymbol{V}$ is a vector of relations of the same length.

The subscript $T[X,\boldsymbol{\rho}]$ to $\mathsf{SimRel}_{T[X,\boldsymbol{\rho}]}$ might occasionally be omitted.

Intuitively, two data types are related by a simulation relation according to Def. 3.3, if there exists a relation R on their respective data representations that is preserved by their corresponding operations. The precise mode of simulation here corresponds to the semantic notion in (Reynolds, 1974, 1981). We will think of both the base-type relation R on the data representations, and also the relation $T[R,\boldsymbol{\rho}]$ generated by the action of types on R, as the simulation relation.

With relational parametricity we get the following central result.

Theorem 3.4 ((Plotkin and Abadi, 1993)) *The following sequent schema is derivable using* PARAM.

$$\forall u : \exists X.T[X, \boldsymbol{U}], v : \exists X.T[X, \boldsymbol{V}] \;.\quad u\ (\exists X.T[X, \boldsymbol{\rho}])\ v \quad\Leftrightarrow\quad u\ \mathsf{SimRel}_{T[X,\boldsymbol{\rho}]}\ v$$

where $\boldsymbol{Z}$ *are the free type variables in* $T[X, \boldsymbol{Z}]$ *other than* X*, and* $\boldsymbol{\rho} \subset \boldsymbol{U} \times \boldsymbol{V}$ *is a vector of relations of the same length.*

Proof: See Appendix B. We give a proof because it is seemingly not written down anywhere else. □

Via the Identity Extension Lemma (Theorem 3.2), Theorem 3.4 gives:

Theorem 3.5 ((Plotkin and Abadi, 1993)) *The following sequent schema is derivable using* PARAM.

$$\forall \boldsymbol{Z}.\forall u, v : \exists X.T[X, \boldsymbol{Z}] \;.\quad u =_{\exists X.T[X,\boldsymbol{Z}]} v \quad\Leftrightarrow\quad u\ \mathsf{SimRel}_{T[X,\mathbf{eq}_{\boldsymbol{Z}}]}\ v$$

Theorem 3.5 states the equivalence of equality at existential type with the existence of a simulation relation. Thus relational parametricity coarsens the granularity of equality to a greater level of abstraction, which furthermore is arguably exactly the level of abstraction one is interested in. From this we also get

Theorem 3.6 $\forall \boldsymbol{Z}.\forall u : \exists X.T[X, \boldsymbol{Z}].\exists A.\exists a : T[A] \;.\; u = (\mathsf{pack} A a)$

In other words, every inhabitant of existential type is equal to a package.

—

We said above that intuitively, two data types are related by simulation relation according to Def. 3.3 if there exists a relation between their two data representations such that their respective operations preserve that relation. However, the converse is not immediate by definition. Suppose we have two data types $(\mathsf{pack} A a) : \exists X.T[X, \boldsymbol{U}]$ and $(\mathsf{pack} B b) : \exists X.T[X, \boldsymbol{V}]$, and that

$$(\mathsf{pack} A a)\ \mathsf{SimRel}_{T[X,\boldsymbol{\rho}]}\ (\mathsf{pack} B b)$$

This does not say there exists $R \subset A \times B$ such that $a\ T[R, \boldsymbol{\rho}]\ b$. It merely says that there are packages $(\mathsf{pack} A' a')$ and $(\mathsf{pack} B' b')$, such that $(\mathsf{pack} A a) = (\mathsf{pack} A' a')$ and $(\mathsf{pack} B b) = (\mathsf{pack} B' b')$, and then some $S \subset A' \times B'$ such that $a'\ T[S, \boldsymbol{\rho}]\ b'$, *i.e.*, we do not immediately have

$$(\mathsf{pack} A a)\ \mathsf{SimRel}_{T[X,\boldsymbol{\rho}]}\ (\mathsf{pack} B b) \;\Rightarrow\; \exists R \subset A \times B \;.\; a\ T[R, \boldsymbol{\rho}]\ b$$

although we do of course have

$$(\mathsf{pack} Aa)\ \mathsf{SimRel}_{T[X,\boldsymbol{\rho}]}\ (\mathsf{pack} Bb)\ \Leftarrow\ \exists R\subset A\times B\ .\ a\ T[R,\boldsymbol{\rho}]\ b$$

In fact we have the following negative result, even for $\boldsymbol{\rho} = \mathbf{eq}_{\boldsymbol{Z}}$.

Theorem 3.7 *The following schema is* not *in general derivable in the logic.*

$$\begin{array}{l}\forall A, B.\forall a{:}T[A,\boldsymbol{Z}], b{:}T[B,\boldsymbol{Z}])\ .\\ \qquad (\mathsf{pack} Aa)\ \mathsf{SimRel}_{T[X,\mathbf{eq}_{\boldsymbol{Z}}]}\ (\mathsf{pack} Bb)\quad \Leftrightarrow\ \exists R\subset A\times B\ .\ a\ T[R,\mathbf{eq}_{\boldsymbol{Z}}]\ b\end{array}$$

where $\boldsymbol{Z}$ are the free type variables in $T[X,\boldsymbol{Z}]$ other than X.

Proof: This follows from Theorem 5.5 (*p.* 114). □

We now point out that the sequent in Theorem 3.5 has a slight circular feature. Since equality at existential type appears in SimRel, equality at existential type occurs on both sides of the equivalence. There is nothing wrong with this; the sequent is derivable. However, this circularity would in the outset render the result less useful as an instrument in proof strategies.

In practice, one would use the sequent as a derivation rule from right to left to establish equality. In particular, if one has two packages and a simulation relation between their data representations, then the packages are equal. In this direction, for given packages (not variables of existential type) the equality clauses in the SimRel expression are trivial. This just uses

$$(\mathsf{pack} Aa)\ \mathsf{SimRel}_{T[X,\boldsymbol{\rho}]}\ (\mathsf{pack} Bb)\ \Leftarrow\ \exists R\subset A\times B\ .\ a\ T[R,\boldsymbol{\rho}]\ b$$

from the discussion above. But it would be nice to know in addition that two packages are equal *only if* there is a simulation relation between their data representations, without involving equality to two proxy packages. This of course hinges on the sequent in Theorem 3.7.

If the sequent in Theorem 3.7 were derivable, this would additionally support the intuition one gets when looking at SimRel, namely that the equalities herein are for converting the variables of existential type to packages so that one may speak in terms of a relation on a data representation, but that when one actually presents two packages, the equalities should not be relevant, *i.e.,* the sequent in Theorem 3.7 should be applicable.

Luckily, although Theorem 3.7 is true, we shall be able to derive the sequent for $T[X]$ containing only first-order types. We will be able to derive a version of the sequent for $T[X]$ containing also higher-order types, when we discuss the alternative notion of simulation relation in Ch. 5.

3.3.3 Packages as Data Types

A data type consists of a data representation A and a collection a of operations on that data representation. In the present setting, we refer to packages as data types, *i.e.,* we would say that $(\mathsf{pack} A a)$ is a data type. Note that by Theorem 3.5, calling $(\mathsf{pack} A a)$ a data type is different from calling the pair $\langle A, a\rangle$ a data type, since packages are equated if there exists a simulation relation between their data representations. Thus, packages are really slightly abstract data types, namely a collection of data types, seen as pairs, related by simulation relations, roughly speaking, see the ensuing discussion. We will continue to refer to packages as data types. This is partly because we cannot form pairs of a type and a term in the type theory we are using, but also because we find the initial abstraction inherent in packages of existential type suitable. We will later see that this abstraction corresponds to observational equivalence, and in a specification refinement context, it is in fact convenient and conceptually pleasing to have observational abstraction already built in to the notion of data type. We do nevertheless recognise that this may be a matter of taste. If one wishes to view data types as pairs, this is still conceptually possible; the main results are immediately transferable to this view, although pairs of the relevant sort are not expressible in the language.

3.3.4 Universal Constructions

We saw in Sect. 3.2.4 how to encode products, sums, inductive types, (initial) T-algebras, and (final) T-coalgebras for any covariant endofunctor T.

All these constructions are in the outset weak notions, in the sense that one can show in the logic the existence of mediating morphisms, *e.g.,* the existence of a homomorphism from any initial T-algebra to any other T-algebra. However, with the assumption of relational parametricity, these construct all become universal constructions, *i.e.,* one can show in the logic that the mediating morphisms are unique. For example, for products we have weakly without parametricity that

$$\forall u{:}U.\forall v{:}V\ .\ (\mathsf{fst}(\mathsf{pair}\, u\, v)) =_U u$$

$$\forall u{:}U.\forall v{:}V\ .\ (\mathsf{snd}(\mathsf{pair}\, u\, v)) =_V v$$

But with relational parametricity we additionally get

$$\forall x{:}U \times V\ .\ (\mathsf{pair}(\mathsf{fst}\, x), (\mathsf{snd}\, x)) =_{U\times V} x$$

and this gives the universal characterisation

$$\forall X.\forall f{:}X \to U.\forall g{:}X \to V.\exists! h{:}X \to U\times V\ .\ f = (comp\, \mathsf{fst}\, h)\ \wedge\ g = (comp\, \mathsf{snd}\, h)$$

where *comp* is the function composition functional defined earlier (with type parameters omitted). This yields the useful characterisation of the action of products on relations. For $\rho \subset U \times V$ and $\rho' \subset U' \times V'$, one defines $(\rho \times \rho')$ as the action $(X \times X')[\rho, \rho']$. Then,

$$\forall u : A \times A', v : B \times B' \ . \ u(\rho \times \rho')v \ \Leftrightarrow \ (\mathsf{fst}(u) \ \rho \ \mathsf{fst}(v) \ \wedge \ \mathsf{snd}(u) \ \rho' \ \mathsf{snd}(v))$$

3.3.5 Induction

One should have induction schemata for inductive types. In particular we would like to have an induction schema for Nat. The standard induction schema for Nat is not as such originally in the logic of (Plotkin and Abadi, 1993), but the schema is sound w.r.t. the most popular parametric models used to validate the logic. On the other hand, with relational parametricity one gets strong initiality for initial T-algebras, and furthermore, one gets a binary induction principle for initial algebras. A unary induction schema can then be encoded using this binary schema. Thus, alternatively one could encode Nat as the initial algebra $\forall X.(((1+X) \to X) \to X)$, and get the induction schema for free. In the parametric models we will be using, the interpretations of Nat as an inductive type and as an initial algebra coincide up to isomorphism. We therefore assume induction for Nat in one way or another; either asserted implicitly as an augmenting axiom schema for Nat as an inductive type, or derived in the logic for Nat as an initial algebra. We need not go into more detail for our purposes, but see (Wadler, 1989). It should also be mentioned that induction for $\mathsf{Nat} = \forall X.X \to (X \to X) \to X$ is in fact derivable in System $\mathcal{R}$ (Abadi et al., 1993).

3.4 Semantics

We give a brief overview of a few of the models for Plotkin and Abadi's logic for relational parametricity described in this chapter. We first summarise the parametric PER-model of (Bainbridge et al., 1990), and then we briefly describe syntactic models according to (Hasegawa, 1991).

3.4.1 The Parametric PER-model

Let **PER** denote the universe of all partial equivalence relations (PERs) over the natural numbers $\mathbb{N}$. Types are interpreted as PERs, but intuitively it helps to think of the associated quotient instead, whose elements are equivalence classes. Terms are thus interpreted as functions mapping equivalence classes from one

PER to another, where the first PER corresponds to the product of the types of the term's free variables, and the second PER corresponds to the type of the term. Relations between PERs relate equivalence classes of the PERs.

Formally this is expressed in elementary terms as follows. A PER $\mathcal{A}$ is a symmetric and transitive binary relation on $\mathbb{N}$. The domain $Dom(\mathcal{A})$ of $\mathcal{A}$ contains those $a \in \mathbb{N}$ for which $a\ \mathcal{A}\ a$. For any $a \in Dom(\mathcal{A})$ we can form the equivalence class $[a]_{\mathcal{A}}$. A morphism from $\mathcal{A}$ to $\mathcal{B}$ is given by $n \in \mathbb{N}$ if for any $a, a' \in \mathbb{N}$, $a \in Dom(\mathcal{A}) \ \Rightarrow\ n(a)\downarrow$ and $a\ \mathcal{A}\ a' \ \Rightarrow\ n(a)\ \mathcal{B}\ n(a')$. Here, $n(a)$ denotes the result of evaluating the n^{th} partial recursive function on a, and $n(a)\downarrow$ denotes that this function is defined for a. We can form a PER $(\mathcal{A}\to\mathcal{B})$ by defining

$$\begin{aligned} e\ (\mathcal{A}\to\mathcal{B})\ e' \ \overset{def}{\Leftrightarrow}\ \forall a, a' \in \mathbb{N}\ .\ &(a \in Dom(\mathcal{A}) \ \Rightarrow\ e(a)\downarrow\ \wedge e'(a)\downarrow) \\ &\wedge (a\ \mathcal{A}\ a' \ \Rightarrow\ e(a)\ \mathcal{B}\ e'(a')) \end{aligned}$$

That is, the equivalence classes in $(\mathcal{A}\to\mathcal{B})$ contain functions that are extensionally equal w.r.t. $\mathcal{A}$ and $\mathcal{B}$. Each such equivalence class is then a morphism from $\mathcal{A}$ to $\mathcal{B}$, with application defined as $[e]_{\mathcal{A}\,\to\,\mathcal{B}}[a]_{\mathcal{A}} \overset{def}{=} [e(a)]_{\mathcal{B}}$. Products are given by

$$n\ (\mathcal{A}\times\mathcal{B})\ n' \ \overset{def}{\Leftrightarrow}\ n.1\ \mathcal{A}\ n'.1\ \wedge\ n.2\ \mathcal{B}\ n'.2$$

where m encodes a pair of natural numbers using some standard encoding, and $m.i$ decodes the i^{th} projection of m.

Relations between $\mathcal{A}$ and $\mathcal{B}$ are given by *saturated relations*. A relation $\mathcal{R}$ between $Dom(\mathcal{A})$ and $Dom(\mathcal{B})$ is saturated on $\mathcal{A}$ and $\mathcal{B}$, if

$$(m\ \mathcal{A}\ n\ \wedge\ n\ \mathcal{R}\ n'\ \wedge\ n'\ \mathcal{B}\ m') \ \Rightarrow\ m\ \mathcal{R}\ m'$$

Thus saturated relations preserve, and can be seen to relate equivalence classes. Any member n of an equivalence class q is called a *realiser* for q.

We get complex relations as follows. Let $\mathcal{A}$ and $\mathcal{B}$ be any PER, and consider any saturated relations $\mathcal{R} \subset Dom(\mathcal{A}) \times Dom(\mathcal{B})$ and $\mathcal{S} \subset Dom(\mathcal{A}') \times Dom(\mathcal{B}')$. For any $n \in Dom(\mathcal{A})$, $m \in Dom(\mathcal{B})$, $n' \in Dom(\mathcal{A}')$, $m' \in Dom(\mathcal{B}')$, we define the saturated relation $\mathcal{R}\times\mathcal{S} \subset Dom(\mathcal{A}\times\mathcal{B}) \times Dom(\mathcal{A}'\times\mathcal{B}')$ by

$$\langle n, n'\rangle\ (\mathcal{R}\times\mathcal{S})\ \langle m, m'\rangle \ \overset{def}{\Leftrightarrow}\ n\ \mathcal{R}\ m\ \wedge\ n'\ \mathcal{S}\ m'$$

For any $e \in Dom(\mathcal{A} \to \mathcal{A}')$ and $e' \in Dom(\mathcal{B} \to \mathcal{B}')$, we define the saturated relation $\mathcal{R}\to\mathcal{S} \subset Dom(\mathcal{A}\to\mathcal{B}) \times Dom(\mathcal{A}'\to\mathcal{B}')$ by

$$e\ (\mathcal{R}\to\mathcal{S})\ e' \ \overset{def}{\Leftrightarrow}\ \forall a \in Dom(\mathcal{A}), \forall b \in Dom(\mathcal{B})\ .\ a\ \mathcal{R}\ b \ \Rightarrow\ e(a)\ \mathcal{S}\ e'(b)$$

If $\mathcal{R}$ is saturated on $\mathcal{A}$ and $\mathcal{B}$, we will often write $[n]_{\mathcal{A}}\ \mathcal{R}\ [m]_{\mathcal{B}}$ in place of $n\ \mathcal{R}\ m$ to aid intuition. We will use this equivalence-class notation elsewhere too.

Type and Relation Semantics

Given the universe of PERs above, type semantics are now defined denotationally w.r.t. an environment δ mapping type variables to PERs.

$$
\begin{aligned}
&[\![\Delta, X \rhd X]\!]_\delta \stackrel{def}{=} \delta(X)\\
&[\![\Delta \rhd U \to V]\!]_\delta \stackrel{def}{=} ([\![\Delta \rhd U]\!]_\delta \to [\![\Delta \rhd V]\!]_\delta)\\
&[\![\Delta \rhd \forall X.U[X]]\!]_\delta \stackrel{def}{=} (\cap_{\mathcal{A}\in\mathbf{PER}}[\![\Delta, X \rhd U[X]]\!]_{\delta[X\mapsto\mathcal{A}]})^\flat
\end{aligned}
$$

where $(\cap_{\mathcal{A}\in\mathbf{PER}}[\![\Delta, X \rhd U[X]]\!]_{\delta[X\mapsto\mathcal{A}]})^\flat$ is the indicated intersection but trimmed down to only those elements invariant over all saturated relations. This trimming is what makes the model relational parametric. We explain in a moment.

The semantics of formulae are as one would expect, given semantics on types and terms. We write

$$\models_{\Gamma,\gamma} \phi$$

to indicate that formula ϕ holds in the model under environment γ on Γ. We omit the full definition of formula semantics, but give the resulting semantics of the action of types on relations. Recall that the action of types on relations are defined in the logic in terms of complex relations defined in terms of formulae, *cf.* Sect. 3.3. Thus w.r.t. environments δ and υ mapping respectively, type variables to PERs, and relation variables to appropriate saturated relations, we have

$$
\begin{aligned}
&[\![\Delta \mid \Upsilon, R \rhd R]\!]_{\delta\upsilon} = \upsilon(R)\\
&[\![\Delta \mid \Upsilon \rhd U \to V]\!]_{\delta\upsilon} = ([\![\Delta \mid \Upsilon \rhd U]\!]_{\delta\upsilon} \to [\![\Delta \mid \Upsilon \rhd V]\!]_{\delta\upsilon})\\
&[\![\Delta \mid \Upsilon \rhd \forall X.U[X]]\!]_{\delta\upsilon} = (\cap_{\mathcal{R}}[\![\Delta, A, B \mid \Upsilon, R\subset A\times B \ \rhd\ U[R]]\!]_{\delta[A\mapsto\mathcal{A},B\mapsto\mathcal{B}]\upsilon[R\mapsto\mathcal{R}]})
\end{aligned}
$$

where the intersection ranges over all saturated $\mathcal{R} \subset Dom(\mathcal{A}) \times Dom(\mathcal{B})$, for all PERs $\mathcal{A}$ and $\mathcal{B}$. We might omit the type information of relation variables.

We can now explain the relational parametric type semantics of universal types. First, the non-relational parametric PER semantics is given by the untrimmed intersection, *i.e.,*

$$n\ (\cap_{\mathcal{A}\in\mathbf{PER}}[\![\Delta, X \rhd U[X]]\!]_{\delta[X\mapsto\mathcal{A}]})\ m \stackrel{def}{\Leftrightarrow} \forall\mathcal{A} \in \mathbf{PER}\ .\ n\ [\![\Delta, X \rhd U[X]]\!]_{\delta[X\mapsto\mathcal{A}]}\ m$$

This gives rise to an interpretation where an instance of a polymorphic element at a particular PER is represented by the same realiser as the polymorphic element itself. This is parametricity according to Strachey. But for relational parametricity, we additionally want

$$
[n]^{(\mathcal{A})}_{\cap_{\mathcal{A}\in\mathbf{PER}}[\![\Delta,X\rhd U[X,\boldsymbol{Z}]]\!]_{\delta[X\mapsto\mathcal{A}]}}
\quad ([\![\Delta \mid \Upsilon, R \rhd U[R, \mathbf{eq}_{\boldsymbol{Z}}]]\!]_{\delta\upsilon[R\mapsto\mathcal{R}]})
\quad [n]^{(\mathcal{B})}_{\cap_{\mathcal{A}\in\mathbf{PER}}[\![\Delta,X\rhd U[X,\boldsymbol{Z}]]\!]_{\delta[X\mapsto\mathcal{A}]}}
$$

for every saturated $\mathcal{R} \subset Dom(\mathcal{A}) \times Dom(\mathcal{B})$. Here the super-scripts suggest for intuition the instances at PERs $\mathcal{A}$ and $\mathcal{B}$ respectively, but there is of course really no difference between the two equivalence classes, since a polymorphic element at a particular PER is represented by the same realiser as the polymorphic element itself. Consequently, this boils down to

$$\begin{aligned} & n\ (\cap_{\mathcal{A}\in\mathbf{PER}}[\![\Delta, X \rhd U[X,\boldsymbol{Z}]]\!]_{\delta[X\mapsto\mathcal{A}]})^{\flat}\ m \\ & \qquad \overset{def}{\Leftrightarrow} \forall \mathcal{A},\mathcal{B} \in \mathbf{PER},\ \textit{saturated}\ \mathcal{R} \subset Dom(\mathcal{A}) \times Dom(\mathcal{B})\ . \\ & n\ [\![\Delta, X \rhd U[X,\boldsymbol{Z}]]\!]_{\delta[X\mapsto\mathcal{A}]}\ m\ \wedge\ n\ [\![\Delta, X \rhd U[X,\boldsymbol{Z}]]\!]_{\delta[X\mapsto\mathcal{B}]}\ m\ \wedge \\ & \qquad n\ [\![\Delta \shortmid \Upsilon, R \rhd U[R,\mathsf{eq}_{\boldsymbol{Z}}]]\!]_{\delta\upsilon[R\mapsto\mathcal{R}]}\ n\ \wedge \\ & \qquad m\ [\![\Delta \shortmid \Upsilon, R \rhd U[R,\mathsf{eq}_{\boldsymbol{Z}}]]\!]_{\delta\upsilon[R\mapsto\mathcal{R}]}\ m \end{aligned}$$

We now have that

$$(\cap_{\mathcal{A}\in\mathbf{PER}}[\![\Delta, X \rhd U[X,\boldsymbol{Z}]]\!]_{\delta[X\mapsto\mathcal{A}]})^{\flat} = (\cap_{\mathcal{R}}[\![\Delta \shortmid R \rhd U[R,\mathsf{eq}_{\boldsymbol{Z}}]]\!]_{\delta\upsilon\,[R\mapsto\mathcal{R}]})$$

where $\mathcal{R}$ ranges over all saturated relations on $Dom(\mathcal{A}) \times Dom(\mathcal{B})$, for all PERs $\mathcal{A}$ and $\mathcal{B}$. In general we have

$$[\![U[\boldsymbol{Z}]]\!]_{\delta} = [\![U[\mathsf{eq}_{\boldsymbol{Z}}]]\!]_{\delta\upsilon}$$

This is a manifestation of the Identity Extension Lemma, Theorem 3.2. Note that $[\![\mathsf{eq}_{U[\boldsymbol{Z}]}]\!]_{\delta\upsilon} = [\![U[\boldsymbol{Z}]]\!]_{\delta}$, *i.e.,* the equality predicate on a type has the same interpretation, *i.e.,* a PER, as the type itself.

Finally, in the parametric PER-model initial constructs interpret to objects isomorphic to interpretations of inductive types, *e.g.,* let $T[X] = 1 + X$. Then

$$[\![\emptyset \rhd \forall X.((T[X] \to X) \to X)]\!] \cong [\![\emptyset \rhd \forall X.(X \to (X \to X) \to X)]\!] \cong \mathbb{N}$$

This justifies the remarks made in Sect. 3.3.5.

Term Semantics

Term semantics follow regular denotational structure, given a type environment, and a term environment mapping term variables to elements of the appropriate PER. We amalgamate contexts into a single context Γ, and environments into a single environment γ.

$$\begin{aligned} & [\![\Gamma, x{:}U \rhd x{:}U]\!]_{\gamma} \overset{def}{=} \gamma(x) \quad \text{where } \gamma(x) \in Dom(\mathcal{A}), \text{ for } \mathcal{A} = [\![\Gamma \rhd U]\!]_{\gamma} \\ & [\![\Gamma \rhd fu{:}V]\!]_{\gamma} \overset{def}{=} [\![\Gamma \rhd f{:}U \to V]\!]_{\gamma}[\![\Gamma \rhd u{:}U]\!]_{\gamma} \\ & [\![\Gamma \rhd \lambda x{:}U.t[x]{:}U \to V]\!]_{\gamma} \overset{def}{=} n \in Dom([\![\Gamma \rhd U \to V]\!]_{\gamma}), \quad \textit{for } n \textit{ the code of a} \\ & \qquad \textit{partial recursive function } \lambda a.[\![\Gamma, x{:}U \rhd t[x]{:}V]\!]_{\gamma[x\mapsto a]} \\ & [\![\Gamma \rhd tA{:}U[A]]\!]_{\gamma} \overset{def}{=} [\![\Gamma \rhd t{:}\forall X.U[X]]\!]_{\gamma} \\ & [\![\Gamma \rhd \Lambda X.t[X]{:}\forall X.U[X]]\!]_{\gamma} \overset{def}{=} n, \quad \textit{such that} \\ & \qquad [\![\Gamma, X \rhd t[X]{:}U[X]]\!]_{\gamma[X\mapsto\mathcal{A}]} = n \quad \textit{for any } \mathcal{A} \in \mathbf{PER} \end{aligned}$$

There is an obligation to check that the term semantics preserves the appropriate equivalence classes. It is not necessary to take precautions for relational parametricity, other than the proviso implicit in the variable rule, ensuring that variables of polymorphic type are interpreted into a trimmed intersection PER interpreting the universal types. The term semantics follow term formation rules and will not yield elements in, as it were, untrimmed intersection PERs.

A crucial observation we will be using in Ch. 6 is the following. The underlying mechanism of the term semantics is the construction of (encodings of) partial recursive functions according to term structure, and in fact without using type information (Bainbridge et al., 1990; Mitchell, 1996). One is thus justified in viewing a realiser as being generated over a set of realisers representing free variables, using term-formation rules. If $t[x_1, \ldots, x_k]$ is a term with free term variables $x_1, \ldots, x_k$, then for $n_1, \ldots, n_k \in \mathbb{N}$, we will write the partial recursive function constructed over $n_1, \ldots, n_k$ according to t as $t[n_1, \ldots, n_k]$.

3.4.2 Syntactic Models

We will also be considering other model candidates for Plotkin and Abadi's logic, namely models constructed over types and terms, and over theories. We will refer to these as *syntactic models*.

First there is the closed type and term model of (Hasegawa, 1991). One considers a structure constructed over the closed types and terms of System F, quotiented by the equational theory. This structure is not a model, because it is not extensional in functional and polymorphic application. However, in (Breazu-Tannen and Coquand, 1988), this structure, dubbed a *polymorphic lambda interpretation*, undergoes an *extensional collapse* via quotienting by logical relations. Thus a model is formed. In fact, (Breazu-Tannen and Coquand, 1988) describes this extensional collapse as a general method for obtaining extensional structures from arbitrary polymorphic lambda interpretations, of which the above is an example. This model has the property that all elements are denotable by closed terms, and moreover, all closed type expressions of the form $\forall X_1. \cdots \forall X_n.T$, where T has no universal quantifiers, have canonical interpretations, *i.e.,* there is a bijection between the closed normal forms of the the type and the elements of the interpretation of the type. Then, in (Hasegawa, 1991) this model is adjoined with binary relations, and it is shown that the resulting structure is *partially* relational parametric, in the sense that all syntactically definable universal types are relational parametric. It is an open question whether or not universal types that are not syntactically definable are parametric, and thus it is not known if this structure is a model for the logic for relational parametricity we are considering. We will nevertheless still refer to this structure in our speculations.

A structure that on the other hand is a model of the logic, is the *parametric minimal model* due to (Hasegawa, 1991). The starting point is the *maximum consistent theory* and the ensuing construction of the *second-order minimal model* due to (Moggi and Statman, 1986). This construction is an instance of the extensional collapse schema mentioned above. This model satisfies the ω-rule, see Sect. A.6 in Appendix A for the ω-rule, and moreover, all closed type expressions of the form $\forall X_1. \cdots \forall X_n.T$, where T has no universal quantifiers, have canonical interpretations, but only if T has rank no greater than two; the rank $\mathit{rank}(T)$ of a type being defined as

$$\mathit{rank}(X) = 0, \text{ for any variable } X,$$
$$\mathit{rank}(U \to V) = \mathit{max}(\mathit{rank}(U) + 1, \mathit{rank}(V)).$$

Then in (Hasegawa, 1991) this model is adjoined with binary relations, and it is shown that the resulting structure is fully relational parametric.

Chapter 4

Specification Refinement in Type Theory

In this chapter, essential concepts of algebraic specification refinement outlined in Ch. 2, are translated into the type-theoretic setting involving System F and Reynolds' relational parametricity assertion as expressed in Plotkin and Abadi's logic for parametric polymorphism. At first order, the type-theoretic setting provides a canonical picture of algebraic specification refinement. At higher order, the type-theoretic setting allows future generalisation of the principles of algebraic specification refinement to higher order and polymorphism; this is the topic of Ch. 5, Ch. 6, and Ch. 7. In this chapter, we will show the equivalence of the acquired type-theoretic notion of specification refinement with that from algebraic specification. We will also import the generic algebraic-specification strategy for behavioural refinement proofs into the type-theoretic setting. Beside being a valuable asset in its own right, this imported proof strategy is necessary for showing the above correspondence between the two notions of refinement.

4.1 Introduction

One of the most appealing and successful endeavours in algebraic specification is that of *stepwise specification refinement*, in which abstract descriptions of processes and data types are methodically refined to concrete executable descriptions, *viz.* programs and program modules. The account of this due to Sannella and Tarlecki was described in Ch. 2. We are now going to express this refinement framework in a type-theoretic environment comprised of System F and the relational logic assuming relational parametricity described in Ch. 3.

The benefit of this to algebraic specification is that inherently first-order concepts are translated into a setting in which they may be generalised through the full force of the chosen type theory. Furthermore, in algebraic specification many concepts have numerous theoretical variants. Here, the setting of type theory may provide a somewhat sobering framework, in that type-theoretic formalisms insist on certain sensibly canonical choices.

Conversely, the benefit to type theory lies in drawing from the rich source of formalisms, development methodology and reasoning techniques developed in the field of algebraic specification. See (Cerioli et al., 1997) for a survey.

Chapter 2 highlighted three essential concepts that make the chosen account of specification refinement apt for real-life development. These are so-called *constructor implementations*, *observational equivalence*, and *stability*. At first order, these concepts fall out naturally in the type-theoretic setting. Relational parametricity plays an essential role in this. It gives the coincidence at first order of observational equivalence with equality at existential type.

In algebraic specification there is a generic proof strategy for proving specification refinement up to observational equivalence. This strategy is formalised in (Bidoit et al., 1997; Bidoit and Hennicker, 1996; Hennicker, 1997), and was outlined in Sect. 2.7. The strategy considers axiomatisations of so-called behavioural partial congruences. As observed in (Poll and Zwanenburg, 1999) and (Hannay, 1999a), although the former does not refer to this particular proof strategy, the type-theoretic setting of System F and Plotkin and Abadi's logic is not sufficient to accommodate this proof strategy. The reason is the lack of mechanisms for handling subobjects and quotients. Handling subobjects is necessary in case the behavioural congruence is partial. One solution would be to add something akin to subset types, and quotient types, *e.g.,* (Hofmann, 1995b), but this requires a dependent type theory, which we avoid in this thesis.

In (Poll and Zwanenburg, 1999) one finds the idea for an alternative solution. This solution is purely logical; one simply adds axioms stating the existence of quotients and subobjects. This is justified by the soundness of the axioms w.r.t. the parametric PER-model (Bainbridge et al., 1990) that already is a justification for Plotkin and Abadi's logic. We do not know if the extended logic is complete for the parametric PER-model.

The imported proof strategy is a valuable tool for proving observational refinements in the type-theoretic setting. Moreover, it lets us show a formal correspondence between the type-theoretical notion of refinement and the notion of refinement from algebraic specification.

Related work to parts of this chapter is primarily (Poll and Zwanenburg, 1999; Zwanenburg, 1999) which also study refinement in the type-theoretic context. We do this with references to existing practices in algebraic specification by establishing a formal link to algebraic specification refinement. In (Poll and Zwanenburg, 1999) quotients are dealt with. Using the same idea of axiomatisation, (Hannay, 1999a) then complements this work by also dealing with subobjects. The axiomatisation of the existence of subobjects also appeared independently in the comprehensive (Zwanenburg, 1999).

Other relevant work linking algebraic specification and type theory includes (Luo, 1993) encoding constructor implementations in the Extended Calculus of Constructions (ECC), (Reus and Streicher, 1993) expressing module-algebra axioms in ECC, (Mylonakis, 1995) encoding behavioural equalities in the Uniform Theory of Dependent Types (UTT), (Aspinall, 1997) treating the specification language ASL+, (Underwood, 1994) using Nuprl as a specification language, and (Streicher and Wirsing, 1990) arguing the necessity for dependent types in specification. Only (Poll and Zwanenburg, 1999; Zwanenburg, 1999) utilise the specific context of System F and relational parametricity.

The next section introduces specification refinement in a type-theoretic setting. Then, the obvious translation of algebraic specification refinement into this environment is presented. The main result of this chapter is a correspondence at first-order between algebraic specification refinement and the type-theoretic notion of specification refinement. This sets the scene for generalising the refinement concepts now implanted in type theory to higher order and polymorphism.

4.2 Type Theoretic Specification Refinement

We now define a notion of *type theoretic specification refinement* that mirrors algebraic specification refinement. It is important to realise that things, which in the algebraic specification approach were semantic notions or meta-level concepts, such as data types, observational equivalence and refinement, now become internalised in syntax. This is possible because of the expressivity of System F and the logic for relational parametricity. We will use the same terminology for the internalised features as for the features themselves.

4.2.1 Data Types

Data types are now terms of existential type. As mentioned earlier, it is tempting to call existential types abstract data types. This would be appropriate if the abstract properties shared by all instances of an abstract data type were simply those inherent in the existential type. However, we are concerned with the additional interface given by specifications. Consequently, we follow (Mitchell and Plotkin, 1988) and call the types of data types merely *abstract types*. In fact, existential types act more like signatures.

Example 4.1 The type of natural-number stack data types is the abstract type

$$Sig_{\mathsf{STACK}_{\mathsf{Nat}}} \stackrel{\mathit{def}}{=} \exists X.\mathfrak{T}_{\mathsf{STACK}_{\mathsf{Nat}}}[X]$$

for $\mathfrak{T}_{\mathsf{STACK}_{\mathsf{Nat}}}[X] \stackrel{\mathit{def}}{=} (\mathsf{empty}\!:\!X,\ \mathsf{push}\!:\!\mathsf{Nat}\to X\to X,\ \mathsf{pop}\!:\!X\to X,\ \mathsf{top}\!:\!X\to\mathsf{Nat})$ ○

As a notational convention, we will in discourse reserve $\mathfrak{T}[X]$ for the *body* of a given arbitrary abstract type $\exists X.\mathfrak{T}[X]$. We will later be interested in parameterised data types, so $\mathfrak{T}[X]$ may have free variables $\boldsymbol{Z}$ other than X, but we only write these when necessary. As in the above example, we henceforth use a labelled product notation as a purely notational convenience for product types and tuples when discussing data types, *i.e.,* we will write $\mathfrak{T}[X]$ in the form

$$(f_1\!:\!T_1[X], \ldots, f_k\!:\!T_k[X])$$

and if $u\!:\!\mathfrak{T}[X]$, we write $u.f_i$ for the i^{th} projection of u. We call each $f_i\!:\!T_i[X]$ a *profile* of the abstract type.

4.2.2 Polymorphic Data Types

We conveniently get *polymorphic data types* by Λ-abstracting regular data types. The type of a polymorphic data type is a *polymorphic abstract type*.

Example 4.2 The type of polymorphic stack data types is given by the polymorphic abstract type

$$Sig_{\mathsf{STACK}} = \forall Z.\exists X.\mathfrak{T}_{\mathsf{STACK}}[X, Z]$$

for $\mathfrak{T}_{\mathsf{STACK}}[X, Z] = (\mathsf{empty} : X, \mathsf{push} : Z \to X \to X, \mathsf{pop} : X \to X, \mathsf{top} : X \to Z \to Z)$. If $u : \mathit{Sig}_{\mathsf{STACK}}$ is a polymorphic stack data type, then one gets natural-number and boolean stack data types by, respectively

$$u\mathsf{Nat} : \exists X.\mathfrak{T}_{\mathsf{STACK}}[X, \mathsf{Nat}] \qquad\qquad u\mathsf{Bool} : \exists X.\mathfrak{T}_{\mathsf{STACK}}[X, \mathsf{Bool}]$$

○

Writing polymorphic data types is of course good programming economy and enhances maintainability by limiting code duplication.

4.2.3 Constructors

In algebraic specification, constructors are essentially parameterised programs, *i.e.,* programs are likened to semantic structures, namely algebras, so constructors are functions from algebras to algebras. In type theory, programs are directly terms. Therefore, constructors are internalised in the type theory as function terms between abstract types, *e.g.,* $F : \exists X.\mathfrak{T}'[X] \to \exists X.\mathfrak{T}[X]$.

Example 4.3 The constructor *Tr* of Example 2.3 is expressed in this setting as

$$\lambda u : \mathit{Sig}_{\mathsf{STACK}_{\mathsf{Nat}}}.\mathsf{unpack}(u)(\mathit{Sig}_{\mathsf{TRIV}})(\Lambda X.\lambda \mathfrak{x} : \mathfrak{T}_{\mathsf{STACK}_{\mathsf{Nat}}} . (\mathsf{pack} X \ (\mathsf{id} = \\ \lambda x, n, z : \mathsf{Nat} \ . \ \mathfrak{x}.\mathsf{top}(\mathit{multipop}\, n \ (\mathit{multipush}\, n \ z \ (\mathfrak{x}.\mathsf{push}\, x \ \mathfrak{x}.\mathsf{empty}))))))$$

Here$\mathit{Sig}_{\mathsf{TRIV}} \stackrel{def}{=} \exists X.(\mathsf{id} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat})$, and *multipop* and *multipush* are defined in terms of iterators. This constructor has type $\mathit{Sig}_{\mathsf{STACK}_{\mathsf{Nat}}} \to \mathit{Sig}_{\mathsf{TRIV}}$. ○

Constructors mapping polymorphic data types are terms between polymorphic abstract types, *e.g.,* $F : \forall \boldsymbol{Z}.\exists X.\mathfrak{T}'[X, \boldsymbol{Z}] \to \forall \boldsymbol{Z'}.\exists X.\mathfrak{T}[X, \boldsymbol{Z'}]$.

4.2.4 Observational Equivalence

In algebraic specification, two Σ-algebras A and B are observationally equivalent w.r.t. observable sorts *Obs* and input sorts *In* if for any observable computation $t \in T_\Sigma(X^{In})_s$, $s \in \mathit{Obs}$ the interpretations t^A and t^B are equal; if we assume built-in observable and input sorts, *cf.* Sect. 2.4.

Defining observational equivalence w.r.t. observable computations matches how observational equivalence is commonly defined for lambda calculi. As in algebraic specification, we define observational equivalence w.r.t. a finite set *Obs* of observable types. The content of *Obs* will be determined in a moment. We discuss input sorts/types in Sect. 4.2.9. A (virtual) observable computation is a (virtual) client computation whose result type is in *Obs*. We thus define observational equivalence in terms of observable computations in the logic as follows.

Definition 4.4 (Observational Equivalence (ObsEq)**)** *Define* observational equivalence ObsEq w.r.t. $\mathfrak{T}[X]$ and observable types *Obs in the logic by*

$$\begin{aligned}\mathsf{ObsEq}^{Obs}_{\mathfrak{T}[X]} \stackrel{def}{=} (u\!:\!\exists X.\mathfrak{T}[X], v\!:\!\exists X.\mathfrak{T}[X])\ .\\ (\exists A, B.\exists \mathfrak{a}\!:\!\mathfrak{T}[A], \mathfrak{b}\!:\!\mathfrak{T}[B]\ .\ u = (\mathsf{pack} A\mathfrak{a})\ \wedge\ v = (\mathsf{pack} B\mathfrak{b})\ \wedge\\ \textstyle\bigwedge_{D\in Obs} \forall f\!:\!\forall X.(\mathfrak{T}[X]\to D)\ .\ (f A\, \mathfrak{a}) = (f B\, \mathfrak{b}))\end{aligned}$$

For example, an observable computation on natural-number stacks is, for $n\!:\!\mathsf{Nat}$:

$$\Lambda X.\lambda \mathfrak{x}\!:\!\mathfrak{T}_{\mathsf{STACK}_{\mathsf{Nat}}}[X]\ .\ \mathfrak{x}.\mathsf{top}(\mathfrak{x}.\mathsf{push}\, n\ \mathfrak{x}.\mathsf{empty})$$

We omit the $\mathfrak{T}[X]$-subscript to ObsEq whenever it is obvious from context what abstract type body is meant. Observational equivalence of polymorphic data types is defined point-wise, viewing the quantified type variables as parameters. We return to this later.

Notice that the definition of observational equivalence is ultimately in terms of package components. This means that the definition in principle also supports the view of data types as pairs, rather than packages, *cf.* Sect. 3.3.3.

—

Recall for the existence of simulation relations that Def. 3.3 does not yield

$$(\mathsf{pack} A\mathfrak{a})\ \mathsf{SimRel}_{T[X,\boldsymbol{\rho}]}\ (\mathsf{pack} B\mathfrak{b})\ \Rightarrow\ \exists R\subset A\times B\ .\ \mathfrak{a}\ T[R,\boldsymbol{\rho}]\ \mathfrak{b}$$

What $(\mathsf{pack} A\mathfrak{a})\ \mathsf{SimRel}_{T[X,\boldsymbol{\rho}]}\ (\mathsf{pack} B\mathfrak{b})$ gives is packages $(\mathsf{pack} A'\mathfrak{a}')$ and $(\mathsf{pack} B'\mathfrak{b}')$, such that $(\mathsf{pack} A\mathfrak{a}) = (\mathsf{pack} A'\mathfrak{a}')$ and $(\mathsf{pack} B\mathfrak{b}) = (\mathsf{pack} B'\mathfrak{b}')$, and then some $S\subset A'\times B'$ such that $\mathfrak{a}'\ T[S,\boldsymbol{\rho}]\ \mathfrak{b}'$.

Since observational equivalence is defined via proxy packages as well, one might expect a similar weakness for specific packages also here. This is not the case. We have:

Theorem 4.5 *The following sequent schema is derivable.*

$$\begin{aligned}&\forall A, B.\forall \mathfrak{a}\!:\!\mathfrak{T}[A], \mathfrak{b}\!:\!\mathfrak{T}[B]\ .\\ &(\mathsf{pack} A\mathfrak{a})\ \mathsf{ObsEq}^{Obs}_{\mathfrak{T}[X]}\ (\mathsf{pack} B\mathfrak{b})\ \Leftrightarrow\ \textstyle\bigwedge_{D\in Obs} \forall f\!:\!\forall X.(\mathfrak{T}[X]\to D)\ .\ (f A\, \mathfrak{a}) = (f B\, \mathfrak{b})\end{aligned}$$

Proof: $\Rightarrow$: Suppose

$$\begin{aligned}\exists A', B'.\exists \mathfrak{a}'\!:\!\mathfrak{T}[A'], \mathfrak{b}'\!:\!\mathfrak{T}[B']\ .\\ (\mathsf{pack} A\mathfrak{a}) = (\mathsf{pack} A'\mathfrak{a}')\ \wedge\ (\mathsf{pack} B\mathfrak{b}) = (\mathsf{pack} B'\mathfrak{b}')\ \wedge\\ \textstyle\bigwedge_{D\in Obs} \forall f\!:\!\forall X.(\mathfrak{T}[X]\to D)\ .\ (f A'\, \mathfrak{a}') = (f B'\, \mathfrak{b}')\end{aligned}$$

But $(f A'\, \mathfrak{a}') = \mathsf{unpack}(\mathsf{pack} A'\mathfrak{a}')Df = \mathsf{unpack}(\mathsf{pack} A\mathfrak{a})Df = (f A\, \mathfrak{a})$, and similarly for $(f B'\, \mathfrak{b}')$.

$\Leftarrow$: By definition. □

4.2.5 Stability

Stability of a constructor F (*cf.* Sect. 2.5) now translates to

$$u \ \mathsf{ObsEq}^{Obs_{SP'}}\ v \ \Rightarrow\ Fu \ \mathsf{ObsEq}^{Obs_{SP}}\ Fv$$

In the case of polymorphic data types, stability amounts to

$$(\forall \boldsymbol{Z} \,.\, u\boldsymbol{Z} \ \mathsf{ObsEq}^{Obs_{SP'}\cup \boldsymbol{Z}}\ v\boldsymbol{Z}\,) \ \Rightarrow\ (\forall \boldsymbol{Z} \,.\, (Fu)\boldsymbol{Z} \ \mathsf{ObsEq}^{Obs_{SP}\cup \boldsymbol{Z}}\ (Fv)\boldsymbol{Z}\,)$$

We will discuss stability more in Sect. 4.3.1 and Sect. 5.4.3 where we take up the question of whether or not System F provides only stable constructors.

4.2.6 Specifying Abstract Data Types

Now we define specification up to observational equivalence. The idea from algebraic specification is mirrored straightforwardly as follows.

Definition 4.6 (ADT Specification) *An* abstract data type specification *SP is a tuple $\langle\langle Sig_{SP}, \Theta_{SP}\rangle, Obs_{SP}\rangle$ where*

$$Sig_{SP} \stackrel{def}{=} \exists X.\mathfrak{T}_{SP}[X],$$
$$\Theta_{SP}(u) \stackrel{def}{=} \exists X.\exists \mathfrak{x}\!:\!\mathfrak{T}_{SP}[X] \,.\, u \ \mathsf{ObsEq}^{Obs_{SP}}\ (\mathsf{pack}X\mathfrak{x}) \ \wedge\ Ax_{SP}[X,\mathfrak{x}],$$

where $Ax_{SP}[X,\mathfrak{x}]$ is a finite conjunction of formulae. If $\Theta_{SP}(u)$ for $u : Sig_{SP}$ is derivable, then u is said to be a realisation *of SP. The finite set Obs of observable types is given by the specifier, but is assumed to have the following content:*

- *any number of closed inductive types, such as* Bool *or* Nat*,*
- *all parameters $\boldsymbol{Z}$, in case $\mathfrak{T}_{SP}[X]$ has free $\boldsymbol{Z}$ other than X.*

Example 4.7 The analogue to the specification of stacks in Example 2.1 is

$$\mathsf{STACK}_{\mathsf{Nat}} \stackrel{def}{=} \big\langle\big\langle Sig_{\mathsf{STACK}_{\mathsf{Nat}}}, \Theta_{\mathsf{STACK}_{\mathsf{Nat}}}\big\rangle, \{\mathsf{Nat}\}\big\rangle, \text{ where}$$
$$Sig_{\mathsf{STACK}_{\mathsf{Nat}}} = \exists X.\mathfrak{T}_{\mathsf{STACK}_{\mathsf{Nat}}}[X],$$
$$\mathfrak{T}_{\mathsf{STACK}_{\mathsf{Nat}}}[X] = (\mathsf{empty}\!:\!X, \mathsf{push}\!:\!\mathsf{Nat}\!\to\! X\!\to\! X, \mathsf{pop}\!:\!X\!\to\! X, \mathsf{top}\!:\!X\!\to\!\mathsf{Nat}),$$
$$\begin{aligned}\Theta_{\mathsf{STACK}_{\mathsf{Nat}}}(u) = \exists X.\exists\mathfrak{x}\!:\!\mathfrak{T}_{\mathsf{STACK}_{\mathsf{Nat}}}[X] \,.\, u \ \mathsf{ObsEq}^{\{\mathsf{Nat}\}}\ (\mathsf{pack}X\mathfrak{x}) \ \wedge&\\ \forall x\!:\!\mathsf{Nat}, s\!:\!X \,.\, \mathfrak{x}.\mathsf{pop}(\mathfrak{x}.\mathsf{push}\,x\ s) = s \ \wedge&\\ \forall x\!:\!\mathsf{Nat}, s\!:\!X \,.\, \mathfrak{x}.\mathsf{top}(\mathfrak{x}.\mathsf{push}\,x\ s) = x&\end{aligned}$$

○

The realisation predicate $\Theta_{SP}(u)$ of Def. 4.6 expresses *u is observationally equivalent to a package* ($\mathsf{pack}X\mathfrak{x}$) *that satisfies the axioms* Ax_{SP}. Hence specification is up to observational equivalence.

It is the specifier's task to supply $Sig_{SP}[X,\mathfrak{x}]$ and $Ax_{SP}[X,\mathfrak{x}]$. We can imagine a specification language built on top of this type-theoretic setting, with a pre-processor compiling specifications written in this language into type-theoretic formalisms for use with a proof tool. Thus the specifier could write specifications in the style of Ch. 2. We, in the rôle of formalism devisers, will here continue to work in terms of the underlying type-theoretic setting.

4.2.7 Specifying Polymorphic Abstract Data Types

We can also specify polymorphic abstract data types.

Example 4.8 The following specifies stacks as a polymorphic abstract data type.

$$\begin{aligned}
&\mathsf{STACK} \stackrel{def}{=} \langle\langle Sig_{\mathsf{STACK}}, \Theta_{\mathsf{STACK}}\rangle, \{\}\rangle, \text{ where}\\
&Sig_{\mathsf{STACK}} = \forall Z.\exists X.\mathfrak{T}_{\mathsf{STACK}}[X,Z],\\
&\mathfrak{T}_{\mathsf{STACK}}[X,Z] = (\mathsf{empty}\!:\!X, \mathsf{push}\!:\!Z\to X\to X, \mathsf{pop}\!:\!X\to X, \mathsf{top}\!:\!X\to Z\to Z),\\
&\Theta_{\mathsf{STACK}}(u) = \forall Z.\exists X.\exists\mathfrak{x}\!:\!\mathfrak{T}_{\mathsf{STACK}}[X,Z]\ .\ uZ\ \mathsf{ObsEq}^{\{Z\}}\ (\mathsf{pack}X\mathfrak{x})\ \wedge\\
&\qquad\qquad\qquad\qquad\forall z\!:\!Z, s\!:\!X\ .\ \mathfrak{x}.\mathsf{pop}(\mathfrak{x}.\mathsf{push}\,z\ s) = s\ \wedge\\
&\qquad\qquad\qquad\qquad\forall z\!:\!Z, s\!:\!X\ .\ \mathfrak{x}.\mathsf{top}(\mathfrak{x}.\mathsf{push}\,z\ s) = z
\end{aligned}$$

Realisations $u\!:\!Sig_{\mathsf{STACK}}$ are polymorphic implementations of stacks up to point-wise observational equivalence for every instance uZ. ○

Example 4.8 is a specification of a polymorphic ADT. However, the ADT itself contains no polymorphic operations. We deal with polymorphism within data types properly in Ch. 7. The general form of the specification of PADTs is:

Definition 4.9 (PADT Specification) *A* polymorphic abstract data type specification *PSP is a tuple* $\langle\langle Sig_{PSP}, \Theta_{PSP}\rangle, Obs_{PSP}\rangle$ *where*

$$\begin{aligned}
&Sig_{PSP} \stackrel{def}{=} \forall \boldsymbol{Z}.\exists X.\mathfrak{T}_{PSP}[X,\boldsymbol{Z}],\\
&\Theta_{PSP}(u) \stackrel{def}{=}\ \forall \boldsymbol{Z}.\exists X.\exists\mathfrak{x}\!:\!\mathfrak{T}_{PSP}[X,\boldsymbol{Z}]\ .\\
&\qquad\qquad u\boldsymbol{Z}\ \mathsf{ObsEq}^{Obs_{PSP}\cup\boldsymbol{Z}}\ (\mathsf{pack}X\mathfrak{x})\ \wedge\ Ax_{PSP}[X,\boldsymbol{Z},\mathfrak{x}]
\end{aligned}$$

If $\Theta_{PSP}(u)$ *for* $u\!:\!Sig_{PSP}$ *is derivable, then u is said to be a* realisation of *PSP. A realisation* $u : Sig_{PSP}$ *is in other words a polymorphic implementation up to point-wise observational equivalence for every instance* $u\boldsymbol{Z}$.

Realisations are polymorphic, *i.e.,* one realisation yields instances at every type. One cannot have, say, a particularly efficient stack of Booleans as an instance.

In closing, we give an example of a parameterised specification, *i.e.,* simply an ADT specification according to Def. 4.6 with a type parameter.

Example 4.10 Parameterised stacks could be specified by

$$\mathsf{STACK}[Z] \stackrel{def}{=} \langle\langle Sig_{\mathsf{STACK}}, \Theta_{\mathsf{STACK}}\rangle, \{Z\}\rangle, \text{ where}$$

$$Sig_{\mathsf{STACK}} = \exists X.\mathfrak{T}_{\mathsf{STACK}}[X,Z],$$

$$\mathfrak{T}_{\mathsf{STACK}}[X,Z] = (\mathsf{empty}\!:\!X, \mathsf{push}\!:\!Z\to X\to X, \mathsf{pop}\!:\!X\to X, \mathsf{top}\!:\!X\to Z\to Z),$$

$$\begin{aligned}\Theta_{\mathsf{STACK}}(u) = \exists X.\exists\mathfrak{x}\!:\!\mathfrak{T}_{\mathsf{STACK}}[X,Z]\ .\ u\ \mathsf{ObsEq}^{\{Z\}}\ (\mathsf{pack}X\mathfrak{x})\ \wedge&\\ \forall z\!:\!Z, s\!:\!X\ .\ \mathfrak{x}.\mathsf{pop}(\mathfrak{x}.\mathsf{push}\,z\ s) = s\ \wedge&\\ \forall z\!:\!Z, s\!:\!X\ .\ \mathfrak{x}.\mathsf{top}(\mathfrak{x}.\mathsf{push}\,z\ s) = z&\end{aligned}$$

○

Note that a parameterised specification is not necessarily a specification of a parameterised ADT, a realisation of the latter being again something that may be instantiated at any type, that is, in effect, a polymorphic data type. Parameterised specifications are in the outset economisers for the specifier. When used for developing a component in a refinement process, the parameters will at some point get instantiated. The point at which this happens determines to what degree the final realisation can be optimised w.r.t. the parameter instances. If this happens at the end, then what has been developed is in effect a polymorphic data type, *i.e.,* if one at this stage refrains from instantiating the remaining parameters and instead lambda-abstracts these, one gets a proper polymorphic data type.

Specifications as defined in Def. 4.6 resembles a form suggested in (Luo, 1993). Our specifications are meta-level tuples, whereas the Extended Calculus of Constructions in Luo's work admits specifications as first-order citizens. This means one can quantify over parameters of specifications, and this would allow further internalisation into type theory of meta-level statements regarding specification refinement and instantiation. System F is nevertheless sufficient for our purposes.

The ensuing discussion is in terms of ADTs. It should be clear that by definition, we can lift the results obtained also to PADTs.

4.2.8 Observable Types

Notice that according to Def. 4.6, any parameters to the abstract type are included in *Obs*. The motivation for this is, in the context of compound data types, to allow observable computations whose result types are directly element types. For example, for stacks of T, the type T should be observable.

The parameters of an abstract type may be instantiated by any type and thus *Obs* is ultimately capable of containing any type. We as formalists are showing the derivability in the logic of certain sequents, and therefore the contents of *Obs* will in this context be as specified in Def. 4.6. Note that allowing any type in *Obs* via parameters is different from allowing any type in *Obs per se*. In the former case, *Obs* will never contain the instantiated abstract type, *e.g.,* if $\exists X.\mathfrak{T}[X, Z]$ is instantiated with any type T, then T goes in *Obs*, but T can of course never be the resulting $\exists X.\mathfrak{T}[X, T]$. On the other hand, if any type were allowed *a priori* in *Obs*, then the instantiated abstract type $\exists X.\mathfrak{T}[X, T]$ could itself be an observable type. It is conceptually unpleasant to define observational equivalence at an abstract type w.r.t. the abstract type itself. We revisit this issue in Sect. 5.1.1.

We did not consider parameterised data types in detail when presenting algebraic specification refinement in Ch. 2. Although we do not deal with this topic in length here either, not considering type parameters to data types, abstract types and specifications at all in the present polymorphic type theory, would be rather conspicuous. Therefore, the parameters $\boldsymbol{Z}$ will be at least implicitly present in most of the ensuing discussion. The results in Ch. 7 relating to polymorphic operations in data types also rely on our treatment of type parameters here.

The virtual data representation X associated with an observable computation $\Lambda X.\lambda x\!:\!\mathfrak{T}[X].t$ cannot be an observable type. On the package level, this is because of ***Abs-Bar3*** (*p.* 55), *i.e.,* the virtual data representation cannot be the result type of a client computation. Recalling the discussion in Sect. 3.3.3, suppose we view data types as pairs. Then the virtual data representation cannot be an observable type because *Obs* is determined externally to any observable computation.

One may wonder whether there is any point in restricting observable types further than to excluding the data representation, since one in the lambda calculus conceivably might convert any computation to give a result type in *Obs*. However, an observable computation will only have a distinguishing effect insofar as it uses operations over the data representation, *cf.* the uniformity aspect of ***Abs-Bar*** (*p.* 54). This means that it is not in general possible to do such a conversion. This also means that the determining factor for observations is the set of operations with result types other than the data representation. A natural norm is to set *Obs* to contain all types in the abstract type (read signature), and as mentioned above, this excludes the virtual data representation. This is what we will do later on, *cf.* ***FADT***$_{Obs}$ (*p.* 87). Notice that declaring *Obs* also allows us to give a finite conjunction in the definition of observational equivalence in Def 4.4.

4.2.9 Input Types

We now deal with input types corresponding to the notion of input sorts, *cf.* Sect. 2.4. An observable computation $f : \forall X.(\mathfrak{T}[X] \to D)$ may have free variables. Importantly, by ***Abs-Bar***, these free variables cannot be of types containing the virtual data representation X. In the world of algebraic specification, there is no formal restriction on the set *In* of input-sorts. Thus one has to explicitly restrict input sorts to not include what we can call the *behavioural sort*, *e.g.,* Set in Example 2.2 (*p.* 27), when defining observational equivalence. Here the type-theoretic formalism deals with this automatically.

Example 4.11 In Example 2.2, the point was that the sort Set was not in *In*. The corresponding type-theoretic specification is

$$\mathsf{SET}_{\mathsf{Nat}} \stackrel{def}{=} \langle\langle Sig_{\mathsf{SET}_{\mathsf{Nat}}}, \Theta_{\mathsf{SET}_{\mathsf{Nat}}}\rangle, \{\mathsf{Bool}\}\rangle, \text{ where}$$

$$Sig_{\mathsf{SET}_{\mathsf{Nat}}} = \exists X.\mathfrak{T}_{\mathsf{SET}_{\mathsf{Nat}}}[X]$$

$$\begin{aligned} \text{for } \mathfrak{T}_{\mathsf{SET}_{\mathsf{Nat}}}[X] = (&\mathsf{empty} : X,\\ &\mathsf{add} : \mathsf{Nat} \to X \to X,\\ &\mathsf{remove} : \mathsf{Nat} \to X \to X,\\ &\mathsf{in} : \mathsf{Nat} \to X \to \mathsf{Bool}) \end{aligned}$$

$$\Theta_{\mathsf{SET}_{\mathsf{Nat}}}(u) = \exists X.\exists \mathfrak{x} : \mathfrak{T}_{\mathsf{SET}_{\mathsf{Nat}}}[X] \;.\; u \;\mathsf{ObsEq}^{\{\mathsf{Bool}\}}\; (\mathsf{pack} X \mathfrak{x}) \;\wedge\; Ax_{\mathsf{SET}_{\mathsf{Nat}}}$$

$$\begin{aligned} &\text{for } Ax_{\mathsf{SET}_{\mathsf{Nat}}} \stackrel{def}{=}\\ &\forall x : \mathsf{Nat}, s : X \;.\; (\mathfrak{x}.\mathsf{add}\, x\; (\mathfrak{x}.\mathsf{add}\, x\; s)) = (\mathfrak{x}.\mathsf{add}\, x\; s) \;\wedge\\ &\forall x, y : \mathsf{Nat}, s : X \;.\; (\mathfrak{x}.\mathsf{add}\, x\; (\mathfrak{x}.\mathsf{add}\, y\; s)) = (\mathfrak{x}.\mathsf{add}\, y\; (\mathfrak{x}.\mathsf{add}\, x\; s)) \;\wedge\\ &\forall x : \mathsf{Nat} \;.\; (\mathfrak{x}.\mathsf{in}\, x\; \mathfrak{x}.\mathsf{empty}) = \mathsf{false} \;\wedge\\ &\forall x, y : \mathsf{Nat}, s : X \;.\; (\mathfrak{x}.\mathsf{in}\, x\; (\mathfrak{x}.\mathsf{add}\, y\; s)) = \mathsf{if}\; x =_{\mathsf{Nat}} y \;\mathsf{then\; true\; else}\; (\mathfrak{x}.\mathsf{in}\, x\; s) \;\wedge\\ &\forall x : \mathsf{Nat}, s : X \;.\; (\mathfrak{x}.\mathsf{in}\, x\; (\mathfrak{x}.\mathsf{remove}\, x\; s)) = \mathsf{false}\\ &\forall x, y : \mathsf{Nat}, s : X \;.\; x \neq_{\mathsf{Nat}} y \;\Rightarrow\; (\mathfrak{x}.\mathsf{in}\, x\; (\mathfrak{x}.\mathsf{remove}\, y\; s)) = (\mathfrak{x}.\mathsf{in}\, x\; s) \end{aligned}$$

Consider now the package $LI \stackrel{def}{=} (\mathsf{pack}\ \mathsf{List}_{\mathsf{Nat}}\ \mathfrak{l}) : Sig_{\mathsf{SET}_{\mathsf{Nat}}}$, where

$$\begin{aligned} \mathfrak{l} \stackrel{def}{=} (&\mathsf{empty} = \mathsf{nil},\\ &\mathsf{add} = \lambda x : \mathsf{Nat}.\lambda l : \mathsf{List}_{\mathsf{Nat}} \;.\; \textit{return } l' \textit{ that is } x \textit{ inserted uniquely in } l,\\ &\mathsf{remove} = \lambda x : \mathsf{Nat}.\lambda l : \mathsf{List}_{\mathsf{Nat}} \;.\; \textit{return } l' \textit{ that is } l \textit{ with}\\ &\qquad\qquad \textit{first occurrence of } x \textit{ removed},\\ &\mathsf{in} = \lambda x : \mathsf{Nat}.\lambda l : \mathsf{List}_{\mathsf{Nat}} \;.\; \textit{return } \mathsf{true} \textit{ if } x \textit{ occurs in } l, \mathsf{false} \textit{ otherwise}) \end{aligned}$$

By ***Abs-Bar***, users of *LI* may only build lists using operations of $\mathfrak{l}$, such as $\mathfrak{l}.\mathsf{empty}$ and $\mathfrak{l}.\mathsf{add}$, and on such lists the efficient $\mathfrak{l}.\mathsf{remove}$ gives the intended result. By the same token, any observable computation $f : \forall X.(\mathfrak{T}_{\mathsf{SET}_{\mathsf{Nat}}}[X] \to D)$, for $D \in \{\mathsf{Bool}\}$

can only refer to such lists, and not to arbitrary lists. This is the crucial point that admits LI as a realisation of $\mathsf{SET}_{\mathsf{Nat}}$ according to Def. 4.6. For example, in the observable computation $\Lambda X.\lambda\mathfrak{x}:\mathfrak{T}_{\mathsf{SET}_{\mathsf{Nat}}} \,.\, \mathfrak{x}.\mathsf{in}(x, \mathfrak{x}.\mathsf{remove}(x, g))$, g must be a term of the bound type X, and the typing rules, *i.e.,* ***Abs-Bar***, ensure that g cannot be a variable, nor contain any variable of a type containing X. ○

Unlike for observable types, there is no way in the current type-theoretic formalism of restricting or specifying *input types* further, without separate measures. This is nevertheless adequate; the relevant issue is that the data representation is not an input type. And in fact, types that do not occur in the abstract type will have no influence as input types, because variables and terms involving these types will be the same in actual observable computations arising from a single virtual observable computation, *cf.* the uniformity aspect of ***Abs-Bar*** (*p.* 54).

4.2.10 Specification-Building Operators

It is possible to define type-theoretic formalisms mirroring the canonical specification-building operators **sum**, **derive**, and **translate** from Sect. 2.6. Again, a pre-processor could compile a syntax akin to that of an algebraic specification language into type-theoretic formalisms for use in a proof tool.

Alternatively, for specifications using only first-order logic, we can use the already existing algorithm for normalising algebraic specifications. Then it suffices to have a type-theoretic counterpart for the **hide** operator. In fact, it suffices to extend the type-theoretic specification formalism so it can deal with hidden parts. We obtain this simply by extending specifications as defined in Def 4.6, with an extended body containing the original profiles as well as the intended hidden ones.

Definition 4.12 (ADT Specification with Hidden Parts) *An* abstract data type specification SP with hidden parts *is a tuple* $\langle\langle Sig_{SP}, \Theta_{SP}\rangle, \mathfrak{T}^e_{SP}, Obs_{SP}\rangle$ *with*

$$Sig_{SP} \stackrel{def}{=} \exists X.\mathfrak{T}_{SP}[X],$$

$$\Theta_{SP}(u) \stackrel{def}{=} \exists X.\exists\mathfrak{x}:\mathfrak{T}^e_{SP}[X] \,.\, u \;\mathsf{ObsEq}^{Obs_{SP}}\; (\mathsf{pack} X\, \mathfrak{x}|_{\mathfrak{T}_{SP}}) \;\wedge\; Ax_{SP}[X, \mathfrak{x}],$$

where all profiles of $\mathfrak{T}_{SP}[X]$ *occur in* $\mathfrak{T}^e_{SP}[X]$*, and* $\mathfrak{x}|_{\mathfrak{T}_{SP}}$ *denotes* $(\mathfrak{x}.g_1, \ldots, \mathfrak{x}.g_n)$ *for all profiles* $g_i : T_i[X]$ *in* $\mathfrak{T}_{SP}[X]$*. If* $\Theta_{SP}(u)$ *for* $u : Sig_{SP}$ *is derivable, then* u *is said to be a* realisation of SP*. Assumptions on* Ax_{SP} *and* Obs_{SP} *are as in Def. 4.6.*

If $\mathfrak{T}^e_{SP}[X] = \mathfrak{T}_{SP}[X]$ in Def. 4.12, then we get the notion of specification as described in Def. 4.6. In this case we drop $\mathfrak{T}^e_{SP}[X]$ from SP.

Note that the set Obs_{SP} of observable types is defined as usual w.r.t. the signature Sig_{SP} of the specification. Thus, no types occurring in $\mathfrak{T}^e_{SP}[X]$ that are not in $\mathfrak{T}_{SP}[X]$ appear in Obs_{SP}.

Example 4.13 Consider the data-type parameterised specification $\mathsf{TRIVe}[S]$ from Example 2.6. We can give a corresponding type-theoretic specification as follows. To get the effect of data-type parameterisation, we can simply let Y and $\mathfrak{y}:\mathfrak{T}_{\mathsf{STACK}_{\mathsf{Nat}}}[Y]$ be free in the specification.

$$\mathsf{TRIVe} \stackrel{def}{=} \langle\langle Sig_{\mathsf{TRIVe}}, \Theta_{\mathsf{TRIVe}}\rangle, \mathfrak{T}^e_{\mathsf{TRIVe}}, \{\mathsf{Nat}\}\rangle, \text{ where}$$

$$Sig_{\mathsf{TRIVe}} \stackrel{def}{=} \exists X.\mathfrak{T}_{\mathsf{TRIVe}}[X],$$

$$\mathfrak{T}_{\mathsf{TRIVe}}[X] \stackrel{def}{=} (\mathsf{id}:\mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Nat}),$$

$$\mathfrak{T}^e_{\mathsf{TRIVe}}[X] \stackrel{def}{=} (\mathsf{id}:\mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Nat}, \quad \mathsf{multipush}:\mathsf{Nat}\to\mathsf{Nat}\to Y\to Y,\ \mathsf{multipop}:\mathsf{Nat}\to Y\to Y)$$

$$\begin{aligned}\Theta_{\mathsf{TRIVe}}(u) \stackrel{def}{=} {} & \exists X.\exists\mathfrak{x}:\mathfrak{T}^e_{\mathsf{TRIVe}}[X]\ .\ u\ \mathsf{ObsEq}^{\{\mathsf{Nat}\}}\ (\mathsf{pack} X\, \mathfrak{x}|_{\mathfrak{T}_{\mathsf{TRIVe}}}) \ \wedge \\ & Ax_{\mathsf{TRIVe}} : \forall n, z:\mathsf{Nat}, s:Y\ .\ (\mathfrak{x}.\mathsf{multipop}\, n\ (\mathfrak{x}.\mathsf{multipush}\, n\ z\ s)) = s \ \wedge \\ & \quad \forall x, n, z:\mathsf{Nat}\ .\ (\mathfrak{x}.\mathsf{id}\, x\ n\ z) = \\ & \quad\quad \mathfrak{y}.\mathsf{top}(\mathfrak{x}.\mathsf{multipop}\, n\ (\mathfrak{x}.\mathsf{multipush}\, n\ z\ (\mathfrak{y}.\mathsf{push}\, x\ \mathfrak{y}.\mathsf{empty})))\end{aligned}$$

According to Def. 4.12, all type parameters in $\mathfrak{T}_{SP}$ get included in Obs_{SP}. Note that although Y occurs as a parameter in $\mathfrak{T}^e_{\mathsf{TRIVe}}$, this type variable does not get included in Obs_{TRIVe}, since Y does not occur in $\mathfrak{T}_{\mathsf{TRIVe}}$. ○

4.2.11 Specification Refinement

Specification refinement up to observational equivalence can now be expressed in the logic as follows.

Definition 4.14 (Specification Refinement) *Let SP and SP′ be ADT specifications. Then SP′ is a* refinement *of SP, via* constructor $F : Sig_{SP'} \to Sig_{SP}$ *if*

$$\forall u:Sig_{SP'}\ .\ \Theta_{SP'}(u) \ \Rightarrow\ \Theta_{SP}(Fu)$$

is derivable. We write $SP \underset{F}{\rightsquigarrow} SP'$ *for this fact.*

Again, given a program P that is a realisation of SP', the instantiation $F(P)$ is then a realisation of SP. Constructors here correspond to refinement maps in (Luo, 1993) and derived signature morphisms in (Honsell et al., 2000). It is evident that the refinement relation of Def. 4.14 is transitive, *i.e.,* we have *vertical composability* (Goguen and Burstall, 1980):

$$SP \underset{F}{\rightsquigarrow} SP' \ \text{ and } \ SP' \underset{F'}{\rightsquigarrow} SP'' \ \Rightarrow\ SP \underset{F\circ F'}{\rightsquigarrow} SP''$$

where $F \circ F' \stackrel{def}{=} \lambda u : Sig_{SP''}.F(F'u)$. This is the essential fact that allows us to perform stepwise refinement where the result program is given by the composition of the constructors developed through the refinement process, *cf.* Sect. 2.3.

4.2.12 Specifying Constructors

In the algebraic specification of constructors in Sect. 2.9, we can capture that output data types depend on input data types. Here, since we have no term-dependent types, we can only have dependencies that involve non-dependent result signatures. For refinement purposes this is adequate, because the more abstract result signature usually pre-exists before the less abstract input signature. Recall that constructors map in the opposite direction of refinement.

Definition 4.15 (Constructor Specification) *For specifications SP and SP', a* constructor specification $\Pi S\!:\!SP'.SP$ *is a tuple* $\langle Sig_{\Pi S:SP'.SP}, \Theta_{\Pi S:SP'.SP}\rangle$ *where*

$$Sig_{\Pi S:SP'.SP} \stackrel{def}{=} Sig_{SP'} \to Sig_{SP},$$

$$\begin{aligned}\Theta_{\Pi S:SP'.SP}(v,u) \stackrel{def}{=} \exists Y.\exists \mathfrak{y}\!:\!\mathfrak{T}^e_{SP'}[Y].\exists X.\exists \mathfrak{x}\!:\!\mathfrak{T}^e_{SP}[X]\ .&\\ v\ \mathsf{ObsEq}^{Obs_{SP'}}\ (\mathsf{pack} Y\mathfrak{y}|_{\mathfrak{T}_{SP'}})\ \wedge&\\ u\ \mathsf{ObsEq}^{Obs_{SP}}\ (\mathsf{pack} X\mathfrak{x}|_{\mathfrak{T}_{SP}})\ \wedge&\\ Ax_{\Pi S:SP'.SP}[Y,\mathfrak{y},X,\mathfrak{x}]&\end{aligned}$$

where $Ax_{\Pi S:SP'.SP}[Y,\mathfrak{y},X,\mathfrak{x}]$ *is a finite conjunction of formulae. If one can derive*

$$\forall v\!:\!Sig_{SP'}\ .\ \Theta_{SP'}(v)\ \Rightarrow\ \Theta_{\Pi S:SP'.SP}(v, Fv)$$

for a given $F\!:\!Sig_{\Pi S:SP'.SP}$*, then* F *is said to be a* realisation of $\Pi S\!:\!SP'.SP$.

Definition 4.15 incorporates hidden parts. Again, letting $\mathfrak{T}^e_{SP}[X] = \mathfrak{T}_{SP}[X]$ and $\mathfrak{T}^e_{SP'}[Y] = \mathfrak{T}_{SP'}[Y]$ takes us to a situation with no hidden symbols.

Often $Ax_{\Pi S:SP'.SP}[Y,\mathfrak{y},X,\mathfrak{x}]$ is $Ax_{SP}[Y,\mathfrak{y},X,\mathfrak{x}]$, where $Y,\mathfrak{y}$ are substituted in for free variables representing a data type parameter. See however Example 4.27.

Note also that the dependent-product notation $\Pi S : SP'.SP$ in Def. 4.15 is merely suggestive. We do not have dependent products in our type theory. However, we are still able to express dependent constructor specifications.

Example 4.16 A type-theoretic analogue to the specification in Example 2.6 (*p.* 38) is obtained as follows. Recalling specifications $\mathsf{STACK}_{\mathsf{Nat}}$ from Example 4.7, and TRIVe from Example 4.13, we produce:

$$\Pi S\!:\!\mathsf{STACK}_{\mathsf{Nat}}.\mathsf{TRIVe} \stackrel{def}{=} \left\langle Sig_{\Pi S:\mathsf{STACK}_{\mathsf{Nat}}.\mathsf{TRIVe}}, \Theta_{\Pi S:\mathsf{STACK}_{\mathsf{Nat}}.\mathsf{TRIVe}}\right\rangle, \text{ where}$$

$$Sig_{\Pi S:\mathsf{STACK}_{\mathsf{Nat}}.\mathsf{TRIV}} \stackrel{def}{=} Sig_{\mathsf{STACK}_{\mathsf{Nat}}} \to Sig_{\mathsf{TRIVe}},$$

$$\begin{aligned}\Theta_{\Pi S:\mathsf{STACK}_{\mathsf{Nat}}.\mathsf{TRIVe}}(v,u) \stackrel{def}{=} \exists Y.\exists \mathfrak{y}\!:\!\mathfrak{T}_{\mathsf{STACK}_{\mathsf{Nat}}}[Y].\exists X.\exists \mathfrak{x}\!:\!\mathfrak{T}^e_{\mathsf{TRIVe}}[X]\ .&\\ v\ \mathsf{ObsEq}^{\{\mathsf{Nat}\}}\ (\mathsf{pack} Y\mathfrak{y})\ \wedge\ u\ \mathsf{ObsEq}^{\{\mathsf{Nat}\}}\ (\mathsf{pack} X\mathfrak{x}|_{\mathfrak{T}_{\mathsf{TRIVe}}})\ \wedge&\\ \forall n,z\!:\!\mathsf{Nat}, s\!:\!Y\ .\ (\mathfrak{x}.\mathsf{multipop}\, n\ (\mathfrak{x}.\mathsf{multipush}\, n\ z\ s)) = s\ \wedge&\\ \forall x,n,z\!:\!\mathsf{Nat}\ .\ (\mathfrak{x}.\mathsf{id}\, x\ n\ z) =&\\ \mathfrak{y}.\mathsf{top}(\mathfrak{x}.\mathsf{multipop}\, n\ (\mathfrak{x}.\mathsf{multipush}\, n\ z\ (\mathfrak{y}.\mathsf{push}\, x\ \mathfrak{y}.\mathsf{empty})))&\end{aligned}$$

Here, $Ax_{\Pi S:\mathsf{STACK}_{\mathsf{Nat}}.\mathsf{TRIVe}}$ is Ax_{TRIVe}. ○

4.3 Results and Simplifications at First Order

If $\mathfrak{T}[X]$ is first-order, we get a string of interesting results in the logic. The main result here is that parametricity gives the coincidence of equality at existential type with observational equivalence. Thus, parametricity implies the true indistinguishability of observationally equivalent data types. This reduces our type-theoretic notions of specification and specification refinement in the previous section to simple ones in terms of equality. Additionally, since observational equivalence coincides with equality, all constructors are automatically stable.

Note first that any profile of $\exists X.\mathfrak{T}[X]$ (or indeed any type) $T_i[X]$ has the form $T_{i1}[X] \to \cdots \to T_{n_i}[X] \to T_{c_i}[X]$ where $T_{c_i}[X]$ is not an arrow type, with the understanding that if $n_i = 0$, then $T_i[X]$ is $T_{c_i}[X]$. We presume a specification scenario, and hence a current set of observable types Obs according to Def. 4.6 or Def. 4.12. We will mainly be working with the following assumption:

FADT_{Obs}**:** Every profile $T_i[X] = T_{i1}[X] \to \cdots \to T_{n_i}[X] \to T_{c_i}[X]$ of $\exists X.\mathfrak{T}[X]$ is first order, and such that $T_{c_i}[X]$ is either X or some $D \in Obs$.

Thus, we wish only to hide the data representation in a data type. Notice the dependency between $\mathfrak{T}[X]$ and Obs.

We first establish that the existence of a simulation relation is equivalent to observational equivalence.

Theorem 4.17 *Suppose $\mathfrak{T}[X]$ adheres to FADT_{Obs}. With* PARAM *we derive*

$$\forall \boldsymbol{Z}.\forall u, v : \exists X.\mathfrak{T}[X, \boldsymbol{Z}] \;.\; u \;\mathsf{SimRel}_{\mathfrak{T}[X,\mathbf{eq}_{\boldsymbol{Z}}]}\; v \quad\Leftrightarrow\quad u \;\mathsf{ObsEq}^{Obs}_{\mathfrak{T}[X,\boldsymbol{Z}]}\; v$$

Proof: This follows from Theorem 4.18 below. □

Theorem 4.18 (Tight Correspondence) *Suppose $\mathfrak{T}[X]$ adheres to FADT_{Obs}. With* PARAM *we derive*

$$\begin{aligned}
&\forall A, B.\forall \mathfrak{a} : \mathfrak{T}[A, \boldsymbol{Z}], \mathfrak{b} : \mathfrak{T}[B, \boldsymbol{Z}] \;.\\
&\qquad \exists R \subset A \times B \;.\; \mathfrak{a}(\mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}])\mathfrak{b}\\
&\qquad\qquad\qquad \Leftrightarrow \quad \textstyle\bigwedge_{D \in Obs} \forall f : \forall X.(\mathfrak{T}[X, \boldsymbol{Z}] \to D) \;.\; (f\,A\,\mathfrak{a}) = (f\,B\,\mathfrak{b})
\end{aligned}$$

Proof: $\Rightarrow$: This follows from the parametricity axiom schema. Consider the PARAM instance $\forall Y.\forall f : \forall X.(\mathfrak{T}[X, \boldsymbol{Z}] \to Y) \;.\; f(\forall X.\mathfrak{T}[X, \mathbf{eq}_{\boldsymbol{Z}}] \to \mathsf{eq}_Y) f$. Unravelling the definitions, this becomes more than sufficient:

$$\begin{aligned}
&\forall Y \forall f : \forall X.(\mathfrak{T}[X, \boldsymbol{Z}] \to Y).\forall A, B, R \subset A \times B \;.\; \forall \mathfrak{a} : \mathfrak{T}[A, \boldsymbol{Z}], \mathfrak{b} : \mathfrak{T}[B, \boldsymbol{Z}] \;.\\
&\qquad\qquad\qquad \mathfrak{a}(\mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}])\mathfrak{b} \;\Rightarrow\; (f\,A\,\mathfrak{a}) = (f\,B\,\mathfrak{b})
\end{aligned}$$

$\Leftarrow$: We must exhibit an R such that $\mathfrak{a}(\mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}])\mathfrak{b}$. On the semantic level, (Mitchell, 1991, 1996) and (Schoett, 1990, 1986) relate elements if they are denotable by some common term. We mimic this: For R give

$$\mathsf{Dfnbl} \stackrel{def}{=} (a\!:\!A, b\!:\!B)\ .\ (\exists f\!:\!\forall X.(\mathfrak{T}[X, \boldsymbol{Z}] \to X)\ .\ (f\,A\,\mathfrak{a}) = a\ \wedge\ (f\,B\,\mathfrak{b}) = b)$$

We must now derive $\mathfrak{a}(\mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}])\mathfrak{b}$. Due to the assumption FADT_{Obs}, it suffices to show for every component $(g\!:\!T_1 \to \cdots \to T_n \to T_c)$ in $\mathfrak{T}[X, \boldsymbol{Z}]$ that

$$\begin{aligned}\forall u_1\!:\!T_1[A, \boldsymbol{Z}], \ldots, \forall u_n\!:\!T_n[A, \boldsymbol{Z}], \forall v_1\!:\!T_1[B, \boldsymbol{Z}], \ldots, \forall v_n\!:\!T_n[B, \boldsymbol{Z}]\ .\\ u_1\ T_1[R, \mathbf{eq}_{\boldsymbol{Z}}]\ v_1\ \wedge\ \cdots\ \wedge\ u_n\ T_n[R, \mathbf{eq}_{\boldsymbol{Z}}]\ v_n\\ \Rightarrow\ (\mathfrak{a}.g\,u_1 \cdots u_n)\ T_c[R, \mathbf{eq}_{\boldsymbol{Z}}]\ (\mathfrak{b}.g\,v_1 \cdots v_n)\end{aligned}$$

Under our present assumptions, T_c and any T_j in the antecedent is either X, some Z_i, or else some other closed type $D \in \mathit{Obs}$. If T_j is X then the antecedent says $u_j\ R\ v_j$ and we may assume $\exists f_j\!:\!\forall X.(\mathfrak{T}[X, \boldsymbol{Z}] \to X).(f_j A\,\mathfrak{a}) = u_j\ \wedge\ (f_j B\,\mathfrak{b}) = v_j$. If T_j is some Z_i, we may immediately assume $u_j =_{Z_i} v_j$. If T_j is a closed observable type $D \in \mathit{Obs}$ we may by the Identity Extension Lemma (Theorem 3.2) also assume $u_j =_D v_j$. Consider $f \stackrel{def}{=} \Lambda X.\lambda\mathfrak{x} : \mathfrak{T}[X, \boldsymbol{Z}]\ .\ (\mathfrak{x}.g\,x_1 \cdots x_n)$, where x_j is $(f_j X \mathfrak{x})$ if T_j is X, and $x_j = u_j$ otherwise.

Suppose now T_c is an observable type $D \in \mathit{Obs}$, including any parameter Z_i. By assumption we have $(f\,A\,\mathfrak{a}) = (f\,B\,\mathfrak{b})$ and by β-equality we are done.

Suppose T_c is X. Then we must derive $(\mathfrak{a}.g\,u_1 \cdots u_n)\ R\ (\mathfrak{b}.g\,v_1 \cdots v_n)$, *i.e.*, that $\exists f : \forall X.(\mathfrak{T}[X, \boldsymbol{Z}] \to X).(f\,A\,\mathfrak{a}) = (\mathfrak{a}.g\,u_1 \cdots u_n)\ \wedge\ (f\,B\,\mathfrak{b}) = (\mathfrak{b}.g\,v_1 \cdots v_n)$. But then we exhibit our f above, and we are done. □

Theorems 4.17 and 4.18 are methodologically important, because in general it is hard to show observational equivalence directly, whereas showing the existence of a simulation relation is more tractable given two specific packages.

We also get the correspondence between equality at existential type and observational equivalence.

Theorem 4.19 *Suppose $\mathfrak{T}[X]$ adheres to FADT_{Obs}. With* PARAM *we derive*

$$\forall u, v\!:\!\exists X.\mathfrak{T}[X, \boldsymbol{Z}]\ .\ u =_{\exists X.\mathfrak{T}[X, \boldsymbol{Z}]} v \quad\Leftrightarrow\quad u\ \mathsf{ObsEq}^{Obs}_{\mathfrak{T}[X, \boldsymbol{Z}]}\ v$$

Proof: Together with Theorem 3.5, Theorem 4.17 gives the result. □

Theorem 4.19 means that, under relational parametricity, observational abstraction is inherent to inhabitants of existential type and hence to our notion of data type, but recall Sect. 3.3.3. By Theorem 4.19 we can substitute equality for

$\mathsf{ObsEq}^{Obs_{SP}}$ in Def. 4.6, the definition of specification. This reduces our expressions of specification and specification refinement to ones in terms of equality.

In the first-order case, we get the tight correspondence between equality at existential type and the existence of a simulation relation. This is due to the following theorem, which ameliorates the situation depicted by Theorem 3.7 (*p.* 64).

Theorem 4.20 *Suppose* $\mathfrak{T}[X]$ *adheres to* FADT_{Obs}. *With* PARAM *we derive*

$$\begin{array}{l}\forall A, B. \forall \mathfrak{a}{:}\mathfrak{T}[A, \boldsymbol{Z}], \mathfrak{b}{:}T[B, \boldsymbol{Z}])\ .\\ \qquad (\mathsf{pack} A\mathfrak{a})\ \mathsf{SimRel}_{\mathfrak{T}[X,\mathbf{eq}_{\boldsymbol{Z}}]}\ (\mathsf{pack} B\mathfrak{b}) \quad \Leftrightarrow \quad \exists R \subset A \times B\ .\ \mathfrak{a}\ \mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}]\ \mathfrak{b}\end{array}$$

Proof: Recall from the discussion at the end of Sect. 3.3.2, that it is the direction from left to right that is the issue. Suppose $(\mathsf{pack} A\mathfrak{a})\ \mathsf{SimRel}_{\mathfrak{T}[X,\mathbf{eq}_{\boldsymbol{Z}}]}\ (\mathsf{pack} B\mathfrak{b})$. Theorem 4.17 gives $(\mathsf{pack} A\mathfrak{a})\ \mathsf{ObsEq}^{Obs}_{\mathfrak{T}[X,\boldsymbol{Z}]}\ (\mathsf{pack} B\mathfrak{b})$. Then Theorem 4.5 and Theorem 4.18 give the result. □

Then for packages, we get a version of Theorem 3.5 (*p.* 63) at first order without any circular features, *i.e.,* without indirection and reference to equality on the simulation relation side.

Theorem 4.21 *Suppose* $\mathfrak{T}[X]$ *adheres to* FADT_{Obs}. *With* PARAM *we derive*

$$\begin{array}{l}\forall A, B. \forall \mathfrak{a}{:}\mathfrak{T}[A, \boldsymbol{Z}], \mathfrak{b}{:}T[B, \boldsymbol{Z}])\ .\\ \qquad (\mathsf{pack} A\mathfrak{a})\ =_{\exists X.\mathfrak{T}[X,\mathbf{eq}_{\boldsymbol{Z}}]}\ (\mathsf{pack} B\mathfrak{b}) \quad \Leftrightarrow \quad \exists R \subset A \times B\ .\ \mathfrak{a}\ \mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}]\ \mathfrak{b}\end{array}$$

Proof: Directly from Theorem 4.20 and Theorem 3.5. □

Recall the discussion in Sect. 3.3.3 concerning the view of data types as pairs, rather than as packages of existential type. The results above relate to this view as well. The central theorem is Theorem 4.5, which speaks in terms of package components. Moreover, even when we speak in terms of packages in Theorem 4.17 and then in Theorem 4.19, and also Theorem 3.5, we may immediately move to the component level by using theorems 4.5, 4.20, and 4.21, for $\mathfrak{T}[X]$ satisfying FADT_{Obs}, *i.e.,* for (abstract) (data) types with first-order operations.

4.3.1 Inherent Stability

Under parametricity, any constructor F is inherently stable, *i.e.,*

$$u\ \mathsf{ObsEq}^{Obs_{SP'}}\ v \ \Rightarrow\ F(u)\ \mathsf{ObsEq}^{Obs_{SP}}\ F(v)$$

simply by congruence for equality. Congruence gives

$$\forall u, v : Sig_{SP'} \, . \; u =_{Sig_{SP'}} v \;\Rightarrow\; F(u) =_{Sig_{SP}} F(v)$$

and equality at existential type is of course observational equivalence. Later on, in Sect. 5.4.3 we will see that the stability of System F constructors is in fact inherent in the type-theoretic setting, independently of this link to equality.

Stability simplifies observational proofs significantly, because one can then assume literal satisfaction of the implementing specification's axioms when proving refinement, see (Sannella and Tarlecki, 1997; Schoett, 1990) and Sect. 2.5.

Note that we automatically get this proof simplification due to stability. Since observational equivalence is simply equality in the type theory, it is sound to substitute any package with an observationally equivalent package that satisfies the axioms of the specification literally.

Observe that the non-stable constructor Tr' from Example 2.4 is not expressible here, because the proposition $x =_X y$ is not allowed in System F terms.

4.3.2 Simulation Relations Compose at First Order

Relating data types by simulation relations goes under the heading of *data refinement*. There are thus two refinement dimensions; one concerning specifications, and within each stage of this refinement process, a second dimension concerning observational equivalence, *i.e.,* simulation relations, *i.e.,* data refinement.

Theorem 4.17 means that we can explain observational equivalence, and thus also specification refinement up to observational equivalence, in terms of the existence of simulation relations. At first order, Theorem 3.5 and Theorem 4.21 give the essential property that the existence of simulation relations is transitive, but we can actually give a more constructive result:

Theorem 4.22 (Composability of Simulation Relations) *Suppose* $\mathfrak{T}[X]$ *adheres to* $\mathtt{FADT}_{Obs}$*. Then we can derive*

$$\begin{aligned}&\forall A, B, C, R \subset A \times B, S \subset B \times C, \mathfrak{a} : \mathfrak{T}[A, \boldsymbol{Z}], \mathfrak{b} : \mathfrak{T}[B, \boldsymbol{Z}], \mathfrak{c} : \mathfrak{T}[C, \boldsymbol{Z}].\\ &\qquad\qquad \mathfrak{a}(\mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}])\mathfrak{b} \;\wedge\; \mathfrak{b}(\mathfrak{T}[S, \mathbf{eq}_{\boldsymbol{Z}}])\mathfrak{c} \;\Rightarrow\; \mathfrak{a}(\mathfrak{T}[S \circ R, \mathbf{eq}_{\boldsymbol{Z}}])\mathfrak{c}\end{aligned}$$

Proof: Assuming $\mathfrak{a}(\mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}])\mathfrak{b} \wedge \mathfrak{b}(\mathfrak{T}[S, \mathbf{eq}_{\boldsymbol{Z}}])\mathfrak{c}$, the goal is to derive for every component $(g : T_1 \to \cdots T_n \to T_c)$ in $\mathfrak{T}[X, \boldsymbol{Z}]$

$$\begin{aligned}&\forall u_1 : T_1[A, \boldsymbol{Z}], \ldots, \forall u_n : T_n[A, \boldsymbol{Z}], \forall w_1 : T_1[C, \boldsymbol{Z}], \ldots, \forall w_n : T_n[C, \boldsymbol{Z}] \; .\\ &\qquad u_1 \; T_1[S \circ R, \mathbf{eq}_{\boldsymbol{Z}}] \; w_1 \;\wedge\; \cdots \;\wedge\; u_n \; T_n[S \circ R, \mathbf{eq}_{\boldsymbol{Z}}] \; w_n\\ &\qquad\qquad \Rightarrow\; (\mathfrak{a}.g \, u_1 \cdots u_n) \; T_c[S \circ R, \mathbf{eq}_{\boldsymbol{Z}}] \; (\mathfrak{c}.g \, w_1 \cdots w_n)\end{aligned}$$

If $T_i[X] = X$, the antecedent says $\exists v_i : B \,.\, u_i \; R \; v_i \wedge v_i \; S \; w_i$. In this case, and also in the remaining case according to FADT_{Obs}, that T_i is in Obs, the assumption gives $(\mathfrak{a}.g\, u_1 \cdots u_n)\; T_c[R, \mathbf{eq}_{\boldsymbol{Z}}]\; (\mathfrak{b}.g\, v_1 \cdots v_n) \;\wedge\; (\mathfrak{b}.g\, v_1 \cdots v_n)\; T_c[S, \mathbf{eq}_{\boldsymbol{Z}}]\; (\mathfrak{c}.g\, w_1 \cdots w_n)$, in other words, $(\mathfrak{a}.g\, u_1 \cdots u_n)\; T_c[S \circ R, \mathbf{eq}_{\boldsymbol{Z}}]\; (\mathfrak{c}.g\, w_1 \cdots w_n)$. □

Again, this result is applicable both to the view of data types as packages and to the view of data types as pairs.

Thus simulation relations explain stepwise refinement, but methodologically this is not enough. Given instances u and v and constructor F one can check that there is a simulation relation relating (Fu) and v. But for proving a specification refinement $SP \underset{F}{\rightsquigarrow} SP'$, one requires that this be done for all $u : Sig_{SP'}$. A pointwise method of verifying a refinement step is thus impractical. One would prefer a general method for proving refinement. Such a method would work on the specifications SP and SP', rather than on individual models. We reviewed such a general method for algebraic specification in Sect. 2.7. In the next section we express this strategy in the type-theoretic setting, using the results on simulation relations, observational equivalence and equality obtained in this section.

4.4 Importing a Universal Proof Strategy

A general method for proving specification refinement up to observational equivalence has been developed in the realm of algebraic specification, *cf.* Sect. 2.7. The method uses only abstract information supplied by the specifications. One proves observational refinements by considering quotients w.r.t. a partial congruence (Bidoit et al., 1995), and then one uses an axiomatisation of this congruence to prove relativised versions of the axioms of the specification to be refined. In general, clauses restricting to the domain of the congruence must also be incorporated (Bidoit et al., 1997; Bidoit and Hennicker, 1996).

As observed in (Poll and Zwanenburg, 1999) and (Hannay, 1999a), this method is evidently not expressible in the type theory or the logic of (Plotkin and Abadi, 1993). One remedy would be to augment the type theory with quotient types, *e.g.,* (Hofmann, 1995b), and something to the effect of subset types. The quotient types of (Hofmann, 1995b) are in fact macro-like definitions in terms of the underlying type theory, but as with subset-types this demands dependent types with accompanying difficulties. The alternative simple solution inspired by (Poll and Zwanenburg, 1999) is to soundly add axioms enabling the axiomatisation of partial congruences. We give the axiom schemata below. The first schema postulates the existence of subobjects, the second schema postulates the existence of quotients. The axiom schema SUB extends the discussion in (Poll and Zwanenburg, 1999) to deal also with partial congruences. The schema appeared independently in (Zwanenburg, 1999) and (Hannay, 1999a), but the one in (Poll and Zwanenburg, 1999) is far more general, in that it treats restrictions via predicates in general, albeit satisfying certain criteria.

Rather than being fundamental, the schemata below are tailored to suit refinement-proof purposes. One could in principle give proper fundamental axioms and then derive the schemata below from these. However, in (Zwanenburg, 1999) it is suggested that this is difficult to do in a proper manner, unless one assumes the following schema for *axiom of choice*:

$$\mathrm{AC} : \forall X, Y \;.\; (\forall x\!:\!X.\exists y\!:\!Y \;.\; \phi[x,y]) \;\Rightarrow\; (\exists f\!:\!X\!\to\!Y.\forall x\!:\!X \;.\; \phi[x,(fx)])$$

for all formula ϕ. However, AC does not hold in the parametric PER-model.

Definition 4.23 (Existence of Subobjects (SUB) (Hannay, 1999a)**)**

$$\begin{array}{ll} \textsc{Sub}: & \forall X\,.\,\forall\mathfrak{x}\!:\!\mathfrak{T}[X,\boldsymbol{Z}]\,.\,\forall R\!\subset\! X\!\times\! X \;.\quad (\mathfrak{x}\;\mathfrak{T}[R,\mathbf{eq}_{\boldsymbol{Z}}]\;\mathfrak{x}) \;\Rightarrow \\ & \quad \exists S\,.\,\exists\mathfrak{s}\!:\!\mathfrak{T}[S,\boldsymbol{Z}]\,.\,\exists R'\!\subset\! S\!\times\! S\,.\,\exists\mathsf{mono}\!:\!S\!\to\! X \;. \\ & \qquad \forall s\!:\!S \;.\; s\;R'\;s \hfill \wedge \\ & \qquad \forall s,s'\!:\!S \;.\; s\;R'\;s' \;\Leftrightarrow\; (\mathsf{mono}\,s)\;R\;(\mathsf{mono}\,s') \quad \wedge \\ & \qquad \mathfrak{x}\;(\mathfrak{T}[(x\!:\!X,s\!:\!S).(x =_X (\mathsf{mono}\,s)),\mathbf{eq}_{\boldsymbol{Z}}])\;\mathfrak{s} \end{array}$$

Intuitively, this essentially states that for any data type or algebra ($\mathsf{pack}X\mathfrak{x}$), if R is a relation that is compatible with the "signature" $\exists X.\mathfrak{T}[X]$, *i.e.,* if R is a partial congruence on ($\mathsf{pack}X\mathfrak{x}$), then there exists a data type ($\mathsf{pack}S\mathfrak{s}$), a relation R', and a map mono, which one can think of as a monomorphism (injection) from the algebra ($\mathsf{pack}S\mathfrak{s}$) to ($\mathsf{pack}X\mathfrak{x}$), such that R' is total on the algebra ($\mathsf{pack}S\mathfrak{s}$) and a restriction of R via mono, and such that ($\mathsf{pack}S\mathfrak{s}$) is a subalgebra of ($\mathsf{pack}X\mathfrak{x}$).

Definition 4.24 (Existence of Quotients (QUOT) (Zwanenburg, 1999)**)**

$$\begin{array}{ll} \textsc{Quot}: & \forall X\,.\,\forall\mathfrak{x}\!:\!\mathfrak{T}[X,\boldsymbol{Z}]\,.\,\forall R\!\subset\! X\!\times\! X \;.\quad (\mathfrak{x}\;\mathfrak{T}[R,\mathbf{eq}_{\boldsymbol{Z}}]\;\mathfrak{x} \;\wedge\; \mathit{equiv}(R)) \;\Rightarrow \\ & \quad \exists Q\,.\,\exists\mathfrak{q}\!:\!\mathfrak{T}[Q,\boldsymbol{Z}]\,.\,\exists\mathsf{epi}\!:\!X\!\to\! Q \;. \\ & \qquad \forall x,y\!:\!X \;.\; xRy \;\Leftrightarrow\; (\mathsf{epi}\,x) =_Q (\mathsf{epi}\,y) \quad \wedge \\ & \qquad \forall q\!:\!Q.\exists x\!:\!X \;.\; q =_Q (\mathsf{epi}\,x) \hfill \wedge \\ & \qquad \mathfrak{x}\;(\mathfrak{T}[(x\!:\!X,q\!:\!Q).((\mathsf{epi}\,x) =_Q q),\mathbf{eq}_{\boldsymbol{Z}}])\;\mathfrak{q} \end{array}$$

where $\mathit{equiv}(R)$ *specifies* R *to be an equivalence relation.*

Intuitively, this states that for any data type or algebra ($\mathsf{pack}X\mathfrak{x}$), if R is an equivalence relation on ($\mathsf{pack}X\mathfrak{x}$), then there exists a data type ($\mathsf{pack}Q\mathfrak{q}$) and a map epi, which can be seen as an epimorphism (surjection) from the algebra ($\mathsf{pack}X\mathfrak{x}$) to ($\mathsf{pack}Q\mathfrak{q}$), such that ($\mathsf{pack}Q\mathfrak{q}$) is a quotient algebra of ($\mathsf{pack}X\mathfrak{x}$).

Theorem 4.25 *Suppose* $\mathfrak{T}[X]$ *adheres to* FADT_{Obs}*. Then* SUB *and* QUOT *hold in the parametric PER-model of* (Bainbridge et al., 1990).

Proof: Easy. See Sections 6.5.1 and 6.5.2 for the definitions of subobject and quotient PERs and a validation of generalisations of SUB and QUOT. □

It is evident that SUB and QUOT are not derivable from the logic for parametric polymorphism, so the logic is by Theorem 4.25 not complete w.r.t. the parametric PER-model. Completeness issues of the extended logic is left as future work.

—

Below we give two basic examples to illustrate the use of the axioms and hence the proof method now imported into type theory. We refer to (Poll and Zwanenburg, 1999; Zwanenburg, 1999) for further examples using variants of QUOT and SUB, and to refinement examples in the literature for sources to other examples using this framework. In Sect. 6.6, we give a general schema for the use of variants of QUOT and SUB that also work for data types with higher-order operations. In Sect. 4.6 the axioms are instrumental in showing the correspondence between refinement in type theory and refinement in algebraic specification.

Example 4.26 Recall the specification of sets from Example 4.11:

$$\mathsf{SET}_{\mathsf{Nat}} \stackrel{def}{=} \langle\langle Sig_{\mathsf{SET}_{\mathsf{Nat}}}, \Theta_{\mathsf{SET}_{\mathsf{Nat}}}\rangle, \{\mathsf{Bool}\}\rangle, \text{ where}$$

$$Sig_{\mathsf{SET}_{\mathsf{Nat}}} = \exists X.\mathfrak{T}_{\mathsf{SET}_{\mathsf{Nat}}}[X]$$

$$\begin{aligned}\text{for } \mathfrak{T}_{\mathsf{SET}_{\mathsf{Nat}}}[X] = (&\mathsf{empty}\!:\!X,\\ &\mathsf{add}\!:\!\mathsf{Nat}\to X\to X,\\ &\mathsf{remove}\!:\!\mathsf{Nat}\to X\to X,\\ &\mathsf{in}\!:\!\mathsf{Nat}\to X\to\mathsf{Bool})\end{aligned}$$

$$\Theta_{\mathsf{SET}_{\mathsf{Nat}}}(u) = \exists X.\exists \mathfrak{x}\!:\!\mathfrak{T}_{\mathsf{SET}_{\mathsf{Nat}}}[X]\ .\ u\ \mathsf{ObsEq}^{\{\mathsf{Bool}\}}\ (\mathsf{pack}X\mathfrak{x})\ \wedge\ Ax_{\mathsf{SET}_{\mathsf{Nat}}}$$

$$\begin{aligned}&\text{for } Ax_{\mathsf{SET}_{\mathsf{Nat}}} \stackrel{def}{=}\\ &\quad\forall x\!:\!\mathsf{Nat}, s\!:\!X\ .\ (\mathfrak{x}.\mathsf{add}\,x\ (\mathfrak{x}.\mathsf{add}\,x\ s)) = (\mathfrak{x}.\mathsf{add}\,x\ s)\ \wedge\\ &\quad\forall x,y\!:\!\mathsf{Nat}, s\!:\!X\ .\ (\mathfrak{x}.\mathsf{add}\,x\ (\mathfrak{x}.\mathsf{add}\,y\ s)) = (\mathfrak{x}.\mathsf{add}\,y\ (\mathfrak{x}.\mathsf{add}\,x\ s))\ \wedge\\ &\quad\forall x\!:\!\mathsf{Nat}\ .\ (\mathfrak{x}.\mathsf{in}\,x\ \mathfrak{x}.\mathsf{empty}) = \mathsf{false}\ \wedge\\ &\quad\forall x,y\!:\!\mathsf{Nat}, s\!:\!X\ .\ (\mathfrak{x}.\mathsf{in}\,x\ (\mathfrak{x}.\mathsf{add}\,y\ s)) = \mathsf{if}\ x =_{\mathsf{Nat}} y\ \mathsf{then\ true\ else}\ (\mathfrak{x}.\mathsf{in}\,x\ s)\ \wedge\\ &\quad\forall x\!:\!\mathsf{Nat}, s\!:\!X\ .\ (\mathfrak{x}.\mathsf{in}\,x\ (\mathfrak{x}.\mathsf{remove}\,x\ s)) = \mathsf{false}\\ &\quad\forall x,y\!:\!\mathsf{Nat}, s\!:\!X\ .\ x \neq_{\mathsf{Nat}} y\ \Rightarrow\ (\mathfrak{x}.\mathsf{in}\,x\ (\mathfrak{x}.\mathsf{remove}\,y\ s)) = (\mathfrak{x}.\mathsf{in}\,x\ s)\end{aligned}$$

A nice feature in refinement settings is the provision for using an implementation of one abstract data type to implement another. In the example below, the specification $\mathsf{SET}_{\mathsf{Nat}}$ is refined by using the specification $\mathsf{BAG}_{\mathsf{Nat}}$ specifying bags or multisets. This reuses any refinement path $\mathsf{BAG}_{\mathsf{Nat}} \underset{F}{\overset{*}{\rightsquigarrow}} SP''$ previously done for $\mathsf{BAG}_{\mathsf{Nat}}$. In particular if $\mathsf{BAG}_{\mathsf{Nat}}$ has been refined to an executable module, then this code is reused when implementing $\mathsf{SET}_{\mathsf{Nat}}$. The specification $\mathsf{BAG}_{\mathsf{Nat}}$ goes like this:

$$
\begin{aligned}
&\mathsf{BAG}_{\mathsf{Nat}} \overset{def}{=} \langle\langle Sig_{\mathsf{BAG}_{\mathsf{Nat}}}, \Theta_{\mathsf{BAG}_{\mathsf{Nat}}}\rangle, \{\mathsf{Nat}\}\rangle, \text{ where}\\
&Sig_{\mathsf{BAG}_{\mathsf{Nat}}} = \exists X . \mathfrak{T}_{\mathsf{BAG}_{\mathsf{Nat}}}[X]\\
&\text{for } \mathfrak{T}_{\mathsf{BAG}_{\mathsf{Nat}}}[X] = (\mathsf{empty} : X,\\
&\qquad\qquad\qquad\quad \mathsf{add} : \mathsf{Nat} \to X \to X,\\
&\qquad\qquad\qquad\quad \mathsf{remove} : \mathsf{Nat} \to X \to X,\\
&\qquad\qquad\qquad\quad \mathsf{count} : \mathsf{Nat} \to \mathsf{Nat} \to X \to \mathsf{Bool})\\
&\Theta_{\mathsf{BAG}_{\mathsf{Nat}}}(u) = \exists X . \exists \mathfrak{x} : \mathfrak{T}_{\mathsf{BAG}_{\mathsf{Nat}}}[X] \;.\; u \; \mathsf{ObsEq}^{\{\mathsf{Nat}\}} \; (\mathsf{pack} X \mathfrak{x}) \;\wedge\; Ax_{\mathsf{BAG}_{\mathsf{Nat}}}\\
&\text{for } Ax_{\mathsf{BAG}_{\mathsf{Nat}}} \overset{def}{=}\\
&\quad \forall x, y : \mathsf{Nat}, s : X \;.\; (\mathfrak{x}.\mathsf{add}\, x \; (\mathfrak{x}.\mathsf{add}\, y \; s)) = (\mathfrak{x}.\mathsf{add}\, y \; (\mathfrak{x}.\mathsf{add}\, x \; s)) \;\wedge\\
&\quad \forall x : \mathsf{Nat} \;.\; (\mathfrak{x}.\mathsf{count}\, x \; \mathfrak{x}.\mathsf{empty}) = 0 \;\wedge\\
&\quad \forall x, y : \mathsf{Nat}, s : X \;.\; (\mathfrak{x}.\mathsf{count}\, x \; (\mathfrak{x}.\mathsf{add}\, y \; s)) = \mathsf{if}\; x =_{\mathsf{Nat}} y \;\mathsf{then}\; (\mathsf{succ}\, (\mathfrak{x}.\mathsf{count}\, x \; s))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\; (\mathfrak{x}.\mathsf{count}\, x \; s) \quad \wedge\\
&\quad \forall x : \mathsf{Nat}, s : X \;.\; (\mathfrak{x}.\mathsf{count}\, x \; (\mathfrak{x}.\mathsf{remove}\, n \; x \; s)) = (\mathfrak{x}.\mathsf{count}\, x \; s) - n\\
&\quad \forall x, y : \mathsf{Nat}, s : X \;.\; x \neq_{\mathsf{Nat}} y \;\Rightarrow\; (\mathfrak{x}.\mathsf{count}\, x \; (\mathfrak{x}.\mathsf{remove}\, n \; y \; s)) = (\mathfrak{x}.\mathsf{count}\, x \; s)
\end{aligned}
$$

We now show that

$$\mathsf{SET}_{\mathsf{Nat}} \underset{F}{\rightsquigarrow} \mathsf{BAG}_{\mathsf{Nat}}$$

for

$$F \overset{def}{=} \lambda u : Sig_{\mathsf{BAG}_{\mathsf{Nat}}} . \mathsf{unpack}(u)(Sig_{\mathsf{SET}_{\mathsf{Nat}}})(\Lambda X . \lambda \mathfrak{x} : \mathfrak{T}_{\mathsf{BAG}_{\mathsf{Nat}}}[X] \;.\; (\mathsf{pack} X \mathfrak{x}'))$$

where

$$
\begin{aligned}
\mathfrak{x}' \overset{def}{=} (&\mathsf{empty} = \mathfrak{x}.\mathsf{empty},\\
&\mathsf{add} = \mathfrak{x}.\mathsf{add}\\
&\mathsf{in} = \lambda x : \mathsf{Nat} . \lambda s : X \;.\; (\mathsf{ifzero}\, (\mathfrak{x}.\mathsf{count}\, x \; s))(\mathsf{false})(\mathsf{true}),\\
&\mathsf{remove} = \lambda x : \mathsf{Nat} . \lambda s : X \;.\; (\mathfrak{x}.\mathsf{remove}\, (\mathfrak{x}.\mathsf{count} \;\; s) \; x \; s) \;)
\end{aligned}
$$

We need to show the derivability of

$$\forall u : Sig_{\mathsf{BAG}_{\mathsf{Nat}}} \;.\; \Theta_{\mathsf{BAG}_{\mathsf{Nat}}}(u) \;\Rightarrow\; \Theta_{\mathsf{SET}_{\mathsf{Nat}}}(Fu)$$

That is, we must for arbitrary $u : Sig_{\mathsf{BAG}_{\mathsf{Nat}}}$ derive

$$\exists B . \exists \mathfrak{b} : \mathfrak{T}_{\mathsf{SET}_{\mathsf{Nat}}}[B] \;.\; (\mathsf{pack} B \mathfrak{b}) \; \mathsf{ObsEq}^{\{\mathsf{Bool}\}}_{\mathfrak{T}_{\mathsf{SET}_{\mathsf{Nat}}}} \; (Fu) \;\;\wedge\;\; Ax_{\mathsf{SET}_{\mathsf{Nat}}}[B, \mathfrak{b}]$$

assuming $\exists A.\exists\mathfrak{a}\!:\!\mathfrak{T}_{\mathsf{BAG}_{\mathsf{Nat}}}[A]\ .\ (\mathsf{pack}A\mathfrak{a})\ \mathsf{ObsEq}^{\{\mathsf{Nat}\}}_{\mathfrak{T}_{\mathsf{BAG}_{\mathsf{Nat}}}}\ u\ \wedge\ Ax_{\mathsf{BAG}_{\mathsf{Nat}}}[A,\mathfrak{a}]$. By Theorem 4.19, observational equivalence is equality, so we can replace u by $(\mathsf{pack}A\mathfrak{a})$, where A and $\mathfrak{a}$ are projected out from the assumption. Then let $(\mathsf{pack}A\mathfrak{a}')$ denote $F(\mathsf{pack}A\mathfrak{a})$.

In order to follow the universal proof strategy, the first thing we must do is axiomatise a suitable partial congruence. The congruence must relate bags that represent the same set. Thus, we define

$$\begin{aligned}\sim\ \stackrel{def}{=}\ (a\!:\!A, a'\!:\!A)\ .\ (\forall x\!:\!\mathsf{Nat}\ .\ \ & (\mathfrak{a}.\mathsf{count}\,x\ a) \geq 0\ \wedge\ (\mathfrak{a}.\mathsf{count}\,x\ a') \geq 0\\ & \wedge\ (\mathfrak{a}.\mathsf{count}\,x\ a) > 0\ \Leftrightarrow\ (\mathfrak{a}.\mathsf{count}\,x\ a') > 0)\end{aligned}$$

Here $\sim$ is total, so we will only need QUOT. To use QUOT, we must first check that $\mathfrak{a}'\ \mathfrak{T}_{\mathsf{SET}_{\mathsf{Nat}}}[\sim]\ \mathfrak{a}'$ and $equiv(\sim)$. Since $\sim$ is total, we easily get $equiv(\sim)$. It is also straightforward to verify $\mathfrak{a}'\ \mathfrak{T}_{\mathsf{SET}_{\mathsf{Nat}}}[\sim]\ \mathfrak{a}'$, for example to check that $\mathfrak{a}'.\mathsf{remove}\ (\mathsf{eq}_{\mathsf{Nat}} \to \sim\ \to\ \sim)\ \mathfrak{a}'.\mathsf{remove}$, we assume $x =_{\mathsf{Nat}} y$ and $a \sim a'$, and show $(\mathfrak{a}.\mathsf{remove}\,(\mathfrak{a}.\mathsf{count}\,x\ a)\ x\ a)\ \sim\ (\mathfrak{a}.\mathsf{remove}\,(\mathfrak{a}.\mathsf{count}\,y\ a')\ y\ a')$. This follows from the lemma $(\mathfrak{a}.\mathsf{count}\,x\ (\mathfrak{a}.\mathsf{remove}\,(\mathfrak{a}.\mathsf{count}\,x\ a)\ x\ a)) = 0$ derivable from $Ax_{\mathsf{BAG}_{\mathsf{Nat}}}$. Note that we are assuming induction for Nat, *cf.* Sect. 3.3.5.

Thus we can use QUOT to get Q and $\mathfrak{q}\!:\!\mathfrak{T}_{\mathsf{SET}_{\mathsf{Nat}}}[Q]$ and $\mathsf{epi}\!:\!A \to Q$ such that

$(q1)\quad \forall a, a'\!:\!A\ .\ a \sim a'\ \Leftrightarrow\ (\mathsf{epi}\,a) =_Q (\mathsf{epi}\,a')$
$(q2)\quad \forall q\!:\!Q.\exists a\!:\!A\ .\ q =_Q (\mathsf{epi}\,a)$
$(q3)\quad \mathfrak{a}'\ (\mathfrak{T}_{\mathsf{SET}_{\mathsf{Nat}}}[(a\!:\!A, q\!:\!Q).((\mathsf{epi}\,a) =_Q q)])\ \mathfrak{q}$

We exhibit Q for B, and $\mathfrak{q}$ for $\mathfrak{b}$. It remains to derive

1. $(\mathsf{pack}Q\mathfrak{q})\ \mathsf{ObsEq}^{\{\mathsf{Bool}\}}_{\mathfrak{T}_{\mathsf{SET}_{\mathsf{Nat}}}}\ (\mathsf{pack}A\mathfrak{a}')$ and
2. $Ax_{\mathsf{SET}_{\mathsf{Nat}}}[Q,\mathfrak{q}]$.

The derivability of (1) follows from $(q3)$ using Theorem 4.17. For (2) we must show the derivability of $\phi[Q,\mathfrak{q}]$ for every conjunct ϕ in $Ax_{\mathsf{SET}_{\mathsf{Nat}}}$. We give the derivations for the first three formulae.

$\phi = \forall x\!:\!\mathsf{Nat}, s\!:\!Q\ .\ (\mathfrak{q}.\mathsf{add}\,x\ (\mathfrak{q}.\mathsf{add}\,x\ s)) = (\mathfrak{q}.\mathsf{add}\,x\ s)$: Let $x\!:\!\mathsf{Nat}$ and $s\!:\!Q$ be arbitrary. By $(q2)$ we get a_s such that $(\mathsf{epi}\,a_s) = s$. We have by definition of $\sim$,

$$(\mathfrak{a}'.\mathsf{add}\,x\ (\mathfrak{a}'.\mathsf{add}\,x\ a_s)) \sim (\mathfrak{a}'.\mathsf{add}\,x\ a_s)$$

By (q1) we get $(\mathsf{epi}\ (\mathfrak{a}'.\mathsf{add}\,x\ (\mathfrak{a}'.\mathsf{add}\,x\ a_s))) =_Q (\mathsf{epi}\ (\mathfrak{a}'.\mathsf{add}\,x\ a_s))$, and $(q3)$ gives

$$x =_{\mathsf{Nat}} y\ \wedge\ (\mathsf{epi}\,a) =_Q q\ \ \Rightarrow\ \ (\mathsf{epi}\ (\mathfrak{a}'.\mathsf{add}\,x\ a)) =_Q (\mathfrak{q}.\mathsf{add}\,y\ q)$$

This therefore gives $(\mathfrak{q}.\mathsf{add}\,x\ (\mathfrak{q}.\mathsf{add}\,x\ s)) =_Q (\mathfrak{q}.\mathsf{add}\,x\ s)$ as desired.

$\phi = \forall x, y{:}\mathsf{Nat}, s{:}Q\ .\ (\mathfrak{q}.\mathsf{add}\, x\ (\mathfrak{q}.\mathsf{add}\, y\ s)) = (\mathfrak{q}.\mathsf{add}\, y\ (\mathfrak{q}.\mathsf{add}\, x\ s))$: Let $x{:}\mathsf{Nat}$ and $s{:}Q$ be arbitrary. By $(q2)$ we get a_s such that $(\mathsf{epi}\, a_s) = s$. We have from $Ax_{\mathsf{BAG}_{\mathsf{Nat}}}$, the definition of F, and the reflexivity of $\sim$,

$$(\mathfrak{a}'.\mathsf{add}\, x\ (\mathfrak{a}'.\mathsf{add}\, y\ a_s)) \sim (\mathfrak{a}'.\mathsf{add}\, y\ (\mathfrak{a}'.\mathsf{add}\, x\ a_s))$$

The desired result follows analogously to the case above.

$\phi = \forall x{:}\mathsf{Nat}\ .\ (\mathfrak{q}.\mathsf{in}\, x\ \mathfrak{q}.\mathsf{empty}) = \mathsf{false}$: We have

$$(\mathfrak{a}'.\mathsf{in}\, x\ \mathfrak{a}'.\mathsf{empty}) = \mathsf{ifzero}(\mathfrak{a}.\mathsf{count}\, x\ \mathfrak{a}.\mathsf{empty})(\mathsf{false})(\mathsf{true}) = \mathsf{false}$$

By $(q3)$ we get first $(\mathsf{epi}\, \mathfrak{a}'.\mathsf{empty}) =_Q \mathfrak{q}.\mathsf{empty}$, and thereafter $(\mathfrak{a}'.\mathsf{in}\, x\ \mathfrak{a}'.\mathsf{empty}) = (\mathfrak{q}.\mathsf{in}\, x\ \mathfrak{q}.\mathsf{empty})$, which gives the result.

The derivation of the other formulae follow similar lines. ○

—

The next example uses both SUB and QUOT.

Example 4.27 Again we seek to implement sets as specified by $\mathsf{SET}_{\mathsf{Nat}}$ from Example 4.11. However, this time we want to implement sets using lists. Lists exists as inductive types in System F, but there is of course room for other implementations of lists. We would probably want these implementations to satisfy the following specification.

$$\mathsf{LIST}_{\mathsf{Nat}} \stackrel{def}{=} \langle\langle Sig_{\mathsf{LIST}_{\mathsf{Nat}}}, \Theta_{\mathsf{LIST}_{\mathsf{Nat}}}\rangle, \{\mathsf{Bool}, \mathsf{Nat}\}\rangle, \text{ where}$$

$$Sig_{\mathsf{LIST}_{\mathsf{Nat}}} = \exists X.\mathfrak{T}_{\mathsf{LIST}_{\mathsf{Nat}}}[X]$$

$$\begin{aligned}\text{for } \mathfrak{T}_{\mathsf{LIST}_{\mathsf{Nat}}}[X] = (&\mathsf{nil}{:}X,\\ &\mathsf{cons}{:}\mathsf{Nat}\to X\to X,\\ &\mathsf{car}{:}X\to\mathsf{Nat},\\ &\mathsf{cdr}{:}X\to X,\\ &\mathsf{in}{:}\mathsf{Nat}\to X\to\mathsf{Bool})\end{aligned}$$

$$\Theta_{\mathsf{LIST}_{\mathsf{Nat}}}(u) = \exists X.\exists\mathfrak{x}{:}\mathfrak{T}_{\mathsf{LIST}_{\mathsf{Nat}}}[X]\ .\ u\ \mathsf{ObsEq}^{\{\mathsf{Bool},\mathsf{Nat}\}}\ (\mathsf{pack}X\mathfrak{x})\ \wedge\ Ax_{\mathsf{LIST}_{\mathsf{Nat}}}$$

$$\begin{aligned}\text{for } Ax_{\mathsf{LIST}_{\mathsf{Nat}}} \stackrel{def}{=}\ & \forall l{:}X\ .\ l \neq \mathfrak{x}.\mathsf{nil}\ \Rightarrow\ \mathfrak{x}.\mathsf{cons}(\mathfrak{x}.\mathsf{car}\, l)(\mathfrak{x}.\mathsf{cdr}\, l) = l\quad\wedge\\ & \forall x{:}\mathsf{Nat}\ .\ (\mathfrak{x}.\mathsf{in}\, x\ \mathfrak{x}.\mathsf{nil}) = \mathsf{false}\quad\wedge\\ & \forall x, y{:}\mathsf{Nat}.\forall l{:}X\ .\\ &\qquad (\mathfrak{x}.\mathsf{in}\, x\ (\mathfrak{x}.\mathsf{cons}\, y\ l)) = \mathsf{true}\ \Leftrightarrow\ (x = y \vee (\mathfrak{x}.\mathsf{in}\, x\ l) = \mathsf{true})\\ & \forall R{\subset}X{\times}X.\forall x, y{:}X\ .\ \mathfrak{x}.\mathsf{nil}\ R\ \mathfrak{x}.\mathsf{nil}\ \wedge\\ &\quad (\forall x{:}\mathsf{Nat}, l{:}X\ .\ l\ R\ l\ \Rightarrow\ (\mathfrak{x}.\mathsf{cons}\, x\ l)\ R\ (\mathfrak{x}.\mathsf{cons}\, x\ l))\\ &\qquad\qquad \Rightarrow\ \forall l{:}X\ .\ l\ R\ l\end{aligned}$$

We now implement sets using lists via an appropriate constructor. However, this time, rather than giving a specific constructor, we consider a range of constructors satisfying a constructor specification. The constructor specification below admits constructors that implement sets by lists where equal elements occur consecutively. (One might at lower levels of implementation wish to keep a record of insertions.) Notice that remove is optimised by using this fact. The constructor specification leaves room to choose in what order the resulting constructor should arrange the blocks of consecutive elements.

$$\Pi S\!:\!\mathsf{LIST_{Nat}}.\mathsf{SET_{Nat}} \stackrel{def}{=} \left\langle Sig_{\Pi S:\mathsf{LIST_{Nat}}.\mathsf{SET_{Nat}}}, \Theta_{\Pi S:\mathsf{LIST_{Nat}}.\mathsf{SET_{Nat}}}\right\rangle, \text{ where}$$

$$Sig_{\Pi S:\mathsf{LIST_{Nat}}.\mathsf{SET_{Nat}}} \stackrel{def}{=} Sig_{\mathsf{LIST_{Nat}}} \to Sig_{\mathsf{SET_{Nat}}},$$

$$\begin{aligned}\Theta_{\Pi S:\mathsf{LIST_{Nat}}.\mathsf{SET_{Nat}}}(v,u) \stackrel{def}{=}\ & \exists X.\exists\mathfrak{x}\!:\!\mathfrak{T}_{\mathsf{LIST_{Nat}}}[X].\exists\mathfrak{x}'\!:\!\mathfrak{T}_{\mathsf{SET_{Nat}}}[X]\times\mathsf{lumped}\!:\!X\to\mathsf{Bool}\ .\\ & v\ \mathsf{ObsEq}^{Obs_{\mathsf{LIST_{Nat}}}}\ (\mathsf{pack}X\mathfrak{x})\ \ \wedge\ \ u\ \mathsf{ObsEq}^{Obs_{\mathsf{SET_{Nat}}}}\ (\mathsf{pack}X\mathfrak{x}'|_{\mathfrak{T}_{\mathsf{SET_{Nat}}}})\\ & \wedge\ Ax_{\Pi S:\mathsf{LIST_{Nat}}.\mathsf{SET_{Nat}}}[X,\mathfrak{x},\mathfrak{x}']\end{aligned}$$

$$\begin{aligned}
&Ax_{\Pi S:\mathsf{LIST_{Nat}}.\mathsf{SET_{Nat}}}[X,\mathfrak{x},\mathfrak{x}'] \stackrel{def}{=}\\
&\quad \mathfrak{x}'.\mathsf{empty} = \mathfrak{x}.\mathsf{nil}\ \wedge\\
&\quad \mathfrak{x}'.\mathsf{in} = \mathfrak{x}.\mathsf{in}\ \wedge\\
&\quad (\mathfrak{x}'.\mathsf{lumped}\,\mathfrak{x}.\mathsf{nil}) = \mathsf{true}\ \wedge\\
&\quad \forall x\!:\!\mathsf{Nat}\ .\ (\mathfrak{x}'.\mathsf{lumped}(\mathfrak{x}.\mathsf{cons}\,x\ \mathfrak{x}.\mathsf{nil})) = \mathsf{true}\ \wedge\\
&\quad \forall x,y\!:\!\mathsf{Nat}.\forall l\!:\!X\ .\ (\mathfrak{x}'.\mathsf{lumped}(\mathfrak{x}.\mathsf{cons}\,x\ (\mathfrak{x}.\mathsf{cons}\,y\ l))) =\\
&\qquad\qquad \mathsf{if}\ x =_{\mathsf{Nat}} y\ \ \mathsf{then}\ (\mathfrak{x}'.\mathsf{lumped}\,(\mathfrak{x}.\mathsf{cons}\,y\ l))\\
&\qquad\qquad\qquad \mathsf{else}\ \mathsf{not}(\mathfrak{x}.\mathsf{in}\,x\ l)\ \wedge\\
&\quad \forall x\!:\!\mathsf{Nat}.\forall l\!:\!X\ .\ (\mathfrak{x}'.\mathsf{lumped}\,l) = \mathsf{true}\ \Rightarrow\ (\mathfrak{x}'.\mathsf{lumped}(\mathfrak{x}'.\mathsf{add}\,x\ l)) = \mathsf{true}\ \wedge\\
&\quad \forall x\!:\!\mathsf{Nat}.\forall l\!:\!X\ .\ (\mathfrak{x}'.\mathsf{in}\,x\ (\mathfrak{x}'.\mathsf{add}\,x\ l)) = \mathsf{true}\ \wedge\\
&\quad \forall x\!:\!\mathsf{Nat}\ .\ (\mathfrak{x}'.\mathsf{remove}\,x\ \mathfrak{x}.\mathsf{nil}) = \mathfrak{x}.\mathsf{nil}\ \wedge\\
&\quad \forall x,y\!:\!\mathsf{Nat}\ .\ (\mathfrak{x}'.\mathsf{remove}\,x\ (\mathfrak{x}.\mathsf{cons}\,y\ \mathfrak{x}.\mathsf{nil})) =\\
&\qquad\qquad \mathsf{if}\ x =_{\mathsf{Nat}} y\ \mathsf{then}\ \mathfrak{x}.\mathsf{nil}\ \mathsf{else}\ (\mathfrak{x}.\mathsf{cons}\,y\ \mathfrak{x}.\mathsf{nil})\ \wedge\\
&\quad \forall x,y,z\!:\!\mathsf{Nat}.\forall l\!:\!X\ .\ (\mathfrak{x}'.\mathsf{remove}\,x\ (\mathfrak{x}.\mathsf{cons}\,y\ (\mathfrak{x}.\mathsf{cons}\,z\ l))) =\\
&\qquad\qquad \mathsf{if}\ x =_{\mathsf{Nat}} y\ \mathsf{then}\\
&\qquad\qquad\qquad \mathsf{if}\ x =_{\mathsf{Nat}} z\ \mathsf{then}\ (\mathfrak{x}'.\mathsf{remove}\,x\ l)\ \mathsf{else}\ (\mathfrak{x}.\mathsf{cons}\,z\ l)\\
&\qquad\qquad \mathsf{else}\ (\mathfrak{x}.\mathsf{cons}\,y\ (\mathfrak{x}'.\mathsf{remove}\,x\ (\mathfrak{x}.\mathsf{cons}\,z\ l)))
\end{aligned}$$

We must now show the derivability of

$$\forall u\!:\!Sig_{\mathsf{LIST_{Nat}}}\ .\ \Theta_{\mathsf{LIST_{Nat}}}(u)\ \Rightarrow\ \Theta_{\mathsf{SET_{Nat}}}(Fu)$$

for any realisation F of $\Pi S\!:\!\mathsf{LIST_{Nat}}.\mathsf{SET_{Nat}}$, *i.e.*, we must for $u\!:\!Sig_{\mathsf{LIST_{Nat}}}$ derive

$$\exists B.\exists\mathfrak{b}\!:\!\mathfrak{T}_{\mathsf{SET_{Nat}}}[B]\ .\ (\mathsf{pack}B\mathfrak{b})\ \mathsf{ObsEq}^{\{\mathsf{Bool}\}}_{\mathfrak{T}_{\mathsf{SET_{Nat}}}}\ (Fu)\ \ \wedge\ \ Ax_{\mathsf{SET_{Nat}}}[B,\mathfrak{b}]$$

assuming $\exists A.\exists\mathfrak{a}\!:\!\mathfrak{T}_{\mathsf{LIST}_{\mathsf{Nat}}}[A]\ .\ (\mathsf{pack}A\mathfrak{a})\ \mathsf{ObsEq}^{\{\mathsf{Nat}\}}_{\mathfrak{T}_{\mathsf{LIST}_{\mathsf{Nat}}}}\ u \wedge Ax_{\mathsf{LIST}_{\mathsf{Nat}}}[A,\mathfrak{a}]$. By Theorem 4.19, we replace u by $(\mathsf{pack}A\mathfrak{a})$ where A and $\mathfrak{a}$ are projected out from the assumption. Let $(\mathsf{pack}A\mathfrak{a}')$ denote $F(\mathsf{pack}A\mathfrak{a})$.

Again, in order to follow the universal proof strategy, we must axiomatise a suitable partial congruence. The congruence must relate lists that represent the same set. However, certain lists do not represent sets; indeed lists that do not store equal elements consecutively must not represent sets, because the remove operation will be erroneous on such lists, *i.e.,* the axioms for remove will not hold, and the proposed refinement will not go through. The domain of the congruence must therefore be restricted to the particular kind of lists we want. These lists are the ones built up using the abstract generators empty and add provided by the constructor F. These lists are exactly the ones users will be able to generate according to `Abs-Bar`. Thus, we define

$$\begin{aligned}\sim\ \stackrel{def}{=}\ (a\!:\!A, a'\!:\!A)\ .\ (\mathsf{lumped}(a) = \mathsf{true}\ \wedge\ \mathsf{lumped}(a') = \mathsf{true})\\ \wedge\ \forall x\!:\!\mathsf{Nat}\ .\ (\mathfrak{a}.\mathsf{in}\,x\ a) = \mathsf{true}\ \Leftrightarrow\ (\mathfrak{a}.\mathsf{in}\,x\ a') = \mathsf{true})\end{aligned}$$

Here $\sim$ is partial. We must therefore use SUB prior to using QUOT. In order to use SUB, we must check that $\mathfrak{a}'\ \mathfrak{T}_{\mathsf{SET}_{\mathsf{Nat}}}[\sim]\ \mathfrak{a}'$. For example, to check that $\mathfrak{a}'.\mathsf{remove}\ (\mathsf{eq}_{\mathsf{Nat}} \to\ \sim\ \to\ \sim)\ \mathfrak{a}'.\mathsf{remove}$, we assume $x =_{\mathsf{Nat}} y$ and $a \sim a'$, and show $(\mathfrak{a}'.\mathsf{remove}\,x\ a) \sim (\mathfrak{a}'.\mathsf{remove}\,y\ a')$. The assumption $a \sim a'$, implies that remove will work as intended on a and a'. We omit further details.

With $\mathfrak{a}'\ \mathfrak{T}_{\mathsf{SET}_{\mathsf{Nat}}}[\sim]\ \mathfrak{a}'$ we use SUB to get S_A, $\mathfrak{s}_{\mathfrak{a}'}$, $\sim'\subset S_A{\times}S_A$, and $\mathsf{mono}\!:\!S_A \to A$ such that we can derive

(s1) $\forall s\!:\!S_A\ .\ s\ \sim'\ s$

(s2) $\forall s.s'\!:\!S_A\ .\ s \sim' s'\ \Leftrightarrow\ (\mathsf{mono}\,s) \sim (\mathsf{mono}\,s')$

(s3) $\mathfrak{a}'\ (\mathfrak{T}_{\mathsf{SET}_{\mathsf{Nat}}}[(a\!:\!A, s\!:\!S_A).(a =_A (\mathsf{mono}\,s))])\ \mathfrak{s}_{\mathfrak{a}'}$

By $(s2)$ we get $\mathfrak{s}_{\mathfrak{a}'}\ \mathfrak{T}_{\mathsf{SET}_{\mathsf{Nat}}}[\sim']\ \mathfrak{s}_{\mathfrak{a}'}$. We also get $equiv(\sim')$ by $(s1)$. We now use QUOT to get Q and $\mathfrak{q}\!:\!\mathfrak{T}_{\mathsf{SET}_{\mathsf{Nat}}}[Q]$ and $\mathsf{epi}\!:\!S_A \to Q$ such that

(q1) $\forall s, s'\!:\!S_A\ .\ s \sim' s'\ \Leftrightarrow\ (\mathsf{epi}\,s) =_Q (\mathsf{epi}\,s')$

(q2) $\forall q\!:\!Q.\exists s\!:\!S_A\ .\ q =_Q (\mathsf{epi}\,s)$

(q3) $\mathfrak{s}_{\mathfrak{a}'}\ (\mathfrak{T}_{\mathsf{SET}_{\mathsf{Nat}}}[(s\!:\!S_A, q\!:\!Q).((\mathsf{epi}\,s) =_Q q)])\ \mathfrak{q}$

We thus exhibit Q for B, and $\mathfrak{q}$ for $\mathfrak{b}$. It remains to derive

1. $(\mathsf{pack}Q\mathfrak{q})\ \mathsf{ObsEq}^{\{\mathsf{Bool}\}}_{\mathfrak{T}_{\mathsf{SET}_{\mathsf{Nat}}}}\ (\mathsf{pack}A\mathfrak{a}')$ and

2. $Ax_{\mathsf{SET}_{\mathsf{Nat}}}[Q,\mathfrak{q}]$.

The derivability of (1) follows from $(s3)$ and $(q3)$ using Theorem 4.17. One verifies (2) using $Ax_{\mathsf{LIST}_{\mathsf{Nat}}}$, $Ax_{\Pi S:\mathsf{LIST}_{\mathsf{Nat}}.\mathsf{SET}_{\mathsf{Nat}}}$, and the definition of $\sim$. ∘

4.5 The Translation

We now define an obvious translation $\mathcal{T}$ mapping algebraic specification refinement to type theoretic specification refinement. Formally, this would be a functor from the category of algebraic specifications and refinement maps $\underset{\kappa}{\leadsto}$ to the category of type-theoretic specifications and refinement maps $\underset{F}{\leadsto}$, but for our purposes the details of this are not worth elaborating.

The starting point is the concept of algebraic specification refinement up to observational equivalence with constructors from Ch. 2:

$$(\textbf{abstract } SP \textbf{ wrt } \equiv^{Obs,In}) \underset{\kappa}{\leadsto} (\textbf{abstract } SP' \textbf{ wrt } \equiv^{Obs',In'})$$

We will here assume normal-form specifications SP and SP', *cf.* Sect. 2.6. To keep things simple, we will at any one refinement stage assume a single behavioural sort b in the signature, see also Sect. 4.2.9; methodologically this means focusing on one data type at a time, and on one thread in a development. Thus we can stick to existential types with one existentially quantified variable. It is straightforward to generalise to multiple existentially quantified variables (Mitchell, 1991). We assume that all other sorts in the signature are observable. This means that we can assume $\texttt{FADT}_{Obs}$ (*p.* 87) for the corresponding type-theoretic specification.

We assume built-in observable and input sorts in algebraic specification. We assume that observable types share names with observable sorts, hence we use Obs to denote both observable sorts and observable types. We assume that the corresponding observable types are inductive, *e.g.,* Bool and Nat.

We need not specify input types, since the type-theoretic setting automatically deals with the corresponding problem of excluding the behavioural sort from input sorts, recall Sect. 4.2.9. This is sufficient for the correspondence we will be showing in this section. We now note that together with $\texttt{FADT}_{Obs}$, this essentially entails $Obs = In$ on the algebraic specification side, which arguably is the sensible choice (Sannella and Tarlecki, 1987). In the following we therefore set $Obs = In$.

Finally, note that we show a correspondence between specification refinements, not between data types.

4.5.1 Translating ADT Specifications

We will map both ADT specifications and constructor specifications to type-theoretic formalisms. The latter is necessary in order to be precise about how to translate the refinement relation. We begin with ADT specifications.

—

Definition 4.28 (Translation) *Let $\Sigma = \langle S, \Omega\rangle$ and $\Sigma^e = \langle S^e, \Omega^e\rangle$ be signatures such that there is an inclusion signature morphism $\iota : \Sigma \hookrightarrow \Sigma^e$, and let $S^h = S^e \setminus S$ and $\Omega^h = \Omega^e \setminus \Omega$. Let $SP =$ **hide sorts** S^h **operations** Ω^h **in** $\langle \Sigma^e, Ax\rangle$. Assume one behavioural sort b in Σ. Define the translation $\mathcal{T}$ by*

$$\mathcal{T}(\textbf{abstract } SP \textbf{ wrt } \equiv^{Obs,Obs}) \stackrel{def}{=} \langle\langle Sig_{SP}, \Theta_{SP}\rangle, \mathfrak{T}^e_{SP}, Obs\rangle$$

where $Sig_{SP} = \exists X.\mathfrak{T}_{SP}[X]$,
where $\mathfrak{T}_{SP}[X] = (f_1 : s_{11} \to \cdots \to s_{1n_1} \to s_1,\ \ldots\ , f_k : s_{k1} \to \cdots \to s_{kn_k} \to s_k)[X/b]$,
for $f_i : s_{i1} \times \cdots \times s_{in_i} \to s_i \in \Omega$,
and $\mathfrak{T}^e_{SP}[X] = (f_1 : s_{11} \to \cdots \to s_{1n_1} \to s_1, \ldots , f_m : s_{m1} \to \cdots \to s_{mn_m} \to s_m)[X/b]$,
for $f_i : s_{i1} \times \cdots \times s_{in_i} \to s_i \in \Omega^e$,
and where $\Theta_{SP}(u) = \exists X.\exists \mathfrak{x} : \mathfrak{T}^e_{SP}[X]\ .\ u = (\mathsf{pack} X\, \mathfrak{x}|_{\mathfrak{T}_{SP}}) \ \wedge\ Ax_{SP}[X, \mathfrak{x}]$.

Here, $Ax_{SP}[X, \mathfrak{x}]$ indicates Ax, where X substitutes b, and every operator symbol in Ax belonging to Ω^e is prefixed with $\mathfrak{x}$.

Thus, the existentially quantified variable corresponds to the behavioural sort. The translation specialises to specifications without hidden symbols in the obvious way according to the convention mentioned after Def. 4.12 (*p.* 84).

Example 4.29 A translation not involving hidden symbols, is

$$\mathcal{T}(\textbf{abstract } \mathsf{STACK} \textbf{ wrt } \{\mathsf{Nat}\}, \{\mathsf{Nat}\}) = \big\langle\big\langle Sig_{\mathsf{STACK}_{\mathsf{Nat}}}, \Theta_{\mathsf{STACK}_{\mathsf{Nat}}}\big\rangle, \{\mathsf{Nat}\}\big\rangle$$

which is $\mathsf{STACK}_{\mathsf{Nat}}$ from Example 4.7. Here, $\Sigma^e = \Sigma$ and $\mathfrak{T}^e_{SP}[X] = \mathfrak{T}_{SP}[X]$, so according to convention, $\mathfrak{T}^e_{SP}[X]$ is dropped. ○

The translation $\mathcal{T}$ can be seen in two ways. First it maps observational specifications **abstract** SP **wrt** $\equiv^{Obs,Obs}$ where SP is in normal form, to type-theoretic specifications. But via the normalisation result for structured specifications, $\mathcal{T}$ can also be seen to map any structured specification to a type-theoretic formalism, hence translating something the specifier wrote, to a type-theoretic formalism for use in a theorem prover.

4.5.2 Translating Constructor Specifications

We can extend the translation $\mathcal{T}$ to constructor specifications as follows. First, on the algebraic specification side it makes sense to consider *normal-form constructor specifications* $\Pi S : SP'.SP[S]$, where SP' and SP are observational specifications based on normal-form specifications. This is justified by the normalisation result for ADT specifications, *cf.* Sect. 2.6; any constructor specification built from basic ADT specifications using the canonical specification-building operators **sum**, **derive**, and **translate**, within **abstract**, can be normalised to the form above.

Definition 4.30 (Translation of Constructor Specification) *Let*

$$SP = \textbf{hide sorts } S^h \textbf{ operations } \Omega^h \textbf{ in } \langle \Sigma^e, Ax \rangle,$$
$$SP' = \textbf{hide sorts } S'^h \textbf{ operations } \Omega'^h \textbf{ in } \left\langle \Sigma'^e, Ax' \right\rangle,$$

where SP may be data-type parameterised by $S : SP'$, but nevertheless such that Σ_{SP} does not rely on S. Let furthermore

$$\langle\langle Sig_{SP}, \Theta_{SP}\rangle, \mathfrak{T}^e_{SP}, Obs\rangle = \mathcal{T}(\textbf{abstract } SP \textbf{ wrt } \equiv^{Obs,Obs}),$$
$$\langle\langle Sig_{SP'}, \Theta_{SP'}\rangle, \mathfrak{T}^e_{SP'}, Obs'\rangle = \mathcal{T}(\textbf{abstract } SP' \textbf{ wrt } \equiv^{Obs',Obs'}).$$

Then

$$\mathcal{T}(\Pi S{:}(\textbf{abstract } SP' \textbf{ wrt } \equiv^{Obs',Obs'}) \;.\; (\textbf{abstract } SP \textbf{ wrt } \equiv^{Obs,Obs})) \stackrel{def}{=} \langle Sig_{\Pi S:SP'.SP}, \Theta_{\Pi S:SP'.SP}\rangle$$

where

$$Sig_{\Pi S:SP'.SP} \stackrel{def}{=} Sig_{SP'} \rightarrow Sig_{SP},$$
$$\begin{aligned}\Theta_{\Pi S:SP'.SP}(v,u) \stackrel{def}{=} \exists Y.\exists \mathfrak{y}{:}\mathfrak{T}^e_{SP'}[Y].\exists X.\exists \mathfrak{x}{:}\mathfrak{T}^e_{SP}[X] \;.\; & v \;\mathsf{ObsEq}^{Obs_{SP'}}\; (\mathsf{pack} Y\, \mathfrak{y}|_{\mathfrak{T}_{SP'}}) \;\wedge \\ & u \;\mathsf{ObsEq}^{Obs_{SP}}\; (\mathsf{pack} X\, \mathfrak{x}|_{\mathfrak{T}_{SP}}) \;\wedge \\ & Ax_{\Pi S:SP'.SP}[Y, \mathfrak{y}, X, \mathfrak{x}]\end{aligned}$$

where $Ax_{\Pi S:SP'.SP}[Y, \mathfrak{y}, X, \mathfrak{x}] = Ax_{SP}[Y, \mathfrak{y}, X, \mathfrak{x}]$, such that Y and $\mathfrak{y}$ replace the free data type variables in Ax_{SP}.

Recall from Def. 4.15, that the realisation predicate $\Theta_{\Pi S:SP'.SP}(v,u)$ is used by saying that F is a realisation of $\Pi S{:}SP'.SP$ if one can derive

$$\forall v{:}Sig_{SP'} \;.\; \Theta_{SP'}(v) \;\Rightarrow\; \Theta_{\Pi S:SP'.SP}(v, Fv)$$

The translation specialises to cases with no hidden symbols in the obvious ways.

4.6 A Correspondence at First Order

We now seek to establish a formal connection between the concept of algebraic specification refinement and its type-theoretic counterpart as defined in Def. 4.14, that is, we wish to show the following meta-result:

$$\begin{aligned}&(\textbf{abstract } SP \textbf{ wrt } \equiv^{Obs,Obs}) \underset{\kappa}{\leadsto} (\textbf{abstract } SP' \textbf{ wrt } \equiv^{Obs',Obs'}) \\ &\quad\Leftrightarrow\; \mathcal{T}(\textbf{abstract } SP \textbf{ wrt } \equiv^{Obs,Obs}) \underset{F_\kappa}{\leadsto} \mathcal{T}(\textbf{abstract } SP' \textbf{ wrt } \equiv^{Obs',Obs'})\end{aligned}$$

where κ and F_κ are constructors that correspond in a sense given below.

That u is a realisation of a type theory specification $\langle\langle Sig_{SP}, \Theta_{SP}\rangle, \mathfrak{T}^{e}_{SP} Obs\rangle$ can in general only be derived by exhibiting an observationally equivalent package u' that satisfies Ax_{SP}. For any particular closed term $g : Sig_{SP}$, one can attempt to construct such a g' perhaps ingeniously, using details of g. But to show that a specification is a refinement of another specification we are asked to consider a term $(\mathsf{pack} A\mathfrak{a})$ where we do not know details of A or $\mathfrak{a}$. To show the desired correspondence, we therefore need a universal method for exhibiting suitable observationally equivalent packages. Of course, it also defies the point of behavioural abstraction having to construct a specific literal implementation to justify a behavioural one.

To show the desired correspondence, we shall therefore utilise the existence of the universal proof strategy sketched in Sect. 2.7 that was imported into type theory in Sect. 4.4. In algebraic specification one proves observational refinements by first considering quotients w.r.t. a possibly partial congruence $\approx^{Obs,In}$ induced by Obs and In (Bidoit et al., 1995), and then using an axiomatisation of this quotienting congruence to prove relativised versions of the axioms of the specification to be refined. In the case that this congruence is partial, clauses restricting to the domain of the congruence must also be incorporated (Bidoit et al., 1997; Bidoit and Hennicker, 1996). The quotients are of the form $Dom(\approx^{Obs,In}_{A})/\approx^{Obs,In}_{A}$, where $Dom(\approx^{Obs,In}_{A})_s \stackrel{def}{=} \{a \in A_s \mid a \approx^{Obs,In}_{A} a\}$, *cf.* Sect. 2.7. As mentioned in Sect. 4.4, this proof method is not available in the type theory and logic of (Plotkin and Abadi, 1993). It can be made available by augmenting the logic with axiom schemata postulating the existence of subobjects Def. 4.23, and quotients Def. 4.24, for dealing with partial congruences.

Algebraic specification uses classical logic, while the logic in (Plotkin and Abadi, 1993) is constructive. However, formulae may be interpreted classically in the parametric PER-model, and it is sound w.r.t. this model to assume the axiom of excluded middle (Plotkin and Abadi, 1993), see Sect. A.3 in Appendix A. For our comparison with algebraic specification, we shall do this.

We can now show our desired correspondence. We must assume that specifications are *behaviourally closed*, but this is an obvious requirement, *cf.* Sect. 2.6. We will show the correspondence in three stages. First we show the case for observational refinement on basic specifications, and with the identity constructor. Then we replace the basic specifications with normal-form specifications, and finally we extend with general stable constructors.

—

Theorem 4.31 (Correspondence (Basic)) *Consider basic algebraic specifications $SP = \langle \Sigma, Ax \rangle$ and $SP' = \langle \Sigma', Ax' \rangle$, for $\Sigma = \langle S, \Omega \rangle$. Assume one behavioural sort b in Σ, and assume $Obs = In = S \setminus b$. Assume that $\equiv^{Obs,Obs}$ on $\Sigma\mathbf{Alg}$ is behaviourally closed w.r.t. the factorising observational congruence $\approx^{Obs,Obs}$. Then*

$$\begin{aligned}&(\mathbf{abstract}\ SP\ \mathbf{wrt}\ \equiv^{Obs,Obs}) \rightsquigarrow (\mathbf{abstract}\ SP'\ \mathbf{wrt}\ \equiv^{Obs',Obs'})\\ &\Leftrightarrow\ \mathcal{T}(\mathbf{abstract}\ SP\ \mathbf{wrt}\ \equiv^{Obs,Obs}) \rightsquigarrow \mathcal{T}(\mathbf{abstract}\ SP'\ \mathbf{wrt}\ \equiv^{Obs',Obs'})\end{aligned}$$

Proof: Note that our assumptions entail that we can assume $\mathtt{FADT}_{Obs}$ for the resulting type-theoretic specifications $\langle\langle Sig_{SP}, \Theta_{SP}\rangle, Obs\rangle$ and $\langle\langle Sig_{SP'}, \Theta_{SP'}\rangle, Obs'\rangle$. We are dealing with the identity constructor. This means $\Sigma = \Sigma'$ and hence $Obs = Obs'$, and also $Sig_{SP} = Sig_{SP'}$.

$\Rightarrow$: We must show the derivability of

$$\forall u{:}Sig_{SP'}\ .\ \Theta_{SP'}(u)\ \Rightarrow\ \Theta_{SP}(u)$$

From $(\mathbf{abstract}\ SP\ \mathbf{wrt}\ \equiv^{Obs,Obs}) \rightsquigarrow (\mathbf{abstract}\ SP'\ \mathbf{wrt}\ \equiv^{Obs,Obs})$, we obtain the proof-theoretical information outlined in the following. By behavioural closedness, we can use the sound and complete calculus $\vdash_{\Pi_{\rightsquigarrow}}$ for specification refinement, based on a calculus $\vdash_{\Pi_S}$ for structured specifications (Bidoit et al., 1997). Since the identity constructor is stable, the above is equivalent to proving $(\mathbf{abstract}\ SP\ \mathbf{wrt}\ \equiv^{Obs,Obs}) \rightsquigarrow SP'$, *cf.* Sect 2.5. By syntax directedness, we must have had $SP'/\!\approx_{Obs,In} \vdash_{\Pi_S} Ax$, where the semantics of $SP'/\!\approx_{Obs,In}$ is $\{Dom(\approx_A^{Obs,In})/\!\approx_A^{Obs,In} \mid A \in [\![SP']\!]\}$. Since specifications here are basic, this boils down to the predicate logic statement of

$$Ax', Ax(\sim)\ \vdash\ \mathcal{L}(Ax) \qquad (\dagger)$$

Here $\sim$ stands for a new symbol representing $\approx_{Obs,In}$ at the behavioural sort b, and $\mathcal{L}(\Phi) \stackrel{\mathit{def}}{=} \{\mathcal{L}(\phi) \mid \phi \in Ax\}$, for $\mathcal{L}(\phi) = (\ \wedge_{y \in FV_b(\phi)}\ y \sim y) \Rightarrow \phi^*$ where $FV_b(\phi)$ is the set of free variables of sort b in ϕ, and where inductively

(*a*) $(u =_b v)^* \stackrel{\mathit{def}}{=} u \sim v$,

(*b*) $(\neg\phi)^* \stackrel{\mathit{def}}{=} \neg(\phi^*)$ and $(\phi\ \wedge\ \psi)^* \stackrel{\mathit{def}}{=} \phi^*\ \wedge\ \psi^*$,

(*c*) $(\forall x{:}b.\phi)^* \stackrel{\mathit{def}}{=} \forall x{:}b.(x \sim x\ \Rightarrow\ \phi^*)$,

(*d*) $\phi^* \stackrel{\mathit{def}}{=} \phi$, otherwise.

and $Ax(\sim) \stackrel{\mathit{def}}{=} \forall x, y{:}b.(x \sim y\ \Leftrightarrow\ Beh_b(x,y))$, where $Beh_b(x,y)$ is an axiomatisation of $\approx_{Obs,In}$ at b (Bidoit and Hennicker, 1996). At $s \in Obs = In$, $\approx_{Obs,In}$ is just equality.

Using this we derive our goal as follows. Let $u\!:\!Sig_{SP'}$ be arbitrary. Let $\mathfrak{T}[X]$ denote $\mathfrak{T}_{SP'}[X](=\mathfrak{T}_{SP}[X])$. We must derive

$$\exists B.\exists \mathfrak{b}\!:\!\mathfrak{T}[B]\ .\ u = (\mathsf{pack}B\mathfrak{b})\ \wedge\ Ax[B,\mathfrak{b}]$$

assuming $\exists A.\exists \mathfrak{a} : \mathfrak{T}[A]\ .\ u = (\mathsf{pack}A\mathfrak{a})\ \wedge\ Ax'[A,\mathfrak{a}]$. Let $\mathfrak{a}$ and A denote the witnesses projected out from that assumption. It suffices to exhibit a B and $\mathfrak{b}$ such that $(\mathsf{pack}B\mathfrak{b}) = (\mathsf{pack}A\mathfrak{a})$ and $Ax[B,\mathfrak{b}]$.

Now, *Beh* is in general infinitary. There are heuristics for producing finite axiomatisations from *Beh* in some cases (Bidoit et al., 1997; Bidoit and Hennicker, 1996), and with our higher-order relational logic one gets finitary axiomatisations in general (Hofmann and Sannella, 1996). In any case one gets a finitary Beh^* equivalent to *Beh*. Thus we form $\sim$ type-theoretically by

$$\sim\ \stackrel{def}{=} (a\!:\!A, a'\!:\!A).(Beh^*_A(a,a'))$$

Since $\sim$ is an axiomatisation of a partial congruence, we have $\mathfrak{a}\ \mathfrak{T}[\sim]\ \mathfrak{a}$. We use SUB to get S_A, $\mathfrak{s}_\mathfrak{a}$ and $\sim' \subset S_A \times S_A$ and $\mathsf{mono}\!:\!S_A \to A$ such that we can derive

$(s1)$ $\forall s\!:\!S_A\ .\ s\ \sim'\ s$

$(s2)$ $\forall s.s'\!:\!S_A\ .\ s \sim' s'\ \Leftrightarrow\ (\mathsf{mono}\, s) \sim (\mathsf{mono}\, s')$

$(s3)$ $\mathfrak{a}\ (\mathfrak{T}[(a\!:\!A, s\!:\!S_A).(a =_A (\mathsf{mono}\, s))])\ \mathfrak{s}_\mathfrak{a}$

By $(s2)$ we get $\mathfrak{s}_\mathfrak{a}\ \mathfrak{T}[\sim']\ \mathfrak{s}_\mathfrak{a}$. We also get $equiv(\sim')$ by $(s1)$. We now use QUOT to get Q and $\mathfrak{q}\!:\!\mathfrak{T}[Q]$ and $\mathsf{epi}\!:\!S_A \to Q$ s.t.

$(q1)$ $\forall s, s'\!:\!S_A\ .\ s \sim' s'\ \Leftrightarrow\ (\mathsf{epi}\, s) =_Q (\mathsf{epi}\, s')$

$(q2)$ $\forall q\!:\!Q.\exists s\!:\!S_A\ .\ q =_Q (\mathsf{epi}\, s)$

$(q3)$ $\mathfrak{s}_\mathfrak{a}\ (\mathfrak{T}[(s\!:\!S_A, q\!:\!Q).((\mathsf{epi}\, s) =_Q q)])\ \mathfrak{q}$

We thus exhibit Q for B, and $\mathfrak{q}$ for $\mathfrak{b}$; it remains to derive

1. $(\mathsf{pack}Q\mathfrak{q}) = (\mathsf{pack}A\mathfrak{a})$ and

2. $Ax[Q,\mathfrak{q}]$.

To show the derivability of (1), it suffices to observe that, through Theorem 3.5, $(s3)$ and $(q3)$ give $(\mathsf{pack}A\mathfrak{a}) = (\mathsf{pack}S_A\mathfrak{s}_\mathfrak{a}) = (\mathsf{pack}Q\mathfrak{q})$. For (2) we must show the derivability of $\phi[Q,\mathfrak{q}]$ for every $\phi \in Ax$. We proceed by induction on the structure of ϕ.

(i) ϕ is $u =_b v$. We must derive $u[\mathfrak{q}] =_Q v[\mathfrak{q}]$. For any variable $q_i\!:\!Q$ in $u[\mathfrak{q}]$ or $v[\mathfrak{q}]$, we may by $(q2)$ assume an $s_{q_i}\!:\!S_A$ s.t. $(\mathsf{epi}\, s_{q_i}) = q_i$. From (†) we can derive $\wedge_{\ i}((\mathsf{mono}\, s_{q_i})\ \sim\ (\mathsf{mono}\, s_{q_i}))\ \Rightarrow\ u[\mathfrak{a}][\cdots(\mathsf{mono}\, s_{q_i})\cdots]\ \sim\ v[\mathfrak{a}][\cdots(\mathsf{mono}\, s_{q_i})\cdots]$,

but by $(s2)$ and $(s3)$ this is equivalent to $\wedge_i(s_{q_i} \sim' s_{q_i}) \Rightarrow u[\mathfrak{s}_\mathfrak{a}][\cdots s_{q_i} \cdots] \sim' v[\mathfrak{s}_\mathfrak{a}][\cdots s_{q_i} \cdots]$, which by $(s1)$ is equivalent to $u[\mathfrak{s}_\mathfrak{a}][\cdots s_{q_i} \cdots] \sim' v[\mathfrak{s}_\mathfrak{a}][\cdots s_{q_i} \cdots]$.

Then from $(q1)$ we can derive $(\mathsf{epi}\, u[\mathfrak{s}_\mathfrak{a}][\cdots s_{q_i} \cdots]) =_Q (\mathsf{epi}\, v[\mathfrak{s}_\mathfrak{a}][\cdots s_{q_i} \cdots])$. By $(q3)$ we then get $(\mathsf{epi}\, u[\mathfrak{s}_\mathfrak{a}][\cdots s_{q_i} \cdots]) = u[\mathfrak{q}]$ and $(\mathsf{epi}\, v[\mathfrak{s}_\mathfrak{a}][\cdots s_{q_i} \cdots]) = v[\mathfrak{q}]$.

(ii) ϕ is $u =_s v$, for s different from the behavioural sort b. This is an easy version of (i).

(iii) Suppose $\phi = \neg\phi'$. By negation n.f. convertibility it suffices to consider ϕ' an atomic formula. Thus, the cases for ϕ' as (i) and (ii) warrants proofs for $\neg\phi'$ similar to those of (i) and (ii).

(iv) Suppose $\phi = \phi' \wedge \phi''$. This is dealt with by i.h. on ϕ' and ϕ''.

(v) $\phi = \forall x\!:\!b.\phi'$. This is dealt with by i.h. on ϕ'.

$\Leftarrow$: Observe that to show $Ax[Q, \mathfrak{q}]$ we must either use $Ax'[Q, \mathfrak{q}]$ and the definition of $\sim$, or else $Ax[Q, \mathfrak{q}]$ was a tautology; in both cases we get (†). $\square$

—

We now show the correspondence for observational refinement on normal-form specifications. As hinted in Sect. 2.7, there is something to say about proving refinements involving **hide**. The issue concerns refinements where the implemented specification has hidden parts. Consider the refinement $SP \rightsquigarrow SP'$, for

$$SP \stackrel{def}{=} (\textbf{hide sorts } S'' \;\; \textbf{operations } \Omega'' \textbf{ in } SP^e)$$

In the calculus $\vdash_{\Pi_\rightsquigarrow}$ for behavioural refinement, based on the calculus $\vdash_{\Pi_S}$ for structured specifications (Bidoit et al., 1997), *cf.* Sect. 2.7, one shows this by showing

$$SP^e \rightsquigarrow SP'^+$$

where SP'^+ is a so-called *persistent* extension of SP', where a specification SP_2 is a persistent extension of SP_1 if $\Sigma_{SP_1} \subseteq \Sigma_{SP_2}$, and for every $A \in [\![SP_1]\!]$, there exists an $A' \in [\![SP_2]\!]$ such that $A = A'|_{\Sigma_{SP_1}}$.

It is shown in (Farrés-Casals, 1990, 1992) that SP'^+ may be constructed by the specification SP' **sum** Hid_{SP}, where Hid_{SP} is a basic specification constructed from the parts of SP involving the hidden symbols of SP. We omit the specifics of Hid_{SP}; for us it is enough that Hid_{SP} exists, but see (Farrés-Casals, 1990, 1992).

This so-called *inheriting strategy* is sound, provided Hid_{SP} is consistent, *i.e.,* provided Hid_{SP} has a model. The proviso for completeness (modulo the underlying predicate logic) is that SP fulfils an *independence* property, namely that for all $B \in [\![SP]\!]$ and $A \in [\![Hid_{SP}]\!]$, such that $B|_{\Sigma_0} = A|_{\Sigma_0}$, where Σ_0 is the greatest subsignature common to both Σ_{SP} and $\Sigma_{Hid_{SP}}$, there exists a $C \in [\![SP^e]\!]$ such that $C|_{\Sigma_{SP}} = B$ and $C|_{\Sigma_{Hid_{SP}}} = A$.

Intuitively, one generally needs Hid_{SP} to show the visible axioms in Ax. The inheriting strategy is used in EML. In the theorem below, we shall assume independence. Moreover, we postulate the following.

`Pers`: If SP is independent, we can derive the following persistency clause in the type-theoretic logic.

$$\forall X.\forall \mathfrak{x}\!:\!Sig_{SP'}[X]\ .\ \Theta_{SP'}(\mathsf{pack}X\mathfrak{x})\ \Rightarrow\ (\exists \mathfrak{x}^e\!:\!Sig^e_{SP}\ .\ \mathfrak{x} = \mathfrak{x}^e|_{Sig_{SP'}}\ \wedge\ Ax_{Hid_{SP}}[X, \mathfrak{x}^e])$$

It seems plausible that `Pers` is true, but we have not verified this, so `Pers` is included as an assumption. To see if `Pers` is true one would look at methods for proving persistency. In some cases persistency can even be checked syntactically.

Hiding on the right-hand side of refinements poses no problem, because the hidden parts are incorporated straightforwardly in the proof assumptions.

Theorem 4.32 (Correspondence (Normal Form)) *Let SP and SP' be algebraic specifications in normal form. Let $\Sigma_{SP} = \langle S, \Omega\rangle$. Assume one behavioural sort b in Σ_{SP}, and assume $Obs = In = S \setminus b$. Assume that $\equiv^{Obs,Obs}$ on Σ_{SP}***Alg** *is behaviourally closed w.r.t. the factorising observational congruence $\approx^{Obs,Obs}$. Assume furthermore that SP is independent, and assume* `Pers`*. Then*

$$(\textbf{abstract}\ SP\ \textbf{wrt}\ \equiv^{Obs,Obs}) \rightsquigarrow (\textbf{abstract}\ SP'\ \textbf{wrt}\ \equiv^{Obs',Obs'})$$
$$\Leftrightarrow\ \mathcal{T}(\textbf{abstract}\ SP\ \textbf{wrt}\ \equiv^{Obs,Obs}) \rightsquigarrow \mathcal{T}(\textbf{abstract}\ SP'\ \textbf{wrt}\ \equiv^{Obs',Obs'})$$

Proof: This proof is an extension of the proof for Theorem 4.31. Let

$$SP = \textbf{hide sorts}\ S^h\ \textbf{operations}\ \Omega^h\ \textbf{in}\ \langle \Sigma^e, Ax\rangle,$$
$$SP' = \textbf{hide sorts}\ S'^h\ \textbf{operations}\ \Omega'^h\ \textbf{in}\ \left\langle \Sigma'^e, Ax\right\rangle'.$$

Let $\langle\langle Sig_{SP}, \Theta_{SP}\rangle, \mathfrak{T}^e_{SP}, Obs\rangle$ and $\langle\langle Sig_{SP'}, \Theta_{SP'}\rangle, \mathfrak{T}^e_{SP'}, Obs'\rangle$ be the resulting type-theoretic specifications. We are still dealing with the identity constructor. This means $\Sigma_{SP} = \Sigma_{SP'}$ and hence $Obs = Obs'$, and also $Sig_{SP} = Sig_{SP'}$.

$\Rightarrow$: We have to show the derivability of

$$\forall u\!:\!Sig_{SP'}\ .\ \Theta_{SP'}(u)\ \Rightarrow\ \Theta_{SP}(u)$$

assuming $(\textbf{abstract}\ SP\ \textbf{wrt}\ \equiv^{Obs,Obs}) \rightsquigarrow (\textbf{abstract}\ SP'\ \textbf{wrt}\ \equiv^{Obs,Obs})$. From this we obtain the proof-theoretical information using $\vdash_{\Pi_\rightsquigarrow}$ along the same lines as before. Since the identity constructor is stable, the above is equivalent to proving (**abstract** SP **wrt** $\equiv^{Obs,Obs}$) $\rightsquigarrow$ SP'. By syntax directedness, we must now have had $SP \rightsquigarrow SP'/\!\approx_{Obs,In}$. By the rules for **hide** this was derived from $\langle\Sigma^e, Ax\rangle \rightsquigarrow (SP'/\!\approx_{Obs,In})^+$, where $(SP'/\!\approx_{Obs,In})^+ = (SP'/\!\approx_{Obs,In}$ **sum** $Hid_{SP})$, for a basic specification Hid_{SP} composed of parts of SP involving its hidden symbols. The above in turn is derived from $(SP'/\!\approx_{Obs,In}$ **sum** $Hid_{SP}) \vdash_{\Pi_S} Ax$, where the semantics of $SP'/\!\approx_{Obs,In}$ is $\{Dom(\approx_A^{Obs,In})/\!\approx_A^{Obs,In} \mid A \in [\![SP']\!]\}$. By the rules for **hide** and **sum** in $\vdash_{\Pi_S}$ we must have had the predicate logic statement

$$Ax', Ax_{Hid_{SP}}, Ax(\sim) \;\vdash\; \mathcal{L}(Ax) \qquad (\ddagger)$$

where $Ax(\sim)$ and $\mathcal{L}(Ax)$ are as before.

Using this we derive our goal as follows. Let $u\!:\!Sig_{SP'}$ be arbitrary. Let $\mathfrak{T}[X]$ denote $\mathfrak{T}_{SP'}[X](=\mathfrak{T}_{SP}[X])$. We must derive

$$\exists B.\exists \mathfrak{b}\!:\!\mathfrak{T}^e_{SP}[B]\;.\; u = (\mathsf{pack} B\, \mathfrak{b}|_{\mathfrak{T}}) \;\wedge\; Ax[B, \mathfrak{b}]$$

assuming $\exists A.\exists \mathfrak{a} : \mathfrak{T}^e_{SP'}[A]\;.\; u = (\mathsf{pack} A\, \mathfrak{a}|_{\mathfrak{T}}) \;\wedge\; Ax'[A, \mathfrak{a}]$. Let $\mathfrak{a}$ and A denote the witnesses projected out from that assumption. It suffices to exhibit a B and $\mathfrak{b}\!:\!\mathfrak{T}^e_{SP}$ such that $(\mathsf{pack} B\, \mathfrak{b}|_{\mathfrak{T}}) = (\mathsf{pack} A\, \mathfrak{a}|_{\mathfrak{T}})$ and $Ax[B, \mathfrak{b}]$.

We form $\sim$ type-theoretically as before by

$$\sim \;\stackrel{\mathit{def}}{=}\; (a\!:\!A, a'\!:\!A).(Beh^*_A(a, a'))$$

and we have $\mathfrak{a}|_{\mathfrak{T}}\ \mathfrak{T}[\sim]\ \mathfrak{a}|_{\mathfrak{T}}$, since $\sim$ is an axiomatisation of a partial congruence. As before, we use SUB to get S_A, $\mathfrak{s}_{\mathfrak{a}}$, $\sim' \subset S_A \times S_A$, and $\mathsf{mono}\!:\!S_A \to A$ such that we can derive

$(s1)$ $\forall s\!:\!S_A\;.\; s \;\sim'\; s$

$(s2)$ $\forall s.s'\!:\!S_A\;.\; s \sim' s' \;\Leftrightarrow\; (\mathsf{mono}\, s) \sim (\mathsf{mono}\, s')$

$(s3)$ $\mathfrak{a}|_{\mathfrak{T}}\ (\mathfrak{T}[(a\!:\!A, s\!:\!S_A).(a =_A (\mathsf{mono}\, s))])\ \mathfrak{s}_{\mathfrak{a}}$

By $(s2)$ we get $\mathfrak{s}_{\mathfrak{a}}\ \mathfrak{T}[\sim']\ \mathfrak{s}_{\mathfrak{a}}$. We also get $equiv(\sim')$ by $(s1)$. We now use QUOT to get Q and $\mathfrak{q}\!:\!\mathfrak{T}[Q]$ and $\mathsf{epi}\!:\!S_A \to Q$ s.t.

$(q1)$ $\forall s, s'\!:\!S_A\;.\; s \sim' s' \;\Leftrightarrow\; (\mathsf{epi}\, s) =_Q (\mathsf{epi}\, s')$

$(q2)$ $\forall q\!:\!Q.\exists s\!:\!S_A\;.\; q =_Q (\mathsf{epi}\, s)$

$(q3)$ $\mathfrak{s}_{\mathfrak{a}}\ (\mathfrak{T}[(s\!:\!S_A, q\!:\!Q).((\mathsf{epi}\, s) =_Q q)])\ \mathfrak{q}$

We now have Q and $\mathfrak{q}\!:\!\mathfrak{T}[Q]$ as candidates for B and $\mathfrak{b}$. We give Q for B, but we must extend $\mathfrak{q}$ to some $\mathfrak{q}^e\!:\!\mathfrak{T}^e_{SP}$ in such a way that

1. $(\mathsf{pack} Q\, \mathfrak{q}^e|_{\mathfrak{T}}) = (\mathsf{pack} A\, \mathfrak{a}|_{\mathfrak{T}})$ and

2. $Ax[Q, \mathfrak{q}^e]$.

By *Pers*, we get

$$\forall X.\forall \mathfrak{x}:\mathfrak{T}[X]\ .\ \Theta_{SP'}(\mathsf{pack}X\mathfrak{x})\ \Rightarrow\ (\exists \mathfrak{x}^e:\mathfrak{T}^e_{SP}\ .\ \mathfrak{x}=\mathfrak{x}^e|_{\mathfrak{T}}\ \wedge\ Ax_{Hid_{SP}}[X,\mathfrak{x}^e])$$

By $(s3)$ and $(q3)$ we have by Theorem 3.5, $(\mathsf{pack}A\,\mathfrak{a}|_{\mathfrak{T}}) = (\mathsf{pack}S_A\mathfrak{s}_{\mathfrak{a}}) = (\mathsf{pack}Q\mathfrak{q})$. Since of course $\Theta_{SP'}(\mathsf{pack}A\,\mathfrak{a}|_{\mathfrak{T}})$, we therefore have $\Theta_{SP'}(\mathsf{pack}Q\mathfrak{q})$. Then by *Pers* we get $\mathfrak{q}^e$, which we exhibit for $\mathfrak{b}$.

We already showed (1). For (2) we must derive $\phi[Q,\mathfrak{q}^e]$ for every $\phi \in Ax$. This goes as before by induction on the structure of ϕ. We show the first case.

(i) ϕ is $u =_b v$. We must derive $u[\mathfrak{q}^e] =_Q v[\mathfrak{q}^e]$. For any variable $q_i:Q$ in $u[\mathfrak{q}^e]$ or $v[\mathfrak{q}^e]$, we may by $(q2)$ assume an $s_{q_i}:S_A$ s.t. $(\mathsf{epi}\,s_{q_i}) = q_i$. From (‡) we can derive $\wedge_i((\mathsf{mono}\,s_{q_i})\ \sim\ (\mathsf{mono}\,s_{q_i}))\ \Rightarrow\ u[\mathfrak{a}][\cdots(\mathsf{mono}\,s_{q_i})\cdots]\ \sim\ v[\mathfrak{a}][\cdots(\mathsf{mono}\,s_{q_i})\cdots]$, but by $(s2)$ and $(s3)$ this is equivalent to $\wedge_i(s_{q_i} \sim' s_{q_i})\ \Rightarrow\ u[\mathfrak{s_a}^e][\cdots s_{q_i}\cdots] \sim' v[\mathfrak{s_a}^e][\cdots s_{q_i}\cdots]$, which by $(s1)$ is equivalent to $u[\mathfrak{s_a}^e][\cdots s_{q_i}\cdots] \sim' v[\mathfrak{s_a}^e][\cdots s_{q_i}\cdots]$. Here $\mathfrak{s_a}^e$ is obtained from *Pers* like $\mathfrak{q}^e$.

Then from $(q1)$ we can derive $(\mathsf{epi}\,u[\mathfrak{s_a}^e][\cdots s_{q_i}\cdots]) =_Q (\mathsf{epi}\,v[\mathfrak{s_a}^e][\cdots s_{q_i}\cdots])$. By $(q3)$ we get $(\mathsf{epi}\,u[\mathfrak{s_a}^e][\cdots s_{q_i}\cdots]) = u[\mathfrak{q}^e]$ and $(\mathsf{epi}\,v[\mathfrak{s_a}^e][\cdots s_{q_i}\cdots]) = v[\mathfrak{q}^e]$.

$\Leftarrow$: Consider (‡) in place of (†). □

—

The extension of the correspondence to include constructors is simpler. We must generalise the assumption *Pers*, to the following.

FPers: If SP is independent, then we can derive the following persistency clause in the type-theoretic logic.

$$\begin{aligned}\forall X,Y.\forall \mathfrak{x}:Sig_{SP'}[X],\mathfrak{y}:Sig_{SP}[Y]\ .\ (\Theta_{SP'}(\mathsf{pack}X\mathfrak{x})\ \wedge\ (\mathsf{pack}Y\mathfrak{y}) = F(\mathsf{pack}X\mathfrak{x}))\\ \Rightarrow\ (\exists \mathfrak{y}^e:Sig^e_{SP}\ .\ \mathfrak{y}=\mathfrak{y}^e|_{\mathfrak{T}_{SP}}\ \wedge\ Ax_{Hid_{SP}}[Y,\mathfrak{y}^e])\end{aligned}$$

Again, it seems plausible that *FPers* is true, but we have not verified this, so *FPers* is included as an assumption. Effectively, *Pers* is a special case of *FPers*.

Theorem 4.33 (Correspondence (Constructors)) *Let SP and SP' be algebraic specifications in normal form. Let $\Sigma_{SP} = \langle S,\Omega\rangle$. Assume one behavioural sort b in Σ_{SP}, and assume $Obs = In = S \setminus b$. Assume that $\equiv^{Obs,Obs}$ on $\Sigma_{SP}\mathbf{Alg}$ is behaviourally closed w.r.t. the factorising observational congruence $\approx^{Obs,Obs}$. Assume furthermore that SP is independent, and assume FPers. Then*

$$\begin{aligned}&(\textbf{abstract}\ SP\ \textbf{wrt}\ \equiv^{Obs,Obs}) \underset{\kappa}{\rightsquigarrow} (\textbf{abstract}\ SP'\ \textbf{wrt}\ \equiv^{Obs',Obs'})\\ &\Leftrightarrow\ \mathcal{T}(\textbf{abstract}\ SP\ \textbf{wrt}\ \equiv^{Obs,Obs}) \underset{F}{\rightsquigarrow} \mathcal{T}(\textbf{abstract}\ SP'\ \textbf{wrt}\ \equiv^{Obs',Obs'})\end{aligned}$$

where κ and F are such that κ is a stable realisation of a constructor specification CSP for which $\mathcal{T}$ is applicable, and F is a realisation of $\mathcal{T}(CSP)$.

Proof: This proof is an extension of the proof for Theorem 4.32. Through the translation $\mathcal{T}$ on CSP, κ and F are both characterised by Ax_{CSP}. Note first that if $CSP = \Pi S : (\textbf{abstract}\ SP'\ \textbf{wrt}\ \equiv^{Obs',Obs'})\ .\ (\textbf{abstract}\ SP\ \textbf{wrt}\ \equiv^{Obs,Obs})$, then by definition we are already done. However, CSP might of course not be this particular specification.

$\Rightarrow$: We have to show the derivability of

$$\forall u : Sig_{SP'}\ .\ \Theta_{SP'}(u)\ \Rightarrow\ \Theta_{SP}(Fu)$$

supposing $(\textbf{abstract}\ SP\ \textbf{wrt}\ \equiv^{Obs,Obs}) \underset{\kappa}{\rightsquigarrow} (\textbf{abstract}\ SP'\ \textbf{wrt}\ \equiv^{Obs',Obs'})$. By similar reasoning to that above, we get

$$Ax', Ax_{Hid_{SP}}, Ax(\sim), Ax_{CSP}\ \vdash\ \mathcal{L}(Ax) \qquad (\S)$$

Assume $\exists A.\exists \mathfrak{a} : \mathfrak{T}^e_{SP'}[A]\ .\ u = (\mathsf{pack} A\, \mathfrak{a}|_{\mathfrak{T}_{SP'}})\ \wedge\ Ax'[A, \mathfrak{a}]$. Let $\mathfrak{a}$ and A denote the witnesses thus projected out. We should exhibit a B and $\mathfrak{b} : \mathfrak{T}^e_{SP}$ such that

1. $(\mathsf{pack} B\, \mathfrak{b}|_{\mathfrak{T}_{SP}}) = F(\mathsf{pack} A\, \mathfrak{a}|_{\mathfrak{T}_{SP'}})$ and
2. $Ax[B, \mathfrak{b}]$.

Let $(\mathsf{pack} A' \mathfrak{a}') = F(\mathsf{pack} A\, \mathfrak{a}|_{\mathfrak{T}_{SP'}})$. As before, we use SUB and QUOT to obtain $S_{A'}$, $\mathfrak{s}_{\mathfrak{a}'} : \mathfrak{T}_{SP}[S_{A'}]$, and Q, $\mathfrak{q} : \mathfrak{T}_{SP}[\mathfrak{q}]$. We give Q for B. We have by construction through SUB and QUOT, that $F(\mathsf{pack} A\, \mathfrak{a}|_{\mathfrak{T}_{SP'}}) = (\mathsf{pack} S_{A'} \mathfrak{s}_{\mathfrak{a}'}) = (\mathsf{pack} Q \mathfrak{q})$ by Theorem 3.5. Thus, by *FPers*, we get the desired $\mathfrak{a}'^e : \mathfrak{T}^e_{SP}[A']$, ${\mathfrak{s}_{\mathfrak{a}'}}^e : \mathfrak{T}^e_{SP}[S_{A'}]$, and $\mathfrak{q}^e : \mathfrak{T}^e_{SP}[X]$ for $\mathfrak{b}$, and the verification of (2) using (§).

$\Leftarrow$: Consider (§) in place of (‡). $\square$

An alternative to requiring that κ and F are related through specifications CSP and $\mathcal{T}(CSP)$, would be to try to extract axioms directly from the definitions of given κ and F. We implicitly extracted axioms for the type-theoretic constructors in Example 4.26 and Example 4.27.

4.7 Summary

In this chapter we have expressed an account of algebraic specification refinement in System F with the logic for relational parametricity of (Plotkin and Abadi, 1993). We have seen how the concepts of observational refinement and stable constructors are inherent in this type-theoretic setting, because at first order, equality at existential type is exactly observational equivalence (Theorem 4.19).

We have shown a correspondence (Theorems 4.31, 4.32 and 4.33) between refinement in the algebraic specification sense, and a notion of type theory specification refinement (Def. 4.14). This correspondence encompasses normal-form specifications and constructors, and is thus general in light of the normalisation results and the view that **abstract**, which in our discussion gives specification up to observational equivalence, should only be applied outermost. We have seen how a proof technique from algebraic specification for proving refinement up to observational equivalence can be mirrored in type theory by extending the logic soundly with axioms QUOT and SUB.

The stage is now set for type-theoretic development in several directions. First, algebraic specification has much more to it than presented here. An obvious extension would be to express specification building operators in System F and the logic. However, since we have shown a correspondence for observational specifications over normal form specifications, we can in fact use any already existing normalisable specification language, utilise the normal form results of (Wirsing, 1993; Farrés-Casals, 1992; Cengarle, 1995), and make the transition to type-theoretic formalism at normal form. This rationale is based on the view that specification building operators are primarily meant as aids in the specification language for the benefit of the specifier. Nevertheless, it might be beneficial to reflect specificational structure in the proof process, and then specification building operators should indeed be expressed in System F and the logic.

Another direction would be to provide a fuller account of specifications of parameterised programs and also parameterised specifications (Sannella et al., 1992). It is for example possible to have refinements of constructor specifications. Also, we have only dealt with constructors with one argument. It should be simple to extend the discussion to constructors with multiple arguments. This allows various development strategies, see also (Sannella and Tarlecki, 1997). We do not pursue these issues further in this thesis.

We can also use our notion of type theory specification refinement as a base to start looking at specification refinement for data types with higher-order polymorphic operations. This is the path we shall take. In the next chapter, we investigate an alternative notion of simulation relation that will among other things, establish a higher-order version of Theorem 4.17. We then use these results in Ch. 6 where we look at specification refinement for data types with higher-order operations. In Ch. 7, we deal with data types with polymorphic operations.

Chapter 5

General Simulation Relations

This chapter continues to investigate specification refinement in a setting consisting of System F and relational parametricity in Reynolds' sense, as expressed in Plotkin and Abadi's logic for parametric polymorphism. This setting allows an elegant formalisation of abstract data types as existential types (Mitchell and Plotkin, 1988), and furthermore, the relational parametricity axiom enables one to derive in the logic that two data types, *i.e.,* inhabitants of existential type, are equal if and only if there exists a simulation relation between their implementation parts. Together with the fact that at first order, equality at existential type is derivably equivalent to a notion of observational equivalence, this formalises the semantic proof principle of (Mitchell, 1991). This also lifts the type-theoretic formalism of refinement to a notion of specification refinement up to observational equivalence; a key issue in program development.

After having established a correspondence at first order between algebraic specification refinement and type-theoretic refinement in Ch. 4, we now cast off and discuss type-theoretic specification refinement in more generality, *i.e.,* we treat data types whose operations may be higher order and polymorphic.

At higher order, we are not able to establish the formal link between the existence of a simulation relation and observational equivalence. Nor are we able to show in the logic that simulation relations compose. This is the topic of this chapter. The solution presented is to use a weak arrow-type relation giving an alternative notion of simulation relation in the logic that observes the abstraction barrier inherent in the existential type. The resulting abstraction barrier-observing simulation relations agree with observational equivalence and also compose at higher-order. In spirit, this relates the syntactic level to recent and on-going work on the semantic level remedying the fact that logical relations traditionally used to describe refinement do not compose at higher order (Honsell et al., 2000; Honsell and Sannella, 1999; Kinoshita and Power, 1999; Kinoshita et al., 1997; Plotkin et al., 2000).

When data types have higher-order operations, it is not generally the case that the existence of simulation relations between the data representations of two packages ($\mathsf{pack}Aa$) and ($\mathsf{pack}Bb$) implies the existence of simulation relations between data representations of packages ($\mathsf{pack}Cc$) = ($\mathsf{pack}Aa$) and ($\mathsf{pack}Dd$) = ($\mathsf{pack}Bb$). This is however the case for the alternative notion of simulation relation. Thus, abstraction barrier-observing simulation relations resolve the dilemma of viewing data types as pairs or packages, and therefore also resolve the circularity problem of Theorem 3.5 (p. 63) for packages. Recall that for data types with first-order operations we do have this tight connection between packages and components for standard simulation relations, *cf.* Theorem 4.20 and Theorem 4.21.

Other relevant work concerning System F and parametricity includes the interesting (Pitts, 1997, 1998) showing that the introduction of non-terminating recursion also breaks down the correspondence between the existence of a simulation relation and observational equivalence.

This chapter deals specifically with the issue of simulation relations. Specification refinement in more generality is tackled in Ch. 6, where the proof method from algebraic specification for proving observational refinements formalised by Bidoit *et al.*, *cf.* Sect. 2.7 and Sect. 4.4, is imported into the higher-order setting.

5.1 The Break-Down at Higher Order

We will now look into what causes the loss at higher order of the correspondence between observational equivalence and simulation relations. We find that the culprit is a disregard for *Abs-Bar*. But first we introduce higher-order profiles.

We no longer assume first-order operations as we did in Ch. 4. We allow higher-order and in principle, polymorphic operations in data types. However, it is hard to find reasonable examples utilising polymorphic operations, without wanting to move to calculus F_3. We do this in Ch. 7. Based on these considerations, we will for clarity defer polymorphism in data types until Ch. 7.

We thus assume a higher-order version of $\mathtt{FADT}_{Obs}$ (*p.* 87). Again we presume a specification scenario, and hence a current set of observable types *Obs* according to Def. 4.6 (*p.* 79) or Def. 4.12 (*p.* 84).

$\mathtt{HADT}_{Obs}$: Every profile $T_i[X] = T_{i1}[X] \to \cdots \to T_{n_i}[X] \to T_{c_i}[X]$ of an abstract type $\exists X.\mathfrak{T}[X]$ is such that $T_{ij}[X]$ has no occurrences of universal types other than those in *Obs*, and $T_{c_i}[X]$ is either X or some $D \in Obs$.

We can relax $\mathtt{HADT}_{Obs}$ in various ways to admit degrees of polymorphism. Mainly, the forthcoming results still hold true, albeit by different arguments. For example, we can permit polymorphic domain types $T_{ij}[X]$. Refer to this relaxation of $\mathtt{HADT}_{Obs}$ as $\mathtt{ADT}_{Obs}$. Moreover, we can relax the restriction on the codomain type $T_{c_i}[X]$. Here, universal types that are not in *Obs* are not admitted in $T_{c_i}[X]$. This restriction can be relaxed to allow $T_{c_i}[X]$ to be a universal type $\forall Y.T[Y, X]$, where $T[Y, X]$ is again a profile for which $\mathtt{ADT}_{Obs}$ holds.

We do not know how severe the restriction posed by $\mathtt{HADT}_{Obs}$ is in practice, nor how useful it is to have polymorphic codomain types that are not closed inductive types. Although it is possible to relax $\mathtt{HADT}_{Obs}$, we do not do this here in order to avoid unmotivated clutter. The discussion at present thus focuses on the higher-order aspect.

5.1.1 The Break-Down

If $\mathfrak{T}[X]$ has higher-order function profiles, we are not able to prove Theorem 4.18 (*p.* 87), *i.e.,* we lose the correspondence between the existence of simulation relations and observational equivalence.

Theorem 5.1 (Failure of Correspondence with Obs. Equiv.) *If* $\mathfrak{T}[X]$ *has higher-order profiles, the following sequent is* not *in general derivable in the logic.*

$$
\begin{aligned}
&\forall A, B.\forall \mathfrak{a}\,{:}\,\mathfrak{T}[A, \boldsymbol{Z}], \mathfrak{b}\,{:}\,\mathfrak{T}[B, \boldsymbol{Z}]\ .\\
&\qquad \exists R \subset A \times B\ .\ \mathfrak{a}(\mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}])\mathfrak{b}\\
&\qquad\qquad\qquad\qquad \Leftrightarrow \quad \textstyle\bigwedge_{D \in Obs} \forall f\,{:}\,\forall X.(\mathfrak{T}[X, \boldsymbol{Z}] \to D)\ .\ (f\,A\,\mathfrak{a}) = (f\,B\,\mathfrak{b})
\end{aligned}
$$

Proof: See Example 5.7 below. □

Furthermore, Theorem 4.22 (*p.* 90) fails, *i.e.,* the composability of simulation relations breaks down.

Theorem 5.2 (Failure of Composability) *If $\mathfrak{T}[X]$ has higher-order profiles, the following sequent is* not *in general derivable in the logic.*

$$\forall A, B, C, R \subset A \times B, S \subset B \times C, \mathfrak{a} : \mathfrak{T}[A, \boldsymbol{Z}], \mathfrak{b} : \mathfrak{T}[B, \boldsymbol{Z}], \mathfrak{c} : \mathfrak{T}[C, \boldsymbol{Z}]. \\ \mathfrak{a}(\mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}])\mathfrak{b} \;\wedge\; \mathfrak{b}(\mathfrak{T}[S, \mathbf{eq}_{\boldsymbol{Z}}])\mathfrak{c} \;\Rightarrow\; \mathfrak{a}(\mathfrak{T}[S \circ R, \mathbf{eq}_{\boldsymbol{Z}}])\mathfrak{c}$$

Proof: See Example 5.7 below. □

In fact, even the transitivity of existence of simulation relations fails.

Theorem 5.3 (Failure of Transitivity of Existence) *If $\mathfrak{T}[X]$ has higher-order profiles, the following sequent is* not *in general derivable in the logic.*

$$\forall A, B, C, \mathfrak{a} : \mathfrak{T}[A, \boldsymbol{Z}], \mathfrak{b} : \mathfrak{T}[B, \boldsymbol{Z}], \mathfrak{c} : \mathfrak{T}[C, \boldsymbol{Z}]. \\ \exists R \subset A \times B \;.\; \mathfrak{a}(\mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}])\mathfrak{b} \;\wedge\; \exists S \subset B \times C \;.\; \mathfrak{b}(\mathfrak{T}[S, \mathbf{eq}_{\boldsymbol{Z}}])\mathfrak{c} \\ \Rightarrow\; \exists Q \subset A \times C \;.\; \mathfrak{a}(\mathfrak{T}[Q, \mathbf{eq}_{\boldsymbol{Z}}])\mathfrak{c}$$

Proof: See Example 5.7 below. □

Note that we still have the transitivity of SimRel according to Def. 3.3 (*p.* 62), due to Theorem 3.5 (*p.* 63) and the transitivity of equality, *i.e.,*

Theorem 5.4 (Transitivity of SimRel) *The following* is *derivable in the logic.*

$$\forall u, v, w : \exists X.\mathfrak{T}[X, \boldsymbol{Z}] \;.\; u \;\mathsf{SimRel}\; v \;\wedge\; v \;\mathsf{SimRel}\; w \;\Rightarrow\; u \;\mathsf{SimRel}\; w$$

Proof: This follows from Theorem 3.5. □

Theorem 5.4 does not contradict Theorem 5.3. That two packages are related by SimRel does not infer a simulation relation between the data representations of the packages; recall the remarks at the end of Sect. 3.3.2. In fact, now we can establish the following theorem which in turn implies Theorem 3.7 (*p.* 64).

Theorem 5.5 (Failure of Tight Connection with Components) *If $\mathfrak{T}[X]$ has higher-order profiles, the schema below is* not *in general derivable in the logic.*

$$\forall A, B.\forall \mathfrak{a} : \mathfrak{T}[A, \boldsymbol{Z}], \mathfrak{b} : T[B, \boldsymbol{Z}]) \;. \\ (\mathsf{pack} A \mathfrak{a}) \;\mathsf{SimRel}_{\mathfrak{T}[X, \mathbf{eq}_{\boldsymbol{Z}}]}\; (\mathsf{pack} B \mathfrak{b}) \;\Leftrightarrow\; \exists R \subset A \times B \;.\; \mathfrak{a} \;\mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}]\; \mathfrak{b}$$

Proof: Suppose the sequent were derivable. Then using Theorem 5.4, we could derive the sequent in Theorem 5.3, thus contradicting Theorem 5.3. □

It is natural to compare the properties of syntactical simulation relations with the situation regarding logical relations on the semantic level (Mitchell, 1996; Tennent, 1997; O'Hearn and Tennent, 1993). Logical relations *e.g.,* between applicative structures or combinatory algebras, have traditionally been used to describe data refinement. For this to be useful, one uses logical relations that are the identity at observable types. As for our simulation relations, the existence of such logical relations does coincide with observational equivalence for data types with only first-order operations, but this correspondence is lost at higher order.

When it comes to composability, note first that for simulation relations, relation composition in the context of higher types is defined by lifting the composition at base type, *e.g.,* if $R \subset A \times B$ and $S \subset B \times C$, then the composite relation between $A \to A$ and $C \to C$ would be $S \circ R \to S \circ R$. This is generally laxer than the composition at higher types, *i.e.,* $f\ (S \to S) \circ (R \to R)\ h \ \Rightarrow\ f\ (S \circ R \to S \circ R)\ h$, but not conversely. Lax composition fails in general in the presence of higher-order operations both for logical relations, and for simulation relations, as demonstrated in Theorem 5.2. Finally, at higher-order, the transitivity of existence fails for logical relations as it does for simulation relations.

We defer a fuller comparison between logical relations and the syntactic simulation relations in this thesis to a different occasion.

—

Before we start, it might be informative to point out the following fact in our setting of System F and relational parametricity.

> *If we choose all types to be observable types, then observational equivalence coincides with equality, and thus also coincides with the existence of a simulation relation.*

Suppose namely that we define observational equivalence as follows.

$$\begin{aligned}\mathsf{ObsEq}^{omni}_{\mathfrak{T}[X]} \stackrel{def}{=} (u\!:\!\exists X.\mathfrak{T}[X], v\!:\!\exists X.\mathfrak{T}[X])\ .\\ (\exists A, B.\exists \mathfrak{a}\!:\!\mathfrak{T}[A], \mathfrak{b}\!:\!\mathfrak{T}[B]\ .\ u = (\mathsf{pack} A \mathfrak{a})\ \wedge\ v = (\mathsf{pack} B \mathfrak{b})\ \wedge\\ \forall Y.\forall f\!:\!\forall X.(\mathfrak{T}[X] \to Y)\ .\ (f A\, \mathfrak{a}) = (f B\, \mathfrak{b}))\end{aligned}$$

Then we have

$$\forall \boldsymbol{Z}.\forall u, v\!:\!\exists X.\mathfrak{T}[X, \boldsymbol{Z}]\ .\ u\ =_{\exists X.\mathfrak{T}[X,\boldsymbol{Z}]}\ v \quad\Leftrightarrow\quad u\ \mathsf{ObsEq}^{omni}_{\mathfrak{T}[X,\boldsymbol{Z}]}\ v$$

Left to right is obvious. For right to left, let A, B, and $\mathfrak{a}\!:\!\mathfrak{T}[A]$, $\mathfrak{b}\!:\!\mathfrak{T}[B]$ be such that $u = (\mathsf{pack} A \mathfrak{a}) \wedge v = (\mathsf{pack} B \mathfrak{b})$ and $\forall Y.\forall f\!:\!\forall X.(\mathfrak{T}[X] \to Y)\ .\ (f A\, \mathfrak{a}) = (f B\, \mathfrak{b})$. Then consider the computation

$$f \stackrel{def}{=} \Lambda X.\lambda \mathfrak{x}\!:\!\mathfrak{T}[X]\ .\ (\mathsf{pack} X \mathfrak{x}) : \forall X.(\mathfrak{T}[X] \to \exists X.\mathfrak{T}[X])$$

whereby we get

$$u = (\mathsf{pack} A \mathfrak{a}) = (f\, A\, \mathfrak{a}) = (f\, B\, \mathfrak{b}) = (\mathsf{pack} B \mathfrak{b}) = v$$

Note that to make this work, it suffices that the "self" type $\exists X.\mathfrak{T}[X]$ is an observable type. In contrast, one usually insists that observable types be the traditional printable types, or at most inductive types. This is what we do in the definition of observational equivalence in Def. 4.4 (*p.* 78). This sensible restriction on what observable types are then introduces the variance between observational equivalence and the existence of simulation relations, and hence also the variance between observational equivalence and equality at existential type.

Note however that the existence of a simulation relation implies observational equivalence, regardless, through Param, *i.e.,*

Theorem 5.6 *The following is derivable in the logic using* Param.

$$\begin{aligned}
&\forall A, B.\forall \mathfrak{a}\!:\!\mathfrak{T}[A, \boldsymbol{Z}], \mathfrak{b}\!:\!\mathfrak{T}[B, \boldsymbol{Z}]\ .\\
&\qquad \exists R \subset A \times B\ .\ \mathfrak{a}(\mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}])\mathfrak{b}\\
&\qquad\qquad\qquad \Rightarrow \quad \textstyle\bigwedge_{D \in Obs} \forall f\!:\!\forall X.(\mathfrak{T}[X, \boldsymbol{Z}] \to D)\ .\ (f\, A\, \mathfrak{a}) = (f\, B\, \mathfrak{b})
\end{aligned}$$

Proof: This follows from the parametricity axiom schema. Consider the Param instance $\forall Y.\forall f\!:\!\forall X.(\mathfrak{T}[X, \boldsymbol{Z}] \to Y)\ .\ f(\forall X.\mathfrak{T}[X, \mathbf{eq}_{\boldsymbol{Z}}] \to \mathsf{eq}_Y)f$. □

This suffices of course for showing observational equivalence. However, to settle the use of simulations relations as a complete method for proving observational equivalence, we must establish the full correspondence.

—

Below is an example demonstrating that observational equivalence does not imply the existence of a simulation relation for abstract types with higher-order profiles, thus providing a proof for Theorem 5.1. In (Mitchell, 1996) there is an example for the simply-typed lambda calculus and applicative structures showing that at higher order, observational equivalence does not coincide with the existence of a logical relation between the two relevant applicative structures. The example uses the full set-theoretic hierarchy, and uses the existence of a function that is able to discern non-computable functions from computable functions. In our case this is perhaps not a natural strategy to follow, firstly because if we want to relate to the PER-model, then all functions are partial recursive. More important however, is the fact that the notion of definability that is relevant for us is not absolute definability, but definability w.r.t. the abstract type. The example also demonstrates that the existence of simulation relations is not transitive at higher order, in particular simulation relations do not compose in general. This provides proofs for theorems 5.2 and 5.3.

Example 5.7 Consider $Sig_{\mathsf{SetCE}} \stackrel{def}{=} \exists X.\mathfrak{T}_{\mathsf{SetCE}}[X]$, where

$$\begin{aligned}\mathfrak{T}_{\mathsf{SetCE}}[X] \stackrel{def}{=} (&\mathsf{empty}\!:\!X,\ \mathsf{add}\!:\!\mathsf{Nat}\!\to\! X\!\to\! X,\ \mathsf{remove}\!:\!\mathsf{Nat}\!\to\! X\!\to\! X,\\ &\mathsf{in}\!:\!\mathsf{Nat}\!\to\! X\!\to\!\mathsf{Bool},\ \mathsf{crossover}\!:\!(\mathsf{Nat}\!\to\! X\!\to\! X)\!\to\!\mathsf{Nat}\!\to\!\mathsf{Bool})\end{aligned}$$

and consider $(\mathsf{pack}\,\mathsf{List}_{\mathsf{Nat}}\ \mathfrak{a})\!:\!Sig_{\mathsf{SetCE}}$ and $(\mathsf{pack}\,\mathsf{List}_{\mathsf{Nat}}\ \mathfrak{b})\!:\!Sig_{\mathsf{SetCE}}$, where

$$\begin{aligned}\mathfrak{a} \stackrel{def}{=} (&\mathsf{empty} = \mathsf{nil},\\ &\mathsf{add} = \textit{cons-uniquesorted} \stackrel{def}{=} \lambda x\!:\!\mathsf{Nat}.\lambda l\!:\!\mathsf{List}_{\mathsf{Nat}}\ .\\ &\qquad\qquad \textit{return } l' \textit{ that is } l \textit{ with } x \textit{ uniquely inserted before first } y > x,\\ &\mathsf{remove} = \textit{del-first} \stackrel{def}{=} \lambda x\!:\!\mathsf{Nat}.\lambda l\!:\!\mathsf{List}_{\mathsf{Nat}}\ .\\ &\qquad\qquad \textit{return } l' \textit{ that is } l \textit{ with first occurrence of } x \textit{ removed},\\ &\mathsf{in} = \textit{in} \stackrel{def}{=} \lambda x\!:\!\mathsf{Nat}.\lambda l\!:\!\mathsf{List}_{\mathsf{Nat}}\ .\ \textit{return } \mathsf{true} \textit{ if } x \textit{ occurs in } l,\ \mathsf{false} \textit{ otherwise},\\ &\mathsf{crossover} \stackrel{def}{=} \lambda f\!:\!(\mathsf{Nat}\!\to\!\mathsf{List}_{\mathsf{Nat}}\!\to\!\mathsf{List}_{\mathsf{Nat}}).\lambda n\!:\!\mathsf{Nat}\ .\\ &\qquad\qquad \mathsf{in}(n)(f(n)(1 :: 0 :: \mathsf{nil})))\ )\end{aligned}$$

Here we use the infix symbol :: to denote cons (*p.* 51). Furthermore,

$$\begin{aligned}\mathfrak{b} \stackrel{def}{=} (&\mathsf{empty} = \mathsf{nil},\\ &\mathsf{add} = \textit{cons-uniquesorted},\\ &\mathsf{remove} = \textit{del-all} \stackrel{def}{=} \lambda x\!:\!\mathsf{Nat}.\lambda l\!:\!\mathsf{List}_{\mathsf{Nat}}\ .\\ &\qquad\qquad \textit{return } l' \textit{ that is } l \textit{ with all occurrences of } x \textit{ removed},\\ &\mathsf{in} = \textit{in},\\ &\mathsf{crossover} \stackrel{def}{=} \lambda f\!:\!(\mathsf{Nat}\!\to\!\mathsf{List}_{\mathsf{Nat}}\!\to\!\mathsf{List}_{\mathsf{Nat}}).\lambda n\!:\!\mathsf{Nat}\ .\\ &\qquad\qquad \mathsf{in}(n)(f(n)(1 :: 1 :: 0 :: \mathsf{nil})))\ )\end{aligned}$$

We will now relate to the parametric minimal model of (Hasegawa, 1991), *cf.* Sect. 3.4.2. In this model, all elements of the interpretation of $\mathsf{List}_{\mathsf{Nat}}$ and Bool are in a one-to-one correspondence with the closed normal forms of the types, and moreover, the ω-rule holds. See Sect. A.6 in Appendix A for the ω-rule.

First, there cannot exist a relation $\mathfrak{R}$ interpreting $R\!\subset\!\mathsf{List}_{\mathsf{Nat}}\!\times\!\mathsf{List}_{\mathsf{Nat}}$ satisfying $\mathfrak{a}\,\mathfrak{T}_{\mathsf{SetCE}}[R]\,\mathfrak{b}$. If there were such an $\mathfrak{R}$, then in order for $\mathfrak{R}$ to simultaneously satisfy

$$\begin{gathered}\mathfrak{a}.\mathsf{empty}\quad R\quad \mathfrak{b}.\mathsf{empty}\\ \mathfrak{a}.\mathsf{add}\ (\mathsf{eq}_{\mathsf{Nat}}\!\to\! R\!\to\! R)\ \mathfrak{b}.\mathsf{add}\\ \mathfrak{a}.\mathsf{remove}\ (\mathsf{eq}_{\mathsf{Nat}}\!\to\! R\!\to\! R)\ \mathfrak{b}.\mathsf{remove}\\ \mathfrak{a}.\mathsf{in}\ (\mathsf{eq}_{\mathsf{Nat}}\!\to\! R\!\to\!\mathsf{eq}_{\mathsf{Bool}})\ \mathfrak{b}.\mathsf{in}\end{gathered}$$

$\mathfrak{R}$ would have to be as follows. First, $\mathfrak{R}$ must be non-empty and relate at least (the interpretations) of $(\mathfrak{a}.\mathsf{empty}, \mathfrak{b}.\mathsf{empty})$. Then $\mathfrak{a}.\mathsf{add}\ (\mathsf{eq}_{\mathsf{Nat}} \to R \to R)\ \mathfrak{b}.\mathsf{add}$

demands that $\Re$ relates any pair of lists generated by corresponding applications of empty and add on equal elements of Nat. Thus $\Re$ relates at least all pairs of equal sorted ascending lists where each element occurs at most once. It is then easy to see that $\Re$ cannot relate any other lists, because this will eventually violate $\mathfrak{a}.\mathsf{in}\ (\mathsf{eq}_{\mathsf{Nat}} \to R \to \mathsf{eq}_{\mathsf{Bool}})\ \mathfrak{b}.\mathsf{in}$. The crux is now that this $\Re$ does not satisfy

$$\mathfrak{a}.\mathsf{crossover} \quad (\mathsf{eq}_{\mathsf{Nat}} \to R \to R) \to \mathsf{eq}_{\mathsf{Nat}} \to \mathsf{eq}_{\mathsf{Bool}} \quad \mathfrak{b}.\mathsf{crossover}$$

because although $\Re$ satisfies *del-all* $(\mathsf{eq}_{\mathsf{Nat}} \to R \to R)$ *del-first*, and $1 = 1$, we have

$$\mathfrak{a}.\mathsf{crossover}(\textit{del-all})(1) \neq_{\mathsf{Bool}} \mathfrak{b}.\mathsf{crossover}(\textit{del-first})(1)$$

since

$$\mathfrak{a}.\mathsf{crossover}(\textit{del-all})(1) = \mathfrak{a}.\mathsf{in}(1)(0 :: \mathsf{nil}) = \mathsf{false}$$
$$\mathfrak{b}.\mathsf{crossover}(\textit{del-first})(1) = \mathfrak{b}.\mathsf{in}(1)(1 :: 0 :: \mathsf{nil}) = \mathsf{true}$$

On the other hand, the interpretations of $\mathfrak{a}$ and $\mathfrak{b}$ satisfy

$$\forall f : \forall X.(\mathfrak{T}[X, \boldsymbol{Z}] \to \mathsf{Nat})\ .\ (f\,\mathsf{List}_{\mathsf{Nat}}\ \mathfrak{a}) = (f\,\mathsf{List}_{\mathsf{Nat}}\ \mathfrak{b})$$

This follows from the ω-rule and Lemma 5.12 below. Thus we have observational equivalence. We get back to this later. In the mean time, we rely on some intuition. It is reasonable that $(\mathsf{pack}\,\mathsf{List}_{\mathsf{Nat}}\ \mathfrak{a})$ and $(\mathsf{pack}\,\mathsf{List}_{\mathsf{Nat}}\ \mathfrak{b})$ are observationally equivalent w.r.t. $\mathit{Obs} = \{\mathsf{Nat}\}$, since actual computations using either of the data types must adhere to ***Abs-Bar***. In particular, the terms

$$\mathfrak{a}.\mathsf{crossover}(\textit{del-all})(1) \qquad\qquad \mathfrak{b}.\mathsf{crossover}(\textit{del-first})(1)$$

do not arise from any one virtual observable computation. On the other hand, the terms $\mathfrak{a}.\mathsf{crossover}(\textit{del-first})(1)$ and $\mathfrak{b}.\mathsf{crossover}(\textit{del-all})(1)$ do arise from a single virtual computation, namely $\Lambda X.\lambda\mathfrak{x} : \mathfrak{T}_{\mathsf{SetCE}}[X]\ .\ \mathfrak{x}.\mathsf{crossover}(\mathfrak{x}.\mathsf{remove})(1)$.

Note that this intuitive argument for observational equivalence is in terms of computations that are somehow closed. It does not follow that we can induce this argument to all $f : \forall X.(\mathfrak{T}_{\mathsf{SetCE}}[X] \to \mathsf{Nat})$ in the logic. We therefore need to relate to models with closedness properties as above. We say more about this later.

To show non-composability and non-transitivity of existence, in addition to $(\mathsf{pack}\,\mathsf{List}_{\mathsf{Nat}}\ \mathfrak{a})$ and $(\mathsf{pack}\,\mathsf{List}_{\mathsf{Nat}}\ \mathfrak{b})$ above, consider $(\mathsf{pack}\,\mathsf{List}_{\mathsf{Nat}}\ \mathfrak{d}) : \mathit{Sig}_{\mathsf{SetCE}}$, where

$$\begin{aligned}
\mathfrak{d} \stackrel{\mathit{def}}{=} (&\mathsf{empty} = \mathsf{nil},\\
&\mathsf{add} = \mathit{cons},\\
&\mathsf{remove} = \textit{del-all}\\
&\mathsf{in} = \mathit{in},\\
&\mathsf{crossover} \stackrel{\mathit{def}}{=} \lambda f : (\mathsf{Nat} \to \mathsf{List}_{\mathsf{Nat}} \to \mathsf{List}_{\mathsf{Nat}}).\lambda n : \mathsf{Nat}\ .\\
&\qquad\qquad \mathsf{in}(n)(f(n)(1 :: 1 :: 0 :: \mathsf{nil})))\ )
\end{aligned}$$

Now, continuing to relate to the parametric minimal model, let $\Re_1$ relate lists a and b if and only if a is sorted ascending and b has the the same content as a, *i.e.,* all and only those items that occur in a, occur, possibly several times, in b. Then, interpreting R as $\Re_1$, we have that $\mathfrak{a}\ \mathfrak{T}_{\mathsf{SetCE}}[R]\ \mathfrak{d}$ holds in the model. Notice that *del-first* $(\mathsf{eq}_{\mathsf{Nat}} \rightarrow R \rightarrow R)$ *del-all* holds, but that in contrast to the situation above, *del-all* $(\mathsf{eq}_{\mathsf{Nat}} \rightarrow R \rightarrow R)$ *del-first* does not hold. Of course, $f\ (\mathsf{eq}_{\mathsf{Nat}} \rightarrow R \rightarrow R)\ g$ holds only for interpretations of $f, g : \mathsf{Nat} \rightarrow \mathsf{List}_{\mathsf{Nat}} \rightarrow \mathsf{List}_{\mathsf{Nat}}$ that maintain the contents invariant on sorted ascending list on the one hand and general lists on the other, as expressed in $\Re_1$. For all such f and g, we have $\mathfrak{a}.\mathsf{crossover}(f)(n) =_{\mathsf{Bool}} \mathfrak{d}.\mathsf{crossover}(g)(n)$, and therefore this time, $\mathfrak{a}.\mathsf{crossover}\ ((R \rightarrow\ R) \rightarrow \mathsf{Nat} \rightarrow \mathsf{Bool})\ \mathfrak{d}.\mathsf{crossover}$ holds.

Analogously, let $\Re_2 \stackrel{def}{=} \Re_1^{-1}$. Then, by interpreting R as $\Re_2$, we also have that $\mathfrak{d}\ \mathfrak{T}_{\mathsf{SetCE}}[R]\ \mathfrak{b}$ holds in the model.

In the parametric minimal model, we thus have simulation relations between (the interpretations of) $(\mathsf{pack}\ \mathsf{List}_{\mathsf{Nat}}\mathfrak{a})$ and $(\mathsf{pack}\ \mathsf{List}_{\mathsf{Nat}}\mathfrak{d})$, namely $\Re_1$, and also between $(\mathsf{pack}\ \mathsf{List}_{\mathsf{Nat}}\mathfrak{d})$ and $(\mathsf{pack}\ \mathsf{List}_{\mathsf{Nat}}\mathfrak{b})$, namely $\Re_2$, but as we demonstrated above, there is no simulation relation between $(\mathsf{pack}\ \mathsf{List}_{\mathsf{Nat}}\mathfrak{a})$ and $(\mathsf{pack}\ \mathsf{List}_{\mathsf{Nat}}\mathfrak{b})$ (in particular not the composition of $\Re_1$ and $\Re_2$).

We could also have given the above argument for non-transitivity in purely syntactic terms, but it is saves effort to give it semantically. ○

5.1.2 To Observe *Abs-Bar*

We here take the view that the current notion of simulation relation is unduly demanding, and fails to observe closely enough the abstraction barrier provided by existential types. Consider the higher-order signature

$$\exists X.(f : (X \rightarrow X) \rightarrow \mathsf{Nat}, g : X \rightarrow X)$$

A requirement for an $R \subset A \times B$ to act as a simulation relation and be respected in the standard sense by two implementations $\mathfrak{a}$ and $\mathfrak{b}$, is that

$$\forall \delta : A \rightarrow A, \forall \gamma : B \rightarrow B\ .\ \delta(R \rightarrow R)\gamma\ \Rightarrow\ \mathfrak{a}.f(\delta) =_{\mathsf{Nat}} \mathfrak{b}.f(\gamma)$$

But according to *Abs-Bar* (*p.* 54), $\mathfrak{a}.f$ and $\mathfrak{b}.f$ can only be applied to arguments expressible by the supplied operations in $\mathfrak{a}$ and $\mathfrak{b}$, unless the supplied operations themselves apply operations to non-expressible arguments. The proof obligation should take this into account. Essentially, one should not have to consider the behaviour of $\mathfrak{a}.f$ and $\mathfrak{b}.f$ on arbitrary operators $\delta : A \rightarrow A$ and $\gamma : B \rightarrow B$ as long as they fulfil the requirements for operators defined in terms of $\mathfrak{a}.g$ and $\mathfrak{b}.g$.

5.2 Abstraction Barrier-Observing Simulation Relations *I*

In this section we present the first of two related solutions to the problems outlined in the previous section. The first solution is sound w.r.t. syntactic models, while the solution in Sect. 5.3 is sound in the non-syntactic parametric PER-model. The idea is to devise an alternative notion of simulation relation.

5.2.1 Abstraction Barrier-Observing Relations *I*

As before $\mathfrak{T}[X]$ denotes the body of an abstract data type $\exists X.\mathfrak{T}[X]$, now possibly with higher-order profiles according to HADT_{Obs}.

Definition 5.8 (*abo*-Relation) *Relative to* $\mathfrak{T}[X]$, *for* k-*ary* $\boldsymbol{Y}$, A, B, $R \subset A \times B$, $\mathfrak{a} : \mathfrak{T}[A]$, $\mathfrak{b} : \mathfrak{T}[B]$. *Define the abo*-relation $U[\mathbf{eq}_{\boldsymbol{Y}}, R]^{\mathsf{abo}} \subset U[\boldsymbol{Y}, A] \times U[\boldsymbol{Y}, B]$, *for the list* $\mathsf{abo} = A, B, \mathfrak{a}, \mathfrak{b}$, *inductively on* $U[\boldsymbol{Y}, X]$ *by*

$$
\begin{array}{lll}
U = X & : & U[\mathbf{eq}_{\boldsymbol{Y}}, R]^{\mathsf{abo}} \overset{def}{=} R \\
U = Y_i & : & U[\mathbf{eq}_{\boldsymbol{Y}}, R]^{\mathsf{abo}} \overset{def}{=} \rho_i \\
U = \forall Y_{k+1}.U'[\boldsymbol{Y}, Y_{k+1}, X] & : & U[\mathbf{eq}_{\boldsymbol{Y}}, R]^{\mathsf{abo}} \overset{def}{=} \\
\multicolumn{3}{r}{(g : \forall Y_{k+1}.U'[\boldsymbol{Y}, Y_{k+1}, A],\ h : \forall Y_{k+1}.U'[\boldsymbol{Y}, Y_{k+1}, B])\ .} \\
\multicolumn{3}{r}{(\forall Y_{k+1}\ .\ gY_{k+1}\ (U'[\mathbf{eq}_{\boldsymbol{Y}}, \mathsf{eq}_{Y_{k+1}}, R]^{\mathsf{abo}})\ hY_{k+1})} \\
U = U' \to U'' & : & U[\mathbf{eq}_{\boldsymbol{Y}}, R]^{\mathsf{abo}} \overset{def}{=} \\
\multicolumn{3}{l}{\quad (g : U'[\boldsymbol{Y}, A] \to U''[\boldsymbol{Y}, A],\ h : U'[\boldsymbol{Y}, B] \to U''[\boldsymbol{Y}, B])\ .} \\
\multicolumn{3}{l}{\quad\quad (\forall x : U'[\boldsymbol{Y}, A], \forall y : U'[\boldsymbol{Y}, B]\ .} \\
\multicolumn{3}{l}{\quad (x\ U'[\mathbf{eq}_{\boldsymbol{Y}}, R]^{\mathsf{abo}}\ y \quad \wedge \quad \mathsf{Dfnbl}^{\mathsf{abo}}_{U'[\boldsymbol{Y},X]}(x, y)) \quad \Rightarrow \quad (gx)\ U''[\mathbf{eq}_{\boldsymbol{Y}}, R]^{\mathsf{abo}}\ (hy))}
\end{array}
$$

where,

$$\mathsf{Dfnbl}^{\mathsf{abo}}_{U'[\boldsymbol{Y},X]}(x, y) \overset{def}{=} \exists f_{U'} : \forall X.(\mathfrak{T}[X] \to U'[\boldsymbol{Y}, X])\ .\ (f_{U'} A\, \mathfrak{a}) = x\ \wedge\ (f_{U'} B\, \mathfrak{b}) = y$$

To avoid clutter, parameters $\boldsymbol{Z}$ *of* $\mathfrak{T}[X, \boldsymbol{Z}]$ *were omitted. Of course,* ${\mathbf{eq}_{\boldsymbol{Z}}}^{\mathsf{abo}} \overset{def}{=} \mathbf{eq}_{\boldsymbol{Z}}$. *Parameters* $\boldsymbol{Z}$ *are subsumed by* $\boldsymbol{Y}$. *However, the latter are intended as type instances introduced at universal type. It is helpful to keep the distinction.*

We usually omit the type subscript on the $\mathsf{Dfnbl}^{\mathsf{abo}}$ clause. The essence of Def. 5.8 is the weakened arrow-type relation via the $\mathsf{Dfnbl}^{\mathsf{abo}}$ clause; an extension of the relation exhibited for proving Theorem 4.17. This clause asserts definability for arguments as stated by *Abs-Bar*. The *abo*-relation will be used relative to a single virtual computation, and therefore the Dfnbl clause incorporates the uniformity aspect of *Abs-Bar* as an essential ingredient.

Example 5.7 (continued) Above, we demonstrated that the relation $\mathfrak{R}$ relating exactly all sorted ascending lists with non-repeated items does not satisfy

$$\mathfrak{a}.\mathsf{crossover} \quad (\mathsf{eq}_{\mathsf{Nat}} \to R \to R) \to \mathsf{eq}_{\mathsf{Nat}} \to \mathsf{eq}_{\mathsf{Bool}} \quad \mathfrak{b}.\mathsf{crossover}$$

in the parametric minimal model of (Hasegawa, 1991). However, $\mathfrak{R}$ does satisfy

$$\mathfrak{a}.\mathsf{crossover} \ ((\mathsf{eq}_{\mathsf{Nat}} \to R \to R) \to \mathsf{eq}_{\mathsf{Nat}} \to \mathsf{eq}_{\mathsf{Bool}})^{\mathsf{abo}} \ \mathfrak{b}.\mathsf{crossover}$$

via the ω-rule. Intuitively, this is because only $\gamma, \delta : \mathsf{Nat} \to \mathsf{List}_{\mathsf{Nat}} \to \mathsf{List}_{\mathsf{Nat}}$ for which we have $\mathsf{Dfnbl}^{\mathsf{abo}}_{\mathsf{Nat} \to X \to X}(\gamma, \delta)$, *i.e.*,

$$\exists f : \forall X. \mathfrak{T}_{\mathsf{SetCE}}[X] \to (\mathsf{Nat} \to X \to X) \ . \ (f\ \mathsf{List}_{\mathsf{Nat}}\ \mathfrak{a}) = \gamma \ \wedge \ (f\ \mathsf{List}_{\mathsf{Nat}}\ \mathfrak{b}) = \delta$$

need be considered. In particular, this should exclude the pair (*del-all*, *del-first*), reflecting exactly the uniformity in actual computations captured by ***Abs-Bar***, a uniformity which is inevitable, since the only way of accessing data types is through virtual computations. ○

Seemingly, Def. 5.8 ignores the fact that data type operations themselves may cause non-definable arguments to be applied; a prime example of the application of non-definable arguments in this way is in fact Example 5.7, where $\mathfrak{b}.\mathsf{crossover}$ causes $\mathfrak{b}.\mathsf{remove}$ to be applied to the non-definable $1 :: 1 :: 0 :: \mathsf{nil}$. However, in the context of data types, the operations that cause non-definable arguments to be applied are themselves subjected to the *abo* arrow-type relation according to Def. 5.8. This means that the application to non-definable arguments are taken care of at that level. For example, the application of $\mathfrak{b}.\mathsf{remove}$ to $1 :: 1 :: 0 :: \mathsf{nil}$ is treated while considering $\mathfrak{b}.\mathsf{crossover}$ with $\mathfrak{a}.\mathsf{crossover}$, and it is adequate to consider the *abo*-relation for the remove operations rather than exceptionally reverting to the standard arrow type relation.

The treatment of universal types in Def. 5.8 demands a comment. According to ***Abs-Bar***, actual computations arising from a virtual computation can only have applications of polymorphic terms of the following two kinds. First, if the instantiating type in the virtual computation does not contain the virtual data representation, then the instantiating type will appear identically in actual computations. Secondly, if the instantiating type in the virtual computation does contain the virtual data representation, then the instantiating type in each actual computation will differ, but then only in harmony with the actual data representations. The first case is what is expressed for universal types in Def. 5.8. To capture type application of the second kind, we must somehow quantify over all types involving the virtual data representation. In the current type theory, we can only do this by infinite conjunction, something the logic does not have. What we can do, though, is to consider a given virtual computation and make a finite conjunction over all types involving the virtual data representation that actually occur in the given virtual computation. This is what we do in Def. B.3 in Sect. B.2 in Appendix B. This however only makes sense in the context of a given virtual computation, and gives no meaning in the general definition in Def. 5.8. At the top-level discussion, we do not really need to go into universal types, since abstract types here adhere to HADT_{Obs}. The exception to this is of course types in *Obs* that are universal types, *e.g.,* Bool and Nat. These are nevertheless inductive, and for such universal types, the treatment in Def. 5.8 suffices; *e.g.,* for Lemma 5.10 below. For further remarks, we refer to Sect. B.2 in Appendix B.

We now define with a slight abuse of notation:

Definition 5.9 *For A, B and $R \subset A \times B$,*

$$\mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}} \stackrel{def}{=} (\mathfrak{a}:\mathfrak{T}[A, \boldsymbol{Z}], \mathfrak{b}:\mathfrak{T}[B, \boldsymbol{Z}]) \ . \ (\wedge_{1\leq i\leq k} \ \mathfrak{a}.g_i \ (T_i[R, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}}) \ \mathfrak{b}.g_i)$$

We want the *abo*-relation of Def. 5.8 to retain the property of being the equality over types in *Obs*. This is easily derivable in the logic, since *Obs* contains only closed inductive types apart from the parameters $\boldsymbol{Z}$.

Lemma 5.10 *We can derive the following in the logic.*

$$\forall g, h{:}D \ . \ g =_D h \ \Leftrightarrow \ g(D^{\mathsf{abo}})h, \qquad \textit{for } D \in Obs$$

Proof: We illustrate with the inductive type Nat. So consider $g(\mathsf{Nat}^{\mathsf{abo}})h$, *i.e.,*

$$\begin{aligned}&(\forall Y.\forall y, y'{:}Y.\forall s, s'{:}Y \to Y \ . \\ &\qquad y =_Y y' \ \wedge \ \mathsf{Dfnbl}^{\mathsf{abo}'}(y, y') \ \wedge \ s(\mathsf{eq}_Y \to \mathsf{eq}_Y)^{\mathsf{abo}'}s' \ \wedge \ \mathsf{Dfnbl}^{\mathsf{abo}'}(s, s') \\ &\qquad\qquad \Rightarrow \ (gYys) =_Y (hYy's'))\end{aligned}$$

The clause $\mathsf{Dfnbl}^{\mathsf{abo}'}(y, y')$ says

$$\exists f{:}\forall X.(\mathfrak{T}[X] \to Y) \ . \ (f A \,\mathfrak{a}) = y \ \wedge \ (f B \,\mathfrak{b}) = y'$$

Since we have $y =_Y y'$, this is derivable by exhibiting $\Lambda X.\lambda\mathfrak{x}{:}\mathfrak{T}[X].y$. Similarly, we derive $\mathsf{Dfnbl}^{\mathsf{abo}'}(s, s')$ by exhibiting $\Lambda X.\lambda\mathfrak{x} : \mathfrak{T}[X].s : \forall X.(\mathfrak{T}[X] \to (Y \to Y))$. This suffices because $s(\mathsf{eq}_Y \to \mathsf{eq}_Y)^{\mathsf{abo}'}s'$ is equivalent to $s(\mathsf{eq}_Y \to \mathsf{eq}_Y)s'$, which by the Identity Extension Lemma gives $s =_{Y\to Y} s'$. Thus, these $\mathsf{Dfnbl}^{\mathsf{abo}'}$ clauses are vacuous. This means that the definition of $g(\mathsf{Nat}^{\mathsf{abo}})h$ is equivalent to

$$\forall Y.\forall y, y'{:}Y.\forall s, s'{:}Y \to Y \ . \ y =_Y y' \ \wedge \ s(\mathsf{eq}_Y \to \mathsf{eq}_Y)s' \ \Rightarrow \ (gYys) =_Y (hYy's')$$

or $\forall Y \ . \ gY(\mathsf{eq}_Y \to (\mathsf{eq}_Y \to \mathsf{eq}_Y) \to \mathsf{eq}_Y)hY$, *i.e.,* $\forall Y \ . \ gY = hY$. By the congruence axiom schema, we get $\Lambda Y.gY = \Lambda Y.hY$, which by η-equality yields $g =_{\mathsf{Nat}} h$. $\Box$

Definition 5.11 (*abo*-Simulation Relation $\mathsf{SimRelA}$) *Relatedness by* abstraction barrier-observing (*abo*) simulation relation *w.r.t.* $\mathfrak{T}[X, \boldsymbol{Z}]$ *is expressed by*

$$\begin{array}{l} \mathsf{SimRelA}_{\mathfrak{T}[X,\boldsymbol{\rho}]} \stackrel{def}{=} (u\!:\!\exists X.\mathfrak{T}[X,\boldsymbol{U}], v\!:\!\exists X.\mathfrak{T}[X,\boldsymbol{V}])\ . \\ \qquad (\exists A, B.\exists \mathfrak{a}\!:\!\mathfrak{T}[A,\boldsymbol{U}], \mathfrak{b}\!:\!\mathfrak{T}[B,\boldsymbol{V}]\ .\ u = (\mathsf{pack} A \mathfrak{a})\ \wedge\ v = (\mathsf{pack} B \mathfrak{b}) \\ \qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \exists R \subset A \times B\ .\ \mathfrak{a}(\mathfrak{T}[R,\boldsymbol{\rho}]^{\mathsf{abo}})\mathfrak{b} \end{array}$$

where $\mathsf{abo} = A, B, \mathfrak{a}, \mathfrak{b}$*, and where* $\boldsymbol{Z}$ *are the free type variables in* $\mathfrak{T}[X, \boldsymbol{Z}]$ *other than* X*, and* $\boldsymbol{\rho} \subset \boldsymbol{U} \times \boldsymbol{V}$ *is a vector of relations of the same length.*

The subscript $\mathfrak{T}[X, \boldsymbol{\rho}]$ to $\mathsf{SimRelA}_{\mathfrak{T}[X,\boldsymbol{\rho}]}$ might occasionally be omitted.

As before, two data types are intuitively related by a simulation relation according to Def. 5.11, if there exists a relation on their respective data representations that is preserved by their respective operations. However, now the action on types within a data type is defined by the *abo*-relations of Def. 5.8 incorporating the weakened arrow-type relation which deploys abstract-type definability.

With *abo*-simulation relations in place one would think that we should be able to re-establish versions of Theorem 4.18 and Theorem 4.22 derivable for $\mathfrak{T}[X]$ of any order. But there is a catch. Since we do not alter the parametricity axiom schema, we can no longer rely directly on parametricity as in Theorem 4.18, when deriving observational equivalence from the existence of a simulation relation. This is because we lose the power of relational parametricity when considering data-type relations in relations over universal types.

One might envision an extended parametricity axiom schema dealing with data-type relations as well, but such a schema would not be sound. One way to see this is to observe that with such a schema, normal simulation relations would coincide with *abo*-simulation relations, and then Example 5.7 would demonstrate inconsistency. Our approach to this problem is to assert only the necessary instances of special parametricity incorporating *abo*-simulation relations. This is the next topic.

5.2.2 Special Parametricity

We will now extend the logic with a special bounded parametricity schema incorporating *abo*-simulation relations. The boundedness consists of restricting the types of client computations, and also involves considering only client computations that are closed in a suitable sense. This mirrors the fact that `Abs-Bar` captures computational aspects, *i.e.,* how data type operations are applied in computational expressions. Notice that `Abs-Bar` does not say much about variables representing computations in logical expressions.

In this section, soundness is w.r.t. models with inherent term denotability. Later, we will use logic to restrict to term-denotable elements in other models.

Closedness must be relative to the given set of input types *In* so as to allow for variables of input types in observable computations. This is automatically taken care of in the notion of observable computations of Def 4.4, but in shifting attention explicitly to closed computations, we are compelled to specify the collection *In* of input types in observations. We set $In = Obs$ as a sensible choice (Sannella and Tarlecki, 1987), *cf.* the discussion around Examples 2.2 (*p.* 27) and 4.11 (*p.* 83). Recall that *Obs*, and hence *In*, includes parameters $\boldsymbol{Z}$ of the relevant abstract type, *cf.* Definitions 4.6 and 4.12.

—

We write $f\ (\forall X.\mathfrak{T}[X, \mathbf{eq}_{\boldsymbol{Z}}]^{\epsilon} \to U[X, \mathbf{eq}_{\boldsymbol{Z}}]^{\epsilon})\ f$, meaning

$$\forall A, B, R \subset A \times B. \forall \mathfrak{a} : \mathfrak{T}[A, \boldsymbol{Z}], \mathfrak{b} : \mathfrak{T}[B, \boldsymbol{Z}]\ .$$
$$\mathfrak{a}(\mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}})\mathfrak{b} \ \Rightarrow\ (f A\, \mathfrak{a})(U[R, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}})(f B\, \mathfrak{b})$$

where $\mathsf{abo} = A, B, \mathfrak{a}, \mathfrak{b}$.

Lemma 5.12 *For $\mathfrak{T}[X, \boldsymbol{Z}]$ adhering to $\mathtt{HADT}_{Obs}$, for $U[X, \boldsymbol{Z}]$ having no occurrences of universal types other than those in Obs, and whose only free variables are among X and $\boldsymbol{Z}$, for $f : \forall X.(\mathfrak{T}[X] \to U[X])$ whose only free variables are term variables of types in In, we derive*

$$f\ (\forall X.\mathfrak{T}[X, \mathbf{eq}_{\boldsymbol{Z}}]^{\epsilon} \to U[X, \mathbf{eq}_{\boldsymbol{Z}}]^{\epsilon})\ f$$

Proof: See Sect. B.2 in Appendix B. □

By Lemma 5.12, the following axiom schema is sound w.r.t. any model in which the ω-rule holds, or in which interpretations of any $f : \forall X.(\mathfrak{T}[X] \to U[X])$ are denotable by terms whose only free variables are term variables of types in *In*. See Sect. A.6 in Appendix A for the ω-rule.

Definition 5.13 (Special Parametricity (SPPARAM)**)** *For $\mathfrak{T}[X, \boldsymbol{Z}]$ adhering to $\mathtt{HADT}_{Obs}$, for $U[X, \boldsymbol{Z}]$ having no occurrences of universal types other than those in Obs, and whose only free variables are among X and $\boldsymbol{Z}$,*

$$\text{SPPARAM: } \forall f : \forall X.(\mathfrak{T}[X, \boldsymbol{Z}] \to U[X, \boldsymbol{Z}])\ .\ f\ (\forall X.\mathfrak{T}[X, \mathbf{eq}_{\boldsymbol{Z}}]^{\epsilon} \to U[X, \mathbf{eq}_{\boldsymbol{Z}}]^{\epsilon})\ f$$

This axiom schema holds in the closed type and term model and the parametric minimal model due to (Hasegawa, 1991), *cf.* Sect. 3.4.2.

5.2.3 The Results

Using spParam we now get a general version of Theorem 4.17:

Theorem 5.14 *Let $\mathfrak{T}[X]$ adhere to HADT_{Obs}. With* spParam *we derive,*

$$\forall \boldsymbol{Z}.\forall u, v\!:\!\exists X.\mathfrak{T}[X, \boldsymbol{Z}]\ .\ u \ \mathsf{SimRelA}_{\mathfrak{T}[X,\mathbf{eq}_{\boldsymbol{Z}}]}\ v \quad\Leftrightarrow\quad u\ \mathsf{ObsEq}^{Obs}_{\mathfrak{T}[X,\boldsymbol{Z}]}\ v$$

Proof: This follows from Theorem 5.15 below. □

If $\mathfrak{T}[X]$ satisfies FADT_{Obs}, (*p.* 87), *i.e.,* has only first-order profiles, then Theorem 5.14 and Theorem 4.17, imply

$$\forall \boldsymbol{Z}.\forall u, v\!:\!\exists X.\mathfrak{T}[X, \boldsymbol{Z}]\ .\ u \ \mathsf{SimRelA}_{\mathfrak{T}[X,\mathbf{eq}_{\boldsymbol{Z}}]}\ v \quad\Leftrightarrow\quad u\ \mathsf{SimRel}_{\mathfrak{T}[X,\mathbf{eq}_{\boldsymbol{Z}}]}\ v$$

This is of course not surprising. At first order, the data-type relation of Def. 5.8 is exactly the simulation relation Dfnbl we displayed in the proof of Theorem 4.18.

Theorem 5.15 (Tight Correspondence) *Let $\mathfrak{T}[X]$ adhere to HADT_{Obs}. We can derive in the logic with* spParam*,*

$$\begin{aligned}
&\forall A, B.\forall \mathfrak{a}\!:\!\mathfrak{T}[A, \boldsymbol{Z}], \mathfrak{b}\!:\!\mathfrak{T}[B, \boldsymbol{Z}]\ .\\
&\qquad \exists R\!\subset\! A\!\times\! B\ .\ \mathfrak{a}(\mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}})\mathfrak{b}\\
&\qquad\qquad\qquad \Leftrightarrow\quad \textstyle\bigwedge_{D\in Obs} \forall f\!:\!\forall X.(\mathfrak{T}[X, \boldsymbol{Z}]\!\to\! D)\ .\ (f\,A\,\mathfrak{a}) = (f\,B\,\mathfrak{b})
\end{aligned}$$

Proof: $\Rightarrow$: This follows from spParam and Lemma 5.10.

$\Leftarrow$: We have to show that $\exists R\!\subset\! A\!\times\! B\ .\ \mathfrak{a}(\mathfrak{T}[R, \boldsymbol{Z}]^{\mathsf{abo}})\mathfrak{b}$ is derivable. We exhibit $R \stackrel{\mathit{def}}{=} (a : A, b : B).(\mathsf{Dfnbl}^{\mathsf{abo}}(a, b))$. Due to the assumption HADT_{Obs}, it suffices by Def 5.9 to show for every component $g\!:\!U_1\!\to\!\cdots\!\to\! U_n\!\to\! U_c$ in $\mathfrak{T}[X]$, that

$$\begin{aligned}
&\forall x_1\!:\!U_1[A], \ldots, x_n\!:\!U_n[A]\ .\ \forall y_1\!:\!U_1[B], \ldots, y_n\!:\!U_n[B]\ .\\
&\qquad\qquad \textstyle\bigwedge_{1\le i\le n}(x_i\ U_i[R, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}}\ y_i \quad\wedge\quad \mathsf{Dfnbl}^{\mathsf{abo}}_{U_i}(x_i, y_i))\\
&\qquad\qquad\qquad\qquad \Rightarrow\quad (\mathfrak{a}.g\,x_1\cdots x_n)\ U_c[R, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}}\ (\mathfrak{b}.g\,y_1\cdots y_n)
\end{aligned}$$

where $U_c[X, \boldsymbol{Z}]$ is X, some Z_i, or some closed $D \in Obs$. Now $\mathsf{Dfnbl}^{\mathsf{abo}}_{U_i}(x_i, y_i)$ gives $\exists f_{U_i} : \forall X.(\mathfrak{T}[X, \boldsymbol{Z}] \to U_i[X, \boldsymbol{Z}])\ .\ (f_{U_i} A\,\mathfrak{a}) = x_i \ \wedge\ (f_{U_i} B\,\mathfrak{b}) = y_i$. Let $f \stackrel{\mathit{def}}{=} \Lambda X.\lambda\mathfrak{x}\!:\!\mathfrak{T}[X, \boldsymbol{Z}]\ .\ (\mathfrak{x}.g(f_{U_1} X\mathfrak{x})\cdots(f_{U_n} X\mathfrak{x}))$.

Suppose $U_c[X, \boldsymbol{Z}] = D \in Obs$, including some Z_i: It suffices, by Lemma 5.10 in the case D is closed, to show that $\mathfrak{a}.g\,x_1\cdots x_n =_D \mathfrak{b}.g\,y_1\cdots y_n$ is derivable. The assumption gives $(f\,A\,\mathfrak{a}) =_D (f\,B\,\mathfrak{b})$ which gives the desired result.

Suppose $U_c[X, \boldsymbol{Z}] = X$: We must then derive

$$\exists f\!:\!\forall X.(\mathfrak{T}[X, \boldsymbol{Z}]\!\to\! U_c[X, \boldsymbol{Z}])\ .\ (f\,A\,\mathfrak{a}) = (\mathfrak{a}.g\,x_1\cdots x_n)\ \wedge\ (f\,B\,\mathfrak{b}) = (\mathfrak{b}.g\,y_1\cdots y_n)$$

For this we display f above. □

Example 5.7 (continued) Theorem 5.15, or more generally Theorem 5.14, gives with spParam,

$$\forall u, v\colon \exists X.\mathfrak{T}_{\mathsf{SetCE}}[X]\ .\ u\ \mathsf{SimRelA}_{\mathfrak{T}_{\mathsf{SetCE}}[X]}\ v \quad \Leftrightarrow \quad u\ \mathsf{ObsEq}^{\{\mathsf{Nat}\}}_{\mathfrak{T}_{\mathsf{SetCE}}[X]}\ v$$

Thus, $(\mathsf{pack}\,\mathsf{List}_{\mathsf{Nat}}\ \mathfrak{a})$ and $(\mathsf{pack}\,\mathsf{List}_{\mathsf{Nat}}\ \mathfrak{b})$ are related by *abo*-simulation relation exactly when they are observationally equivalent w.r.t. $Obs = \{\mathsf{Nat}\}$.

In the parametric minimal model of (Hasegawa, 1991), we have that the $\mathfrak{R}$ relating exactly all sorted ascending lists with non-repeated items, satisfies

$$\mathfrak{a}\ (\mathfrak{T}_{\mathsf{SetCE}}[R])^{\mathsf{abo}}\ \mathfrak{b}$$

for $\mathsf{abo} = \mathsf{List}_{\mathsf{Nat}}, \mathsf{List}_{\mathsf{Nat}}, \mathfrak{a}, \mathfrak{b}$, and R representing $\mathfrak{R}$. Thus in this model, the denotations of $(\mathsf{pack}\,\mathsf{List}_{\mathsf{Nat}}\ \mathfrak{a})$ and $(\mathsf{pack}\,\mathsf{List}_{\mathsf{Nat}}\ \mathfrak{b})$ satisfy observational equivalence, *i.e.,* $(\mathsf{pack}\,\mathsf{List}_{\mathsf{Nat}}\ \mathfrak{a})\ \mathsf{ObsEq}^{\{\mathsf{Nat}\}}_{\mathfrak{T}_{\mathsf{SetCE}}[X]}\ (\mathsf{pack}\,\mathsf{List}_{\mathsf{Nat}}\ \mathfrak{b})$ holds. ○

Example 5.16 The reasoning of Example 5.7 can also be recast in terms of the polymorphic closed type and term model of (Hasegawa, 1991). Let

$$\begin{aligned} R \stackrel{def}{=} \mathsf{Dfnbl} \stackrel{def}{=} {} & (a\colon \mathsf{List}_{\mathsf{Nat}}, b\colon \mathsf{List}_{\mathsf{Nat}})\ . \\ & (\exists f\colon \forall X.(\mathfrak{T}_{\mathsf{SetCE}}[X] \to X)\ .\ (f(\mathsf{List}_{\mathsf{Nat}})(\mathfrak{a})) = a\ \wedge\ (f(\mathsf{List}_{\mathsf{Nat}})(\mathfrak{b})) = b) \end{aligned}$$

The denotation of R in the polymorphic closed type and term model is exactly the $\mathfrak{R}$ of Example 5.7. We get the analogous results, namely that

$$\mathfrak{a}\ (\mathfrak{T}_{\mathsf{SetCE}}[R])\ \mathfrak{b}$$

does *not* hold, but

$$\mathfrak{a}\ (\mathfrak{T}_{\mathsf{SetCE}}[R])^{\mathsf{abo}}\ \mathfrak{b}$$

does hold for $\mathsf{abo} = \mathsf{List}_{\mathsf{Nat}}, \mathsf{List}_{\mathsf{Nat}}, \mathfrak{a}, \mathfrak{b}$. Since the structure satisfies spParam, and is otherwise sufficiently parametric (*cf.* Sect. 3.4.2), we may use Theorem 5.15 and Theorem 5.14 to derive that the denotations of $(\mathsf{packList}_{\mathsf{Nat}}\mathfrak{a})$ and $(\mathsf{packList}_{\mathsf{Nat}}\mathfrak{b})$ satisfy observational equivalence. ○

Using the abstraction-barrier observing notion of simulation relation together with spParam, we also regain composability and the transitivity of existence of simulation relations. This relates the syntactic level to on-going work on refinement relations on the semantic level, namely the *pre-logical relations* of (Honsell et al., 2000; Honsell and Sannella, 1999), the *lax logical relations* of (Plotkin et al., 2000; Kinoshita and Power, 1999), and the *L-relations* of (Kinoshita et al., 1997). A seemingly interesting difference to these approaches is that we require a looser definability notion, namely definability relative to the virtual ADT-operations, rather than absolute definability.

Theorem 5.17 (Composability of Simulation Relations) *Given* spParam*, for* $\mathfrak{T}[X]$ *adhering to* HADT_{Obs}*, we can derive*

$$\forall A, B, C, R \subset A \times B, S \subset B \times C, \mathfrak{a} : \mathfrak{T}[A, \boldsymbol{Z}], \mathfrak{b} : \mathfrak{T}[B, \boldsymbol{Z}], \mathfrak{c} : \mathfrak{T}[C, \boldsymbol{Z}].$$
$$\mathfrak{a}(\mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}})\mathfrak{b} \;\wedge\; \mathfrak{b}(\mathfrak{T}[S, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}})\mathfrak{c} \;\Rightarrow\; \mathfrak{a}(\mathfrak{T}[S \circ R, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}})\mathfrak{c}$$

Proof: Assuming $\mathfrak{a}(\mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}})\mathfrak{b} \wedge \mathfrak{b}(\mathfrak{T}[S, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}})\mathfrak{c}$, the goal is to derive for every component $g : U_1 \to \cdots \to U_n \to U_c$ in $\mathfrak{T}$,

$$\forall x_1 : U_1[A], \ldots, x_n : U_n[A] \;.\; \forall z_1 : U_1[C], \ldots, z_n : U_n[C] \;.$$
$$\bigwedge_{1 \le i \le n} (x_i \; U_i[S \circ R, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}} \; z_i \;\wedge\; \mathsf{Dfnbl}^{\mathsf{abo}}_{U_i}(x_i, z_i))$$
$$\Rightarrow \quad (\mathfrak{a}.g\, x_1 \cdots x_n) \; U_c[S \circ R, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}} \; (\mathfrak{c}.g\, z_1 \cdots z_n)$$

From $\mathsf{Dfnbl}^{\mathsf{abo}}_{U_i}(x_i, z_i)$, we construct $f \stackrel{\mathit{def}}{=} \Lambda X.\lambda \mathfrak{x} : \mathfrak{T}[X] \;.\; (\mathfrak{x}.g(f_{U_1} X \mathfrak{x}) \cdots (f_{U_n} X \mathfrak{x}))$.

$U_c[X, \boldsymbol{Z}] = D \in \mathit{Obs}$, including some Z_i: By assumption and Theorem 5.14, $(f A\, \mathfrak{a}) = (f B\, \mathfrak{b}) = (f C \mathfrak{c})$, and $\mathfrak{a}.g\, x_1 \cdots x_n = (f A\, \mathfrak{a})$ and $(f C \mathfrak{c}) = \mathfrak{c}.g\, z_1 \cdots z_n$.

$U_c[X, \boldsymbol{Z}] = X$: We must show $\exists b : B \;.\; (\mathfrak{a}.g\, x_1 \cdots x_n) \; R \; b \;\wedge\; b \; S \; (\mathfrak{c}.g\, z_1 \cdots z_n)$. Simply exhibit $f B\, \mathfrak{b} = (\mathfrak{b}.g(f_{U_1} B \mathfrak{b}) \cdots (f_{U_n} B \mathfrak{b}))$ for b. Then in order to show *e.g.,* $(\mathfrak{a}.g\, x_1 \cdots x_n) \; R \; (\mathfrak{b}.g(f_{U_1} B \mathfrak{b}) \cdots (f_{U_n} B \mathfrak{b}))$ it suffices by assumption to show $x_i \; U_i[R, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}} \; (f_{U_i} B \mathfrak{b}) \;\wedge\; \mathsf{Dfnbl}^{\mathsf{abo}}(x_i, (f_{U_i} B \mathfrak{b}))$. But $x_i = (f_{U_i} A \mathfrak{a})$, rendering $\mathsf{Dfnbl}^{\mathsf{abo}}(x_i, (f_{U_i} B \mathfrak{b}))$ trivial, and $(f_{U_i} A \mathfrak{a}) \; U_i[R, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}} \; (f_{U_i} B \mathfrak{b})$ follows by assumption $\mathfrak{a}(\mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}})\mathfrak{b}$ and spParam. □

For SimRelA, we have the close connection with components *i.e.,* we get a version of Theorem 4.20 (*p.* 89) for data types with operations of any order. Thus, the notion of *abo*-simulation relation solves the problem of Theorem 5.5.

Theorem 5.18 *Suppose* $\mathfrak{T}[X]$ *adheres to* HADT_{Obs}*. With* spParam *we derive*

$$\forall A, B. \forall \mathfrak{a} : \mathfrak{T}[A, \boldsymbol{Z}], \mathfrak{b} : T[B, \boldsymbol{Z}]) \;.$$
$$(\mathsf{pack} A \mathfrak{a}) \; \mathsf{SimRelA}_{\mathfrak{T}[X, \mathbf{eq}_{\boldsymbol{Z}}]} \; (\mathsf{pack} B \mathfrak{b}) \quad \Leftrightarrow \quad \exists R \subset A \times B \;.\; \mathfrak{a} \; \mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}} \; \mathfrak{b}$$

Proof: For the non-obvious direction, $(\mathsf{pack} A \mathfrak{a}) \; \mathsf{SimRelA}_{\mathfrak{T}[X, \mathbf{eq}_{\boldsymbol{Z}}]} \; (\mathsf{pack} B \mathfrak{b})$ gives by Theorem 5.14, $(\mathsf{pack} A \mathfrak{a}) \; \mathsf{ObsEq}^{\mathit{Obs}}_{\mathfrak{T}[X, \boldsymbol{Z}]} \; (\mathsf{pack} B \mathfrak{b})$. Then Theorem 4.5 and Theorem 5.15 give the result. □

At present, it is not known whether or not, the parametric PER-model of (Bainbridge et al., 1990) satisfies spParam, even for $U = \mathsf{Bool}$. But since we can show spParam in syntactic models, we have at least re-established the correspondence between simulation relations and observational equivalence, as well as the composability of simulation relations, w.r.t. these.

In this thesis we use semantic structures primarily to show the soundness of new axioms, and for this in isolation it does not matter what kind of model one relates to. In future elaborations, however, we want to investigate the semantic meaning of our type-theoretical notions. This is most interestingly done in non-syntactic models. We therefore wish to relate our discussion also to such models.

5.3 Abstraction Barrier-Observing Simulation Relations *II*

In this section we seek validation w.r.t. the parametric PER-model. Recall that we are relying on appropriately closed computations. Now, instead of restricting the model class accordingly, we use linguistic means to focus on appropriately closed computations.

5.3.1 Closedness in the Logic

We want closedness to be qualified by the set of input types *In* so as to allow for variables of input types in observable computations. Recall that this is automatically taken care of in the notion of general observable computations of Def 4.4, but as mentioned in Sect. 5.2.2 we must now explicitly specify *In*. According to our previous discussion in Sect. 4.5, we set $In = Obs$.

We thus want a predicate Closed_{In}. Unfortunately, the possibility of defining Closed_{In} to the desired effect in the existing logic is very unlikely. Nevertheless, we can circumvent this problem by way of a language extension, *i.e.,* by introducing Closed_{In} as a family of new predicates together with a predefined semantics. We are primarily interested in expressing the closedness of terms, rather than types. We thus extend the language as follows.

Definition 5.19 (Closed) *The logical language is extended with families of basic predicates* $\mathsf{Closed}^U_{\urcorner}(u)$ *ranging over terms* $u\!:\!U$*, relative to a given set of types* $\urcorner$*. This syntax is given the following predefined semantics. For any type* $\Gamma \rhd U$*, term* $\Gamma \rhd u\!:\!U$*, and environment* γ *on* Γ*,*

$$\models_{\Gamma,\gamma} \mathsf{Closed}^U_{\urcorner}(u) \overset{def}{\Leftrightarrow} \text{exists } \widehat{\Gamma}, \text{ such that } \widehat{\Gamma} \rhd U,\ \widehat{\Gamma} \rhd \widehat{u}\!:\!U, \text{ and } \widehat{\gamma} \text{ on } \widehat{\Gamma}, \text{ such that } \widehat{\Gamma} \text{ contains types only from } \urcorner, \text{ and } [\![\widehat{\Gamma} \rhd \widehat{u}\!:\!U]\!]_{\widehat{\gamma}} = [\![\Gamma \rhd u\!:\!U]\!]_{\gamma}$$

We say that $\widehat{u}$ *and* U *here are* closed qualified by $\urcorner$. *We use the same notational conventions for* $\urcorner$ *as we do for type contexts.*

Suppose $\urcorner = In$. For $\Gamma \rhd u\!:\!U$ and any environment γ on Γ, we have according to Def. 5.19, that $\mathsf{Closed}^U_\urcorner(u)$ is true under γ in a given model, exactly when there exists $\widehat{u}\!:\!U$, where any free type variables in U are parameters $\boldsymbol{Z}$ in In (remember we set $In = Obs$, and Obs includes parameters $\boldsymbol{Z}$ of abstract types, but if there are no parameters in In, then U is closed), and where free term variables in $\widehat{u}$ are of types in In, such that the denotation of $u\!:\!U$ under γ is the denotation of $\widehat{u}\!:\!U$ for some valuation of the free variables. In other words, the denotation of $u:U$ under γ is denotable by a closed term qualified by In.

Introducing new names with predefined semantics requires that we check soundness *i.e.,* we must reaffirm the logical axioms expressed in the derivation rules (Sect. A.2 in Appendix A) of the calculus.

Lemma 5.20 *The logic extended with the* Closed *predicates remains sound.*

Proof: The only potential problems arise with universal instantiation ($\forall$-*elim*), and existential generalisation ($\exists$-*intro*). For terms, these are the rules

$$\textit{te-}\forall\textit{-elim}: \frac{\Phi \vdash_\Gamma \forall x\!:\!T.\phi[x] \quad \Gamma \rhd v\!:\!T}{\Phi \vdash_\Gamma \phi[v]} \qquad\qquad \textit{te-}\exists\textit{-intro}: \frac{\Phi \vdash_\Gamma \phi[v] \quad \Gamma \rhd v\!:\!T}{\Phi \vdash_\Gamma \exists x\!:\!T.\phi[x]}$$

where ϕ (and formulae in Φ) may have occurrences of Closed. To justify these rules, it suffices to establish the substitution property. For all formulae ϕ,

$$\models_{\Gamma,x:T,\gamma[x\mapsto[\![\Gamma\rhd v:T]\!]_\gamma]} \phi[x] \quad\Leftrightarrow\quad \models_{\Gamma,\gamma} \phi[v]$$

and thus it remains for us to show

$$\models_{\Gamma,x:T,\gamma[x\mapsto[\![\Gamma\rhd v:T]\!]_\gamma]} \mathsf{Closed}^U_\urcorner(u[x]) \quad\Leftrightarrow\quad \models_{\Gamma,\gamma} \mathsf{Closed}^U_\urcorner(u[v])$$

This is reasonably easy. The left-hand side gives $\widehat{\Gamma}$, $\widehat{u}$, and $\widehat{\gamma}$, such that

$$[\![\widehat{\Gamma} \rhd \widehat{u}\!:\!U]\!]_{\widehat{\gamma}} = [\![\Gamma, x\!:\!T \rhd u[x]\!:\!U]\!]_{\gamma[x\mapsto[\![\Gamma\rhd v:T]\!]_\gamma]}$$

Since $[\![\Gamma, x : T \rhd u[x] : U]\!]_{\gamma[x\mapsto[\![\Gamma\rhd v:V]\!]_\gamma]} = [\![\Gamma \rhd u[v] : U]\!]_\gamma$, and $\urcorner$ remains the same over the implication, we may use $\widehat{\Gamma}$, $\widehat{u}$, and $\widehat{\gamma}$ to establish $\models_{\Gamma,\gamma} \mathsf{Closed}^U_\urcorner(u[v])$. Likewise, we may use $\widehat{\Gamma}'$, $\widehat{u[v]}$, and $\widehat{\gamma}'$ given by the right-hand side to establish $\models_{\Gamma,x:T,\gamma[x\mapsto[\![\Gamma\rhd v:T]\!]_\gamma]} \mathsf{Closed}^U_\urcorner(u[v])$. □

Thus we are now in a position to express closedness in the logic by basic predicate symbols Closed. Note that the predefined semantics of Closed is not specific to a particular model, but relates, roughly spoken, to any denotational semantics of the form considered in this thesis.

Now, just as we have non-logical axioms for the other basic predicate in the logic, namely equality, we should appreciate some axioms for Closed as well. In particular, we will need to be able to derive the closedness of terms in the logic, and in order to do this, we will need to derive the closedness of types too. For example, we would like to be able to assert something like

$$\vdash_\Gamma \mathsf{Closed}_\urcorner(U) \ \wedge\ \mathsf{Closed}^V_\urcorner(v) \ \Rightarrow\ \mathsf{Closed}^{U\to V}_\urcorner(\lambda x\!:\!U.v)$$

where we have extended Closed somehow to talk explicitly about types as well. But with our present semantics for Closed, we are not able to validate the indicated axiom. The problem is that we cannot control the form of the witness $\widehat{v}$ that would give $\mathsf{Closed}^V_\urcorner(v)$; more specifically, we cannot guarantee that $\widehat{v}$ has a free variable x to be bound by lambda abstraction. This makes it hard to get the desired results, since we cannot in general construct $\widehat{\lambda x\!:\!U.v}$ from $\widehat{v}$.

We mentioned above that to be able to derive the closedness of terms in the logic, we have to extend Closed to deal with types. Then we encounter similar problems as above. If we define Closed for types in the style of Def. 5.19, *i.e.,*

$$\begin{aligned}\models_{\Gamma,\gamma} \mathsf{Closed}_\urcorner(U) \stackrel{\mathit{def}}{\Leftrightarrow} \textit{exists } \widehat{\Gamma},\ \widehat{\Gamma} \rhd \widehat{U},\ \textit{and } \widehat{\gamma} \textit{ on } \widehat{\Gamma},\ \textit{such that}\\ \widehat{\Gamma} \textit{ contains type variables only from } \urcorner,\\ \textit{and } [\![\widehat{\Gamma} \rhd \widehat{U}]\!]_{\widehat{\gamma}} = [\![\Gamma \rhd U]\!]_\gamma\end{aligned}$$

we will for example, have problems verifying the desired axiom

$$\vdash_\Gamma \mathsf{Closed}_{\urcorner,X}(U) \ \Rightarrow\ \mathsf{Closed}_\urcorner(\forall X.U)$$

Again, the problem is that we cannot control the form of the witness $\widehat{U}$ giving $\mathsf{Closed}_{\urcorner,X}(U)$. For example, for $\Gamma = \{X, Y\}$ and γ such that $\gamma(X) = \mathcal{A}$ and $\gamma(Y) = \mathcal{B}$, we have $\models_{\Gamma,\gamma} \mathsf{Closed}_{\{X\}}(X \to Y)$, simply by exhibiting $\widehat{X\to Y} = X$ and $\widehat{\gamma}$ such that $\widehat{\gamma}(X) = \mathcal{A}\to\mathcal{B}$. However, unless $\mathcal{B}$ is denotable by a closed type, we do not have $\models_{\Gamma,\gamma} \mathsf{Closed}_{\{\}}(\forall X.X\to Y)$. On the other hand, by insisting on $\widehat{U}$ being on the same form as U, and in particular that the free occurrences of X in U are preserved for binding in $\widehat{U}$, one can easily validate the axiom by displaying a $\widehat{\forall X.U}$ constructed from $\widehat{U}$. Similar considerations apply for the desired axiom

$$\vdash_\Gamma \mathsf{Closed}_\urcorner(U) \ \wedge\ \mathsf{Closed}_\urcorner(V) \ \Rightarrow\ \mathsf{Closed}_\urcorner(U\to V)$$

Here we would like to construct $\widehat{U\to V}$ from $\widehat{U}$ and $\widehat{V}$. This is hard unless we insist that the environments $\widehat{\gamma}_U$ and $\widehat{\gamma}_V$ do not clash on common variables.

The upshot of all this is that we need a predicate or something predicate-like, that expresses closedness, and which guarantees that the terms and types witnessing closedness preserve form.

We now present a second family of predicate-like symbols $\mathsf{ClosedS}$ with a semantics extending that of Closed according to what we have just said. With $\mathsf{ClosedS}$ we can establish the desired axioms for deriving closedness in the logic.

Definition 5.21 ($\mathsf{ClosedS}$) *The logical language is extended with families of basic predicate-like symbols* $\mathsf{ClosedS}_{\urcorner}(U)$ *ranging over types, and* $\mathsf{ClosedS}^{U}_{\urcorner}(u)$ *ranging over terms* $u\!:\!U$*, both qualified by a given set of types* $\urcorner$*. The following predefined semantics is given. For any type* $\Gamma \rhd U$*, term* $\Gamma \rhd u\!:\!U$*, and environment* γ *on* Γ*,*

$$\begin{aligned}\models_{\Gamma,\gamma} \mathsf{ClosedS}_{\urcorner}(U) \overset{def}{\Leftrightarrow}\ & \textit{exists } \widehat{\Gamma},\ \textit{type } \widehat{\Gamma} \rhd \widehat{U},\ \textit{and a } \widehat{\gamma} \textit{ on } \widehat{\Gamma},\ \textit{such that}\\ & \widehat{\Gamma} \textit{ contains type variables only from } \urcorner,\\ & \widehat{U} = U[\boldsymbol{T}/\boldsymbol{X}],\ \textit{where } \boldsymbol{X} = FTyV(U),\\ & \textit{and } T_i \textit{ is } X_i,\ \textit{if } X_i \textit{ is in } \urcorner,\ \textit{or else some closed type},\\ & \textit{and } [\![\widehat{\Gamma} \rhd T_i]\!]_{\widehat{\gamma}} = \gamma(X_i).\end{aligned}$$

$$\begin{aligned}\models_{\Gamma,\gamma} \mathsf{ClosedS}^{U}_{\urcorner}(u) \overset{def}{\Leftrightarrow}\ & \textit{exists } \widehat{\Gamma},\ \widehat{\Gamma} \rhd \widehat{U},\ \widehat{\Gamma} \rhd \widehat{u}\!:\!\widehat{U},\ \textit{and } \widehat{\gamma} \textit{ on } \widehat{\Gamma},\ \textit{such that}\\ & \widehat{\Gamma} \textit{ contains types only from } \urcorner,\\ & \widehat{U} = U[\boldsymbol{T}/\boldsymbol{X}],\ \textit{and } \widehat{u} = u[\boldsymbol{T}/\boldsymbol{X}, \boldsymbol{t}/\boldsymbol{x}],\\ & \textit{where } \boldsymbol{X} = FTyV(U),\ \textit{and } \boldsymbol{x}\!:\!\boldsymbol{U} = FTeV(u),\\ & \textit{and } T_i \textit{ is } X_i,\ \textit{if } X_i \textit{ is in } \urcorner,\ \textit{or else some closed type},\\ & \textit{and } t_i \textit{ is } x_i,\ \textit{if } U_i \textit{ is in } \urcorner,\ \textit{and } FTeV(t_i) \cap FTeV(t_j) = \emptyset,\ \textit{if } i \neq j,\\ & \textit{and } [\![\widehat{\Gamma} \rhd T_i]\!]_{\widehat{\gamma}} = \gamma(X_i),\ \textit{and } [\![\widehat{\Gamma} \rhd t_i]\!]_{\widehat{\gamma}} = \gamma(x_i).\end{aligned}$$

We use the same notational conventions for $\urcorner$ *as we do for type contexts.*

The additional clauses in Def. 5.19 concerning the form of $\widehat{U}$ and $\widehat{u}$ are necessary for the semantic justification of the non-logical axioms in Def. 5.23 below, which enable us to derive closedness in the logic. The additional clauses insist that we choose the most open type and term $\widehat{U}$ and $\widehat{u}$ as possible, given $\urcorner$, as well as insisting on $\widehat{U}$ and $\widehat{u}$ being as much of the same form as U and u as possible. This is necessary for validating the axioms of Def. 5.23 concerning abstraction, *e.g.,* axiom (3). For example, for $\urcorner = \{\mathsf{Nat}, X\}$, let $\Gamma = X, Y$, let $U = X \to Y$, and let γ be such that $[X \mapsto [\![\mathsf{Nat}]\!], Y \mapsto [\![\mathsf{Nat}]\!]]$. Let us establish $\mathsf{ClosedS}_{\{\mathsf{Nat},X\}}(U)$ by finding a $\widehat{U}$. Since $\urcorner$ includes the type variable X, we have in the outset many choices for $\widehat{\Gamma}$ and thence $\widehat{U}$. But, the additional clauses prevent us from choosing $\widehat{U}$ to be $\mathsf{Nat} \to \mathsf{Nat}$, or X with $\widehat{\gamma}(X) = [\![\mathsf{Nat} \to \mathsf{Nat}]\!]$, or $X \to X$ or $\mathsf{Nat} \to X$ with $\widehat{\gamma}(X) = [\![\mathsf{Nat}]\!]$. The clauses insist on the choice $\widehat{U} = X \to \mathsf{Nat}$, with $\widehat{\gamma}(X) = \gamma(X)$. This then implies that of $\mathsf{ClosedS}_{\{\mathsf{Nat}\}}(\forall X.U)$ holds; simply give $\widehat{\forall X.U} = \forall X.\widehat{U}$. In contrast, the other choices for $\widehat{U}$ leave no suitable free X in $\widehat{U}$ for binding.

However, with this semantics, $\mathsf{ClosedS}$ is not a predicate, in the sense that if we were to use $\mathsf{ClosedS}$ as a predicate symbol with the above semantics, the logic would no longer be sound; hence we refer to $\mathsf{ClosedS}$ as a *predicate-like* symbol. In particular, instantiating from universal quantification ($\forall$-*elim*) no longer holds. For example, we cannot give a semantic justification for

$$\frac{\vdash_\Gamma \forall x\!:\!T.\mathsf{ClosedS}^{U}_{\urcorner}(u[x])}{\vdash_\Gamma \mathsf{ClosedS}^{U}_{\urcorner}(u[v])}$$

To see this, suppose we have $\models_{\Gamma,\gamma} \forall x\!:\!T.\mathsf{ClosedS}^{T}_{\urcorner}(x)$. This is the case for example in the parametric PER-model for $T = \mathsf{Nat}$. We then want to establish $\models_{\Gamma,\gamma} \mathsf{ClosedS}^{T}_{\urcorner}(v)$. However, we are now insisting that $\widehat{v} = v[\boldsymbol{t}/\boldsymbol{x}]$, where t_i is a closed term w.r.t. $\urcorner$, if the type of x_i is not in $\urcorner$. But v might introduce a variable whose type is not in $\urcorner$, and which γ maps to an element which is not denotable by an appropriately closed term.

The lack of substitutivity is a result of an intentional semantics; the semantics of $\mathsf{ClosedS}$ refers to syntax. Nonetheless, with this intentional semantics, we can validate axioms that allow us to derive closedness.

This dilemma is resolved as follows. First of all, we separate out the rules for deriving closedness in a special calculus, $\vdash^{\text{CTF}}$ in Def. 5.23 below, where logical instantiation has no part. This calculus is then sound. Secondly, we maintain the predicate symbols Closed of Def. 5.19. These have no part in deriving closedness. The idea is that it suffices for the specifier to know about and use the Closed predicates when doing refinement proofs using the strategy embodied by SUB and QUOT. We as formalists on the other hand, will need the ability to derive closedness in order to prove our results, but we will not need the rules relying on substitutivity. The specifier need know only about the main results, and then perhaps only indirectly through the actions of a proof checker, but need not know the details of the derivations, in particular not those using the predicate-like symbols $\mathsf{ClosedS}$ and the calculus $\vdash^{\text{CTF}}$. For us, the crucial link between the two notions of closedness is established by the next result.

Lemma 5.22 *If U is a type closed qualified by $\urcorner$, then it is the case that*

1. $\models_{\Gamma,\gamma} \mathsf{ClosedS}^{U}_{\urcorner}(t) \;\Rightarrow\; \models_{\Gamma,\gamma} \mathsf{Closed}^{U}_{\urcorner}(t)$, *for any $t\!:\!U$,*

2. $\models_{\Gamma,\gamma} \mathsf{ClosedS}^{U}_{\urcorner}(x) \;\Leftrightarrow\; \models_{\Gamma,\gamma} \mathsf{Closed}^{U}_{\urcorner}(x)$, *for any variable $x\!:\!U$.*

Proof: The first claim, and the implication from left to right of the second claim follow easily. The only thing to check is that the $\widehat{U}$ provided by the left-hand side is in fact U. But this is necessarily so, since $\widehat{U} = U[\boldsymbol{T}/\boldsymbol{X}]$ is required to preserve all variables from $\urcorner$. Since U is closed qualified by $\urcorner$, this entails that $\widehat{U} = U$.

For the converse implication of the second claim, suppose $\models_{\Gamma,\gamma} \mathsf{Closed}^U(x)$. Then there simply exists $\widehat{\Gamma}$ containing types only from $\urcorner$, and $\widehat{x} : \widehat{U}$ such that $[\![\widehat{\Gamma} \rhd \widehat{x}:U]\!]_{\widehat{\gamma}} = [\![\Gamma \rhd x:U]\!]_{\gamma}$. We construct the witnesses $\widehat{\Gamma}'$, $\widehat{\Gamma}' \rhd \widehat{U}'$, $\widehat{\Gamma}' \rhd \widehat{u}':\widehat{U}'$, and $\widehat{\gamma}'$ to get $\models_{\Gamma,\gamma} \mathsf{ClosedS}^U(x)$ as follows. Since U is closed qualified by $\urcorner$, we may set $\widehat{U}' = U$. For $\widehat{x}'$, if $U \in \urcorner$, we set $\widehat{x}' = x$ and choose $\widehat{\gamma}'(x) = [\![\widehat{\Gamma} \rhd \widehat{x}]\!]_{\widehat{\gamma}}$, and if $U \notin \urcorner$, we set $\widehat{x}' = \widehat{x}$ and $\widehat{\gamma}' = \widehat{\gamma}$. $\square$

Lemma 5.22 allows us to import results from the sound calculus for deriving closedness following in Def. 5.23 below, as lemmas into the regular calculus, and at the same time keeping the two calculi separated from each other.

Definition 5.23 (Calculus for Closed Type and Term Formation $\vdash^{\mathrm{CTF}}$) *We admit all logical axioms (Sect. A.2 in Appendix A), except universal quantification instantiation* ($\forall$*-elim*) *and existential quantification generalisation* ($\exists$*-intro*) *into the calculus* $\vdash^{\mathrm{CTF}}$. *In addition we admit the following axioms* CTF.

1. $\vdash^{\mathrm{CTF}}_{\Gamma} \mathsf{ClosedS}_{\urcorner,X}(X)$
2. $\vdash^{\mathrm{CTF}}_{\Gamma} \mathsf{ClosedS}_{\urcorner}(U) \;\wedge\; \mathsf{ClosedS}_{\urcorner}(V) \;\Rightarrow\; \mathsf{ClosedS}_{\urcorner}(U \to V)$
3. $\vdash^{\mathrm{CTF}}_{\Gamma} \mathsf{ClosedS}_{\urcorner,X}(U) \;\Rightarrow\; \mathsf{ClosedS}_{\urcorner}(\forall X.U)$
4. $\vdash^{\mathrm{CTF}}_{\Gamma} \mathsf{ClosedS}^{U}_{\urcorner,\boldsymbol{X},U}(x)$, *for variable* x, *and* $\boldsymbol{X} = FTyV(U)$
5. $\vdash^{\mathrm{CTF}}_{\Gamma} \mathsf{ClosedS}_{\urcorner,\boldsymbol{X}}(U) \;\wedge\; \mathsf{ClosedS}^{V}_{\urcorner,\boldsymbol{X},U}(v) \;\Rightarrow\; \mathsf{ClosedS}^{U\to V}_{\urcorner,\boldsymbol{X}}(\lambda x\!:\!U.v)$, *for* $\boldsymbol{X} = FTyV(U)$
6. $\vdash^{\mathrm{CTF}}_{\Gamma} \mathsf{ClosedS}^{U\to V}_{\urcorner}(g) \;\wedge\; \mathsf{ClosedS}^{U}_{\urcorner}(u) \;\Rightarrow\; \mathsf{ClosedS}^{V}_{\urcorner}(gu)$
7. $\vdash^{\mathrm{CTF}}_{\Gamma} \mathsf{ClosedS}^{U}_{\urcorner,X}(t) \;\Rightarrow\; \mathsf{ClosedS}^{\forall X.U}_{\urcorner}(\Lambda X.t)$
8. $\vdash^{\mathrm{CTF}}_{\Gamma} \mathsf{ClosedS}^{\forall X.U[X]}_{\urcorner}(f) \;\wedge\; \mathsf{ClosedS}_{\urcorner}(A) \;\Rightarrow\; \mathsf{ClosedS}^{U[A]}_{\urcorner}(fA)$
9. $\vdash^{\mathrm{CTF}}_{\Gamma} \mathsf{ClosedS}_{\urcorner}(U) \;\Rightarrow\; \mathsf{ClosedS}_{\urcorner'}(U)$, $\urcorner \subseteq \urcorner'$, $X \in FTyV(U) \cap \urcorner' \;\Rightarrow\; X \in \urcorner$
10. $\vdash^{\mathrm{CTF}}_{\Gamma} \mathsf{ClosedS}^{U}_{\urcorner}(u) \;\Rightarrow\; \mathsf{ClosedS}^{U}_{\urcorner'}(u)$, $\urcorner \subseteq \urcorner'$, $X \in FTyV(U) \cap \urcorner' \;\Rightarrow\; X \in \urcorner$

Additionally, we have the inference rules:

$$\textit{import}: \frac{\Phi \vdash_{\Gamma} \mathsf{Closed}^{U}_{\urcorner}(x)}{\Phi \vdash^{\mathrm{CTF}}_{\Gamma} \mathsf{ClosedS}^{U}_{\urcorner}(x)} \qquad \textit{export}: \frac{\Phi \vdash^{\mathrm{CTF}}_{\Gamma} \mathsf{ClosedS}^{U}_{\urcorner}(t)}{\Phi \vdash_{\Gamma} \mathsf{Closed}^{U}_{\urcorner}(t)}$$

for U *a type closed qualified by* $\urcorner$, x *a variable, any term* t, *and appropriate* Φ.

Lemma 5.24 *The calculus* $\vdash^{\text{CTF}}$ *is sound.*

Proof: The axiom schemata CTF are sound, see Appendix B. It is easy to see that the admitted logical axioms are sound. The additional rules *import* and *export* are justified by Lemma 5.22. □

Henceforth, we usually omit the type in the term families of Closed and $\mathsf{ClosedS}$.

Ideally, we would like to avoid having the separate external calculus $\vdash^{\text{CTF}}$. Instead, it would be better to have a definition of Closed that satisfies substitutivity and at the same time allows derivation of closedness. This might not be possible in the current semantic framework, due to the inherent intentional properties that seem to emerge as necessities for deriving closedness. A solution might be to use a Kripke-style semantics, and it does seem possible to achieve parametricity in such a model. We choose not to investigate this here, because a change of semantics would require too much over-head.

5.3.2 Closed Computations

Since we are now able to talk about closedness in the logic, it is possible to relate to other models than those in which we somehow rely on term denotability. In any model, we can now simply focus the attention on term-denotable elements by using the Closed clauses. For example, the interpretation in any model of $\forall f : \forall X.(\mathfrak{T}[X, \boldsymbol{Z}] \to D)\ .\ \mathsf{Closed}_{In}(f) \ \Rightarrow\ \phi(f)$ restricts attention in ϕ to those interpretations of all $f : \forall X.(\mathfrak{T}[X] \to D)$ that are denotable by terms whose only free variables are of types in *In*.

The usual semantic notion of observational equivalence for lambda calculus is defined w.r.t. contexts that when filled, are closed terms (Mitchell, 1991). Since we can express observational equivalence via closed observable computations, qualified by the set of input types *In*, we can now give a reasonable alternative definition in the logic of observational equivalence.

Definition 5.25 (Closed Observational Equivalence (ObsEqC)**)** *Define* closed observational equivalence ObsEqC w.r.t. $\mathfrak{T}[X, \boldsymbol{Z}]$ and *Obs by*

$$\begin{array}{l}\mathsf{ObsEqC}^{Obs}_{\mathfrak{T}[X,\boldsymbol{Z}]} \stackrel{def}{=} (u : \exists X.\mathfrak{T}[X, \boldsymbol{Z}], v : \exists X.\mathfrak{T}[X, \boldsymbol{Z}]).\\ (\exists A, B.\exists \mathfrak{a} : \mathfrak{T}[A, \boldsymbol{Z}], \mathfrak{b} : \mathfrak{T}[B, \boldsymbol{Z}]\ .\ u = (\mathsf{pack} A \mathfrak{a})\ \wedge\ v = (\mathsf{pack} B \mathfrak{b})\ \wedge\\ \qquad \bigwedge_{D \in Obs} \forall f : \forall X.(\mathfrak{T}[X, \boldsymbol{Z}] \to D)\ .\ \mathsf{Closed}_{In}(f)\ \Rightarrow\ (f A\, \mathfrak{a}) = (f B\, \mathfrak{b}))\end{array}$$

Of course we have a tight connection to package components also for this notion of observational equivalence.

Theorem 5.26 *The following sequent schema is derivable.*

$$\begin{array}{l}\forall A, B.\forall \mathfrak{a}\!:\!\mathfrak{T}[A], \mathfrak{b}\!:\!\mathfrak{T}[B]\ .\ (\mathsf{pack}A\mathfrak{a})\ \mathsf{ObsEq}^{Obs}\ (\mathsf{pack}B\mathfrak{b})\\ \qquad \Leftrightarrow\ \bigwedge_{D\in Obs} \forall f\!:\!\forall X.(\mathfrak{T}[X]\to D)\ .\ \mathsf{Closed}_{In}(f)\ \Rightarrow\ (f\,A\,\mathfrak{a}) = (f\,B\,\mathfrak{b})\end{array}$$

Proof: The same argument as for Theorem 4.5. □

—

We can now use something like Lemma 5.12 to show the necessary bounded parametricity instance also for the parametric PER-model. We incorporate Closed into SPPARAM, and then to make the induction spiral work, we have to strengthen Lemma 5.12 by incorporating Closed into the $\mathsf{Dfnbl}^{\mathsf{abo}}$ clause.

5.3.3 Abstraction Barrier-Observing Relations *II*

We first extend the notion of *abo*-relation to incorporate closed computations. The definition is the same as for the data type relation of Def. 5.8, except that we use a different definability clause for arrow-type relations.

Definition 5.27 (*abo*-Relation by Closed Computations) *Relative to* $\mathfrak{T}[X]$, *for k-ary* $\boldsymbol{Y}$, A, B, $R\subset A\times B$, $\mathfrak{a}\!:\!\mathfrak{T}[A]$, $\mathfrak{b}\!:\!\mathfrak{T}[B]$. *Define the abo*-relation by closed computations $U[\mathbf{eq}_{\boldsymbol{Y}}, R]^{\mathsf{abo}}_{\mathsf{C}} \subset U[\boldsymbol{Y}, A] \times U[\boldsymbol{Y}, B]$, *for the list* $\mathsf{abo} = A, B, \mathfrak{a}, \mathfrak{b}$, *inductively on* $U[\boldsymbol{Y}, X]$ *by*

$$\begin{array}{lll}U = X & : & U[\mathbf{eq}_{\boldsymbol{Y}}, R]^{\mathsf{abo}}_{\mathsf{C}} \stackrel{def}{=} R\\ U = Y_i & : & U[\mathbf{eq}_{\boldsymbol{Y}}, R]^{\mathsf{abo}}_{\mathsf{C}} \stackrel{def}{=} \rho_i\\ U = \forall Y_{k+1}.U'[\boldsymbol{Y}, Y_{k+1}, X] & : & U[\mathbf{eq}_{\boldsymbol{Y}}, R]^{\mathsf{abo}}_{\mathsf{C}} \stackrel{def}{=}\\ \multicolumn{3}{r}{(g\!:\!\forall Y_{k+1}.U'[\boldsymbol{Y}, Y_{k+1}, A],\ h\!:\!\forall Y_{k+1}.U'[\boldsymbol{Y}, Y_{k+1}, B])\ .}\\ \multicolumn{3}{r}{(\forall Y_{k+1}\ .\ gY_{k+1}\ (U'[\mathbf{eq}_{\boldsymbol{Y}}, \mathsf{eq}_{Y_{k+1}}, R]^{\mathsf{abo}}_{\mathsf{C}})\ hY_{k+1})}\\ U = U'\to U'' & : & U[\mathbf{eq}_{\boldsymbol{Y}}, R]^{\mathsf{abo}}_{\mathsf{C}} \stackrel{def}{=}\\ \multicolumn{3}{l}{\quad (g\!:\!U'[\boldsymbol{Y}, A]\to U''[\boldsymbol{Y}, A],\ h\!:\!U'[\boldsymbol{Y}, B]\to U''[\boldsymbol{Y}, B])\ .}\\ \multicolumn{3}{l}{\qquad (\forall x\!:\!U'[\boldsymbol{Y}, A], \forall y\!:\!U'[\boldsymbol{Y}, B]\ .}\\ \multicolumn{3}{l}{\ (x\ U'[\mathbf{eq}_{\boldsymbol{Y}}, R]^{\mathsf{abo}}_{\mathsf{C}}\ y\ \ \wedge\ \ \mathsf{DfnblC}^{\mathsf{abo}}_{U'[\boldsymbol{Y},X]}(x, y))\ \Rightarrow\ \ (gx)\ U''[\mathbf{eq}_{\boldsymbol{Y}}, R]^{\mathsf{abo}}_{\mathsf{C}}\ (hy))}\end{array}$$

where,

$$\begin{array}{l}\mathsf{DfnblC}^{\mathsf{abo}}_{U'[\boldsymbol{Y},X]}(x, y) \stackrel{def}{=} \exists f_{U'}\!:\!\forall X.(\mathfrak{T}[X]\to U'[\boldsymbol{Y}, X])\ .\\ \qquad\qquad\qquad \mathsf{Closed}_{In,\boldsymbol{Y}}(f_{U'})\ \wedge\ (f_{U'}A\,\mathfrak{a}) = x\ \wedge\ (f_{U'}B\,\mathfrak{b}) = y\end{array}$$

Again, $\mathbf{eq}_{\boldsymbol{Z}}{}^{\mathsf{abo}}_{\mathsf{C}} \stackrel{def}{=} \mathbf{eq}_{\boldsymbol{Z}}$ *for parameters* $\boldsymbol{Z}$ *of* $\mathfrak{T}[X, \boldsymbol{Z}]$.

We might omit the subscript to the $\mathsf{DfnblC}^{\mathsf{abo}}$ clause. As for Def. 5.8, the essence of Def. 5.27 is the weakened arrow-type relation, now via the $\mathsf{DfnblC}^{\mathsf{abo}}$ clause. Notice that the predicate Closed for the case when $\mathsf{DfnblC}^{\mathsf{abo}}$ is dealing within universal quantification, gets qualified by type variables $\boldsymbol{Y}$ in addition to In. This is because according to *Abs-Bar*, we can have arbitrary term variables over types instantiating polymorphic functionals. The proviso is that these types cannot involve the actual data representation, unless the instantiating type coincides with the actual data representation. This is of course taken care of by the type of $f_{U'}$, since the virtual data representation is a bound variable.

We define with a slight abuse of notation:

Definition 5.28 *For A, B and $R \subset A \times B$,*

$$\mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}}_{\mathsf{C}} \stackrel{def}{=} (\mathfrak{a} : \mathfrak{T}[A, \boldsymbol{Z}], \mathfrak{b} : \mathfrak{T}[B, \boldsymbol{Z}]) \ . \ (\wedge_{1 \leq i \leq k} \ \mathfrak{a}.g_i \ (T_i[R, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}}_{\mathsf{C}}) \ \mathfrak{b}.g_i)$$

We get analogous results to those in Sect. 5.2.2.

Lemma 5.29 *We have using $\vdash^{\mathrm{CTF}}$, the derivability of*

$$\forall g, h : D \ . \ g =_D h \ \Leftrightarrow \ g(D^{\mathsf{abo}}_{\mathsf{C}})h, \qquad \text{for } D \in Obs$$

Proof: This follows the lines of the proof for Lemma 5.10. We illustrate with the inductive type Nat. So consider $g(\mathsf{Nat}^{\mathsf{abo}})h$, *i.e.*,

$$\begin{aligned}(\forall Y . \forall y, y' : Y . \forall s, s' : Y \to Y \ . \\ y =_Y y' \ \wedge \ \mathsf{DfnblC}^{\mathsf{abo}'}(y, y') \ \wedge \ s(\mathsf{eq}_Y \to \mathsf{eq}_Y)^{\mathsf{abo}'} s' \ \wedge \ \mathsf{DfnblC}^{\mathsf{abo}'}(s, s') \\ \Rightarrow \ (gYys) =_Y (hYy's'))\end{aligned}$$

The clause $\mathsf{DfnblC}^{\mathsf{abo}'}(y, y')$ says

$$\exists f : \forall X . (\mathfrak{T}[X] \to Y) \ . \ \mathsf{Closed}_{In,Y}(f) \ \wedge \ (f A \, \mathfrak{a}) = y \ \wedge \ (f B \, \mathfrak{b}) = y'$$

By $y =_Y y'$, this is derivable by exhibiting $\Lambda X . \lambda \mathfrak{x} : \mathfrak{T}[X] . y$. The clause $\mathsf{Closed}_{In,Y}(f)$ is derivable using the calculus $\vdash^{\mathrm{CTF}}$ and applying Lemma 5.22. Similarly, we get $\mathsf{DfnblC}^{\mathsf{abo}'}(s, s')$ by exhibiting $\Lambda X . \lambda \mathfrak{x} : \mathfrak{T}[X] . s : \forall X . (\mathfrak{T}[X] \to (Y \to Y))$. This means that the definition of $g(\mathsf{Nat}^{\mathsf{abo}}_{\mathsf{C}})h$ is equivalent to

$$\forall Y . \forall y, y' : Y . \forall s, s' : Y \to Y \ . \ y =_Y y' \ \wedge \ s(\mathsf{eq}_Y \to \mathsf{eq}_Y)s' \ \wedge \quad \Rightarrow \ (gYys) =_Y (hYy's')$$

or $\forall Y \ . \ gY(\mathsf{eq}_Y \to (\mathsf{eq}_Y \to \mathsf{eq}_Y) \to \mathsf{eq}_Y)hY$, *i.e.*, $\forall Y \ . \ gY = hY$. By the congruence axiom schema, we get $\Lambda Y . gY = \Lambda Y . hY$, which by η-equality yields $g =_{\mathsf{Nat}} h$. $\square$

Definition 5.30 (*abo*-Simulation Relation $\mathsf{SimRelC}$) *Relatedness by* abstraction barrier-observing closed computation (*abo*) simulation relation *w.r.t.* $\mathfrak{T}[X, \boldsymbol{Z}]$ *is expressed in the logic by*

$$\begin{array}{l}\mathsf{SimRelC}_{\mathfrak{T}[X,\boldsymbol{\rho}]} \stackrel{def}{=} (u\!:\!\exists X.\mathfrak{T}[X,\boldsymbol{U}], v\!:\!\exists X.\mathfrak{T}[X,\boldsymbol{V}])\ .\\ \qquad (\exists A,B.\exists \mathfrak{a}\!:\!\mathfrak{T}[A,\boldsymbol{U}], \mathfrak{b}\!:\!\mathfrak{T}[B,\boldsymbol{V}]\ .\ u = (\mathsf{pack} A \mathfrak{a})\ \wedge\ v = (\mathsf{pack} B \mathfrak{b})\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \exists R \subset A \times B\ .\ \mathfrak{a}(\mathfrak{T}[R,\boldsymbol{\rho}]^{\mathsf{abo}}_{\mathsf{C}})\mathfrak{b})\end{array}$$

where $\mathsf{abo} = A, B, \mathfrak{a}, \mathfrak{b}$*, and where* $\boldsymbol{Z}$ *are the free type variables in* $\mathfrak{T}[X, \boldsymbol{Z}]$ *other than* X*, and* $\boldsymbol{\rho} \subset \boldsymbol{U} \times \boldsymbol{V}$ *is a vector of relations of the same length.*

The subscript $\mathfrak{T}[X, \boldsymbol{\rho}]$ to $\mathsf{SimRelC}_{\mathfrak{T}[X,\boldsymbol{\rho}]}$ might occasionally be omitted.

5.3.4 Special Parametricity for Closed Computations

We can now establish a special version of parametricity w.r.t. the non-syntactic parametric PER-model. We write $f\ (\forall X.\mathfrak{T}[X, \mathbf{eq}_{\boldsymbol{Z}}]^{\epsilon}_{\mathsf{C}} \to U[X, \mathbf{eq}_{\boldsymbol{Z}}]^{\epsilon}_{\mathsf{C}})\ f$, meaning

$$\begin{array}{l}\forall A, B, R \subset A \times B. \forall \mathfrak{a}\!:\!\mathfrak{T}[A, \boldsymbol{Z}], \mathfrak{b}\!:\!\mathfrak{T}[B, \boldsymbol{Z}].\\ \qquad\qquad\qquad \mathfrak{a}(\mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}}_{\mathsf{C}})\mathfrak{b}\ \Rightarrow\ (f A\, \mathfrak{a})(U[R, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}}_{\mathsf{C}})(f B\, \mathfrak{b}))\end{array}$$

where $\mathsf{abo} = A, B, \mathfrak{a}, \mathfrak{b}$.

Lemma 5.31 *For* $\mathfrak{T}[X, \boldsymbol{Z}]$ *adhering to* HADT_{Obs}*, for* $U[X, \boldsymbol{Z}]$ *having no occurrences of universal types other than those in Obs, and whose only free variables are among* X *and* $\boldsymbol{Z}$*, for* $f\!:\!\forall X.(\mathfrak{T}[X] \to U[X])$ *whose only free variables are term variables of types in In, we derive*

$$f\ (\forall X.\mathfrak{T}[X, \mathbf{eq}_{\boldsymbol{Z}}]^{\epsilon}_{\mathsf{C}} \to U[X, \mathbf{eq}_{\boldsymbol{Z}}]^{\epsilon}_{\mathsf{C}})\ f$$

Proof: Along the lines of the proof of Lemma 5.12. □

By Lemma 5.31, the following axiom schema is sound w.r.t. the parametric PER-model of (Bainbridge et al., 1990).

Definition 5.32 (Special Parametricity (spParamC)) *For* $\mathfrak{T}[X, \boldsymbol{Z}]$ *adhering to* HADT_{Obs}*, for* $U[X, \boldsymbol{Z}]$ *having no occurrences of universal types other than those in Obs, and whose only free variables are among* X *and* $\boldsymbol{Z}$*,*

$$\begin{array}{l}\textsc{spParamC}\text{:}\ \forall f\!:\!\forall X.(\mathfrak{T}[X, \boldsymbol{Z}] \to U[X, \boldsymbol{Z}])\ .\\ \qquad\qquad\qquad\qquad \mathsf{Closed}_{In}(f)\ \Rightarrow\ f\ (\forall X.\mathfrak{T}[X, \mathbf{eq}_{\boldsymbol{Z}}]^{\epsilon}_{\mathsf{C}} \to U[X, \mathbf{eq}_{\boldsymbol{Z}}]^{\epsilon}_{\mathsf{C}})\ f\end{array}$$

5.3.5 The Results

We can now show the higher-order generalisation of Theorem 4.17, now with reference to the parametric PER-model. Note that we decided at the beginning of this section that $In = Obs$.

Theorem 5.33 *Let $\mathfrak{T}[X]$ adhere to $\mathtt{HADT}_{Obs}$. Extending the language with the predicates* Closed *of Def. 5.19, with* SPPARAMC *and* $\vdash^{\mathrm{CTF}}$*, we derive*

$$\forall \boldsymbol{Z}.\forall u,v\!:\!\exists X.\mathfrak{T}[X,\boldsymbol{Z}]\ .\ u\ \mathsf{SimRelC}_{\mathfrak{T}[X,\mathbf{eq}_{\boldsymbol{Z}}]}\ v \quad \Leftrightarrow \quad u\ \mathsf{ObsEqC}^{Obs}_{\mathfrak{T}[X,\boldsymbol{Z}]}\ v$$

Proof: This follows from Theorem 5.34 below. □

Theorem 5.34 (Tight Correspondence) *Let $\mathfrak{T}[X]$ adhere to $\mathtt{HADT}_{Obs}$. With* SPPARAMC *and* $\vdash^{\mathrm{CTF}}$ *we derive*

$$\begin{aligned}&\forall A,B.\forall \mathfrak{a}\!:\!\mathfrak{T}[A,\boldsymbol{Z}],\mathfrak{b}\!:\!\mathfrak{T}[B,\boldsymbol{Z}]\ .\\ &\quad \exists R\!\subset\! A\!\times\! B\ .\ \mathfrak{a}(\mathfrak{T}[R,\mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}}_{\mathsf{C}})\mathfrak{b}\ \Leftrightarrow\\ &\qquad \textstyle\bigwedge_{D\in Obs}\forall f\!:\!\forall X.(\mathfrak{T}[X,\boldsymbol{Z}]\to D)\ .\ \mathsf{Closed}_{In}(f)\ \Rightarrow\ (f\,A\,\mathfrak{a}) = (f\,B\,\mathfrak{b})\end{aligned}$$

Proof: $\Rightarrow$: This follows from SPPARAMC and Lemma 5.29.

$\Leftarrow$: Along the lines of the proof of Theorem 5.15, using $\vdash^{\mathrm{CTF}}$ (Lemma 5.24) to obtain $\mathsf{Closed}_{In}(f)$ from $\mathsf{Closed}_{In}(f_U)$, via Lemma 5.22, and using Lemma 5.29 in place of Lemma 5.10. □

Example 5.35 Recall the definitions of $\mathfrak{a}$ and $\mathfrak{b}$ from Example 5.7 and Example 5.16. In Example 5.16, we said that for

$$\begin{aligned}&R = \mathsf{Dfnbl} \stackrel{def}{=} (a\!:\!\mathsf{List}_{\mathsf{Nat}}, b\!:\!\mathsf{List}_{\mathsf{Nat}})\ .\\ &\qquad (\exists f\!:\!\forall X.(\mathfrak{T}_{\mathsf{SetCE}}[X]\to X)\ .\ (f(\mathsf{List}_{\mathsf{Nat}})(\mathfrak{a})) = a\ \wedge\ (f(\mathsf{List}_{\mathsf{Nat}})(\mathfrak{b})) = b)\end{aligned}$$

we have that the polymorphic closed type and term model of (Hasegawa, 1991) does not satisfy $\mathfrak{a}\ (\mathfrak{T}_{\mathsf{SetCE}}[R])\ \mathfrak{b}$, but does satisfy $\mathfrak{a}\ (\mathfrak{T}_{\mathsf{SetCE}}[R])^{\mathsf{abo}}\ \mathfrak{b}$, and that through Theorem 5.15 we get observational equivalence.

We can now relate this to the parametric PER-model of (Bainbridge et al., 1990). In our extended language, let

$$\begin{aligned}RC = \mathsf{DfnblC}^{\mathsf{abo}} \stackrel{def}{=} (a\!:\!\mathsf{List}_{\mathsf{Nat}}, b\!:\!\mathsf{List}_{\mathsf{Nat}})\ .\ (\exists f\!:\!\forall X.(\mathfrak{T}_{\mathsf{SetCE}}[X]\to X)\ .\\ \mathsf{Closed}_{In}(f)\ \wedge\ (f(\mathsf{List}_{\mathsf{Nat}})(\mathfrak{a})) = a\ \wedge\ (f(\mathsf{List}_{\mathsf{Nat}})(\mathfrak{b})) = b)\end{aligned}$$

We now have that the parametric PER-model does not satisfy $\mathfrak{a}\ (\mathfrak{T}_{\mathsf{SetCE}}[RC])\ \mathfrak{b}$, but does satisfy $\mathfrak{a}\ (\mathfrak{T}_{\mathsf{SetCE}}[RC])^{\mathsf{abo}}_{\mathsf{C}}\ \mathfrak{b}$. Theorem 5.34, or more generally Theorem 5.33, gives with spParamC,

$$\forall u, v : \exists X.\mathfrak{T}_{\mathsf{SetCE}}[X]\ .\ u\ \mathsf{SimRelC}_{\mathfrak{T}_{\mathsf{SetCE}}[X]}\ v \quad \Leftrightarrow \quad u\ \mathsf{ObsEqC}^{\{\mathsf{Nat}\}}_{\mathfrak{T}_{\mathsf{SetCE}}[X]}\ v$$

Thus, the interpretations of $(\mathsf{pack}\ \mathsf{List}_{\mathsf{Nat}}\ \mathfrak{a})$ and $(\mathsf{pack}\ \mathsf{List}_{\mathsf{Nat}}\ \mathfrak{b})$ also satisfy observationally equivalence, in the sense of Def. 5.25. ○

We also now get composability with reference to the parametric PER-model.

Theorem 5.36 (Composability of Simulation Relations) *With* spParamC *and* $\vdash^{\mathrm{CTF}}$*, for* $\mathfrak{T}[X]$ *adhering to* HADT_{Obs}*, we can derive*

$$\begin{aligned}&\forall A, B, C, R \subset A \times B, S \subset B \times C, \mathfrak{a} : \mathfrak{T}[A, \boldsymbol{Z}], \mathfrak{b} : \mathfrak{T}[B, \boldsymbol{Z}], \mathfrak{c} : \mathfrak{T}[C, \boldsymbol{Z}].\\ &\qquad \mathfrak{a}(\mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}}_{\mathsf{C}})\mathfrak{b}\ \wedge\ \mathfrak{b}(\mathfrak{T}[S, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}}_{\mathsf{C}})\mathfrak{c}\ \Rightarrow\ \mathfrak{a}(\mathfrak{T}[S \circ R, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}}_{\mathsf{C}})\mathfrak{c}\end{aligned}$$

Proof: As for Theorem 5.17, but using spParamC instead of spParam. □

For $\mathsf{SimRelC}$, we again have the close connection with components, *i.e.,* we get a version of Theorem 4.20 (*p.* 89) for data types with operations of any order.

Theorem 5.37 *Suppose* $\mathfrak{T}[X]$ *adheres to* HADT_{Obs}*. With* spParamC *and* $\vdash^{\mathrm{CTF}}$ *we derive*

$$\begin{aligned}&\forall A, B. \forall \mathfrak{a} : \mathfrak{T}[A, \boldsymbol{Z}], \mathfrak{b} : T[B, \boldsymbol{Z}])\ .\\ &\quad (\mathsf{pack} A \mathfrak{a})\ \mathsf{SimRelC}_{\mathfrak{T}[X, \mathbf{eq}_{\boldsymbol{Z}}]}\ (\mathsf{pack} B \mathfrak{b}) \quad \Leftrightarrow\ \exists R \subset A \times B\ .\ \mathfrak{a}\ \mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}}_{\mathsf{C}}\ \mathfrak{b}\end{aligned}$$

Proof: For the non-obvious direction, suppose $(\mathsf{pack} A \mathfrak{a})\ \mathsf{SimRelC}_{\mathfrak{T}[X, \mathbf{eq}_{\boldsymbol{Z}}]}\ (\mathsf{pack} B \mathfrak{b})$. Theorem 5.33 gives $(\mathsf{pack} A \mathfrak{a})\ \mathsf{ObsEqC}^{Obs}_{\mathfrak{T}[X, \boldsymbol{Z}]}\ (\mathsf{pack} B \mathfrak{b})$. Then Theorem 5.26 and Theorem 5.34 give the result. □

5.4 Equality, Transitivity and Stability

We have defined alternative simulation relations that observe the abstraction barrier provided by existential types. Hence, we have established the correspondence between simulation relations and observational equivalence also at higher order. But we have not related the new simulation relations to equality; indeed if we could do this, then that would mean by Theorem 3.5 that the new notions of simulation relation were not necessary.

5.4.1 Equality

In fact, it is not unreasonable to ask whether spParam and spParamC are strong enough to obtain a correspondence between equality at existential type and SimRelA and SimRelC. In the case of SimRelA, let $(\exists X.\mathfrak{T}[X, \mathbf{eq}_{\boldsymbol{Z}}]^\epsilon)$ be the relation defined by

$$
\begin{aligned}
&(\exists X.\mathfrak{T}[X, \mathbf{eq}_{\boldsymbol{Z}}]^\epsilon) \stackrel{def}{=} (u{:}\exists X.\mathfrak{T}[X,\boldsymbol{Z}], v{:}\exists X.\mathfrak{T}[X,\boldsymbol{Z}]) \;.\\
&\quad (\,\forall Y.\forall Z.\forall S\subset Y\times Z\;.\quad \forall f{:}\forall X.\mathfrak{T}[X,\boldsymbol{Z}]\to Y.\;\forall g{:}\forall X.\mathfrak{T}[X,\boldsymbol{Z}]\to Z\;.\\
&\qquad\qquad\qquad f\;(\forall X.\mathfrak{T}[X, \mathbf{eq}_{\boldsymbol{Z}}]^\epsilon \to S)\; g \;\Rightarrow\; (uYf)\;\; S\;\; (vZg)\,)
\end{aligned}
$$

This is just the unravelling of $(\exists X.\mathfrak{T}[X, \mathbf{eq}_{\boldsymbol{Z}}])$ but using the *abo*-relations from Sect. 5.2. We have, using standard parametricity

$$\forall u, v{:}\exists X.\mathfrak{T}[X,\boldsymbol{Z}]\;.\; u\;(\exists X.\mathfrak{T}[X, \mathbf{eq}_{\boldsymbol{Z}}]^\epsilon)\; v \quad\Leftrightarrow\quad u\;\mathsf{SimRelA}\; v$$

If it were possible to derive

$$\forall u, v{:}\exists X.\mathfrak{T}[X,\boldsymbol{Z}]\;.\; u =_{\exists X.\mathfrak{T}[X,\boldsymbol{Z}]} v \;\;\Leftrightarrow\;\; u\;(\exists X.\mathfrak{T}[X, \mathbf{eq}_{\boldsymbol{Z}}]^\epsilon)\; v$$

we would have a correspondence between equality and SimRelA. However, the derivation of the latter sequent seems to depend on the lemma

$$f\;\;(\forall X.(\mathfrak{T}[X, \mathbf{eq}_{\boldsymbol{Z}}]^\epsilon \to \mathsf{eq}_Y))\;\; f$$

for free Y. This is *not* an instance of spParam, and of course, due to the remarks in Example 5.7, this lemma cannot hold in the syntactic models of (Hasegawa, 1991) that we have considered so far. See also the comment preceding Lemma B.5 (p. 222) in Sect. B.2 in Appendix B.

Similar remarks hold for SimRelC and spParamC.

5.4.2 Transitivity

When observational equivalence and the existence of simulation relations coincide with equality, we of course get transitivity for those notions. But it is worth mentioning that transitivity holds for these notions independently of any coincidence with equality.

Theorem 5.38 *We have*

$$u\;\mathsf{ObsEq}\;v \;\wedge\; v\;\mathsf{ObsEq}\;w \;\;\Rightarrow\;\; u\;\mathsf{ObsEq}\;w$$

$$u\;\mathsf{ObsEqC}\;v \;\wedge\; v\;\mathsf{ObsEqC}\;w \;\;\Rightarrow\;\; u\;\mathsf{ObsEqC}\;w$$

Proof: From the antecedent u ObsEq $v \;\wedge\; v$ ObsEq w, we get A, B, C, D and $\mathfrak{a} : \mathfrak{T}[A], \mathfrak{b} : \mathfrak{T}[B], \mathfrak{c} : \mathfrak{T}[C], \mathfrak{d} : \mathfrak{T}[D]$ such that $u = (\mathsf{pack} A \mathfrak{a}) \;\wedge\; v = (\mathsf{pack} B \mathfrak{b})$ and $v = (\mathsf{pack} C \mathfrak{c}) \;\wedge\; w = (\mathsf{pack} D \mathfrak{d})$ such that $(f A\, \mathfrak{a}) = (f B\, \mathfrak{b})$ and $(f\, C\, \mathfrak{c}) = (f D\, \mathfrak{d})$ for any observable computation f. The result follows if $(f B\, \mathfrak{b}) = (f C\, \mathfrak{c})$. But $(f B\, \mathfrak{b}) = \mathsf{unpack}(\mathsf{pack} B \mathfrak{b}) = \mathsf{unpack}(v) = \mathsf{unpack}(\mathsf{pack} C \mathfrak{c}) = (f\, C\, \mathfrak{c})$. The case for ObsEqC is similar. □

We get the corresponding statement in terms of packages and their components.

Theorem 5.39 *We have*

$$\begin{array}{l}
\forall A, B, C, \mathfrak{a}{:}\mathfrak{T}[A], \mathfrak{b}{:}\mathfrak{T}[B], \mathfrak{c}{:}\mathfrak{T}[C]\ .\\
\qquad\qquad (\bigwedge_{D\in Obs} \forall f{:}\forall X.(\mathfrak{T}[X]\to D)\ .\ (f A\, \mathfrak{a}) = (f B\, \mathfrak{b}) \;\wedge\\
\qquad\qquad\quad \bigwedge_{D\in Obs} \forall f{:}\forall X.(\mathfrak{T}[X]\to D)\ .\ (f B\, \mathfrak{b}) = (f\, C\, \mathfrak{c}))\\
\qquad\qquad\qquad\quad \Rightarrow\; \bigwedge_{D\in Obs} \forall f{:}\forall X.(\mathfrak{T}[X]\to D)\ .\ (f A\, \mathfrak{a}) = (f\, C\, \mathfrak{c})
\end{array}$$

$$\begin{array}{l}
\forall A, B, C, \mathfrak{a}{:}\mathfrak{T}[A], \mathfrak{b}{:}\mathfrak{T}[B], \mathfrak{c}{:}\mathfrak{T}[C]\ .\\
\quad (\bigwedge_{D\in Obs} \forall f{:}\forall X.(\mathfrak{T}[X]\to D)\ .\ \mathsf{Closed}_{In}(f) \;\Rightarrow\; (f A\, \mathfrak{a}) = (f B\, \mathfrak{b}) \;\wedge\\
\qquad \bigwedge_{D\in Obs} \forall f{:}\forall X.(\mathfrak{T}[X]\to D)\ .\ \mathsf{Closed}_{In}(f) \;\Rightarrow\; (f B\, \mathfrak{b}) = (f\, C\, \mathfrak{c}))\\
\qquad \Rightarrow\; \bigwedge_{D\in Obs} \forall f{:}\forall X.(\mathfrak{T}[X]\to D)\ .\ \mathsf{Closed}_{In}(f) \;\Rightarrow\; (f A\, \mathfrak{a}) = (f\, C\, \mathfrak{c})
\end{array}$$

Proof: This follows from Theorem 5.38 using Theorem 4.5 and Theorem 5.26, but of course the theorem also follows trivially by definition. □

Theorem 5.40 *With* SPPARAM SPPARAMC *and* $\vdash^{\text{CTF}}$, *for* $\mathfrak{T}[X]$ *adhering to* HADT_{Obs}, *for* $u, v, w{:}\exists X.\mathfrak{T}[X]$, *we can derive*

$$u \;\mathsf{SimRelA}\; v \;\wedge\; v \;\mathsf{SimRelA}\; w \;\Rightarrow\; u \;\mathsf{SimRelA}\; w$$

$$u \;\mathsf{SimRelC}\; v \;\wedge\; v \;\mathsf{SimRelC}\; w \;\Rightarrow\; u \;\mathsf{SimRelC}\; w$$

Proof: This follows from Theorem 5.38, by Theorem 5.14 and Theorem 5.33 □

Because of Theorem 5.18 and Theorem 5.37, Theorem 5.40 also follows from composability, *i.e.,* Theorem 5.17 and Theorem 5.36. For the standard notion of simulation relation, this is not the case for data types with higher-order operations, *i.e.,* Theorem 5.4 neither infers nor follows from the composability or transitivity of existence of simulation relations in terms of data type components, *cf.* the discussion around Theorem 5.4 (*p.* 114).

5.4.3 Stability

For first-order signatures, we have by parametricity that constructors are inherently stable, if we are content with the notion of observational equivalence ObsEq of Def. 4.4 (*p.* 78). This is a corollary of the fact that equality at existential type then coincides with observational equivalence (Theorem 4.19).

However, for higher-order signatures the two-way link to equality is lost. In this case, we have to show the stability of constructors explicitly. As mentioned in (Sannella and Tarlecki, 1997) stability is a trait of the programming language as a whole and can be verified once and for all.

Here this is simple. In fact, stability follows independently of the link between observational equivalence and equality. The only assumption we make is that the observable types of the implemented specification are a subset of the observable types of the implementing specification. This is a sensible assumption for refinement scenarios, see also *e.g.,* (Honsell et al., 2000). For closed observations, we also assume the schemata CTF and calculus $\vdash^{\text{CTF}}$ in Def. 5.23.

Theorem 5.41 (Stability) *We have with* $\vdash^{\text{CTF}}$*, assuming* $\mathit{Obs} \subseteq \mathit{Obs}'$*,*

$$\forall F\!:\!\exists X.\mathfrak{T}'[X,\boldsymbol{Z}]\!\rightarrow\!\exists X.\mathfrak{T}[X,\boldsymbol{Z}].\quad \forall u,v\!:\!\exists X.\mathfrak{T}[X,\boldsymbol{Z}]\ .\ u\ \mathsf{ObsEqC}^{\mathit{Obs}'}\ v\ \Rightarrow\ Fu\ \mathsf{ObsEqC}^{\mathit{Obs}}\ Fv$$

Proof: We must derive

$$\exists A,B.\exists\mathfrak{a}\!:\!\mathfrak{T}[A,\boldsymbol{Z}],\mathfrak{b}\!:\!\mathfrak{T}[B,\boldsymbol{Z}]\ .\ Fu=(\mathsf{pack}A\mathfrak{a})\ \wedge\ Fv=(\mathsf{pack}B\mathfrak{b})\ \wedge \bigwedge_{D\in\mathit{Obs}}\forall f\!:\!\forall X.(\mathfrak{T}[X,\boldsymbol{Z}]\!\rightarrow\! D)\ .\ \mathsf{Closed}_{\mathit{In}}(f)\ \Rightarrow\ (f\,A\,\mathfrak{a})=(f\,B\,\mathfrak{b})$$

From Theorem 3.6 we get

$$\exists A,B.\exists\mathfrak{a}\!:\!\mathfrak{T}[A,\boldsymbol{Z}],\mathfrak{b}\!:\!\mathfrak{T}[B,\boldsymbol{Z}]\ .\ Fu=(\mathsf{pack}A\mathfrak{a})\ \wedge\ Fv=(\mathsf{pack}B\mathfrak{b})$$

From $u\ \mathsf{ObsEqC}^{\mathit{Obs}'}\ v$ we get

$$\exists A',B'.\exists\mathfrak{a}'\!:\!\mathfrak{T}'[A',\boldsymbol{Z}],\mathfrak{b}'\!:\!\mathfrak{T}'[B',\boldsymbol{Z}]\ .\ u=(\mathsf{pack}A'\mathfrak{a}')\ \wedge\ v=(\mathsf{pack}B'\mathfrak{b}')\ \wedge \bigwedge_{D\in\mathit{Obs}'}\forall f'\!:\!\forall X'.(\mathfrak{T}'[X',\boldsymbol{Z}]\!\rightarrow\! D)\ .\ \mathsf{Closed}_{\mathit{In}'}(f')\ \Rightarrow\ (f'A'\,\mathfrak{a}')=(f'B'\,\mathfrak{b}')$$

For any $f\!:\!\forall X.(\mathfrak{T}[X,\boldsymbol{Z}]\!\rightarrow\! D)$, where $D\in\mathit{Obs}\subseteq\mathit{Obs}'$, let

$$f'\stackrel{\mathit{def}}{=}\Lambda X'.\lambda\mathfrak{x}'\!:\!\mathfrak{T}'[X',\boldsymbol{Z}]\ .\ \mathsf{unpack}(F(\mathsf{pack}X'\mathfrak{x}'))(D)(f)$$

Then from the above, we get in particular that

$$(f\,A\,\mathfrak{a})=(f'A'\,\mathfrak{a}')=(f'B'\,\mathfrak{b}')=(f\,B\,\mathfrak{b})$$

as required. We must also check that the closedness requirement is upheld, *i.e.,* we must have $\mathsf{Closed}_{In'}(f')$. But this follows from $\mathsf{Closed}_{In}(f)$ using $\vdash^{\text{CTF}}$, since $In \subseteq In'$ by our assumption that input types are the same as output types. □

The proof of Theorem 5.41 shows that System F constructors are indeed stable under any of our notions of observational equivalence, *i.e.,* we can adapt the theorem to use ObsEq rather than ObsEqC and then omit the reference to $\vdash^{\text{CTF}}$. When the correspondence between observational equivalence and equality exists, then stability can be seen to be inherent because of this, but here we see that stability is fundamental and independent of this correspondence.

5.5 Summary and Further Work

In this chapter we dealt primarily with higher-order signatures and the subsequent loss of both the composability of simulation relations, and the correspondence between observational equivalence and the existence of a simulation relation. We developed two variants of the notion of abstraction barrier-observing simulation relation that reflect the aspect ***Abs-Bar*** of the abstraction barrier supplied by existential types. The first variant relates to syntactic models, while the second can be used when relating to non-syntactic models. The second variant thus allows future investigations into the semantic meaning of the type-theoretical notions we develop in this thesis.

Then at any order, observational equivalence corresponds to the existence of abstraction barrier-observing simulation relations. This means that the notion of abstraction barrier-observing simulation relations bears exactly the desired strength for explaining observational refinement. This connection is methodologically relevant, because observational equivalence is generally not easily shown directly, whereas the construction of simulation relations (abstraction-barrier observing, or not) is based on the given data type operations.

The methodological remark that finding a simulation relation, or alternatively, finding an abstraction barrier-observing simulation relation, is easier than proving observational equivalence directly, needs commenting. The remark certainly bears much truth on the semantic level. However, these considerations do not necessarily apply to the logic since this involves issues of completeness. For instance, the present logic is not strong enough to derive, relating to Example 5.7,

$$(\mathsf{pack}\ \mathsf{List}_{\mathsf{Nat}}\ \mathfrak{a})\ \mathsf{SimRelA}_{\mathfrak{T}_{\mathsf{SetCE}}[X]}\ (\mathsf{pack}\ \mathsf{List}_{\mathsf{Nat}}\ \mathfrak{b})$$

Thus, although we have made proving observational equivalence more tractable in principle by providing the option of exhibiting *abo*-simulation relations when there is no standard simulation relation, we have in methodological terms not come far enough. What we lack are (semantic) proof heuristics manifested in the logic, for establishing relations as simulation relations. One might also include the ω-rule (Sect. A.6 in Appendix A) in conjunction with such heuristics. All this requires semantic validation, and would, if the ω-rule is included, restrict attention to syntactic models. We will not go deeper into heuristics in this thesis. The focus is instead on developing the *connection* between notions of simulation relations and observational equivalence.

Due to a seemingly orthogonal demands, the solution we presented for the Closed predicates is two-tiered; we have a proper predicate expressing closedness, and when needed, we may derive closedness using a predicate-like symbol and an external calculus. In this manner, we can have the cake and eat parts of it too. But preferably one would want a predicate expressing closedness that can be used for deriving closedness while remaining a proper predicate. This seems difficult to achieve due to the intentional aspects involved in describing closedness. Nevertheless, a possible alternative that might lead to success is to use a Kripke-style semantics. This would however require a reworking of the subsequent results in order to relate to this semantics.

Chapter 6

General Specification Refinement

In Sect. 2.7 we outlined the universal proof method for proving observational refinements formalised in (Bidoit et al., 1995; Bidoit and Hennicker, 1996; Bidoit et al., 1997), and in Sect. 4.4 we imported this method into the type-theoretic context of System F and relational parametricity. The results in Sect. 4.4 are for data types with first-order operations. We now generalise the universal proof method to handle higher-order operations in data types. There are two issues: At higher order, constructing data types over quotients and subobjects is not as straightforward as it is at first order. Secondly, as we have seen in Ch. 5, it is in general not the case that standard simulation relations can describe data refinement at higher order. We should therefore provide a version of the imported proof strategy adapted to speak in terms of abstraction barrier-observing simulation relations.

6.1 Introduction

We are in the process of generalising the concepts of specification refinement established in the type theory at first order, to data types with operations that may be higher-order and polymorphic. The higher-order aspect presents challenges. Firstly, as we have seen in Ch. 5, the formal link between the existence of simulation relations and observational equivalence breaks down, and we lose composability of simulation relations. By analysing the existential-type abstraction barrier, one can devise an abstraction barrier-observing notion of simulation relation in the logic. This notion composes at higher-order, and re-establishes the correspondence with observational equivalence. All this was done in Ch. 5. The second challenge, and the topic of this chapter, arises when attempting to import the proof method of Bidoit *et al.* at higher order. The basis of the problem is the difficulty of constructing data types over quotients and subobjects at higher order, but additionally, we must adapt the proof method to take into consideration abstraction barrier-observing simulation relations.

We need to put things into perspective for a moment. When using the proof method formalised by Bidoit *et al.* to show refinements, we only use one direction of the correspondence between simulation relations and observational equivalence, *cf.* Examples 4.26 (*p.* 93) and 4.27 (*p.* 96), and also Theorems 4.31, 4.32 and 4.33. That is, using Sub and Quot (*p.* 92) we get the existence of simulation relations, and then we use that this implies observational equivalence. Ostensibly, we do not need the converse direction, and since the existence of simulation relations implies observational equivalence at any order through relational parametricity, the results in Ch. 5 would seem superfluous for proving refinements.

This is not the whole story, of course. First of all, it is desirable to have the composability of simulation relations, and this is in general lost at higher order, but is firmly reestablished in Ch. 5 using the abstraction barrier-observing notion of simulation relation relation. Secondly, at higher order it may not be possible to show the existence of standard simulation relations. In particular it may not be possible to obtain the required simulation relations postulated by Sub and Quot. Example 6.24 below illustrates this. This means that we shall need versions of Sub and Quot that make use of abstraction barrier-observing simulation relations. These are presented as SubG and QuotG below. The two-way correspondence between abstraction barrier-observing simulation relations and observational equivalence shown in Ch. 5 then guarantees that abstraction barrier-observing simulation relations are indeed exactly what one wants.

We thus have two possible scenarios for showing an observational refinement for data types with higher-order operations. If we can exhibit standard simulation relations using Sub and Quot, this is fine and and we get observational equivalence as before, depicted as follows.

$$\begin{tikzcd}[row sep=small] \textsc{Quot} \arrow[dr] & & \\ & \mathsf{SimRel} \arrow[r, phantom, "\Rightarrow"] & \mathsf{ObsEq} \\ \textsc{Sub} \arrow[ur] & & \end{tikzcd}$$

If this is not possible, we can instead try exhibiting abstraction barrier-observing simulation relations using SubG and QuotG, and then use the results of Ch. 5 to get observational equivalence. This gives the scenario for observational refinement at any order depicted as follows.

$$\begin{tikzcd}[row sep=small] \textsc{QuotG} \arrow[dr] & & \\ & \mathsf{SimRelC} \arrow[r, phantom, "\Rightarrow"] & \mathsf{ObsEqC} \\ \textsc{SubG} \arrow[ur] & & \end{tikzcd}$$

To import the proof method of Bidoit *et al.* is in our scheme to show soundness of the logic augmented by axioms postulating the existence of subobjects and quotients. We have already done this for Sub and Quot for first-order profiles by showing that the axioms hold in the parametric PER-model.

We do not know whether or not Sub and Quot for data types with higher-order operations hold in the parametric PER-model. The difficulty in the PER model is to construct higher-order operations over quotients and subobjects. It is conjectured in (Zwanenburg, 1999) but not proven, that Quot and a more general version of Sub are at least sound w.r.t. some model. We are here able to give proof that the logic augmented by Sub and Quot is sound for datatype operations of any order. This we do by showing that Sub and Quot hold in a parametric setoid model. Instead of interpreting types as PERs, one now interprets them as a pair consisting of a PER together with a relation on that PER giving the equality of the interpreted type. This then more or less directly gives us the ability to construct quotient and subobject data types at any order.

The axioms Sub and Quot speak in terms of standard simulation relations, which are of limited use at higher order. To import the proof method fully for higher-order profiles, we formulate axioms SubG and QuotG, versions of Sub and Quot that make use of the abstraction barrier observing simulation relation developed in Ch. 5. Showing soundness of the logic augmented with these axioms involves the above difficulty of constructing higher-order operations over quotients and subobjects, but with the added challenge of the context of abstraction barrier-observing simulation relations. The setoid semantics above only solves the problem in the context of standard simulation relations.

Our solution to this again follows the core idea in this thesis. One can observe the same phenomenon that we saw for simulation relations: The standard formalisms do not accommodate the abstraction barrier in existential types, as described in particular by *Abs-Bar*. Starting from this, we extend the parametric PER-model interpretation in such a way as to observe data type abstraction barriers more closely. The result is a special *data type semantics* for use inside data types. This gives an interpretation for encapsulated operations that mirrors their applicability according to *Abs-Bar*. This in turn allows us to construct quotient and subobject data types at any order, in the context of abstraction barrier observing simulation relations. The idea behind this approach is a natural continuation of the development so far, in that we directly use *Abs-Bar* as a guidance in developing more suitable notions. Using the data type semantics, we can show the soundness of the logic augmented by SUBG and QUOTG. We develop data type semantics w.r.t. the PER model. It is also possible to formulate a data type semantics w.r.t. the setoid model.

—

We will present the solutions in the order indicated above; we first present the setoid semantics in Sect. 6.2 and then in Sect. 6.4 we develop data type semantics. The data type semantics do not depend on the setoid semantics. The setoid semantics will be presented as an instance of a more general scheme, *i.e.,* we will give certain axiomatic criteria for when a model has subobject and quotient maps, and then show that the setoid model satisfies these.

As an initial motivation, suppose we attempt to show that QUOT with higher-order operations holds in the parametric PER-model. For any PER $\mathcal{X}$ and element x of $[\![X \rhd \mathfrak{T}[X]]\!]_{[X\mapsto\mathcal{X}]}$, we should display a quotienting PER $\mathcal{Q}$ and element q of $[\![X \rhd \mathfrak{T}[X]]\!]_{[X\mapsto\mathcal{Q}]}$ with the desired characteristics. Quotients over an algebra A are usually constructed directly from A with the new operations derived from the original operations. The natural approach to constructing the operations of q in a PER model is to use the same realisers that give the operations of x.

At first order this is straight-forward; but at higher order it is not. Suppose $x = \langle [e_0]_{\mathcal{X}}, [e_1]_{\mathcal{X}\to\mathcal{X}}, [e_2]_{(\mathcal{X}\to\mathcal{X})\to\mathcal{X}}, \ldots\rangle$. We would like to simply define q as $\langle [e_0]_{\mathcal{Q}}, [e_1]_{\mathcal{Q}\to\mathcal{Q}}, [e_2]_{(\mathcal{Q}\to\mathcal{Q})\to\mathcal{Q}}, \ldots\rangle$. For this we must check that the indicated equivalence classes actually exist. To show $e_2\ ((\mathcal{Q}\to\mathcal{Q})\to\mathcal{Q})\ e_2$, we must show $n\ (\mathcal{Q}\to\mathcal{Q})\ n\ \Rightarrow\ e_2(n)\!\downarrow$. Now, this does not follow from the running assumption $n\ (\mathcal{X}\to\mathcal{X})\ n\ \Rightarrow\ e_2(n)\!\downarrow$; even if $n\ (\mathcal{Q}\to\mathcal{Q})\ n$ we may not have $n\ (\mathcal{X}\to\mathcal{X})\ n$.

Example 6.1 We consider sequences over $\mathbb{N}$. We will write the encoding of a sequence as the sequence itself. Consider now a function *rfi* on $\mathbb{N}$ that takes an encoding of a sequence and returns the encoding of the sequence with the first item repeated, *e.g.,* $rfi(123) = 1123$. Define the PER $\mathcal{L}ist$ by

$$n \; \mathcal{L}ist \; m \;\; \Leftrightarrow \;\; n \text{ and } m \textit{ encode the same list}$$

Now consider the PERs $\mathcal{B}ag$ and $\mathcal{S}et$ defined as follows

$$n \; \mathcal{B}ag \; m \;\; \Leftrightarrow \;\; n \text{ and } m \textit{ encode the same list, modulo permutation}$$

$$n \; \mathcal{S}et \; m \;\; \Leftrightarrow \;\; n \text{ and } m \textit{ encode the same list, modulo permutation and repetition}$$

Now, *rfi* is a realiser for a map $f_{rfi} : \mathcal{S}et \to \mathcal{S}et$, but is not a realiser for any map in $\mathcal{B}ag \to \mathcal{B}ag$, *i.e.,* we have $rfi \; (\mathcal{S}et \to \mathcal{S}et) \; rfi$ but not $rfi \; (\mathcal{B}ag \to \mathcal{B}ag) \; rfi$. Thus, even if we assume that $rfi \; (\mathcal{B}ag \to \mathcal{B}ag) \; rfi \;\Rightarrow\; e(rfi)\downarrow$ for some e, and also that $rfi \; (\mathcal{S}et \to \mathcal{S}et) \; rfi$, this information on its own is useless for showing $e(rfi)\downarrow$. ○

When establishing QUOT, one has the additional assumption of $\mathsf{x} \; [\![\mathfrak{T}[R]]\!]_{[R \mapsto \Re]} \; \mathsf{x}$, where $\Re$ is the quotienting relation. From this we get, for e a component of x, $[e]_{(\mathcal{X} \to \mathcal{X}) \to \mathcal{X}} \; ((\Re \to \Re) \to \Re) \; [e]_{(\mathcal{X} \to \mathcal{X}) \to \mathcal{X}}$, or written in elementary terms, $n \; (\Re \to \Re) \; m \;\Rightarrow\; e(n) \; \Re \; e(m)$. Seemingly, this gives $e \; ((\mathcal{Q} \to \mathcal{Q}) \to \mathcal{Q}) \; e$, but this is not the case because we again do not know if $[n]_{\mathcal{X} \to \mathcal{X}}$ and $[m]_{\mathcal{X} \to \mathcal{X}}$ exist.

Similar difficulties arise for subobjects.

6.2 Criteria giving Subobject and Quotient Maps

We here give axiomatic criteria sufficient to ensure higher-order subobject and quotient maps based on ideas in (Hofmann, 1995a). At the end we present the setoid model as an instance. This will show soundness of the logic augmented by SUB and QUOT for higher-order signatures according to HADT_{Obs} (*p.* 113).

6.2.1 Higher-Order Quotient Maps

The trouble with higher-order quotients is illustrated by the following: For a given map $f : X/R \to X/R$, there need not exist a map $g : X \to X$ such that for all $x : X$, $[g(x)] = f([x])$, *i.e.,* $epi(g(x)) = f(epi(x))$, where $epi : X \to X/R$ maps an element to its equivalence class. In particular this is the case for the parametric PER model. This is illustrated in Example 6.1 and is the basic difficulty motivating this chapter. The axiom of choice AC (*p.* 92) gives such a map g, because we can then define an inverse for epi, and the desired g is then given by $\lambda x : X. epi^{-1}(f \, epi(x))$.

The axiom of choice does not hold in the parametric PER-model, nor in the model we will look at in this section. However, we shall look at a weaker condition derived from (Hofmann, 1995a) that suffices to give higher-order quotients.

We omit polymorphism in data types from the discussion, that is, we shall assume $\mathtt{HADT}_{Obs}$ (*p.* 113). Accordingly, we consider arbitrary order arrow types, composed over types $U_0, U_1, \ldots$, where any U_i is either X or some $D \in Obs$. From this, define the families U^i by

$$\begin{array}{lcl} U^0 & = & U_0 \\ U^{i+1} & = & (U^i) \to U_{i+1} \end{array}$$

For example, $U^2 = ((U_0 \to U_1) \to U_2)$. Now for a given $U = U^n$, define $Q(U)^i$ for some R as follows,

$$\begin{array}{lcl} Q(U)^0 & = & U_0 \\ Q(U)^1 & = & U_0/R \to U_1 \\ Q(U)^{i+1} & = & (Q(U)^{i-1} \to U_i/R) \to U_{i+1}, \quad 1 \leq i \leq n-1 \end{array}$$

where, if U_i is X then $U_i/R = X/R$, and if U_i is $D \in Obs$ then $U_i/R = D$. For example $Q(U^2)^2 = ((U_0 \to U_1/R) \to U_2)$. In any $Q(U)^i$, we have that quotients U_j/R occur only negatively. Finally, for a given $U = U^n$, define $R(U)^i$ as follows for some R. We denote identities by their domains,

$$\begin{array}{lcl} R(U)^0 & = & R_0 \\ R(U)^1 & = & U_0/R \to R_1 \\ R(U)^{i+1} & = & (R(U)^{i-1} \to U_i/R) \to R_{i+1}, \quad 1 \leq i \leq n-1 \end{array}$$

where, if U_i is X then $R_i = R$, and if U_i is $D \in Obs$ then $R_i = D$, *i.e.,* the identity on D. In any $R(U)^i$, we have that R occurs positively, and identities U_j/R occur only negatively. The point of all this is that, if R is an equivalence relation on X, then $R(U)^i$ is an equivalence relation on $Q(U)^i$. This means that we may form the quotient $Q(U)^i/R(U)^i$. For example, consider a particular $U = U^1 = X \to X$. Then $Q(U)^1 = X/R \to X$ and $R(U)^1 = X/R \to R$, and $X/R \to R$ is an equivalence relation on $X/R \to X$. Note on the other hand that $R \to X/R$ is not necessarily an equivalence relation on $X \to X/R$. However, $(R \to X/R) \to R$ is an equivalence relation on $(X \to X/R) \to X$, that is, $R(U^2)^2$ is an equivalence relation on $Q(U^2)^2$, for $U^2 = (X \to X) \to X$. Now consider the relation

$$\mathit{graph}(\mathsf{epi}) \stackrel{\mathit{def}}{=} (x\!:\!X, q\!:\!X/R)\ .\ ((\mathsf{epi}\, x) =_{X/R} q)$$

where the map $\mathsf{epi} : X \to X/R$ maps elements to their R-equivalence class. We define the relation $graph(\mathsf{epi})(U)^n$ by

$$\begin{array}{lcl} graph(\mathsf{epi})(U)^0 & = & graph(\mathsf{epi}) \\ graph(\mathsf{epi})(U)^1 & = & U_0/R \to graph(\mathsf{epi}) \\ graph(\mathsf{epi})(U)^{i+1} & = & (graph(\mathsf{epi})(U)^{i-1} \to U_i/R) \to graph(\mathsf{epi})(U)_{i+1}, \\ & & \qquad\qquad\qquad\qquad 1 \leq i \leq n-1 \end{array}$$

where, if U_i is X then $graph(\mathsf{epi})(U)_i = graph(\mathsf{epi})(U)$, and if U_i is $D \in Obs$ then $R_i = D$, *i.e.,* the identity on D.

A sufficient condition for obtaining quotients at higher types is now

***Quot-Arr*:** For R an equivalence relation on X, and any given $U = U^n$,

$$Q(U)^n/R(U)^n \;\cong\; U[(U_i/R)/U_i]$$

where the isomorphism $\mathsf{iso} : U[(U_i/R)/U_i] \to Q(U)^n/R(U)^n$ is such that any f in the equivalence class $\mathsf{iso}(\beta)$ is such that

$$f\ (graph(\mathsf{epi})(U)^n)\ \beta$$

where $U[(U_i/R)/U_i]$ denotes U with every U_i, $0 \leq i \leq n$ replaced by U_i/R.

Note that *Quot-Arr* is not an axiom in our logic; we do not have quotient types. Rather, *Quot-Arr* is here a condition that we can check in the models we are considering, and in which the terminology concerning quotients has a well-defined meaning. In the setting of (Hofmann, 1995a), *Quot-Arr* is expressible in the logic, and it follows from results herein that *Quot-Arr* is strictly weaker than the axiom of choice. This is shown by establishing *Quot-Arr* in a syntactic setoid model for which AC fails. The model we will present is a semantic analogue to this model.

Let us exemplify why *Quot-Arr* suffices. The challenge that motivates this entire chapter is higher-order operations in data types, and then the soundness of Quot and Sub where $\mathfrak{T}[X]$ has higher-order operation profiles. To illustrate the use of *Quot-Arr* in obtaining Quot, let X be any type and suppose $\mathfrak{T}[X]$ has a profile $g : (X \to X) \to X$. Suppose that $R \subset X \times X$ is an equivalence relation. Consider now any $\mathfrak{x} : \mathfrak{T}[X]$. Under the assumption that $\mathfrak{x}\ (\mathfrak{T}[R])\ \mathfrak{x}$, we must produce a $\mathfrak{q} : \mathfrak{T}[X/R]$, such that $\mathfrak{x}\ (\mathfrak{T}[graph(\mathsf{epi})])\ \mathfrak{q}$. For $\mathfrak{x}.g : (X \to X) \to X$, this involves finding a $\mathfrak{q}.g : (X/R \to X/R) \to X/R$, such that

$$\mathfrak{x}.g\ ((graph(\mathsf{epi}) \to graph(\mathsf{epi})) \to graph(\mathsf{epi}))\ \mathfrak{q}.g$$

Consider now the following instance of *Quot-Arr*.

Quot-Arr$_1$**:** For R an equivalence relation on X,

$$(X/R \to X)/(X/R \to R) \;\cong\; (X/R \to X/R)$$

With *Quot-Arr*$_1$ we can construct the following commuting diagram.

$$\begin{tikzcd}
(X/R \to X) \arrow[r, "\mathsf{epi} \to X"] \arrow[d, "\mathsf{epi}_{X/R \to X}"'] & (X \to X) \arrow[r, "\mathfrak{x}.g"] & X \arrow[r, "\mathsf{epi}"] & X/R \\
(X/R \to X)/(X/R \to R) \arrow[urrr, "lift(\mathsf{epi} \circ \mathfrak{x}.g \circ (\mathsf{epi} \to X))"'] & & & \\
X/R \to X/R \arrow[u, "\mathsf{iso}"] & & &
\end{tikzcd}$$

where $\mathsf{epi} \to X$ maps any $f : X/R \to X$ to $\lambda a : X.f(\mathsf{epi}\, a)$. Then, the sought-after $\mathsf{q}.g : (X/R \to X/R) \to X/R$ is given by $lift(\mathsf{epi} \circ \mathfrak{x}.g \circ (\mathsf{epi} \to X)) \circ \mathsf{iso}$. Here *lift* is the operation that lifts any $\alpha : X \to Y$ to $lift(\alpha) : X/\!\sim \to Y$, given an equivalence relation $\sim$ on X, provided that α satisfies $x \sim y \;\Rightarrow\; \alpha x = \alpha y$ for all $x, y : X$. Then, $lift(\alpha)$ is the map satisfying $lift(\alpha) \circ \mathsf{epi} = \alpha$. To be able to lift $\mathsf{epi} \circ \mathfrak{x}.g \circ (\mathsf{epi} \to X)$ in this way, we must check that $\mathsf{epi} \circ \mathfrak{x}.g \circ (\mathsf{epi} \to X)$ satisfies $f\ (\mathsf{eq}_{X/R} \to R)\ f' \;\Rightarrow\; (\mathsf{epi} \circ \mathfrak{x}.g \circ (\mathsf{epi} \to X))(f) =_{X/R} (\mathsf{epi} \circ \mathfrak{x}.g \circ (\mathsf{epi} \to X))(f')$, for all $f, f' : (X/R \to X)$. Assuming $f\ (\mathsf{eq}_{X/R} \to R)\ f'$, we immediately get that $(\mathsf{epi} \to X)(f)\ (R \to R)\ (\mathsf{epi} \to X)(f')$. Since we are assuming $\mathfrak{x}\ \mathfrak{T}[R]\ \mathfrak{x}$, which in particular gives $\mathfrak{x}.g\ ((R \to R) \to R)\ \mathfrak{x}.g$, the result follows.

The general form of this diagram for any given $U = U^n$ and U_c, is

$$\begin{tikzcd}
Q(U)^n \arrow[r, "\mathsf{epi}(U)^n"] \arrow[d, "\mathsf{epi}_{Q(U)^n}"'] & U_n \arrow[r, "\mathfrak{x}.g"] & U_c \arrow[r, "\mathsf{epi}"] & U_c/R \\
Q(U)^n/R(U)^n \arrow[urrr, "lift(\mathsf{epi} \circ \mathfrak{x}.g \circ (\mathsf{epi}(U)^n))"'] & & & \\
U[(U_i/R)/U_i] \arrow[u, "\mathsf{iso}"] & & &
\end{tikzcd}$$

where for a given $U = U^n$, we define $\mathsf{epi}(U)^i$ as follows. First, for any $f : B \to A$ and $g : C \to D$, the map $f \to g : (A \to C) \to (B \to D)$ is given by

$$\lambda h : (A \to C).\lambda b : B\ .\ g(h(fb))$$

Then we define, where identities are denoted by their domains,

$$
\begin{array}{lcl}
\mathsf{epi}(U)^0 & = & U_0 \\
\mathsf{epi}(U)^1 & = & (\mathsf{epi} \to U_1) \\
\mathsf{epi}(U)^{i+1} & = & (\mathsf{epi}(U)^{i-1} \to \mathsf{epi}) \to U_{i+1}, \quad 1 \leq i \leq n-1
\end{array}
$$

When relating to a particular model, we have to check that ***Quot-Arr*** holds, and also as mentioned earlier, that the terminology used concerning quotients makes sense. If this goes through, we have succeeded in constructing data types over quotients at higher order, according to Quot.

6.2.2 Higher-Order Subobject Maps

A similar story applies to subobjects. For any predicate P on X, we write $R_P(X)$ for the subobject of X containing only those $x{:}X$ such that $P(x)$ is satisfied. We construct a binary relation from P, also denoted P, by

$$P \stackrel{def}{=} (x{:}X, y{:}X) \;.\; (x =_X y \;\wedge\; P(x))$$

The binary version is for use in arrow-type relations.

Now for a given $U = U^n$, define $S(U)^i$ for some P as follows

$$
\begin{array}{lcl}
S(U)^0 & = & R_P(U_0) \\
S(U)^1 & = & U_0 \to R_P(U_1) \\
S(U)^{i+1} & = & (S(U)^{i-1} \to U_i) \to R_P(U_{i+1}), \quad 1 \leq i \leq n-1
\end{array}
$$

where, if U_i is X then $R_P(U_i) = R_P(X)$, and if U_i is $D \in \mathit{Obs}$ then $R_P(U_i) = D$. For example $S(U^2)^2 = ((R_P(U_0) \to U_1) \to R_P(U_2))$. In any $S(U)^i$, we have that subobjects $R_P(U_j)$ occur only positively. For a given $U = U^n$ define $P(U)^i$ as follows for some P. We denote identities by their domains.

$$
\begin{array}{lcl}
P(U)^0 & = & R_P(U_0) \\
P(U)^1 & = & P_0 \to R_P(U_1) \\
P(U)^{i+1} & = & (P(U)^{i-1} \to P_i) \to R_P(U_{i+1}), \quad 1 \leq i \leq n-1
\end{array}
$$

where, if U_i is X then $P_i = P$, and if U_i is $D \in \mathit{Obs}$ then $P_i = D$, *i.e.*, the identity on D. In any $P(U)^i$, we have that P occurs negatively, and identities $R_P(U_i)$ occur only positively.

Intuitively, one would think that for any given $U = U^n$, we should now postulate an isomorphism between $R_{P(U)^n}(S(U)^n)$ and $U[(R_R(U_i))/U_i]$. This would be in dual analogy to ***Quot-Arr***. However, this isomorphism does not exist even in the setoid model we will be looking at. For example, we will not be able to find an isomorphism between $R_{P\to R_P(X)}((X\to R_P(X))$ and $R_P(X)\to R_P(X)$. However, it turns out that we can in fact use an outermost quotient instead of subobjects for the isomorphism, in the same way as we did for *Quot-Arr*. The quotient itself holds the duality, as it were.

Thus, if P is a predicate on X, then $P(U)^i$ is an equivalence relation on $S(U)^i$. This means that we may form the quotient $S(U)^i/P(U)^i$. For example, consider a particular $U = U^1 = X \to X$. Then $S(U)^1 = X \to R_R(X)$ and $P(U)^1 = P\to R_P(X)$, and $P\to R_P(X)$ is an equivalence relation on $X\to R_R(X)$. Again, note on the other hand that $R_P(X)\to P$ is not necessarily an equivalence relation on $R_P(X) \to X$. However, $(R_P(X) \to P) \to R_P(X)$ is an equivalence relation on $(R_P(X)\to X)\to R_P(X)$, that is, $P(U^2)^2$ is an equivalence relation on $S(U^2)^2$, for $U^2 = (X\to X)\to X$.

Now consider the relation

$$graph(\mathsf{mono}) \stackrel{\text{def}}{=} (x:X, s:R_P(X))\ .\ (x =_X (\mathsf{mono}\, s))$$

for $\mathsf{mono}\colon R_P(X)\to X$ mapping elements to their correspondents in X. We define the relation $graph(\mathsf{mono})(U)^n$ by

$$\begin{array}{lcl} graph(\mathsf{mono})(U)^0 & = & graph(\mathsf{mono}) \\ graph(\mathsf{mono})(U)^1 & = & graph(\mathsf{mono})\to R_P(U_1) \\ graph(\mathsf{mono})(U)^{i+1} & = & (R_P(U_{i-1})\to graph(\mathsf{mono})(U)^i)\to R_P(U_{i+1}), \\ & & \hfill 1\leq i\leq n-1 \end{array}$$

where, if U_i is X then $graph(\mathsf{mono})(U)_i = graph(\mathsf{mono})(U)$, and if U_i is $D \in Obs$ then $R_i = D$, *i.e.,* the identity on D.

A sufficient condition for obtaining subobjects at higher types is now

Sub-Arr: For P a predicate on X, and any given $U = U^n$,

$$S(U)^n/P(U)^n \;\cong\; U[(R_P(U_i))/U_i]$$

where the isomorphism $\mathsf{iso} : U[(R_P(U_i))/U_i] \to S(U)^n/P(U)^n$ is such that any f in the equivalence class $\mathsf{iso}(\beta)$ is such that

$$f\ (graph(\mathsf{mono})(U)^n)\ \beta$$

where $U[(R_P(U_i))/U_i]$ denotes U with every U_i, $0\leq i\leq n$ replaced by $R_P(U_i)$.

Again, *Sub-Arr* is not an axiom in our logic, but is a condition that we can check for models in which the terminology in *Sub-Arr* has a well-defined meaning.

Let us illustrate how *Sub-Arr* gives subobjects at higher order. Again let X be any type and suppose $\mathfrak{T}[X]$ has a profile $g : (X \to X) \to X$. Consider now any $\mathfrak{x} : \mathfrak{T}[X]$. Now assume that $\mathfrak{x}\ \mathfrak{T}[P]\ \mathfrak{x}$. We must exhibit a $\mathfrak{s}_\mathfrak{a} : \mathfrak{T}[R_P(X)]$, such that $\mathfrak{x}\ \mathfrak{T}[\mathit{graph}(\mathsf{mono})]\ \mathfrak{s}_\mathfrak{a}$. For our $\mathfrak{x}.g : (X \to X) \to X$, this involves finding a map $\mathfrak{s}_\mathfrak{a}.g : (R_P(X) \to R_P(X)) \to R_P(X)$, such that

$$\mathfrak{x}.g\ ((\mathit{graph}(\mathsf{mono}) \to \mathit{graph}(\mathsf{mono})) \to \mathit{graph}(\mathsf{mono}))\ \mathfrak{s}_\mathfrak{a}.g$$

Consider now the following instance of *Sub-Arr*.

***Sub-Arr*$_1$:** For a predicate P on X,

$$(X \to R_P(X))/(P \to R_P(X)) \ \cong\ R_P(X) \to R_P(X)$$

Using *Sub-Arr*$_1$, we can construct the following commuting diagram.

$$\begin{tikzcd}
(X \to R_P(X)) \arrow[r, "X \to \mathsf{mono}"] \arrow[d, "\mathsf{epi}_{X \to R_P(X)}"'] & (X \to X) \arrow[r, "\mathfrak{x}.g"] & X & R_P(X) \arrow[l, "\mathsf{mono}"'] \\
(X \to R_P(X))/(P \to R_P(X)) \arrow[urrr, "\mathit{lift}(\mathfrak{x}.g \circ (X \to \mathsf{mono}))"'] & & & \\
R_P(X) \to R_P(X) \arrow[u, "\mathsf{iso}"] & & &
\end{tikzcd}$$

Then, $\mathfrak{s}_\mathfrak{a}.g : (R_R(X) \to R_R(X)) \to R_R(X)$ is given by $\mathit{lift}(\mathfrak{x}.g \circ (X \to \mathsf{mono})) \circ \mathsf{iso}$. To justify the lifting of $\mathfrak{x}.g \circ (X \to \mathsf{mono})$, we must show for all $f, f' : X \to R_P(X)$ satisfying $f\ (P \to R_P(X))\ f'$, that $\mathfrak{x}.g \circ (X \to \mathsf{mono})(f) =_X \mathfrak{x}.g \circ (X \to \mathsf{mono})(f')$. Note that $\mathit{lift}(\mathfrak{x}.g \circ (X \to \mathsf{mono}))$ then maps to X, so in addition we must show that $\mathit{lift}(\mathfrak{x}.g \circ (X \to \mathsf{mono}))$ in fact maps to $R_P(X)$. Now, if $f\ (P \to R_P(X))\ f'$, we get $(X \to \mathsf{mono})(f)\ (P \to P)\ (X \to \mathsf{mono})(f')$. By assumption we have $\mathfrak{x}\ \mathfrak{T}[P]\ \mathfrak{x}$, which in particular gives $\mathfrak{x}.g\ ((P \to P) \to P)\ \mathfrak{x}.g$, and the result follows.

Here is the general form of this diagram for any given $U = U^n$ and U_c, is

$$\begin{tikzcd}
S(U)^n \arrow[r, "\mathsf{mono}(U)^n"] \arrow[d, "\mathsf{epi}_{S(U)^n}"'] & U_n \arrow[r, "\mathfrak{x}.g"] & U_c & R_P(U_c) \arrow[l, "\mathsf{mono}"'] \\
S(U)^n/P(U)^n \arrow[urrr, "\mathit{lift}(\mathfrak{x}.g \circ (\mathsf{mono}(U)^n))"'] & & & \\
U[(R_P(U_i))/U_i] \arrow[u, "\mathsf{iso}"] & & &
\end{tikzcd}$$

where for a given $U = U^n$, we define $\mathsf{mono}(U)^i$ as follows.

$$\begin{array}{lcl}
\mathsf{mono}(U)^0 & = & \mathsf{mono}_{U_0} \\
\mathsf{mono}(U)^1 & = & (U_1 \to \mathsf{mono}_{U_1}) \\
\mathsf{mono}(U)^{i+1} & = & (\mathsf{mono}(U)^{i-1} \to U_i) \to \mathsf{mono}_{U_{i+1}}, \quad 1 \leq i \leq n-1
\end{array}$$

This schema is more general than what is called for in the refinement-specific Sub. In Sub, the starting point is a relation R, and the predicate with which one restricts the domain X, is $P_R(x) \stackrel{def}{=} x\ R\ x$. The corresponding binary relation is then $P_R \stackrel{def}{=} (x{:}X, y{:}X)\ .\ (x =_X y\ \wedge\ x\ R\ x))$.

In the example above, we relied on the assumption $\mathfrak{x}\ \mathfrak{T}[P]\ \mathfrak{x}$. Notice that Sub only assumes $\mathfrak{x}\ \mathfrak{T}[R]\ \mathfrak{x}$. At first order, this implies $\mathfrak{x}\ \mathfrak{T}[P_R]\ \mathfrak{x}$, but this is not the case at higher order. We must therefore explicitly include $\mathfrak{x}\ \mathfrak{T}[P_R]\ \mathfrak{x}$ in the antecedent of Sub, and we get the following general formulation.

Definition 6.2 (Existence of Subobjects (Sub)**)**

$$\begin{array}{l}
\textsc{Sub}: \ \forall X\,.\,\forall \mathfrak{x}{:}\mathfrak{T}[X,\boldsymbol{Z}]\,.\,\forall R \subset X \times X\ .\ (\mathfrak{x}\ \mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}]\ \mathfrak{x})\ \ \wedge\ \ (\mathfrak{x}\ \mathfrak{T}[P_R, \mathbf{eq}_{\boldsymbol{Z}}]\ \mathfrak{x})\ \Rightarrow \\
\quad \exists S\,.\,\exists \mathfrak{s}{:}\mathfrak{T}[S,\boldsymbol{Z}]\,.\,\exists R' \subset S \times S\,.\,\exists \mathsf{mono}{:}S \to X\ . \\
\qquad\qquad \forall s{:}S\ .\ s\ R'\ s \qquad\qquad\qquad\qquad\qquad\qquad \wedge \\
\qquad\qquad \forall s, s'{:}S\ .\ s\ R'\ s' \ \Leftrightarrow\ (\mathsf{mono}\, s)\ R\ (\mathsf{mono}\, s') \quad \wedge \\
\qquad\qquad \mathfrak{x}\ (\mathfrak{T}[(x{:}X, s{:}S)\ .\ (x =_X (\mathsf{mono}\, s)), \mathbf{eq}_{\boldsymbol{Z}}])\ \mathfrak{s}
\end{array}$$

When relating to a particular model, we check that *Sub-Arr* holds, presuming that the terminology concerning subobjects makes sense. If this goes through, we have succeeded in constructing data types over subobjects at higher order.

6.2.3 A Setoid Model

We now give a model for the logic augmented by Sub and Quot for data types with higher-order profiles. The model is PER-based semantic analogue to a syntactic setoid model in (Hofmann, 1995a).

Types are now interpreted as setoids, *i.e.,* pairs $\langle \mathcal{A}, \sim_{\mathcal{A}} \rangle$, consisting of a PER $\mathcal{A}$ together with a partial equivalence relation $\sim_{\mathcal{A}}$ on $\mathcal{A}$, *i.e.,* a saturated PER on $Dom(\mathcal{A}) \times Dom(\mathcal{A})$, intuitively giving the desired equality on the interpreted type. Given two setoids $\langle \mathcal{A}, \sim_{\mathcal{A}} \rangle$ and $\langle \mathcal{B}, \sim_{\mathcal{B}} \rangle$, we form a new setoid $\langle \mathcal{A}, \sim_{\mathcal{A}} \rangle \to \langle \mathcal{B}, \sim_{\mathcal{B}} \rangle$ giving function spaces in the universe of setoids, by

$$\langle \mathcal{A}, \sim_{\mathcal{A}} \rangle \to \langle \mathcal{B}, \sim_{\mathcal{B}} \rangle \stackrel{def}{=} \langle \mathcal{A} \to \mathcal{B}, \sim_{\mathcal{A} \to \mathcal{B}} \rangle$$

where $\sim_{\mathcal{A}\to\mathcal{B}}$ is the saturated relation $\sim_{\mathcal{A}} \to \sim_{\mathcal{B}} \subset Dom(\mathcal{A} \to \mathcal{B}) \times Dom(\mathcal{A} \to \mathcal{B})$, *cf.* Sect.3.4. Products are given by

$$\langle \mathcal{A}, \sim_{\mathcal{A}} \rangle \times \langle \mathcal{B}, \sim_{\mathcal{B}} \rangle \stackrel{def}{=} \langle \mathcal{A} \times \mathcal{B}, \sim_{\mathcal{A} \times \mathcal{B}} \rangle$$

where $\sim_{\mathcal{A}\times\mathcal{B}}$ is the saturated relation $\sim_{\mathcal{A}} \times \sim_{\mathcal{B}}$.

Whereas elements of a PER $\mathcal{A}$ in the PER model intuitively are $\mathcal{A}$ equivalence classes, elements of a setoid $\langle \mathcal{A}, \sim_{\mathcal{A}} \rangle$ are now intuitively $\sim_{\mathcal{A}}$ equivalence classes of $\mathcal{A}$ equivalence classes. Of course, elements in elementary terms are simply natural numbers giving realisers respecting these equivalence classes.

Thus, a relation $\mathcal{R}$ between two setoids $\langle \mathcal{A}, \sim_{\mathcal{A}} \rangle$ and $\langle \mathcal{B}, \sim_{\mathcal{B}} \rangle$ intuitively relates $\sim_{\mathcal{A}}$ equivalence classes (of $\mathcal{A}$ equivalence classes) to $\sim_{\mathcal{B}}$ equivalence classes (of $\mathcal{B}$ equivalence classes). Thus, $\mathcal{R}$ must be, and is therefore, saturated w.r.t. $\sim_{\mathcal{A}}$ and $\sim_{\mathcal{B}}$, *i.e.,* $\mathcal{R}$ is a saturated relation on $Dom(\sim_{\mathcal{A}}) \times Dom(\sim_{\mathcal{B}})$. We then get complex relations by exactly the same mechanics as for the universe of PERs in Sect.3.4, except that now saturation is of course w.r.t. the setoid equalities. So if $\mathcal{R} \subset Dom(\sim_{\mathcal{A}}) \times Dom(\sim_{\mathcal{B}})$ and $\mathcal{S} \subset Dom(\sim_{\mathcal{A}'}) \times Dom(\sim_{\mathcal{B}'})$ are saturated, we obtain a relation between $\langle \mathcal{A}, \sim_{\mathcal{A}} \rangle \to \langle \mathcal{A}', \sim_{\mathcal{A}'} \rangle$ and $\langle \mathcal{B}, \sim_{\mathcal{B}} \rangle \to \langle \mathcal{B}', \sim_{\mathcal{B}'} \rangle$ by the saturated relation $\mathcal{R} \to \mathcal{S} \subset Dom(\sim_{\mathcal{A}} \to \sim_{\mathcal{A}'}) \times Dom(\sim_{\mathcal{B}} \to \sim_{\mathcal{B}'})$.

Type and Relation Semantics

Given the universe of setoids above, type semantics are now defined denotationally w.r.t. to an environment δ mapping type variables to setoids. Here δ consists of a component δ_1 mapping to PERs, and a component δ_2 mapping to relations on PERs, such that $\delta_2(X)$ is a saturated PER on $Dom(\delta_1(X)) \times Dom(\delta_1(X))$.

$$\begin{aligned}
&[\![\Delta, X \rhd X]\!]_\delta \overset{def}{=} \delta(X)\\
&[\![\Delta \rhd U \to V]\!]_\delta \overset{def}{=} ([\![\Delta \rhd U]\!]_\delta \to [\![\Delta \rhd V]\!]_\delta)\\
&[\![\Delta \rhd \forall X.U[X]]\!]_\delta \overset{def}{=} \left\langle (\cap_{\mathcal{A}} [\![\Delta, X \rhd U[X]]\!]_{\delta_1[X\mapsto\mathcal{A}]}), (\cap_{\sim_{\mathcal{A}}} [\![\Delta, X \rhd U[X]]\!]_{\delta_2[X\mapsto\sim_{\mathcal{A}}]})^\flat \right\rangle
\end{aligned}$$

where $\cap_{\sim_{\mathcal{A}}}$ ranges over all saturated PERs $\sim_{\mathcal{A}} \subset Dom(\mathcal{A}) \times Dom(\mathcal{A})$.

Relational semantics are defined in form exactly as for PER semantics. Thus w.r.t. environments δ and υ mapping respectively, type variables to setoids, and relation variables to saturated relations on the the appropriate domains, we have

$$\begin{aligned}
&[\![\Delta \mid \Upsilon, R \rhd R]\!]_{\delta\upsilon} = \upsilon(R)\\
&[\![\Delta \mid \Upsilon \rhd U \to V]\!]_{\delta\upsilon} = ([\![\Delta \mid \Upsilon \rhd U]\!]_{\delta\upsilon} \to [\![\Delta \mid \Upsilon \rhd V]\!]_{\delta\upsilon})\\
&[\![\Delta \mid \Upsilon \rhd \forall X.U[X]]\!]_{\delta\upsilon} = (\cap_{\mathcal{R}} [\![\Delta \mid \Upsilon, R \rhd U[R]]\!]_{\delta\upsilon[R\mapsto\mathcal{R}]})
\end{aligned}$$

The intersection ranges over all saturated $\mathcal{R} \subset Dom(\sim_{\mathcal{A}}) \times Dom(\sim_{\mathcal{B}})$, for all setoids $\langle \mathcal{A}, \sim_{\mathcal{A}} \rangle$ and $\langle \mathcal{B}, \sim_{\mathcal{B}} \rangle$.

Recall that for the PER model, equality on a type has the same interpretation as the type itself. Thus relational parametricity is effectively expressed in terms of the equality predicate. In the setoid semantics, relational parametricity is analogously expressed in terms of the setoid equality component, giving the trimmed intersection over equalities in the above semantics for universal types. The definition is in the same form as before, *i.e.*,

$$\begin{array}{l} n\ (\cap_{\sim_{\mathcal{A}}}[\![\Delta, X \rhd U[X,\boldsymbol{Z}]]\!]_{\delta_2[X\mapsto\sim_{\mathcal{A}}]})^\flat\ m \\ \qquad\qquad\qquad\overset{def}{\Leftrightarrow} \forall \sim_{\mathcal{A}}, \sim_{\mathcal{B}},\ \mathit{saturated}\ \mathcal{R} \subset \mathit{Dom}(\sim_{\mathcal{A}}) \times \mathit{Dom}(\sim_{\mathcal{B}})\ . \\ \quad n\ [\![\Delta, X \rhd U[X,\boldsymbol{Z}]]\!]_{\delta_2[X\mapsto\sim_{\mathcal{A}}]}\ m\ \wedge\ n\ [\![\Delta, X \rhd U[X,\boldsymbol{Z}]]\!]_{\delta_2[X\mapsto\sim_{\mathcal{B}}]}\ m\ \wedge \\ \qquad\qquad\qquad n\ [\![\Delta \shortmid R \rhd U[R,\mathbf{eq}_{\boldsymbol{Z}}]]\!]_{\delta\,[R\mapsto\mathcal{R}]}\ n\ \wedge \\ \qquad\qquad\qquad m\ [\![\Delta \shortmid R \rhd U[R,\mathbf{eq}_{\boldsymbol{Z}}]]\!]_{\delta\,[R\mapsto\mathcal{R}]}\ m \end{array}$$

We now have that

$$(\cap_{\sim_{\mathcal{A}}}[\![\Delta, X \rhd U[X,\boldsymbol{Z}]]\!]_{\delta_2[X\mapsto\sim_{\mathcal{A}}]})^\flat = (\cap_{\mathcal{R}}[\![\Delta \shortmid R \rhd U[R,\mathbf{eq}_{\boldsymbol{Z}}]]\!]_{\delta\,[R\mapsto\mathcal{R}]})$$

where $\mathcal{R}$ ranges over all saturated relations on $\mathit{Dom}(\sim_{\mathcal{A}}) \times \mathit{Dom}(\sim_{\mathcal{B}})$, for all setoids $\langle\mathcal{A}, \sim_{\mathcal{A}}\rangle$ and $\langle\mathcal{B}, \sim_{\mathcal{B}}\rangle$. In general we get the following manifestation of the Identity Extension Lemma.

$$[\![U[\boldsymbol{Z}]]\!]_{\delta_2} = [\![U[\mathbf{eq}_{\boldsymbol{Z}}]]\!]_{\delta\upsilon}$$

As expected, $[\![\mathsf{eq}_{U[\boldsymbol{Z}]}]\!]_{\delta\upsilon} = [\![U[\boldsymbol{Z}]]\!]_{\delta_2}$, *i.e.*, the equality predicate on a type has the same interpretation as the equality component of the interpretation of the type.

Term semantics

Term semantics follow regular denotational structure as before. We amalgamate contexts into a single context Γ, and environments into a single environment γ.

$$\begin{array}{l} [\![\Gamma, x\!:\!U \rhd x\!:\!U]\!]_\gamma \overset{def}{=} \gamma(x) \quad \text{where } \gamma(x) \in \mathit{Dom}(\sim_{\mathcal{A}}), \text{ for } \langle\mathcal{A},\sim_{\mathcal{A}}\rangle = [\![\Gamma \rhd U]\!]_\gamma \\ [\![\Gamma \rhd fu\!:\!V]\!]_\gamma \overset{def}{=} [\![\Gamma \rhd f\!:\!U \to V]\!]_\gamma [\![\Gamma \rhd u\!:\!U]\!]_\gamma \\ [\![\Gamma \rhd \lambda x\!:\!U.t[x]\!:\!U \to V]\!]_\gamma \overset{def}{=} n \in \mathit{Dom}([\![\Gamma \rhd U \to V]\!]_\gamma), \quad \textit{for } n \textit{ the code of a} \\ \qquad\qquad \textit{partial recursive function } \lambda a.[\![\Gamma, x\!:\!V \rhd t[x]]\!]_{\gamma[x\mapsto a]} \\ [\![\Gamma \rhd tA\!:\!U[A]]\!]_\gamma \overset{def}{=} [\![\Gamma \rhd t\!:\!\forall X.U[X]]\!]_\gamma \\ [\![\Gamma \rhd \Lambda X.t[X]\!:\!\forall X.U[X]]\!]_\gamma \overset{def}{=} n, \quad \textit{such that} \\ \qquad\qquad [\![\Gamma, X \rhd t[X]\!:\!U[X]]\!]_{\gamma[X\mapsto\langle\mathcal{A},\sim_{\mathcal{A}}\rangle]} = n \quad \textit{for any } \langle\mathcal{A},\sim_{\mathcal{A}}\rangle \end{array}$$

As usual, there is an obligation to check that the term semantics preserves the appropriate equivalence classes. We omit the details.

—

6.3 Soundness for QUOT and SUB at Higher Order

We are now in a position to show the soundness of the logic with SUB and QUOT for data types with higher-order operations according to HADT_{Obs}. This thereby resolves some questions posed in (Zwanenburg, 1999). It suffices to show that *Sub-Arr* and *Quot-Arr* are satisfied by the setoid semantics just presented. First we define quotients and subobjects.

Definition 6.3 (Quotient Setoid) *Let $\langle \mathcal{X}, \sim_{\mathcal{X}} \rangle$ be a setoid, and let $\mathcal{R}$ be any equivalence relation on $\langle \mathcal{X}, \sim_{\mathcal{X}} \rangle$, i.e., $\mathcal{R} \subset Dom(\sim_{\mathcal{X}}) \times Dom(\sim_{\mathcal{X}})$ is a saturated equivalence relation. Define the* quotient $\langle \mathcal{X}, \sim_{\mathcal{X}} \rangle / \mathcal{R}$ *of* $\langle \mathcal{X}, \sim_{\mathcal{X}} \rangle$ w.r.t. $\mathcal{R}$ *by*

$$\langle \mathcal{X}, \mathcal{R} \rangle$$

Since $\mathcal{R}$ is an equivalence, $\mathcal{R} \subset Dom(\mathcal{X}) \times Dom(\mathcal{X})$ is a saturated PER, and is thereby eligible as a setoid equality.

Definition 6.4 (Subobject Setoid) *Let $\langle \mathcal{X}, \sim_{\mathcal{X}} \rangle$ be a setoid, and let $\mathcal{P}$ be any predicate on $\langle \mathcal{X}, \sim_{\mathcal{X}} \rangle$, meaning that $\mathcal{P}$ fulfils the unary saturation condition $\mathcal{P}(x) \wedge x \sim_{\mathcal{X}} y \;\Rightarrow\; \mathcal{P}(y)$. Consider the relation, also denoted $\mathcal{P}$, on $\langle \mathcal{X}, \sim_{\mathcal{X}} \rangle$ derived from $\mathcal{P}$, defined by $x \; \mathcal{P} \; y \overset{def}{\Leftrightarrow} x \sim_X y \;\wedge\; \mathcal{P}(x)$. Then the* subobject $R_{\mathcal{P}}(\langle \mathcal{X}, \sim_{\mathcal{X}} \rangle)$ *of* $\langle \mathcal{X}, \sim_{\mathcal{X}} \rangle$ restricted on $\mathcal{P}$, *is defined by*

$$\langle \mathcal{X}, \mathcal{P} \rangle$$

It is easy to check that the relation $\mathcal{P} \subset Dom(\mathcal{X}) \times Dom(\mathcal{X})$ is a saturated PER, and is thereby eligible as a setoid equality. Note that it is the fact that setoid equalities are PERs and not total equivalence relations, that allows the definition of subobjects. The setoid equality at once defines equality and the intended domain of usage.

Theorem 6.5 *The setoid semantics satisfies* ***Quot-Arr****, by the isomorphism being denotational equality.*

Proof: We proceed by induction on the length i of $U = U^n$.

$i = 0$: We must essentially verify $U_0/R \cong U_0/R$. This is immediate by definition and universally true.

$i = 1$: We must verify $(U_0/R \to U_1)/(U_0/R \to R_1) \cong U_0/R \to U_1/R$. In setoid terms, U_0/R is $\langle U_0, R_0 \rangle$, and the equality, denoted also by U_0/R, is of course R. Thus, $(U_0/R \to U_1)/(U_0/R_0 \to R_1)$ is $(\langle U_0, R_0 \rangle \to \langle U_1, \sim_{U_1} \rangle)/(R_0 \to R_1)$. But this is $(\langle U_0 \to U_1, R_0 \to \sim_{U_1} \rangle)/(R_0 \to R_1)$, which is simply $\langle U_0 \to U_1, R_0 \to R_1 \rangle$, and which is $U_0/R \to U_1/R$ in setoid terms. The indices i on R_i are put on to match U_i being either X, in which case R_i is R, or some $D \in Obs$, in which case R_i is the identity.

$i = i+1$, $1 \leq i \leq n-1$: We must verify

$$((Q(U)^{i-1} \to U_i/R) \to U_{i+1})/((R(U)^{i-1} \to U_i/R) \to R_{i+1}) \cong (U^{i-1}[(U_j/R)/U_j] \to U^i[(U_j/R)/U_j]) \to U^{i+1}[(U_j/R)/U_j]$$

In setoid terms, $((Q(U)^{i-1} \to U_i/R) \to U_{i+1})$ is

$$\langle (Q(U)^{i-1} \to U_i) \to U_{i+1},\ (\sim_{Q(U)^{i-1}} \to R_i) \to \sim_{U_{i+1}} \rangle$$

Therefore $((Q(U)^{i-1} \to U_i/R) \to U_{i+1})/((R(U)^{i-1} \to U_i/R) \to R_{i+1})$ is

$$\langle (Q(U)^{i-1} \to U_i) \to U_{i+1},\ (R(U)^{i-1} \to U_i/R) \to R_{i+1} \rangle$$

But this is $(Q(U)^{i-1}/R(U)^{i-1} \to U_i/R) \to U_{i+1}/R_{i+1}$. By induction hypothesis, $Q(U)^{i-1}/R(U)^{i-1} \cong U^{i-1}[(U_j/R)/U_j]$, and we are done. □

Theorem 6.6 *The setoid semantics satisfies* ***Sub-Arr****, by the isomorphism being denotational equality.*

Proof: This follows essentially the same course as the proof of Theorem 6.5. □

In closing, we mention that the PER model, parametric or not, does not satisfy ***Quot-Arr*** nor ***Sub-Arr***.

There are more things one can envision doing with the setoid semantics. Firstly, one could try to extend ***Quot-Arr*** and ***Sub-Arr*** to universal types. However, this is problematic, because the action of a universal type on relations is not necessarily an equivalence relation, even if all the free relation variables are substituted by equivalence relations. It is therefore not immediately possible to form the desired quotients necessary in ***Quot-Arr*** and ***Sub-Arr***. Also, the motivation for attempting this is perhaps not great, because as we have argued before, polymorphic operations are probably best dealt with in F_3, *i.e.,* there is perhaps not much in favour of insisting on the generality in ADT_{Obs} over HADT_{Obs}, *cf. p.* 113.

6.4 Abstraction Barrier-Observing Semantics

In the previous section we showed soundness of the logic augmented by SUB and QUOT, for data types with operations of any order, thus expressing the proof method for observational refinement also at higher order. However, we did this in the context of standard simulation relations. Now we will express the proof method in the context of abstraction barrier-observing simulation relations, by soundly adding appropriate versions of SUB and QUOT to the logic.

The model we use to show soundness in this case is constructed by the same basic consideration as for abstraction barrier-observing simulation relations. At the beginning of this chapter we argued that the natural approach to constructing the operations of q in a PER model is to use the same realisers that give the operations of x. Thus, if $\mathsf{x} = \langle [e_0]_{\mathcal{X}}, [e_1]_{\mathcal{X}\to\mathcal{X}}, [e_2]_{(\mathcal{X}\to\mathcal{X})\to\mathcal{X}}, \ldots\rangle$, we would like to define q as $\langle [e_0]_{\mathcal{Q}}, [e_1]_{\mathcal{Q}\to\mathcal{Q}}, [e_2]_{(\mathcal{Q}\to\mathcal{Q})\to\mathcal{Q}}, \ldots\rangle$. The problem is then to verify that the indicated equivalence classes exist, *e.g.,* to show e_2 $((\mathcal{Q} \to \mathcal{Q}) \to \mathcal{Q})$ e_2, we must show $n\ (\mathcal{Q}\to\mathcal{Q})\ n \ \Rightarrow\ e_2(n)\downarrow$, and this does not follow from the running assumption $n\ (\mathcal{X}\to\mathcal{X})\ n \ \Rightarrow\ e_2(n)\downarrow$; because even if $n\ (\mathcal{Q}\to\mathcal{Q})\ n$ we may not have $n\ (\mathcal{X}\to\mathcal{X})\ n$; recall Example 6.1.

Suppose on the other hand that the arguments given to a data type operation e_i of x were not arbitrary, but definable in terms of the e_j's of x using function definition. This is the principle of ***Abs-Bar***. This would ensure that arguments to any operation e_i in $\mathsf{x} = \langle [e_0]_{\mathcal{X}}, [e_1]_{\mathcal{X}\to\mathcal{X}}, [e_2]_{(\mathcal{X}\to\mathcal{X})\to\mathcal{X}}, \ldots\rangle$ would be given only by certain realisers, namely those freely generated on $\{e_0, e_1, e_2, \ldots\}$ according to term formation, *cf.* Sect. 3.4; notice how this eliminates *rfi* from consideration in Example 6.1. Since ***Abs-Bar*** essentially says that in actual use, these are the arguments that e_i will ever be applied to, we can try redefining data type operations to accommodate this fact. This in turn would allow the desired definition of q. The proposed solution here acts on the observation that the standard notions do not observe ***Abs-Bar***. The angle of approach is thus simply a natural continuation of tactics so far. We refine the interpretation of data type operations to reflect their actual applicability as captured by ***Abs-Bar***.

To illustrate the basic idea, suppose $g_i\colon(X\to X)\to X$ is a data type operation in some $\mathfrak{x}\colon\mathfrak{T}[X]$. Then, for some environment γ, if $\gamma(X) = \mathcal{A}$, and x is a realiser of $\gamma(\mathfrak{x})$, the interpretation of g_i will lie in

$$(\mathcal{A}\cap\mathcal{D}_X\to\mathcal{A})\cap\mathcal{D}_{X\to X}\to\mathcal{A}$$

where

$n\ \mathcal{D}_{U[X]}\ n' \overset{def}{\Leftrightarrow}$ *there exist terms $t\colon U[X]$ and $t'\colon U[X]$,*
without free variables of types involving X,
such that the interpretation under γ of t is n and that of t' is n'

This is analogous to the way *abo*-simulation relations are defined in Ch. 5. Note that the uniformity aspect of ***Abs-Bar*** is not expressed in $\mathcal{D}_{U[X]}$. The uniformity aspect states how data type operations are applied in a uniform manner in all actual computations arising from a virtual computation. Here on the other hand, we are talking about the interpretation of single operations within a data type.

The scene here is that we want to give a special semantics to data type operations, but otherwise leave the semantics as it is. This 'special' semantics is really not that special; it simply conforms to the use of data type operations in computations. Thus, in computations, the result will be the same whether we use regular semantics or we use the special data type semantics. However, in logical deductions, one cannot rely on *Abs-Bar*, and we then have to determine when to apply which semantics. For example, for X and $\mathfrak{x} : \mathfrak{T}[X]$, suppose $\mathfrak{x}.g : (X \to X) \to X$. Consider the interpretation of a formula

$$[\![\; \forall f : X \to X . \exists z : X \; . \; (\mathfrak{x}.g\, f) = (f\, z) \;]\!]_\gamma$$

where $\gamma(X) = \mathcal{X}$ and $\gamma(\mathfrak{x}) = x \in Dom([\![X \rhd \mathfrak{T}[X]]\!]_\gamma)$. One possibility is to have a dynamic interpretation. Let ϕ be the scope of a variable α. Then the denotation of α is a semantic variable belonging to a PER restricted according to the strictest usage of the variable in ϕ. Thus, the denotation of f above is a variable belonging to $(\mathcal{X} \cap \mathcal{D}_X \to \mathcal{X}) \cap \mathcal{D}_{X \to X}$. This is because the denotation of $\mathfrak{x}.g : (X \to X) \to X$ lies in $((\mathcal{X} \cap \mathcal{D}_X \to \mathcal{X}) \cap \mathcal{D}_{X \to X}) \to \mathcal{X}$, and the denotation of f must match this. Moreover, this entails that the denotation of z belongs to $\mathcal{X} \cap \mathcal{D}_X$.

On the other hand, one could argue that the above formula is not really what we or the specifier should write. We at least, know that $\mathfrak{x}.g$ will not be applied to arbitrary $f : X \to X$, but only definable ones, and then f will only be applied to definable z. So we should really write

$$[\![\; \forall f : X \to X . \exists z : X \; . \; \mathsf{Dfnbl}(f) \; \wedge \; \mathsf{Dfnbl}(z) \; \Rightarrow \; (\mathfrak{x}.g\, f) = (f\, z) \;]\!]_\gamma$$

for suitable Dfnbl clauses. The specifier would be less happy to have to think about definability. In this case, the proper Dfnbl clauses might be inserted at least semi-automatically. The appropriate Dfnbl clauses, directly derived from the Dfnbl and DfnblC clauses used in Ch. 5, would guarantee application according to *Abs-Bar*, and hence the regular term semantics would suffice. Yet another alternative would be to prune proof trees so that free variables of types involving the virtual data representation are disallowed, but this would not let us quantify over virtual data representations.

6.4.1 Annotated Types

The path we choose in this thesis is to have special types for data type operators. We can get the desired effect by introducing a type constructor $(-)^{\mathsf{a}} : * \to *$, for marking types when using them in data types. There are in the outset several plausible schemes for doing this. One might conceivably mark types according to what is indicated by ***Abs-Bar***. However, unless we introduce subtypes, this will not work; *e.g.,* for $\mathfrak{x}.g_0 : X, \mathfrak{x}.g_1 : X^{\mathsf{a}} \to X$, we cannot form $\mathfrak{x}.g_1\mathfrak{x}.g_0$. We could set $\mathfrak{x}.g_0 : X^{\mathsf{a}}$; then we can form $\mathfrak{x}.g_1\mathfrak{x}.g_0$, but not $\mathfrak{x}.g_1\mathfrak{x}.g_1\mathfrak{x}.g_0$. Thus we must set $\mathfrak{x}.g_1 : X^{\mathsf{a}} \to X^{\mathsf{a}}$ as well; in fact we must set $\mathfrak{x}.g_1 : (X^{\mathsf{a}} \to X^{\mathsf{a}})^{\mathsf{a}}$, in the case we have $\mathfrak{x}.g_2 : (X \to X) \to X$ (which should be marked $\mathfrak{x}.g_2 : ((X^{\mathsf{a}} \to X^{\mathsf{a}})^{\mathsf{a}} \to X^{\mathsf{a}})^{\mathsf{a}}$). The result is that we mark every single type occurrence involving the virtual data representation, *i.e.,* the existentially quantified type variable, in the types of data-type operators. Types that do not involve the virtual data representation need not be marked, so *e.g.,* types in *Obs* are not marked, and neither are types built purely from these. The idea here is to mark types automatically after type formation, that is, one writes an existential type and then marks the types of data-type operations as described above. We define this recursively as follows.

Definition 6.7 (Type Marking) *With respect to $\exists X.\mathfrak{T}[X]$, we define*

$$T^{r\mathsf{a}} \stackrel{def}{=} T$$

if T has no occurrence of (free) X, and otherwise

$$\begin{aligned} X^{r\mathsf{a}} &\stackrel{def}{=} X^{\mathsf{a}} \\ (T \to T')^{r\mathsf{a}} &\stackrel{def}{=} (T^{r\mathsf{a}} \to T'^{r\mathsf{a}})^{\mathsf{a}} \\ (\forall Y.T[Y,X])^{r\mathsf{a}} &\stackrel{def}{=} \forall Y.T[Y,X]^{r\mathsf{a}} \end{aligned}$$

Alternatively, one could give formation rules for forming marked types.

Semantically, these marked types are to be interpreted as subdomains containing the elements that are term denotable. Now, the ubiquitous type marking does not exactly mirror the semantic intention according to ***Abs-Bar***, but the type marking is merely at places superfluous, not actually in miscorrespondence to the intended semantics. For example, according to our earlier deliberations and ***Abs-Bar***, the data type operation $\mathfrak{x}.g_1 : X \to X$ should be in $(\mathcal{X} \cap \mathcal{D}_X) \to \mathcal{X}$, while the interpretation of $(X^{\mathsf{a}} \to X^{\mathsf{a}})^{\mathsf{a}}$ will be $((\mathcal{X} \cap \mathcal{D}_X) \to (\mathcal{X} \cap \mathcal{D}_X)) \cap \mathcal{D}_{X \to X})$. But the $\mathcal{D}_{X \to X}$ clause is trivially fulfilled for $\mathfrak{x}.g_1$, and then the $\mathcal{D}_X$ clause of the codomain is also trivial, since we have this clause on the domain of definition.

Term formation w.r.t. marked types follows the usual format, but in addition one needs the following formation rules. These are necessary so that no syntactic restrictions are introduced; note that type marking is for semantic purposes.

Definition 6.8 (Additional Term Formation Rules)

$$\frac{\Gamma, x\!:\!T'^{r\mathsf{a}} \rhd t\!:\!T^{r\mathsf{a}}}{\Gamma \rhd \lambda x\!:\!T'^{r\mathsf{a}}.t : (T'^{r\mathsf{a}} \to T^{r\mathsf{a}})^{\mathsf{a}}}$$

$$\frac{\Gamma \rhd t\!:\!(T'^{r\mathsf{a}} \to T^{r\mathsf{a}})^{\mathsf{a}} \qquad \Gamma \rhd t'\!:\!T'^{r\mathsf{a}}}{\Gamma \rhd tt'\!:\!T^{r\mathsf{a}}}$$

$$\frac{\Gamma, X \rhd t\!:\!T^{r\mathsf{a}}}{\Gamma \rhd \Lambda X.t\!:\!(\forall X.T^{r\mathsf{a}})^{\mathsf{a}}}$$

$$\frac{\Gamma \rhd t\!:\!(\forall X.T^{r\mathsf{a}})^{\mathsf{a}} \qquad \Gamma \rhd A}{\Gamma \rhd tA\!:\!T[A/X]^{r\mathsf{a}}}$$

We owe it to the specifier that he should not have to think about type marking, since type marking fills a gap in the logic in relation to the abstraction barrier inherent in existential types. When the specifier uses a proof checker, the proof assistant should mark all abstract types according to the scheme above. This begs the issue of other types with similar properties to existential types that need special treatment in the logic. But here we only treat existential types.

6.4.2 Data Type Semantics

The semantics given to marked types is as follows. We keep the parametric PER-model as structure, but supply a modified interpretation for data type operations. For clarity we omit parameters $\boldsymbol{Z}$ in the following discussion. We assume that the proof checker has recognised an existential type $\exists X.\mathfrak{T}[X]$ and has marked the body of the abstract type by $\mathfrak{T}[X]^{i\mathsf{a}}$.

Definition 6.9 (Data Type Semantics for Types) *For any PER $\mathcal{X}$ and any $x, y \in \mathbb{N}$, we define the* Data Type Semantics *for marked types as follows.*

$$x\ [\![X \rhd \mathfrak{T}[X]^{i\mathsf{a}}]\!]_{[X\mapsto\mathcal{X}]}\ y \quad \overset{def}{\Leftrightarrow} \quad \textit{for all components } g_i\!:\!T_i[X]^{r\mathsf{a}} \textit{ in } \mathfrak{T}[X]^{i\mathsf{a}}$$
$$x.i\ ([\![X \rhd T_i[X]^{r\mathsf{a}}]\!]^{x,y,X}_{[X\mapsto\mathcal{X}]})\ y.i$$

where, for the list $\mathsf{adt} = x, y, X$ *and where δ is an environment on $\boldsymbol{Y}, X$ such that $\delta(X) = \mathcal{X}$,*

$$[\![\boldsymbol{Y}, X \rhd T]\!]^{\mathsf{adt}}_{\delta} \overset{def}{=} [\![\boldsymbol{Y}, X \rhd T]\!]_{\delta}, \qquad \textit{for unmarked } T$$
$$[\![\boldsymbol{Y}, X \rhd X^{\mathsf{a}}]\!]^{\mathsf{adt}}_{\delta} \overset{def}{=} \delta(X) \cap \mathcal{D}^{\mathsf{adt}}_{X}$$
$$[\![\boldsymbol{Y}, X \rhd \forall Y.U[Y, X]^{r\mathsf{a}}]\!]^{\mathsf{adt}}_{\delta} \overset{def}{=} (\cap_{\mathcal{E}\in\mathbf{PER}}[\![\boldsymbol{Y}, Y_{k+1}, X \rhd U[\boldsymbol{Y}, Y_{k+1}, X]^{r\mathsf{a}}]\!]^{\mathsf{adt}}_{\delta[Y_{k+1}\mapsto\mathcal{E}]})^{\flat}$$
$$[\![\boldsymbol{Y}, X \rhd (U[\boldsymbol{Y}, X]^{r\mathsf{a}} \to U'[\boldsymbol{Y}, X]^{r\mathsf{a}})^{\mathsf{a}}]\!]^{\mathsf{adt}}_{\delta} \overset{def}{=}$$
$$([\![\boldsymbol{Y}, X \rhd U[\boldsymbol{Y}, X]^{r\mathsf{a}}]\!]^{\mathsf{adt}}_{\delta} \to [\![\boldsymbol{Y}, X \rhd U'[\boldsymbol{Y}, X]^{r\mathsf{a}}]\!]^{\mathsf{adt}}_{\delta}) \cap \mathcal{D}^{\mathsf{adt}}_{U[\boldsymbol{Y},X]\to U'[\boldsymbol{Y},X]}$$

where

$$n \; \mathcal{D}^{\mathsf{adt}}_{U[\boldsymbol{Y},X]} \; n' \;\overset{\mathit{def}}{\Leftrightarrow}\; \begin{array}{c}\textit{there exist terms } \Gamma \rhd t\!:\!U[\boldsymbol{Y},X] \textit{ and } \Gamma \rhd t'\!:\!U[\boldsymbol{Y},X] \\ \textit{where } \Gamma = X,\boldsymbol{Y},\boldsymbol{y}\!:\!\boldsymbol{V}[\boldsymbol{Y}] \textit{ for } \boldsymbol{V}[\boldsymbol{Y}] \textit{ not containing } X, \\ \textit{such that} \;\; t[x,\boldsymbol{z}] = n \;\; \textit{and} \;\; t'[y,\boldsymbol{z'}] = n'\end{array}$$

where $t[x,\boldsymbol{z}]$ *and* $t'[y,\boldsymbol{z'}]$ *denote the realisers generated over* $x,\boldsymbol{z}$ *and* $y,\boldsymbol{z'}$ *according to* t *and* t'*, respectively, cf. Sect. 3.4. The* $\boldsymbol{z}$ *and* $\boldsymbol{z'}$ *interpret any free term variables* $\boldsymbol{y}$ *in* t *and* t'*, respectively.*

For Def. 6.9, recall that free variables $\boldsymbol{y}$ in t and t' are due to type instantiations, represented by $\boldsymbol{Y}$, of polymorphic subterms in t and t'. The relevant term environment determines $\boldsymbol{z}$ and $\boldsymbol{z'}$. The details of this mutual recursion with the term semantics are omitted to avoid clutter, but the idea should be clear.

For data type operations, the semantics defined in Def. 6.9 in effect captures the actual use of data type operations according to the definability aspect of ***Abs-Bar*** (*p.* 54). The essence is the definability condition for arguments in arrow-type semantics. The PER $\mathcal{D}^{\mathsf{adt}}_{U[\boldsymbol{Y},X]}$ relates two natural numbers if there exist respective terms that are definable as stated by ***Abs-Bar***. The line of reasoning for *abo*-relations on *p.* 121 applies again here for data type operations that cause non-definable arguments to be applied.

Theorem 6.10 *The top-level data type semantics* $[\![X \rhd \mathfrak{T}[X]^{ia}]\!]_{[X\mapsto\mathcal{X}]}$ *is a PER.*

Proof: By induction. Use the fact that for any $t\!:\!U[\boldsymbol{Y},X]^{\mathsf{ra}}$ without free variables of types involving X, we have $t[x] \; ([\![\boldsymbol{Y},X \rhd U[\boldsymbol{Y},X]^{\mathsf{ra}}]\!]^{x,y,X}_{\delta}) \; t[y]$ □

Theorem 6.10 asserts that the data type semantics is a PER at the top level. But we also need to establish what the function spaces inside data types are; so far they appear only as parameterised entities in the overall definition of the top-level data type semantics. The issue is resolved in the following.

Theorem 6.11 *For any* x,y *such that* $x \; [\![X \rhd \mathfrak{T}[X]^{ia}]\!]_{[X\mapsto\mathcal{X}]} \; y$*, the data type semantics* $[\![\boldsymbol{Y},X \rhd U[\boldsymbol{Y},X]^{\mathsf{a}}]\!]^{x,y,X}_{[X\mapsto\mathcal{X}]}$ *is a PER. Moreover, for any* x',y' *such that* $x' \; [\![X \rhd \mathfrak{T}[X]^{ia}]\!]_{[X\mapsto\mathcal{X}]} \; x$ *and* $y \; [\![X \rhd \mathfrak{T}[X]^{ia}]\!]_{[X\mapsto\mathcal{X}]} \; y'$*, it is the case that*

$$[\![\boldsymbol{Y},X \rhd U[\boldsymbol{Y},X]^{\mathsf{a}}]\!]^{x,y,X}_{[X\mapsto\mathcal{X}]} = [\![\boldsymbol{Y},X \rhd U[\boldsymbol{Y},X]^{\mathsf{a}}]\!]^{x',y',X}_{[X\mapsto\mathcal{X}]}$$

Proof: By induction. □

Within a given data type we can now determine the appropriate function spaces for data type operations. If x is any realiser for the data type, by Theorem 6.11,

$[\![\boldsymbol{Y}, X \rhd U[\boldsymbol{Y}, X]^{\mathsf{a}}]\!]^{x,x,X}_{[X \mapsto \mathcal{X}]}$ is a PER, and by convention, we choose this form for function spaces for data type operations. Data type semantics are thus qualified, or parameterised, according to the data type in question.

—

We can now give the relational semantics for data type operations. This inherits the effectual weakening at arrow types. The relations in question here are between data type operations possibly from distinct data types. Therefore, relational semantics bear the qualifications adt and adt' of both data types.

Definition 6.12 (Data Type Semantics for Relations) *For $\Delta = \boldsymbol{E}, \boldsymbol{F}, A, B$, $\Upsilon = \boldsymbol{\rho} \subset \boldsymbol{E} \times \boldsymbol{F}$, $R \subset A \times B$, and environments δ on Δ and υ on Υ, we define*

$$[\![\Delta \shortmid \Upsilon \rhd U[\boldsymbol{\rho}, R]^{ra}]\!]^{\mathsf{adt},\mathsf{adt}'}_{\delta\upsilon} \subset Dom([\![\Delta \rhd U[\boldsymbol{E}, A]^{ra}]\!]^{\mathsf{adt}}_{\delta}) \times Dom([\![\Delta \rhd U[\boldsymbol{F}, B]^{ra}]\!]^{\mathsf{adt}'}_{\delta})$$

where $\mathsf{adt} = x, x, A$, for some $x \in Dom([\![A \rhd \mathfrak{T}[A]^{ia}]\!]_{\delta})$, and $\mathsf{adt}' = x', x', B$, for some $x' \in Dom([\![B \rhd \mathfrak{T}[B]^{ia}]\!]_{\delta})$, by

$$\begin{array}{l}
[\![\Delta \shortmid \Upsilon \rhd U[\boldsymbol{\rho}]]\!]^{\mathsf{adt},\mathsf{adt}'}_{\delta\upsilon} = [\![\Delta \shortmid \Upsilon \rhd U[\boldsymbol{\rho}]]\!]_{\delta\upsilon}, \qquad \textit{for unmarked } U \\
[\![\Delta \shortmid \Upsilon \rhd R^{\mathsf{a}}]\!]^{\mathsf{adt},\mathsf{adt}'}_{\delta\upsilon} = \upsilon(R) \cap \mathcal{D}^{\mathsf{adt},\mathsf{adt}'}_{X} \\
[\![\Delta \shortmid \Upsilon \rhd \forall Y.U[\boldsymbol{\rho}, Y]^{ra}]\!]^{\mathsf{adt},\mathsf{adt}'}_{\delta\upsilon} = (\cap_{\mathcal{R}} [\![\Delta \shortmid \Upsilon, \rho_{k+1} \rhd U[\boldsymbol{\rho}, \rho_{k+1}, R]^{ra}]\!]^{\mathsf{adt},\mathsf{adt}'}_{\delta\upsilon[\rho_{k+1} \mapsto \mathcal{R}]}) \\
[\![\Delta \shortmid \Upsilon \rhd (U[\boldsymbol{\rho}, R]^{ra} \to V[\boldsymbol{\rho}, R]^{ra})^{\mathsf{a}}]\!]^{\mathsf{adt},\mathsf{adt}'}_{\delta\upsilon} = \\
\quad ([\![\Delta \shortmid \Upsilon \rhd U[\boldsymbol{\rho}, R]^{ra}]\!]^{\mathsf{adt},\mathsf{adt}'}_{\delta\upsilon} \to [\![\Delta \shortmid \Upsilon \rhd V[\boldsymbol{\rho}, R]^{ra}]\!]^{\mathsf{adt},\mathsf{adt}'}_{\delta\upsilon}) \cap \mathcal{D}^{\mathsf{adt},\mathsf{adt}'}_{U[\boldsymbol{Y},X] \to V[\boldsymbol{Y},X]}
\end{array}$$

where

$$\begin{array}{rcl}
[n]\ \mathcal{D}^{\mathsf{adt},\mathsf{adt}'}_{U[\boldsymbol{Y},X]}\ [n'] & \stackrel{def}{\Leftrightarrow} & \textit{there exist terms } \Gamma_A \rhd t\!:\!U[\boldsymbol{Y}, A] \textit{ and } \Gamma_B \rhd t'\!:\!U[\boldsymbol{Y}, B] \\
& & \textit{where } \Gamma_X = X, \boldsymbol{Y}, \boldsymbol{y}\!:\!\boldsymbol{V}[\boldsymbol{Y}] \textit{ for } \boldsymbol{V}[\boldsymbol{Y}] \textit{ not containing } X, \\
& & \textit{such that } t[x, \boldsymbol{z}] = n \textit{ and } t'[x', \boldsymbol{z'}] = n'
\end{array}$$

The $\boldsymbol{z}$ and $\boldsymbol{z'}$ interpret $\boldsymbol{y}$. The intersection $\cap_{\mathcal{R}}$ above for universal types ranges over all saturated relations $\mathcal{R} \subset Dom(\mathcal{E}) \times Dom(\mathcal{F})$, for all PERs $\mathcal{E}$ and $\mathcal{F}$.

It is easy to verify that $[\![\Delta \shortmid \Upsilon \rhd U[\boldsymbol{\rho}, R]]\!]^{\mathsf{adt},\mathsf{adt}'}_{\delta\upsilon}$ is saturated w.r.t. $[\![\Delta \rhd U[\boldsymbol{E}, A]]\!]^{\mathsf{adt}}_{\delta}$ and $[\![\Delta \rhd U[\boldsymbol{F}, B]]\!]^{\mathsf{adt}'}_{\delta}$. Using this, one can show that

$$\begin{array}{l}
(\cap_{\mathcal{E} \in \mathbf{PER}} [\![\boldsymbol{Y}, Y_{k+1}, X \rhd U[\boldsymbol{Y}, Y_{k+1}, X]]\!]^{\mathsf{adt}}_{\delta[Y_{k+1} \mapsto \mathcal{E}]})^{\flat} \\
\qquad\qquad = (\cap_{\mathcal{R}} [\![\Delta \shortmid \rho_{k+1} \rhd U[\mathbf{eq}_{\boldsymbol{E}}, \rho_{k+1}, \mathsf{eq}_A]]\!]^{\mathsf{adt},\mathsf{adt}}_{\delta\upsilon[\rho_{k+1} \mapsto \mathcal{R}]})
\end{array}$$

In general we again have

$$[\![U[\boldsymbol{Z}]]\!]^{\mathsf{adt}}_{\delta} = [\![U[\mathbf{eq}_{\boldsymbol{Z}}]]\!]^{\mathsf{adt},\mathsf{adt}}_{\delta\upsilon}$$

This is a manifestation of the Identity Extension Lemma, Theorem 3.2. We have that $[\![\mathsf{eq}_{U[\boldsymbol{Z}]}]\!]^{\mathsf{adt},\mathsf{adt}}_{\delta\upsilon} = [\![U[\boldsymbol{Z}]]\!]^{\mathsf{adt}}_{\delta}$, *i.e.,* the equality predicate on a type has the same interpretation, *i.e.,* a PER, as the type itself.

—

Term semantics follow regular denotational structure. We amalgamate contexts into a single context Γ, and environments into a single environment γ.

Definition 6.13 (Data Type Semantics for Terms)

$$\llbracket \Gamma \triangleright t\!:\!T \rrbracket^{\mathsf{adt}}_{\gamma} \stackrel{def}{=} \llbracket \Gamma \triangleright t\!:\!T \rrbracket_{\gamma}, \qquad \textit{for unmarked } T$$
$$\llbracket \Gamma, x\!:\!U^{\mathsf{ra}} \triangleright x\!:\!U^{\mathsf{ra}} \rrbracket^{\mathsf{adt}}_{\gamma} \stackrel{def}{=} \gamma(x), \qquad \textit{where } \gamma(x) \in \textit{Dom}(\mathcal{A}), \textit{ for } \mathcal{A} = \llbracket \Gamma \triangleright U^{\mathsf{ra}} \rrbracket^{\mathsf{adt}}_{\gamma}$$
$$\llbracket \Gamma \triangleright fu\!:\!V^{\mathsf{ra}} \rrbracket^{\mathsf{adt}}_{\gamma} \stackrel{def}{=} \llbracket \Gamma \triangleright f\!:\!(U \to V)^{\mathsf{ra}} \rrbracket^{\mathsf{adt}}_{\gamma} \llbracket \Gamma \triangleright u\!:\!U^{\mathsf{ra}} \rrbracket^{\mathsf{adt}}_{\gamma}$$
$$\llbracket \Gamma \triangleright \lambda x\!:\!U^{\mathsf{ra}}.t[x]\!:\!(U \to V)^{\mathsf{ra}} \rrbracket^{\mathsf{adt}}_{\gamma} \stackrel{def}{=} n \in \textit{Dom}(\llbracket \Gamma \triangleright (U \to V)^{\mathsf{ra}} \rrbracket^{\mathsf{adt}}_{\gamma}), \quad \textit{for } n \textit{ the}$$
$$\textit{code of a partial recursive function } \lambda a.\llbracket \Gamma, x\!:\!U^{\mathsf{ra}} \triangleright t[x]\!:\!V^{\mathsf{ra}} \rrbracket^{\mathsf{adt}}_{\gamma[x \mapsto a]}$$
$$\llbracket \Gamma \triangleright tA\!:\!U[A]^{\mathsf{ra}} \rrbracket^{\mathsf{adt}}_{\gamma} \stackrel{def}{=} \llbracket \Gamma \triangleright t\!:\!\forall X.U[X]^{\mathsf{ra}} \rrbracket^{\mathsf{adt}}_{\gamma}$$
$$\llbracket \Gamma \triangleright \Lambda X.t[X]\!:\!\forall X.U[X]^{\mathsf{ra}} \rrbracket^{\mathsf{adt}}_{\gamma} \stackrel{def}{=} n, \quad \textit{such that}$$
$$\llbracket \Gamma, X \triangleright t[X]\!:\!U[X]^{\mathsf{ra}} \rrbracket^{\mathsf{adt}}_{\gamma[X \mapsto \mathcal{A}]} = n \textit{ for any PER } \mathcal{A}$$

Note that Γ contains $\mathfrak{x}\!:\!\mathfrak{T}[X]$, and γ must be consistent with the list adt.

—

For the ensuing development, we now need the marked versions of the relevant results from Ch. 5. These follow immediately, since the type marking scheme does not introduce any logical restrictions. The marked version of the axiom schema spParamC also immediately holds under data type semantics.

6.5 New Axioms SubG and QuotG

We now give versions of Sub and Quot that employ abstraction barrier-observing simulation relations. These new axiom schemata hold at any order w.r.t. the data type semantics proposed in the previous section.

There is now the issue of variables of higher-order and universal type. Recall from Sect. 4.4 how the imported proof strategy uses Sub and Quot to effectively replace variables of the quotienting Q in formulae. Here we need to take into account variables of higher-order types over Q. Ostensibly, we must therefore have maps $\mathsf{mono}_U : U[S] \to U[X]$ and $\mathsf{epi}_U : U[X] \to U[Q]$ for any type $U[X]$ over X occurring in the relevant specification axioms Ax. This is feasible, but the semantic validation of these maps is troublesome. Luckily however, it usually suffices to have maps at first order. Consider an occurrence of a term t in Ax involving higher-order variables of types over X. If t is first order of type X, then we can use the first-order maps on the term t as a whole, rather than mapping the higher-order constituents of t. If t is of higher-type $U[X]$ over X, then we cannot do this, of course. However, the occurrence of t must ultimately find itself in an atomic formula, *i.e.,* in contexts of the forms $t =_{U[X]} u$ or $R(t,u)$, where R is a relation variable at higher type. In the first case, we can use the congruence axioms (Sect. 3.3) and replace $t =_{U[X]} u$ by its equivalent formula according to extensional equality, thereby obtaining a first-order formula. In the latter case, we can do nothing, except assume that relation variables at higher-order types do not occur in the axioms of implemented specifications. We do not know how restrictive this is in practice. Looking at some present examples, this does not seem to be an unreasonable assumption, but this is no indication that one will never need relation variables at higher-order types. Nonetheless, to keep things simple, we leave the formalisms using higher-order maps mono_U and epi_U to future development, and work with the above restriction on relation variables now.

What we have just said about variables of higher-order types over X, also applies to variables of universal types over X. Congruence applies here too, and we must assume no relation variables of universal types over X in the axioms of any implemented specification. To summarise, we thus assume:

***FAx*:** The axioms Ax of a specification to be refined have no relation variables at higher-order or universal types. This in turn implies without loss of generality, that Ax has no terms of higher types over X.

—

We omit parameters $\boldsymbol{Z}$; they would appear as in Defs. 4.23 and 4.24.

Definition 6.14 (Existence of Subobjects (SUBG)**)** *For* $\mathsf{abo} = X, S, \mathfrak{r}, \mathfrak{s}$,

$$\begin{array}{l}\textsc{SubG}:\ \forall X\,.\,\forall \mathfrak{r}{:}\mathfrak{T}[X]^{i\mathsf{a}}\,.\,\forall R\subset X\times X\ .\quad (\mathfrak{r}\ \mathfrak{T}[R]^{i\mathsf{a}}{}_{\mathsf{C}}^{X,X,\mathfrak{r},\mathfrak{r}}\ \mathfrak{r})\ \Rightarrow\\ \quad \exists S\,.\,\exists \mathfrak{s}{:}\mathfrak{T}[S]^{i\mathsf{a}}\,.\,\exists R'\subset S\times S\,.\,\exists\,\mathsf{mono}{:}S\to X\ .\\ \qquad \forall s{:}S\ .\ s\ R'\ s \quad\wedge\\ \qquad \forall s,s'{:}S\ .\ s\ R'\ s'\ \Leftrightarrow\ (\mathsf{mono}\,s)\ R\ (\mathsf{mono}\,s')\quad\wedge\\ \qquad \mathfrak{r}\ (\mathfrak{T}[(x{:}X,s{:}S).(x =_X (\mathsf{mono}\,s))]^{i\mathsf{a}}{}_{\mathsf{C}}^{\mathsf{abo}})\ \mathfrak{s}\end{array}$$

Definition 6.15 (Existence of Quotients (QUOTG)**)** *For* $\mathsf{abo} = X, Q, \mathfrak{r}, \mathfrak{q}$,

$$\begin{array}{l}\textsc{QuotG}:\ \forall X\,.\,\forall \mathfrak{r}{:}\mathfrak{T}[X]^{i\mathsf{a}}\,.\,\forall R\subset X\times X\ .\quad (\mathfrak{r}\ \mathfrak{T}[R]^{i\mathsf{a}}{}_{\mathsf{C}}^{X,X,\mathfrak{r},\mathfrak{r}}\ \mathfrak{r}\ \wedge\ \mathit{equiv}(R))\ \Rightarrow\\ \quad \exists Q\,.\,\exists \mathfrak{q}{:}\mathfrak{T}[Q]^{i\mathsf{a}}\,.\,\exists\,\mathsf{epi}{:}X\to Q\ .\\ \qquad \forall x,y{:}X\ .\ x\ R\ y\ \Leftrightarrow\ (\mathsf{epi}\,x) =_Q (\mathsf{epi}\,y)\quad\wedge\\ \qquad \forall q{:}Q.\exists x{:}X\ .\ q =_Q (\mathsf{epi}\,x)\quad\wedge\\ \qquad \mathfrak{r}\ (\mathfrak{T}[(x{:}X,q{:}Q).((\mathsf{epi}\,x) =_Q q)]^{i\mathsf{a}}{}_{\mathsf{C}}^{\mathsf{abo}})\ \mathfrak{q}\end{array}$$

where $\mathit{equiv}(R)$ *specifies* R *to be an equivalence relation.*

Theorem 6.16 SUBG *and* QUOTG *hold in the parametric PER-model of* (Bainbridge et al., 1990)*, under the assumption of data type semantics, and* $\mathtt{HADT}_{Obs}$*.*

In the following, we will write *e.g.,* $[\![U[\mathcal{X}]]\!]$ in place of $[\![X \rhd U[X]]\!]_{[X \mapsto \mathcal{X}]}$, and similar natural abuses of notation. In the following $\Gamma \stackrel{def}{=} X, \mathfrak{x}{:}\mathfrak{T}[X]^{i\mathsf{a}}$.

6.5.1 Soundness of SUBG (proof of Theorem 6.16)

Definition 6.17 (Subobject PER) *Let* $\mathcal{X}$ *be any PER, and* $\Re$ *any relation on* $\mathcal{X}$*. Define the* subobject $R_{\Re}(\mathcal{X})$ *of* $\mathcal{X}$ restricted on $\Re$ *by*

$$n\ R_{\Re}(\mathcal{X})\ m \ \stackrel{def}{\Leftrightarrow}\ n\ \mathcal{X}\ m \ \ \textit{and}\ \ [n]_{\mathcal{X}}\ \Re\ [m]_{\mathcal{X}}$$

As expected we do not in general have $n\ R_{\Re}(\mathcal{X})\ m \ \Leftarrow\ [n]_{\mathcal{X}}\ \Re\ [m]_{\mathcal{X}}$. We do have by definition and symmetry $n\ R_{\Re}(\mathcal{X})\ m \ \Rightarrow\ [n]_{\mathcal{X}}\ \Re\ [m]_{\mathcal{X}}$, and also of course $n\ \mathcal{X}\ m \ \Leftarrow\ n\ R_{\Re}(\mathcal{X})\ m$, but not necessarily the converse implication.

To show the soundness of SUBG, consider an arbitrary PER $\mathcal{X}$, $\mathsf{x} \in [\![\mathfrak{T}[\mathcal{X}]^{i\mathsf{a}}]\!]$, and relation $\Re$ on $\mathcal{X}$. We must exhibit a PER $\mathcal{S}$, a relation $\Re'$ on $\mathcal{S}$, an $\mathsf{s} \in [\![\mathfrak{T}[\mathcal{S}]^{i\mathsf{a}}]\!]$, and map $mono{:}\mathcal{S} \to \mathcal{X}$, all satisfying the following properties,

SUB-1. For all $s \in \mathcal{S}$, $s\ \Re'\ s$

SUB-2. For all $s, s' \in \mathcal{S}$, $s\ \Re'\ s' \ \Leftrightarrow\ mono(s)\ \Re\ mono(s')$

SUB-3. $\mathsf{x}\ [\![\mathfrak{T}[(x \in \mathcal{X}, s \in \mathcal{S}).(x =_{\mathcal{X}} (mono\ s))]^{i\mathsf{a}\,\mathsf{abo}}_{\ \ \mathsf{C}}]\!]\ \mathsf{s}$

We exhibit $\mathcal{S} \stackrel{def}{=} R_{\Re}(\mathcal{X})$, define $mono[n]_{\mathcal{S}} \stackrel{def}{=} [n]_{\mathcal{X}}$, and define $\Re'$ by

$$s\ \Re'\ s' \ \stackrel{def}{\Leftrightarrow}\ mono(s)\ \Re\ mono(s')$$

Well-definedness and (SUB-1) and (SUB-2) follow by definition.

We now postulate that we can construct s as the k-tuple where the i^{th} component is $[e_i]_{[\![T_i[\mathcal{S}]^{r\mathsf{a}}]\!]^{\mathsf{adt}}}$ derived from the i^{th} component $[e_i]_{[\![T_i[\mathcal{X}]^{r\mathsf{a}}]\!]^{\mathsf{adt}'}}$ of x, where $\mathsf{adt} = s, s, S$ for s a realiser of s, and $\mathsf{adt}' = x, x, X$ for x a realiser of x. For each component $g_i : (U[X] \to V[X])^{r\mathsf{a}}$ in $\mathfrak{T}[X]^{i\mathsf{a}}$ we must show for all n, n' s.t. there exist terms t, t' s.t. $\Gamma \rhd t{:}U[X]$, $\Gamma \rhd t'{:}U[X]$, and $t[s] = n$, $t'[s] = n'$, that

verSUB-1. $n\ [\![U[\mathcal{S}]^{r\mathsf{a}}]\!]^{\mathsf{adt}}\ n \ \Rightarrow\ e_i(n)\downarrow$,

verSUB-2. $n\ [\![U[\mathcal{S}]^{r\mathsf{a}}]\!]^{\mathsf{adt}}\ n' \ \Rightarrow\ e_i(n)\ [\![V[\mathcal{S}]^{r\mathsf{a}}]\!]^{\mathsf{adt}}\ e_i(n')$.

The crucial observation that now lets us show the well-definedness of s, is that the realiser s can be assumed to be a realiser x for the existing x. It therefore suffices to show (verSUB-1) and (verSUB-2) for n, n' s.t. $t[x] = n$ and $t'[x] = n'$.

Lemma 6.18 *For any PER $\mathcal{X}$ and relation $\Re$ on $\mathcal{X}$ such that x $[\![\mathfrak{T}[\Re]^{ia}{}_{\mathsf{C}}^{X,X,\mathfrak{x},\mathfrak{x}}]\!]$ x, we have for x a realiser of x, that for any n for which there exists a term t such that $\Gamma \rhd t\!:\!X$ and $t[x] = n$,*

$$[n]_{\mathcal{X}} \ \Re \ [n]_{\mathcal{X}}$$

Proof: This follows from Lemma 5.31. □

Lemma 6.19 *For any n, n' s.t. there exist terms t, t' s.t. $\Gamma \rhd t\!:\!X$, $\Gamma \rhd t'\!:\!X$, and $t[x] = n$, $t'[x] = n'$, where x is a realiser of x,*

$$n \ \mathcal{X} \ n' \ \Leftrightarrow \ n \ \mathcal{S} \ n'$$

Proof: Suppose $n \ \mathcal{X} \ n'$. It suffices to show $[n]_{\mathcal{X}} \ \Re \ [n']_{\mathcal{X}}$. By Lemma 6.18 we get $[n']_{\mathcal{X}} \ \Re \ [n']_{\mathcal{X}}$, and $n \ \mathcal{X} \ n'$ gives $[n]_{\mathcal{X}} = [n']_{\mathcal{X}}$. Suppose $n \ \mathcal{S} \ n'$. By definition this gives $n \ \mathcal{X} \ n'$. □

Lemma 6.20 $n \ [\![U[\mathcal{X}]^{\mathsf{ra}}]\!]^{\mathsf{adt}} \ n' \ \Leftrightarrow \ n \ [\![U[\mathcal{S}]^{\mathsf{ra}}]\!]^{\mathsf{adt}} \ n'$

Proof: This follows from Lemma 6.19 by induction on type structure. □

So let n, n' be as proposed, and recall that we are assuming the existence of x, thus we have $e_i \ [\![(U[\mathcal{X}] \to V[\mathcal{X}])^{\mathsf{ra}}]\!]^{\mathsf{adt}} \ e_i$. We may now use Lemma 6.20 directly and immediately get what we want since e_i satisfies its conditions. However, for illustrative purposes we go a level down. By assumption we have

assmpSUB-1. $n \ [\![U[\mathcal{X}]^{\mathsf{ra}}]\!]^{\mathsf{adt}} \ n \ \Rightarrow \ e_i(n)\!\downarrow$,
assmpSUB-2. $n \ [\![U[\mathcal{X}]^{\mathsf{ra}}]\!]^{\mathsf{adt}} \ n' \ \Rightarrow \ e_i(n) \ [\![V[\mathcal{X}]^{\mathsf{ra}}]\!]^{\mathsf{adt}} \ e_i(n')$.

Showing (verSUB-1) is now easy. By assumption on n, we use Lemma 6.20, and then (assmpSUB-1) yields $e_i(n)\downarrow$. For (verSUB-2) assume $n \ [\![U[\mathcal{S}]^{\mathsf{ra}}]\!]^{\mathsf{adt}} \ n'$. Lemma 6.20 gives $n \ [\![U[\mathcal{X}]^{\mathsf{ra}}]\!]^{\mathsf{adt}} \ n'$, and (assmpSUB-2) gives $e_i(n) \ [\![V[\mathcal{X}]^{\mathsf{ra}}]\!]^{\mathsf{adt}} \ e_i(n')$. Lemma 6.20 gives $e_i(n) \ [\![V[\mathcal{S}]^{\mathsf{ra}}]\!]^{\mathsf{adt}} \ e_i(n')$. This concludes the definition of s.

It is time to verify x $[\![\mathfrak{T}[(x \in \mathcal{X}, s \in \mathcal{S}).(x =_{\mathcal{X}} (mono\ s))]^{ia}{}_{\mathsf{C}}^{\mathsf{abo}}]\!]$ s. Let then $\rho \stackrel{def}{=} (x : \mathcal{X}, s : \mathcal{S}).(x =_{\mathcal{X}} (mono\ s))$. For any component $g_i : (U[X] \to V[X])^{\mathsf{ra}}$ in $\mathfrak{T}[X]^{ia}$. we must show for all $[n] \in [\![U[\mathcal{X}]^{\mathsf{ra}}]\!]^{\mathsf{adt}}$ and $[m] \in [\![U[\mathcal{S}]^{\mathsf{ra}}]\!]^{\mathsf{adt}}$ that

$$[n] \ [\![U[\rho]^{ra}{}_{\mathsf{C}}^{\mathsf{abo}}]\!]^{\mathsf{adt}} \ [m] \ \wedge \ [\![\mathsf{DfnblC}^{\mathsf{abo}}([n],[m])]\!] \ \Rightarrow \ [e_i(n)] \ [\![V[\rho]^{\mathsf{ra}}]\!]^{\mathsf{adt}} \ [e_i(m)]$$

Let e be the realiser of the polymorphic functional of which $\mathsf{DfnblC}^{\mathsf{abo}}$ asserts the existence. This realiser is the same for all instances of the functional, and by construction the realisers for x and s are the same, say x. Hence the $\mathsf{DfnblC}^{\mathsf{abo}}$ clause asserts that $e(x) = n$ and $e(x) = m$, thus $n = m$. If $V[X]$ is X we must show $[e_i(n)]_{\mathcal{X}} = \mathit{mono}([e_i(m)]_{\mathcal{S}})$, *i.e.*, $[e_i(n)]_{\mathcal{X}} = [e_i(m)]_{\mathcal{X}}$, and if $V[X]$ is some $D \in \mathit{Obs}$ we must show $[e_i(n)]_D = [e_i(m)]_D$. Both cases follow since $n = m$.

6.5.2 Soundness of QuotG (proof of Theorem 6.16)

Definition 6.21 (Quotient PER) *Let $\mathcal{X}$ be any PER, and $\Re$ any equivalence relation on $\mathcal{X}$. Define the* quotient $\mathcal{X}/\!\Re$ *of $\mathcal{X}$* w.r.t. $\Re$ *by*

$$n\ \mathcal{X}/\!\Re\ m \ \stackrel{def}{\Leftrightarrow}\ n\ \mathcal{X}\ n \ \ \textit{and}\ \ m\ \mathcal{X}\ m \ \ \textit{and}\ \ [n]_{\mathcal{X}}\ \Re\ [m]_{\mathcal{X}}$$

The following lemma follows immediately by definition.

Lemma 6.22 *For all n, m, we have, provided that $[n]_{\mathcal{X}}$ and $[m]_{\mathcal{X}}$ exist,*

$$n\ \mathcal{X}/\!\Re\ m \ \Leftrightarrow\ [n]_{\mathcal{X}}\ \Re\ [m]_{\mathcal{X}}$$

We also have by definition and reflexivity of $\Re$, $n\ \mathcal{X}\ m \ \Rightarrow\ n\ \mathcal{X}/\!\Re\ m$, but not necessarily the converse implication.

To show the soundness of QuotG, consider any PER $\mathcal{X}$, $\mathsf{x} \in [\![\mathfrak{T}[\mathcal{X}]^{i\mathsf{a}}]\!]$, and equivalence relation $\Re$ on $\mathcal{X}$. We must exhibit a PER $\mathcal{Q}$, a $\mathsf{q} \in [\![\mathfrak{T}[\mathcal{Q}]^{i\mathsf{a}}]\!]$, and map $\mathit{epi} : \mathcal{X} \to \mathcal{Q}$, all satisfying the following properties.

- Quot-1. For all $x, y \in \mathcal{X}$, $x\ \Re\ y \ \Leftrightarrow\ \mathit{epi}(x) =_{\mathcal{Q}} \mathit{epi}(y)$
- Quot-2. For all $q \in \mathcal{Q}$, there exists $x \in \mathcal{X}$ s.t. $q =_{\mathcal{Q}} \mathit{epi}(x)$
- Quot-3. $\mathsf{x}\ [\![\mathfrak{T}[(x \in \mathcal{X}, s \in \mathcal{Q}).(\mathit{epi}(x) =_{\mathcal{X}} q)]^{i\mathsf{a}\,\mathsf{abo}}_{\ \ \mathsf{C}}]\!]\ \mathsf{q}$

We exhibit $\mathcal{Q} \stackrel{def}{=} \mathcal{X}/\!\Re$, and define $\mathit{epi}([n]_{\mathcal{X}}) \stackrel{def}{=} [n]_{\mathcal{Q}}$. Well-definedness, (Quot-1) and (Quot-2) follow by definition and Lemma 6.22.

Both the construction of q and the rest of the proof follow analogously to the case for SubG. Things are a bit simpler, because we have Lemma 6.22.

6.6 Using QuotG and SubG

We first illustrate schematically the general use of QuotG and SubG in proving a refinement $SP \underset{F}{\rightsquigarrow} SP'$. Then we instantiate this schema by giving a specific example illustrating the use of QuotG and SubG. It might be more instructive to go straight to the specific example. The general schema is more for future use and for completeness of presentation.

First let us make precise what the definitions of specification and specification refinement are, now in general versions after our work in the previous chapter.

If we choose to relate to the syntactic models of (Hasegawa, 1991), we can keep our notion of observational equivalence ObsEq from Def. 4.4, and all the definitions of specification and specification refinement remain the same.

On the other hand, if we want to relate to the non-syntactic parametric PER-model, or the model obtained using the data type semantics in the previous section, we have to use the notion of closed observational equivalence ObsEqC from Def. 5.25. This then percolates to all definitions involving ObsEq, whence the general versions of these definitions are obtained by replacing ObsEq by ObsEqC. For example, the general definition of abstract data type specification now reads:

Definition 6.23 (General ADT Specification) *An* abstract data type specification *SP is a tuple $\langle\langle Sig_{SP}, \Theta_{SP}\rangle, \mathfrak{T}^e_{SP}, Obs_{SP}\rangle$ where*

$$Sig_{SP} \stackrel{def}{=} \exists X.\mathfrak{T}_{SP}[X],$$

$$\Theta_{SP}(u) \stackrel{def}{=} \exists X.\exists \mathfrak{x}{:}\mathfrak{T}^e_{SP}[X]\ .\ u\ \mathsf{ObsEqC}^{Obs_{SP}}\ (\mathsf{pack}X\mathfrak{x}|_{\mathfrak{T}_{SP}})\ \wedge\ Ax_{SP}[X,\mathfrak{x}],$$

where $Ax_{SP}[X,\mathfrak{x}]$ is a finite set of formulae in the logic. If $\Theta_{SP}(u)$ is derivable, then u is said to be a realisation *of SP. We assume that Obs_{SP} contains*

- *none or more closed inductive types, such as* Bool *or* Nat*,*
- *all parameters $\boldsymbol{Z}$, in case $\mathfrak{T}_{SP}[X]$ has free $\boldsymbol{Z}$ other than X.*

The only difference here from Def. 4.6 and Def. 4.12 is the use of ObsEqC instead of ObsEq. The definitions of polymorphic abstract data type specification (Def. 4.9), and constructor specification (Def. 4.15) are modified similarly. Specification refinement $SP \underset{F}{\rightsquigarrow} SP'$ goes as before, *i.e.*, SP' is a refinement of SP, via constructor $F : Sig_{SP'} \rightarrow Sig_{SP}$ if

$$\forall u{:}Sig_{SP'}\ .\ \Theta_{SP'}(u)\ \Rightarrow\ \Theta_{SP}(Fu)$$

is derivable. Note that this now involves ObsEqC in $\Theta_{SP'}$ and Θ_{SP}.

6.6.1 A General Schema

We now illustrate schematically the general use of QuotG and SubG in proving a refinement $SP \underset{F}{\rightsquigarrow} SP'$, for a stable constructor $F : Sig_{SP'} \rightarrow Sig_{SP}$. We will assume the following persistency clause that we introduced in Sect. 4.6 when showing the correspondence between refinement notions.

$$\begin{array}{l}\forall X, Y.\forall \mathfrak{x}\colon Sig_{SP'}[X], \mathfrak{y}\colon Sig_{SP}[Y]\ .\\ \qquad (\Theta_{SP'}(\mathsf{pack}X\mathfrak{x})\ \wedge\ (\mathsf{pack}Y\mathfrak{y})\ \mathsf{ObsEqC}^{Obs_{SP}}_{\mathfrak{T}_{SP}}\ F(\mathsf{pack}X\mathfrak{x}))\\ \qquad\qquad \Rightarrow\ (\exists \mathfrak{y}^e\colon Sig^e_{SP}\ .\ \mathfrak{y} = \mathfrak{y}^e|_{\mathfrak{T}_{SP}}\ \wedge\ Ax(Hid_{SP})[Y, \mathfrak{y}^e])\end{array}$$

where $Ax(Hid_{SP})$ are the axioms in SP that contain operations other than those given in $\mathfrak{T}_{SP}$. The main task is to derive

$$\forall u\colon Sig_{SP'}\ .\ \Theta_{SP'}(u)\ \Rightarrow\ \Theta_{SP}(Fu)$$

In other words, for arbitrary $u\colon Sig_{SP}$, and assuming

$$\exists A.\exists \mathfrak{a}\colon \mathfrak{T}^e_{SP'}[A]\ .\ u\ \mathsf{ObsEqC}^{Obs_{SP'}}_{\mathfrak{T}_{SP'}}\ (\mathsf{pack}A\,\mathfrak{a}|_{\mathfrak{T}_{SP'}})\ \ \wedge\ \ Ax_{SP'}[A, \mathfrak{a}]$$

we must derive

$$\exists B.\exists \mathfrak{b}\colon \mathfrak{T}^e_{SP}[B]\ .\ (Fu)\ \mathsf{ObsEqC}^{Obs_{SP}}_{\mathfrak{T}_{SP}}\ (\mathsf{pack}B\,\mathfrak{b}|_{\mathfrak{T}_{SP}})\ \ \wedge\ \ Ax_{SP}[B, \mathfrak{b}]$$

Let A and $\mathfrak{a}$ be projected out from the assumption. Since F is assumed stable, we may substitute $(\mathsf{pack}A\,\mathfrak{a}|_{\mathfrak{T}_{SP'}})$ for u, *i.e.,* if we show

$$(\mathsf{pack}B\,\mathfrak{b}|_{\mathfrak{T}_{SP}})\ \mathsf{ObsEqC}^{Obs_{SP}}_{\mathfrak{T}_{SP}}\ F(\mathsf{pack}A\,\mathfrak{a}|_{\mathfrak{T}_{SP'}})$$

we have also shown $(\mathsf{pack}B\,\mathfrak{b}|_{\mathfrak{T}_{SP}})\ \mathsf{ObsEqC}^{Obs_{SP}}_{\mathfrak{T}_{SP}}\ Fu$ since

$$(\mathsf{pack}A\,\mathfrak{a}|_{\mathfrak{T}_{SP'}})\ \mathsf{ObsEqC}^{Obs_{SP'}}_{\mathfrak{T}_{SP'}}\ u\ \Rightarrow\ F(\mathsf{pack}A\,\mathfrak{a}|_{\mathfrak{T}_{SP'}})\ \mathsf{ObsEqC}^{Obs_{SP}}_{\mathfrak{T}_{SP}}\ Fu$$

and observational equivalence is transitive. Let $(\mathsf{pack}A'\mathfrak{a}')$ denote $F(\mathsf{pack}A\,\mathfrak{a}|_{\mathfrak{T}_{SP'}})$.

In accordance with our earlier discussion, we assume no relation variables of higher or universal types over X in Ax_{SP}. Without further loss of generality, we also assume that all equalities in Ax_{SP} at types over X are first order, *i.e.,* simply over X. As pointed out, this is justified by extensionality which follows by the congruence axioms in Sect. 3.3.

Let $\mathfrak{a}'^e$ be as asserted by persistency from $\mathfrak{a}$. Following the strategy of algebraic specification, we attempt to define a possibly partial congruence $\sim$ on A' such that one can show $Ax_{SP}[A', \mathfrak{a}'^e]_{rel}$, where $Ax_{SP}[A', \mathfrak{a}'^e]_{rel}$ is obtained from $Ax_{SP}[A', \mathfrak{a}'^e]$ by replacing all occurrences of $=_{A'}$ by $\sim$. If $\sim$ is partial, every formula ϕ whose free variables of type A' are among $x_1, \ldots, x_n$, must be conditioned by requiring that these variables are in $Dom(\sim)$, *viz.* $\wedge_i(x_i\ \sim\ x_i)\ \Rightarrow\ \phi$. In showing $Ax_{SP}[A', \mathfrak{a}'^e]_{rel}$, one uses $Ax(Hid_{SP})$, and information about the constructor F; either direct definitional information as expressed in $\mathfrak{a}'$, or perhaps abstract information provided by a specification of F.

Suppose now $Ax_{SP}[A', \mathfrak{a}'^e]_{rel}$ is derivable. First, since $\sim$ is an axiomatisation of a partial congruence, we have $\mathfrak{a}'\ \mathfrak{T}_{SP}[\sim]^{A',A',\mathfrak{a}',\mathfrak{a}'}_{\mathsf{C}}\ \mathfrak{a}'$. We use SUBG to get $S_{A'}$, $\mathfrak{s}_{\mathfrak{a}'}$ and $\sim' \subset S_{A'} \times S_{A'}$, and map $\mathsf{mono}\colon S_{A'} \to A$ such that we can derive

(s1) $\forall s\!:\!S_{A'}\ .\ s\ \sim'\ s$
(s2) $\forall s.s'\!:\!S_{A'}\ .\ s\ \sim'\ s'\ \Leftrightarrow\ (\mathsf{mono}\, s)\ \sim\ (\mathsf{mono}\, s')$
(s3) $\mathfrak{a}'\ (\mathfrak{T}_{SP}[(a\!:\!A, s\!:\!S_{A'}).(a =_A (\mathsf{mono}\, s))]^{\mathsf{abo}}_{\mathsf{C}})\ \mathfrak{s}_{\mathfrak{a}'}$

By $(s2)$ we get $\mathfrak{s}_{\mathfrak{a}'}\ \mathfrak{T}_{SP}[\sim']^{S_{A'},S_{A'},\mathfrak{s}_{\mathfrak{a}'},\mathfrak{s}_{\mathfrak{a}'}}_{\mathsf{C}}\ \mathfrak{s}_{\mathfrak{a}'}$. We also get $equiv(\sim')$ by $(s1)$. We now use QuotG to get Q and $\mathfrak{q}\!:\!\mathfrak{T}_{SP}[Q]$ and maps $\mathsf{epi}\!:\!S_{A'}\!\to\! Q$ s.t.

(q1) $\forall s,s'\!:\!S_{A'}\ .\ s\ \sim'\ s'\ \Leftrightarrow\ (\mathsf{epi}\, s) =_Q (\mathsf{epi}\, s')$
(q2) $\forall q\!:\!Q.\exists s\!:\!S_{A'}\ .\ q =_Q (\mathsf{epi}\, s)$
(q3) $\mathfrak{s}_{\mathfrak{a}'}\ (\mathfrak{T}_{SP}[(s\!:\!S_{A'}, q\!:\!Q).((\mathsf{epi}\, s) =_Q q)]^{\mathsf{abo}'}_{\mathsf{C}})\ \mathfrak{q}$

Thus we should exhibit Q for B, and $\mathfrak{q}^e$ for $\mathfrak{b}$, where we postulate that we can obtain $\mathfrak{q}^e$ by persistency. It then remains to derive

1. $(\mathsf{pack}Q\mathfrak{q})\ \mathsf{ObsEqC}^{Obs_{SP}}_{\mathfrak{T}_{SP}}\ (\mathsf{pack}A'\mathfrak{a}')$
2. $Ax_{SP}[Q, \mathfrak{q}^e]$

To show the derivability of (1), it suffices to observe that, through Theorem 5.33, $(s3)$ and $(q3)$ give

$$(\mathsf{pack}A'\mathfrak{a}')\ \mathsf{ObsEqC}^{Obs_{SP}}_{\mathfrak{T}_{SP}}\ (\mathsf{pack}S_{A'}\mathfrak{s}_{\mathfrak{a}'})\ \mathsf{ObsEqC}^{Obs_{SP}}_{\mathfrak{T}_{SP}}\ (\mathsf{pack}Q\mathfrak{q})$$

This thereby warrants the existence of $\mathfrak{q}^e$ according to persistency. For (2) we must show the derivability of $\phi[Q,\mathfrak{q}^e]$ for every conjunct ϕ in Ax_{SP}. We indicate how to deal with equations at type X, since these encompass the main issues. Other formulae are then dealt with in a standard manner. Consider therefore any $\forall \boldsymbol{x}\ .\ u[Q,\mathfrak{q}^e] =_Q v[Q,\mathfrak{q}^e]$ in $Ax_{SP}[Q,\mathfrak{q}^e]$.

We must now deal with the variables of types over Q occurring in the equation. As argued previously, it suffices to look at first-order terms in which these variables occur. For clarity we illustrate the situation with one such term. The generalisation is obvious. Let $q : Q$ be a term in $u[Q,\mathfrak{q}^e]$ or $v[Q,\mathfrak{q}^e]$, involving variables of types over Q. We may by $(q2)$ assume an $s_q : S_{A'}$ s.t. $(\mathsf{epi}\, s_q) =_Q q$. Since we succeeded in deriving $Ax_{SP}[A',\mathfrak{a}'^e]_{rel}$, we can derive $((\mathsf{mono}\, s_q)\ \sim\ (\mathsf{mono}\, s_q))\ \Rightarrow\ u[\mathfrak{a}'^e][(\mathsf{mono}\, s_q)]\ \sim\ v[\mathfrak{a}'^e][(\mathsf{mono}\, s_q)]$. By $(s2)$ and $(s3)$ this is equivalent to $s_q\ \sim'\ s_q\ \Rightarrow\ u[\mathfrak{s}_{\mathfrak{a}'}{}^e][s_q]\ \sim'\ v[\mathfrak{s}_{\mathfrak{a}'}{}^e][s_q]$, which by $(s1)$ is equivalent to $u[\mathfrak{s}_{\mathfrak{a}'}{}^e][s_q]\ \sim'\ v[\mathfrak{s}_{\mathfrak{a}'}{}^e][s_q]$. Here $\mathfrak{s}_{\mathfrak{a}'}{}^e$ is obtained by persistency in the same way as $\mathfrak{q}^e$ through (1). Then from $(q1)$, we derive $(\mathsf{epi}\, u[\mathfrak{s}_{\mathfrak{a}'}{}^e][s_q]) =_Q (\mathsf{epi}\, v[\mathfrak{s}_{\mathfrak{a}'}{}^e][s_q])$. By $(q3)$ we get $(\mathsf{epi}\, u[\mathfrak{s}_{\mathfrak{a}'}{}^e][s_q]) = u[\mathfrak{q}^e]$ and $(\mathsf{epi}\, v[\mathfrak{s}_{\mathfrak{a}'}{}^e][s_q]) = v[\mathfrak{q}^e]$.

—

The schema just presented has variations. In practice one might come up with a partial congruence $\sim$, abstain from showing $Ax_{SP}[A', \mathfrak{a}'^{e}]_{rel}$ as a whole, and instead attempt to show $Ax_{SP}[Q, \mathfrak{q}^{e}]$ directly, using the definition of $\sim$.

6.6.2 A Specific Example

On the following pages we present a specific example instantiating the above schema in principle. The example is an adaptation of an example in (Honsell et al., 2000) on the semantic level demonstrating refinement at higher-order with pre-logical relations. Pre-logical relations are shown in (Honsell et al., 2000) to be exactly the adequate notion for explaining refinement. Their example also shows that refinement via logical relations fails. Here, we wish to express the example in the logical syntactic framework of System F and the logic for relational parametricity. In Ch. 5 we showed the equivalence between observational equivalence and the existence of abstraction barrier-observing simulation relations. This suggests that in our setting, abstraction barrier-observing simulation relations are exactly the adequate notion for explaining refinement.

The importance of this example is that it demands the use of abstraction barrier-observing simulation relations. The desired refinement does not go through using standard simulation relations. Recall that standard simulation relations could in theory suffice also at higher order, because the existence of simulation relations implies observational equivalence by PARAM, regardless. However, this does not take into account that a standard simulation relation may not exist.

For clarity, we will display axioms in a non-standard way, omitting the conjunction symbol between axioms, and refraining from dot-prefixing operations with the relevant data type. We will also use functions defined earlier in the text, in particular some from Sect. 3.2.4.

—

Example 6.24 (Adapted from (Honsell et al., 2000)**)** Here, real numbers are represented by sequences of natural numbers. Sequences are implemented by functions. Real-number operations are thus implemented by operations on natural-number functions.

$\mathsf{REAL} \stackrel{def}{=} \langle\langle Sig_{\mathsf{REAL}}, \Theta_{\mathsf{REAL}}\rangle, \{\mathsf{Bool}_\perp\}\rangle$, where
$Sig_{\mathsf{REAL}} = \exists X.\mathfrak{T}_{\mathsf{REAL}}[X]$
for $\mathfrak{T}_{\mathsf{REAL}}[X] = (\mathsf{0}\!:\!X,\ \mathsf{1}\!:\!X,$
$\quad +\!:\!X \to X \to X,\ \ *\!:\!X \to X \to X,\ \ -\!:\!X \to X,$
$\quad \mathsf{max}\!:\!X \to X \to X,\ \ \mathsf{sup}\!:\!(X \to X) \to X,$
$\quad <\!:\!X \to X \to \mathsf{Bool}_\perp)$
$\Theta_{\mathsf{REAL}}(u) = \exists X.\exists\mathfrak{x}\!:\!\mathfrak{T}_{\mathsf{REAL}}[X]\ .\ u\ \mathsf{ObsEqC}^{\{\mathsf{Bool}_\perp\}}\ (\mathsf{pack}X\mathfrak{x})\ \wedge\ Ax_{\mathsf{REAL}}$
for $Ax_{\mathsf{REAL}} \stackrel{def}{=}$ *open* $\mathfrak{x}$ *in* `i.e., open access, can omit prefixing with` $\mathfrak{x}$

$$\forall x, y, z\!:\!X\ .$$
$$x + y = y + x \qquad x * y = y * x$$
$$x + (y + z) = (x + y) + z \qquad x * (y * z) = (x * y) * z$$
$$x * (y + z) = (x * y) + (x * z)$$
$$x + \mathsf{0} = x \qquad x * \mathsf{1} = x$$
$$x + -x = \mathsf{0}$$
$$x * y = \mathsf{0}\ \Rightarrow\ x = \mathsf{0} \vee y = \mathsf{0}$$
`integral domain axioms`

$$\forall x, y\!:\!X\ .\ x \leq y \vee y \leq x$$
X `totally ordered by` $\leq$

$$[\leq\ \stackrel{def}{=} (x\!:\!X, y\!:\!X)\ .\ (\exists z\!:\!X.y = x + (z * z))]$$

$$\forall x, y\!:\!X\ .\ x \leq y\ \Rightarrow\ (\mathsf{max}\,x\,y) = y\ \wedge\ (y \leq x\ \Rightarrow\ (\mathsf{max}\,x\,y) = x)$$

$$\forall f\!:\!X \to X\ .\ (\exists z\!:\!X.\forall x\!:\!X\ .\ \mathsf{0} \leq x\ \wedge\ x \leq \mathsf{1}\ \Rightarrow\ f(x) \leq z)$$
$$\Rightarrow\ (\forall z\!:\!X\ .\ (\mathsf{sup}f) \leq z\ \Leftrightarrow\ \forall x\!:\!X.\mathsf{0} \leq x\ \wedge\ x \leq \mathsf{1}\ \Rightarrow\ f(x) \leq z)$$
`sup is w.r.t.` $[0,1]$

$$\forall x, y\!:\!X\ .$$
$$\neg(y \leq x)\ \Rightarrow\ x < y = \mathsf{true}$$
$$\neg(x \leq y)\ \Rightarrow\ x < y = \mathsf{false}$$
$$x = y\ \Rightarrow\ x < y = \mathsf{bot}$$

Division is not included in this specification, but the limited signature ensures total operations. Every algebraic real is definable by a closed term in the language of REAL. Algebraic reals are a subset of the recursive or computable reals, since not every recursive real is definable by a closed term. Models for REAL include the full set-theoretic hierarchy over $\mathbb{R}$, and also models in which the first-order function space only contains continuous functions (Normann, 1998).

The specification REAL will be refined to a specification specifying natural numbers and operators including a primitive recursion combinator. The specification actually gives the language for PCF (Plotkin, 1977), and the axioms ensure that models of the specification are models of PCF in the sense of Scott-domains (Plotkin, 1977) or game-models (Abramsky et al., 1996).

$$
\begin{array}{l}
\mathsf{PCF} \stackrel{\mathit{def}}{=} \langle\langle \mathit{Sig}_{\mathsf{PCF}}, \Theta_{\mathsf{PCF}}\rangle, \{\mathsf{Nat}\}\rangle, \text{ where}\\
\mathit{Sig}_{\mathsf{PCF}} = \exists X.\mathfrak{T}_{\mathsf{PCF}}[X]\\
\text{for } \mathfrak{T}_{\mathsf{PCF}}[X] = (0\!:\!X, \quad \mathsf{succ}\!:\!X\to X, \quad \mathsf{pred}\!:\!X\to X,\\
\qquad\qquad\qquad \mathsf{ifzero}\!:\!X\to X\to X\to X,\\
\qquad\qquad\qquad \mathsf{Y}^T\!:\!(T\to T)\to T \qquad \textit{for all types } T \textit{ over } X)\\
\Theta_{\mathsf{PCF}}(u) = \exists X.\exists\mathfrak{x}\!:\!\mathfrak{T}_{\mathsf{PCF}}[X]\;.\;u\;\mathsf{ObsEqC}^{\{\mathsf{Nat}\}}\;(\mathsf{pack}X\mathfrak{x})\;\wedge\;\mathit{Ax}_{\mathsf{PCF}}\\
\text{for } \mathit{Ax}_{\mathsf{PCF}} \stackrel{\mathit{def}}{=} \textit{open } \mathfrak{x} \textit{ in}
\end{array}
$$

$$
\begin{array}{ll}
\forall x\!:\!X\;.\;\neg(x\downarrow_{\mathfrak{x}}) \;\Leftrightarrow\; x = \bot_{\mathfrak{x}} & \texttt{unique non-terminating}\\
& \texttt{inhabitant of type } X\\
\quad [x\downarrow_{\mathfrak{x}} \stackrel{\mathit{def}}{=} (\mathsf{ifzero}\,x\,0\,0) = 0] & \\
\quad [\bot_{\mathfrak{x}} \stackrel{\mathit{def}}{=} \mathsf{Y}^X(\lambda z\!:\!X.z)] &
\end{array}
$$

$$
\begin{array}{lll}
\forall x,y,z\!:\!X\;. & & \\
\quad\quad \mathsf{succ}\,x = \mathsf{succ}\,y \;\Rightarrow\; x = y & & \texttt{Peano axioms for}\\
0 \neq \mathsf{succ}\,x & x \neq 0 \;\Rightarrow\; \exists y\;.\;x = \mathsf{succ}\,y & \texttt{succ and 0}\\
x + 0 = x & x + \mathsf{succ}\,y = \mathsf{succ}\,(x+y) & \\
(\phi(0) \;\wedge\; \forall x.(\phi(x) \;\Rightarrow\; \phi(\mathsf{succ}\,x))) \;\Rightarrow\; \forall x.\phi(x) & & \textit{for all } \phi
\end{array}
$$

$$
\begin{array}{ll}
\forall x,y,z\!:\!X\;. & \\
\quad \mathsf{pred}\,0 = 0 & \mathsf{pred}(\mathsf{succ}\,x) = x\\
\mathsf{ifzero}\,0\,y\,z = y & \mathsf{ifzero}(\mathsf{succ}\,x)\,y\,z = z\\
\quad\quad \mathsf{ifzero}\,\bot_{\mathfrak{x}}\,y\,z = \bot_{\mathfrak{x}} &
\end{array}
$$

$$
\begin{array}{ll}
\forall f\!:\!T\to T\;.\;\forall z\!:\!T\;. & \\
\mathsf{Y}^T f = f(\mathsf{Y}^T f) & z = (fz) \;\Rightarrow\; \mathsf{Y}^T f \sqsubseteq_T z \;\textit{ for all types } T \textit{ over } X
\end{array}
$$

$$
[\sqsubseteq_T \;\stackrel{\mathit{def}}{=}\; (z\!:\!T, z'\!:\!T)\;.\;(\forall f\!:\!T\to X\;.\;(fz\downarrow_{\mathfrak{x}}) \;\Rightarrow\; (fz'\downarrow_{\mathfrak{x}}))]
$$

This concludes the specifications of REAL and PCF. We will use syntactic sugar for PCF terms. Sugared terms will bear the subscript of the relevant collection of operations. Examples are the terms $\downarrow_{\mathfrak{r}}$ and $\perp_{\mathfrak{r}}$ used in Ax_{PCF}. Usually, the underlying definitions will be obvious, but when this is not the case, a sugared term will signify that the indicated function is definable using the given operations.

We now show that $\mathsf{REAL} \underset{F}{\rightsquigarrow} \mathsf{PCF}$ for

$$F \stackrel{def}{=} \lambda u\!:\!Sig_{\mathsf{PCF}}.\mathsf{unpack}(u)(Sig_{\mathsf{REAL}})(\Lambda X.\lambda\mathfrak{r}\!:\!\mathfrak{T}_{\mathsf{PCF}}[X]\ .\ (\mathsf{pack}\,(X\to X)\,\mathfrak{r}'))$$

where

$$\begin{aligned}\mathfrak{r}' \stackrel{def}{=} (&0 = \lambda x\!:\!X\ .\ 1_{\mathfrak{r}},\ 1 = \lambda x\!:\!X\ .\ \mathfrak{r}.\mathsf{ifzero}(x, 2_{\mathfrak{r}}, 1_{\mathfrak{r}}),\\ &+ = Plus_{\mathfrak{r}},\ * = Mult_{\mathfrak{r}},\ - = Neg_{\mathfrak{r}},\\ &\mathsf{max} = Max_{\mathfrak{r}},\\ &\mathsf{sup} = Sup_{\mathfrak{r}},\\ &< = Less_{\mathfrak{r}}\ )\end{aligned}$$

The chosen encoding of reals is in terms of sequences of natural numbers r, where $r_i \leq 2$, for all $i \geq 2$, thus representing the real number $r_0 - r_1 + \sum_{i=2}^{\infty} 2^{1-i}(r_i - 1)$. Sequences are represented by functions on natural numbers.

Details for the PCF terms $Plus_{\mathfrak{r}}$, $Mult_{\mathfrak{r}}$, $Neg_{\mathfrak{r}}$, and $Less_{\mathfrak{r}}$ appear in (Plume, 1998). The terms $Max_{\mathfrak{r}}$ and $Sup_{\mathfrak{r}}$ are derivable from code in (Simpson, 1998).

We now need to show the derivability of

$$\forall u\!:\!Sig_{\mathsf{PCF}}\ .\ \Theta_{\mathsf{PCF}}(u)\ \Rightarrow\ \Theta_{\mathsf{REAL}}(Fu)$$

That is, we must for arbitrary $u\!:\!Sig_{\mathsf{PCF}}$ derive

$$\exists B.\exists\mathfrak{b}\!:\!\mathfrak{T}_{\mathsf{REAL}}[B]\ .\ (\mathsf{pack}B\mathfrak{b})\ \mathsf{ObsEqC}^{\{\mathsf{Bool}_\perp\}}_{\mathfrak{T}_{\mathsf{REAL}}}\ (Fu)\quad\wedge\quad Ax_{\mathsf{REAL}}[B,\mathfrak{b}]$$

assuming $\exists A.\exists\mathfrak{a}\!:\!\mathfrak{T}_{\mathsf{PCF}}[A]\ .\ (\mathsf{pack}A\mathfrak{a})\ \mathsf{ObsEqC}^{\{\mathsf{Nat}\}}_{\mathfrak{T}_{\mathsf{PCF}}}\ u\quad\wedge\quad Ax_{\mathsf{PCF}}[A,\mathfrak{a}]$. Let A and $\mathfrak{a}$ be projected out from the assumption. Since F is stable, we may substitute $(\mathsf{pack}A\mathfrak{a})$ for u, *i.e.,* if we show

$$(\mathsf{pack}B\mathfrak{b})\ \mathsf{ObsEqC}^{\{\mathsf{Bool}_\perp\}}_{\mathfrak{T}_{\mathsf{REAL}}}\ F(\mathsf{pack}A\mathfrak{a})$$

we have also shown $(\mathsf{pack}B\mathfrak{b})\ \mathsf{ObsEqC}^{\{\mathsf{Bool}_\perp\}}_{\mathfrak{T}_{\mathsf{REAL}}}\ Fu$ since

$$(\mathsf{pack}A\mathfrak{a})\ \mathsf{ObsEqC}^{\{\mathsf{Nat}\}}_{\mathfrak{T}_{\mathsf{PCF}}}\ u\ \Rightarrow\ F(\mathsf{pack}A\mathfrak{a})\ \mathsf{ObsEqC}^{\{\mathsf{Bool}_\perp\}}_{\mathfrak{T}_{\mathsf{REAL}}}\ Fu$$

and observational equivalence is transitive. Let $(\mathsf{pack}(A\to A)\mathfrak{a}')$ denote $F(\mathsf{pack}A\mathfrak{a})$.

In order to follow the universal proof strategy, we must axiomatise a suitable congruence. The congruence must relate functions that represent the same real.

Moreover, certain functions do not represent reals, so the congruence should be partial. First define for any $r\!:\!A\!\rightarrow\!A$,

$$IsReal(r) \stackrel{def}{=} \forall a\!:\!A\ .\ \neg(a = 0_{\mathfrak{a}} \vee a = 1_{\mathfrak{a}}) \ \Rightarrow\ (r\,a) = 0_{\mathfrak{a}} \vee (r\,a) = 1_{\mathfrak{a}} \vee (r\,a) = 2_{\mathfrak{a}}$$

$$Val\!:\!(A\!\rightarrow\!A)\!\rightarrow\!A$$

$$Val \stackrel{def}{=} \lambda r\!:\!A\!\rightarrow\!A\ .\ (r\,0_{\mathfrak{a}}) - (r\,1_{\mathfrak{a}}) + \Sigma_{\mathfrak{a}\,i=2_{\mathfrak{a}}}^{\infty} 2_{\mathfrak{a}}^{1_{\mathfrak{a}}-i}((r\,i) - 1_{\mathfrak{a}})$$

Then we define

$$\sim\ \stackrel{def}{=} (r\!:\!A\!\rightarrow\!A,\, r'\!:\!A\!\rightarrow\!A)\ .\ (IsReal(r)\ \wedge\ IsReal(r')\ \wedge\ Val(r) = Val(r'))$$

Since $\sim$ is partial, we use SUBG prior to using QUOTG. In order to use SUBG, we must first show $\mathfrak{a}'\ \mathfrak{T}_{\mathsf{REAL}}[\sim]^{\mathsf{abo}}_{\mathsf{C}}\ \mathfrak{a}'$. For example, $\mathfrak{a}'.0\ \sim\ \mathfrak{a}'.0$ is derivable, since $\mathfrak{a}'.0 = \lambda x\!:\!A\ .\ 1_{\mathfrak{r}}$, whereby $IsReal(\mathfrak{a}'.0)$ and $Val(\mathfrak{a}'.0) = Val(\mathfrak{a}'.0)$ are derivable.

We cannot use standard simulation relations here, because it is not the case that $\mathfrak{a}'\ \mathfrak{T}_{\mathsf{REAL}}[\sim]\ \mathfrak{a}'$ is derivable. One of the things one must derive in order to derive $\mathfrak{a}'\ \mathfrak{T}_{\mathsf{REAL}}[\sim]\ \mathfrak{a}'$, is

$$\mathfrak{a}'.\mathsf{sup}\ ((\sim\rightarrow\sim)\rightarrow\sim)\ \mathfrak{a}'.\mathsf{sup}$$

This, in fact, fails. To derive $\mathfrak{a}'.\mathsf{sup}\ ((\sim\rightarrow\sim)\rightarrow\sim)\ \mathfrak{a}'.\mathsf{sup}$, we must for any $f\!:\!(A\!\rightarrow\!A)\!\rightarrow\!(A\!\rightarrow\!A)$ and $g\!:\!(A\!\rightarrow\!A)\!\rightarrow\!(A\!\rightarrow\!A)$ that satisfy $f\ (\sim\rightarrow\sim)\ g$, show that $(\mathfrak{a}'.\mathsf{sup}\,f)\ \sim\ (\mathfrak{a}'.\mathsf{sup}\,g)$. But there happens to exist a for us pathological, PCF term $funny_{\mathfrak{a}} : (A \rightarrow A) \rightarrow (A \rightarrow A)$, that implements the constant zero function, and that satisfies $funny_{\mathfrak{a}}\ (\sim\rightarrow\sim)\ 0^{\rightarrow}_{\mathfrak{a}}$, for $0^{\rightarrow}_{\mathfrak{a}} \stackrel{def}{=} \lambda r : (A\rightarrow A).\mathfrak{a}.0$, but for which $\mathfrak{a}'.\mathsf{sup}$ fails by yielding $(\mathfrak{a}'.\mathsf{sup}\,funny_{\mathfrak{a}}) = \lambda a : A\ .\ if_{\mathfrak{a}}\ x <_{\mathfrak{a}} 2_{\mathfrak{a}}\ then_{\mathfrak{a}}\ 1_{\mathfrak{a}}\ else_{\mathfrak{a}}\ \bot_{\mathfrak{a}}$, thereby violating $(\mathfrak{a}'.\mathsf{sup}\,funny_{\mathfrak{a}})\ \sim\ (\mathfrak{a}'.\mathsf{sup}\,0^{\rightarrow}_{\mathfrak{a}})$.

In contrast, the abstraction barrier-observing notion of simulation relation weeds out this counter-example. To show $\mathfrak{a}'\ \mathfrak{T}_{\mathsf{REAL}}[\sim]^{\mathsf{abo}}_{\mathsf{C}}\ \mathfrak{a}'$, we have to derive, among other things,

$$\mathfrak{a}'.\mathsf{sup}\ ((\sim\rightarrow\sim)\rightarrow\sim)^{\mathsf{abo}}_{\mathsf{C}}\ \mathfrak{a}'.\mathsf{sup}$$

That is, we must for any $f\!:\!(A\!\rightarrow\!A)\!\rightarrow\!(A\!\rightarrow\!A)$ and $g\!:\!(A\!\rightarrow\!A)\!\rightarrow\!(A\!\rightarrow\!A)$ satisfying $f\ (\sim\rightarrow\sim)^{\mathsf{abo}}_{\mathsf{C}}\ g \wedge \mathsf{DfnblC}^{A\rightarrow A, A\rightarrow A, \mathfrak{a}', \mathfrak{a}'}(f, g)$, show that $(\mathfrak{a}'.\mathsf{sup}\,f)\ \sim\ (\mathfrak{a}'.\mathsf{sup}\,g)$. But this relieves us from having to consider $funny_{\mathfrak{a}}$ and $0^{\rightarrow}_{\mathfrak{a}}$ in conjunction for the clause $(\mathfrak{a}'.\mathsf{sup}\,funny_{\mathfrak{a}})\ \sim\ (\mathfrak{a}'.\mathsf{sup}\,0^{\rightarrow}_{\mathfrak{a}})$, because $\mathsf{DfnblC}^{A\rightarrow A, A\rightarrow A, \mathfrak{a}', \mathfrak{a}'}(funny_{\mathfrak{a}}, 0^{\rightarrow}_{\mathfrak{a}})$ does not hold. There is no $f : \forall X.(\mathfrak{T}_{\mathsf{REAL}}[X] \rightarrow (X \rightarrow X))$ such that both $(f(A \rightarrow A)\,\mathfrak{a}') = funny_{\mathfrak{a}}$ and $(f(A \rightarrow A)\,\mathfrak{a}') = 0^{\rightarrow}_{\mathfrak{a}}$. In fact, the crux of the matter is that $funny_{\mathfrak{a}}$ cannot be expressed in terms of $\mathfrak{T}_{\mathsf{REAL}}[X]$, *i.e.,* there is not even a term $f\!:\!\forall X.(\mathfrak{T}_{\mathsf{REAL}}[X]\!\rightarrow\!(X\!\rightarrow\!X))$ such that $(f(A\!\rightarrow\!A)\,\mathfrak{a}') = funny_{\mathfrak{a}}$.

Notice again how abstraction barrier-observing simulation relations complies firstly with the uniformity part of *Abs-Bar*, and secondly with the encapsulation part of *Abs-Bar*, *i.e.,* clients of REAL data types cannot access *funny*, so we should not have to consider *funny* when verifying properties of REAL.

This thereby excludes the pathological *funny* from consideration. In other words, *funny* exists as a curio in the implementing abstract data type PCF, but is not accessible in REAL which is implemented in terms of PCF.

With $\mathfrak{a}' \; \mathfrak{T}_{\mathsf{REAL}}[\sim]^{\mathsf{abo}}_{\mathsf{C}} \; \mathfrak{a}'$ established, we can use SubG to get S_A, $\mathfrak{s}_{\mathfrak{a}}\!:\!\mathfrak{T}_{\mathsf{REAL}}[S_A]$ and $\sim' \subset S_A \times S_A$, and a map $\mathsf{mono}\!:\!S_A \to (A \to A)$ such that

$$
\begin{array}{ll}
(s1) & \forall s\!:\!S_A \;.\; s \;\sim'\; s \\
(s2) & \forall s.s'\!:\!S_A \;.\; s \;\sim'\; s' \;\Leftrightarrow\; (\mathsf{mono}\, s) \;\sim\; (\mathsf{mono}\, s') \\
(s3) & \mathfrak{a}' \; (\mathfrak{T}_{\mathsf{REAL}}[(a\!:\!A \to A, s\!:\!S_A).(a =_{A\to A} (\mathsf{mono}\, s))]^{\mathsf{abo}}_{\mathsf{C}}) \; \mathfrak{s}_{\mathfrak{a}}
\end{array}
$$

By $(s2)$ we get $\mathfrak{s}_{\mathfrak{a}} \; \mathfrak{T}_{\mathsf{REAL}}[\sim']^{\mathsf{abo}}_{\mathsf{C}} \; \mathfrak{s}_{\mathfrak{a}}$, and we get $equiv(\sim')$ by $(s1)$. We now use QuotG to get Q and $\mathfrak{q}\!:\!\mathfrak{T}_{\mathsf{REAL}}[Q]$ and map $\mathsf{epi}\!:\!S_A \to Q$ such that

$$
\begin{array}{ll}
(q1) & \forall s, s'\!:\!S_A \;.\; s \;\sim'\; s' \;\Leftrightarrow\; (\mathsf{epi}\, s) =_Q (\mathsf{epi}\, s') \\
(q2) & \forall q\!:\!Q.\exists s\!:\!S_A \;.\; q =_Q (\mathsf{epi}\, s) \\
(q3) & \mathfrak{s}_{\mathfrak{a}} \; (\mathfrak{T}_{\mathsf{REAL}}[(s\!:\!S_A, q\!:\!Q).((\mathsf{epi}\, s) =_Q q)]^{\mathsf{abo}'}_{\mathsf{C}}) \; \mathfrak{q}
\end{array}
$$

Thus we should exhibit Q for B, and $\mathfrak{q}$ for $\mathfrak{b}$; and it then remains to derive

1. $(\mathsf{pack} Q \mathfrak{q}) \; \mathsf{ObsEqC}^{\{\mathsf{Bool}_\perp\}} (\mathsf{pack}\, (A \to A)\, \mathfrak{a}')$, and

2. $Ax_{\mathsf{REAL}}[Q, \mathfrak{q}]$.

To show the derivability of (1), observe that through Theorem 5.33, $(s3)$ and $(q3)$ give $(\mathsf{pack}\, (A \to A)\, \mathfrak{a}') \; \mathsf{ObsEqC}^{\{\mathsf{Bool}_\perp\}} \; (\mathsf{pack} S_A \mathfrak{s}_{\mathfrak{a}}) \; \mathsf{ObsEqC}^{\{\mathsf{Bool}_\perp\}} \; (\mathsf{pack} Q \mathfrak{q})$. The verification of (2) is done using Ax_{PCF} and the definitions of $\sim$ and F. $\circ$

6.7 Final Remarks

In this chapter we described specification refinement up to observational equivalence for specifications involving operations of any order. This was done in System F using both logical and linguistic extensions of Plotkin and Abadi's logic for parametric polymorphism, developed in this chapter and in Ch. 5.

The results of Ch. 5 suggest that abstraction barrier-observing simulation relations, through the correspondence to observational equivalence at any order, constitute exactly the notion we need for data refinement. In this chapter, we therefore presented axiom schemata that would incorporate this alternative notion of simulation relation into the proof strategy for proving observational refinement. We gave a model to show the soundness of the logic augmented by these schemata, namely the PER model with the data type semantics interpretation. The main stratagem in devising the data type semantics is the same as we used in developing the alternative notion of simulation relation, namely to observe the aspect *Abs-Bar* of the abstraction barrier provided by existential types.

On the other hand, the existence of a standard simulation relation implies by parametricity observational equivalence, at any order. Thus, if it is possible to find a standard simulation relation at higher order, then this suffices to prove refinement. It is therefore relevant to establish the axiom schemata that incorporate standard simulation relations, also at higher order. This we did w.r.t. a parametric setoid model, based on work in (Hofmann, 1995a).

An interesting alternative to the type marking associated with the data type semantics would be to impose suitable abstraction barriers in the logical deduction part of the system instead, extending ideas in (Hannay, 1998). The idea is that abstraction barriers in the deduction system allow intensional aspects of abstract data types to co-exists with extensional aspects. This is interesting, because when reasoning about the extensional interface of an abstract data type, one needs in general to consider certain intensional implementation issues. This is particularly relevant for stepwise refinement, since the transition from abstract specification to concrete specifications yields heterogeneous situations where abstract and concrete aspects are mixed.

The co-existence of intensional and extensional aspects is not just a quirk related to specification refinement. It is also relevant for constructive mathematics. At the very foundations of real analysis, the reals may be defined as a quotient of a set of Cauchy-sequences. The n^{th} approximant function is then intensional, and from a constructivist point of view, so is every discontinuous function (Troelstra and van Dalen, 1988). Indeed in a constructive setting it might be prudent to add intensional operators to a data type. See *e.g.,* (Hofmann, 1995b) where a choice operator is provided for quotient types.

Chapter 7

Polymorphic Specification Refinement

In this chapter we address polymorphism *in* data types. This is a different principle from that of the polymorphic data types of Ch. 4; polymorphic data types there really being type-parameterised data types.

We have in principle admitted polymorphism in data types earlier, *i.e.*, according to the discussion around HADT_{Obs} and ADT_{Obs} (*p.* 113), but the development in the preceding chapters was based on HADT_{Obs} and focused on making things work for higher-order operations, rather than for polymorphic operations. Indeed, it seems hard to utilise the amount of polymorphism in data types admitted by ADT_{Obs}. Now we address polymorphism in data types properly. Consider the archetypical example of a polymorphic functional

$$\mathsf{map}\colon\forall Z.\forall Z'.(Z\to Z')\to\mathsf{List}_Z\to\mathsf{List}_{Z'}$$

But how would we define a polymorphic function between two *abstract* types, say a map function between a stack of Z and a stack of Z'? The problem is that the stack types are indeed abstract and not readily available. One could of course steal a march on this problem by defining a double stack data type:

$$\begin{aligned}\forall Z, Z'.\exists X, X' \ . \ (&\mathsf{empty}:X, \mathsf{empty}':X', \mathsf{push}:Z\to X\to X, \mathsf{push}':Z'\to X'\to X',\\ &\ldots, \mathsf{map}:\forall Z, Z'.(Z\to Z')\to X\to X' \ )\end{aligned}$$

This is clearly not a nice way of defining an abstract type. Aside from the duplication of all stack operations for two separate data representations, the universal quantification of the element types has to be duplicated as well.

7.1 Specification in F_3

It seems evident that truly polymorphic data types and specifications have a different form from what we have seen so far. In particular, we would like a way of expressing that the data representation might depend on (element) types. One way of resolving this is to use type constructors. This we do not have in the calculus F_2 (System F), but we do have type constructors in calculus F_3. See (Pierce et al., 1989) for a related account of *polymorphic inductive types*.

Example 7.1 (Specification of Polymorphic Stack ADT) The following specification $\mathsf{polySTACK} \stackrel{def}{=} \langle\langle Sig_{\mathsf{polySTACK}}, \Theta_{\mathsf{polySTACK}}\rangle, \mathfrak{T}^e_{\mathsf{polySTACK}}, \{\}\rangle$ specifies a polymorphic stack ADT, where

$$\begin{aligned}
&Sig_{\mathsf{polySTACK}} = \exists\mathfrak{X}:*\to *.\mathfrak{T}_{\mathsf{polySTACK}}[\mathfrak{X}],\\
&\mathfrak{T}_{\mathsf{polySTACK}}[\mathfrak{X}] = (\mathsf{empty}:\forall Z\,.\,\mathfrak{X}Z,\ \mathsf{push}:\forall Z\,.\,Z\to\mathfrak{X}Z\to\mathfrak{X}Z,\\
&\qquad\qquad \mathsf{pop}:\forall Z\,.\,\mathfrak{X}Z\to\mathfrak{X}Z,\ \mathsf{top}:\forall Z\,.\,\mathfrak{X}Z\to Z\to Z,\\
&\qquad\qquad \mathsf{map}:\forall Z, Z'\,.\,(Z\to Z')\to\mathfrak{X}Z\to\mathfrak{X}Z' \ ),\\
&\mathfrak{T}_{\mathsf{polySTACK}}[\mathfrak{X}]^e = \mathfrak{T}_{\mathsf{polySTACK}}[\mathfrak{X}] \times \mathsf{multipop}:\forall Z.\mathsf{Nat}\to\mathfrak{X}Z\to\mathfrak{X}Z\\
&\Theta_{\mathsf{polySTACK}}(u) = \exists\mathfrak{X}:*\to *.\exists\mathfrak{x}:\mathfrak{T}^e_{\mathsf{polySTACK}}[\mathfrak{X}]\ .\ u\ \mathsf{PolyObsEq}^{\{\}}\ (\mathsf{pack}\mathfrak{X}\,\mathfrak{x}|_{\mathfrak{T}_{\mathsf{polySTACK}}})\ \wedge\\
&\quad \forall Z.\forall x:Z, s:\mathfrak{X}Z\ .\ (\mathfrak{x}.\mathsf{pop}Z(\mathfrak{x}.\mathsf{push}Z\ x\ s)) = s\ \wedge\\
&\quad \forall Z.\forall x, z:Z, s:\mathfrak{X}Z\ .\ (\mathfrak{x}.\mathsf{top}Z(\mathfrak{x}.\mathsf{push}Z\ x\ s)\ z) = x\ \wedge\\
&\quad \forall Z.\forall s:\mathfrak{X}Z\ .\ (\mathfrak{x}.\mathsf{multipop}Z\,0\ s) = s\ \wedge\\
&\quad \forall Z.\forall n:\mathsf{Nat}, s:\mathfrak{X}Z\ .\ (\mathfrak{x}.\mathsf{multipop}Z\,(\mathsf{succ}\,n\ s)) = (\mathfrak{x}.\mathsf{multipop}Z\,n\ (\mathfrak{x}.\mathsf{pop}Z\,s))\ \wedge\\
&\quad \forall Z, Z'.\forall n:\mathsf{Nat}.\forall s:\mathfrak{X}Z.\forall g:Z\to Z'\ .\\
&\qquad (\mathfrak{x}.\mathsf{top}Z'\,(\mathfrak{x}.\mathsf{multipop}Z'\,n\ (\mathfrak{x}.\mathsf{map}ZZ'gs))) = g(\mathfrak{x}.\mathsf{top}Z(\mathfrak{x}.\mathsf{multipop}Z\,ns))
\end{aligned}$$

○

In this example, the existentially bound variable $\mathfrak{X}$ is a type constructor. Thus the data representation generated by the stack operations empty and push (and pop) depends on the element type Z. We say what PolyObsEq is in a moment.

Example 7.1 (continued) A data type realising polySTACK is for example

$$(\mathsf{pack}\,(\lambda Z\!:\!*.\mathsf{List}_Z)\ \mathfrak{l})$$

where

$$\begin{aligned}
\mathfrak{l} \stackrel{def}{=} (&\mathsf{empty} = \Lambda Z.\mathsf{nil}_Z\\
&\mathsf{push} = \Lambda Z.\lambda z\!:\!Z.\lambda l\!:\!\mathsf{List}_Z.(\mathsf{cons}_Z zl)\\
&\mathsf{pop} = \Lambda Z.\lambda l\!:\!\mathsf{List}_Z.(\mathsf{cond}\,\mathsf{List}_Z\,(\mathsf{isnil}\,l)\,\mathsf{nil}_Z\,(\mathsf{cdr}_Z\,l))\\
&\mathsf{top} = \Lambda Z.\lambda l\!:\!\mathsf{List}_Z.\lambda z\!:\!Z.(\mathsf{cond}\,\mathsf{List}_Z\,(\mathsf{isnil}\,l)\,z\,(\mathsf{car}_Z\,l))\\
&\mathsf{map} = \Lambda Z.\Lambda Z'.\lambda g\!:\!Z\to Z'.\lambda l\!:\!\mathsf{List}_Z\,.\\
&\qquad(\mathsf{cond}\,\mathsf{List}_Z\,(\mathsf{isnil}\,l)\,\mathsf{nil}_{Z'}\,(\mathsf{cons}_{Z'}(g(\mathsf{car}_Z l))(\mathsf{map}ZZ'g(\mathsf{cdr}_Z l)))))
\end{aligned}$$

where the parameter z for *top* is a default value in case of an empty stack, and

$$\mathsf{isnil} \stackrel{def}{=} \lambda l\!:\!\mathsf{List}_Z\,.\,l\,\mathsf{Bool}\,\mathsf{true}\,(\lambda b\!:\!\mathsf{Bool}.\mathsf{false})$$

returns true when l is nil_Z, and false otherwise. ○

Here is the definition of specification of polymorphic abstract data types. We will have various notions of observational equivalence, so the definition below is in this sense generic.

Definition 7.2 (ADT Specification in F_3) *An* abstract data type specification *is a tuple* $\langle\langle Sig_{SP}, \Theta_{SP}\rangle, \mathfrak{T}^e_{SP}, Obs_{SP}\rangle$*, where*

$Sig_{SP} \stackrel{def}{=} \exists\mathfrak{X}\!:\!*\to*.\mathfrak{T}_{SP}[\mathfrak{X}]$,

$\Theta_{SP}(u) \stackrel{def}{=} \exists\mathfrak{X}\!:\!*\to*.\exists\mathfrak{x}\!:\!\mathfrak{T}^e_{SP}[\mathfrak{X}]\,.\,u\ \mathsf{PolyObsEq}X^{Obs_{SP}}_{\mathfrak{T}_{SP}[\mathfrak{X}]}\ (\mathsf{pack}\mathfrak{X}\,\mathfrak{x}|_{\mathfrak{T}_{SP}}) \wedge Ax_{SP}[\mathfrak{X},\mathfrak{x}]$,

for $\mathsf{PolyObsEq}X$ *any of the polymorphic notions of observational equivalence below. Here, all profiles of* $\mathfrak{T}_{SP}[\mathfrak{X}]$ *occur in* $\mathfrak{T}^e_{SP}[\mathfrak{X}]$*, and* $\mathfrak{x}|_{\mathfrak{T}_{SP}}$ *denotes* $(\mathfrak{x}.g_1,\ldots,\mathfrak{x}.g_n)$ *for all* $g_i\!:\!T_i[\mathfrak{X}]$ *such that* $T_i[\mathfrak{X}]$ *is a profile in* $\mathfrak{T}_{SP}[\mathfrak{X}]$*.* $Ax_{SP}[\mathfrak{X},\mathfrak{x}]$ *is a finite conjunction of formulae in the logic. If* $\Theta_{SP}(u)$ *is derivable, then* u *is said to be a* realisation of SP*. We assume that* Obs_{SP} *contains*

- *possibly closed inductive types, such as* Bool *or* Nat*,*
- *parameters* $\boldsymbol{Z'}$*, in case* $\mathfrak{T}[\mathfrak{X}]$ *has free* $\boldsymbol{Z'}$ *other than* $\mathfrak{X}$*,*

Specification refinement goes as always:

Definition 7.3 (Specification Refinement in F_3) *Let SP and SP′ be abstract data type specifications in* F_3*. Then SP′ is a* refinement *of SP,* via constructor $F\!:\!Sig_{SP'}\to Sig_{SP}$ *if*

$$\forall u\!:\!Sig_{SP'}\,.\,\Theta_{SP'}(u)\ \Rightarrow\ \Theta_{SP}(Fu)$$

is derivable. We write $SP \underset{F}{\leadsto} SP'$ *for this fact.*

To avoid clutter, we now assume exactly one existentially bound type constructor of kind $* \to *$ in our abstract types. It should be plain sailing to generalise to type constructors of higher arity and to multiple type constructors.

Again, $\mathfrak{T}[\mathfrak{X}]$ is reserved for the *body* of a given abstract type $\exists \mathfrak{X}.\mathfrak{T}[\mathfrak{X}]$. In the ensuing discussion, we use $\boldsymbol{Z'}$ for type parameters, *i.e.,* free type variables in $\exists \mathfrak{X}.\mathfrak{T}[\mathfrak{X}, \boldsymbol{Z'}]$, as usual only writing these when necessary. On the other hand, we use $\boldsymbol{Z}$ for polymorphic variables, *i.e.,* for the universally bound type variables for polymorphic operations. We assume $\boldsymbol{Z}$ and $\boldsymbol{Z'}$ to be chosen distinct. We admit no free type constructors in abstract types, although these could probably be added without difficulty, albeit adding clutter.

We continue to use a labelled product notation as a notational convenience when discussing data types, *i.e.,* we will write $\mathfrak{T}[\mathfrak{X}]$ in the form

$$(f_1 : T_1[\mathfrak{X}], \dots, f_k : T_k[\mathfrak{X}])$$

where each $T_i[\mathfrak{X}]$ is a *profile* of the abstract type. Again we presume a specification scenario, and hence a current set of observable types *Obs* according to Def. 7.2. We assume the following polymorphic generalisation of $\mathtt{HADT}_{Obs}$.

$\mathtt{PADT}_{Obs}$**:** Every profile $T_i[\mathfrak{X}]$ of an abstract type $\exists \mathfrak{X}.\mathfrak{T}[\mathfrak{X}]$ is of the form $\forall \boldsymbol{Z}.T_i{}'[\mathfrak{X}, \boldsymbol{Z}]$ where the *subprofile* $T_i{}'[\mathfrak{X}, \boldsymbol{Z}]$ satisfies the clause $\mathtt{HADTF3}_{Obs \cup \boldsymbol{Z}}$**:** The subprofile $T_i{}'[\mathfrak{X}, \boldsymbol{Z}] = T'_{i1} \to \cdots \to T'_{n_i} \to T'_{c_i}$ is such that T_{ij} has no occurrences of universal types other than those in *Obs*, and T'_{c_i} is either $\mathfrak{X} Z_j$ for some Z_j of $\boldsymbol{Z}$, or some $D \in Obs$.

7.2 Observational Equivalence in F_3

We now have to decide what observational equivalence means in this context, and for this we have to decide what observable computations are. An observational computation could intuitively be a functional

$$f : \forall \mathfrak{X} : * \to *. \forall \boldsymbol{Z}.(\mathfrak{T}[\mathfrak{X}] \to D)$$

where D is in *Obs* including any Z'_i from any parameters $\boldsymbol{Z'}$ (recall our assumption on *Obs* (*p.* 79)), or some Z_j from $\boldsymbol{Z}$. Alas, this does not really work. For example, we would perhaps want to form the observable computation

$$\Lambda \mathfrak{X} : * \to *. \Lambda Z : *. \lambda \mathfrak{x} : \mathfrak{T}[\mathfrak{X}] \,.\, \mathfrak{x}.\mathsf{top} Z(\mathfrak{x}.\mathsf{push} Z(z, \mathfrak{x}.\mathsf{empty} Z))$$

But this is ill-typed. The free variable z cannot be of type Z, and there is nothing else we can put in place for z to form the kind of computation we have in mind.

A solution would be to have $\boldsymbol{Z}$ free, and have observational computations of the form

$$f : \forall \mathfrak{X} : * \to *. (\mathfrak{T}[\mathfrak{X}] \to D)$$

Then we could form the observable computation

$$\Lambda \mathfrak{X} : * \to *. \lambda \mathfrak{x} : \mathfrak{T}[\mathfrak{X}] \;.\; \mathfrak{x}.\mathsf{top}Z(\mathfrak{x}.\mathsf{push}Z(z, \mathfrak{x}.\mathsf{empty}Z))$$

for $z : Z$. We can then define observational equivalence as follows.

Definition 7.4 (Observational Equivalence in F_3) *Define* observational equivalence $\mathsf{PolyObsEq}$ w.r.t. $\mathfrak{T}[\mathfrak{X}]$ and observable types *Obs, in the logic by*

$$\begin{aligned}
&\mathsf{PolyObsEq}^{Obs}_{\mathfrak{T}[\mathfrak{X}]} \stackrel{def}{=} (u : \exists \mathfrak{X} : * \to *. \mathfrak{T}[\mathfrak{X}], v : \exists \mathfrak{X} : * \to *. \mathfrak{T}[\mathfrak{X}]). \\
&\quad (\exists \mathfrak{A} : * \to *, \mathfrak{B} : * \to *. \exists \mathfrak{a} : \mathfrak{T}[\mathfrak{A}], \mathfrak{b} : \mathfrak{T}[\mathfrak{B}] \;.\; u = (\mathsf{pack}\mathfrak{A}\mathfrak{a}) \;\wedge\; v = (\mathsf{pack}\mathfrak{B}\mathfrak{b}) \;\wedge \\
&\qquad\qquad \forall \boldsymbol{Z} : *. \textstyle\bigwedge_{D \in Obs \cup \{\boldsymbol{Z}\}} \forall f : \forall \mathfrak{X} : * \to *. (\mathfrak{T}[\mathfrak{X}] \to D) \;.\; (f \mathfrak{A}\, \mathfrak{a}) =_D (f \mathfrak{B}\, \mathfrak{b}))
\end{aligned}$$

On the other hand, we would like to lift our results from previous chapters to this new scenario, and if we choose the above form for observable computations and observational equivalence, we will have to extend the parametricity axiom to F_3, and much more work will be involved. Therefore, it would in fact be desirable to somehow keep the notions of observational equivalence and simulation relation within F_2, in the hope that we might get the desired results for free.

—

In Sect. 7.3 we derive the desired results for observational equivalence and simulation relations for F_3 formalisms represented in F_2. However, when we look at specification refinement in Sect. 7.4, we will take a more direct approach using System F notions of observational equivalence and simulation relations directly, and not using the results of Sect. 7.3 as such.

In the following, we shall use existential types with more than one existentially bound type variable. Our treatment of existential types up to now has involved one single existentially bound type variable, but it is completely straight-forward to generalise everything so far to existential types with any number of existentially bound variables; one simply performs component-wise treatments for each variable. See also Sect. 3.2.7.

7.3 Representations in F_2

We now attempt to retain for F_3 the notion of observable computation that we had for F_2. How can we make this work for inhabitants of existential types that are now F_3 terms and involve type constructors? Our solution here is to represent F_3-inhabitants of F_3-existential types using families of F_2-inhabitants of F_2-existential types. For example,

$$(\mathsf{pack}\mathfrak{A}\mathfrak{a}) : \exists\mathfrak{X}{*}\to{*}.\mathfrak{T}_{\mathsf{polySTACK}}[\mathfrak{X}]$$

is represented by the family

$$\begin{aligned}\{(\mathsf{pack}\,(\mathfrak{A}Z)\,(\mathfrak{A}Z')\ (&\mathfrak{a}.\mathsf{empty}Z, \mathfrak{a}.\mathsf{push}Z, \mathfrak{a}.\mathsf{pop}Z, \mathfrak{a}.\mathsf{top}Z,\\ &\mathfrak{a}.\mathsf{empty}Z', \mathfrak{a}.\mathsf{push}Z', \mathfrak{a}.\mathsf{pop}Z', \mathfrak{a}.\mathsf{top}Z',\\ &\mathfrak{a}.\mathsf{map}ZZ, \mathfrak{a}.\mathsf{map}ZZ', \mathfrak{a}.\mathsf{map}Z'Z, \mathfrak{a}.\mathsf{map}Z'Z'))\end{aligned}$$

$$\begin{gathered}: \exists X, X'.(\mathsf{empty}_1 : X,\ \mathsf{push}_1 : Z\to X\to X,\ \mathsf{pop}_1 : X\to X,\ \mathsf{top}_1 : X\to Z\to Z,\\ \mathsf{empty}_2 : X',\ \mathsf{push}_2 : Z'\to X'\to X',\ \mathsf{pop}_2 : X'\to X',\ \mathsf{top}_2 : X'\to Z'\to Z',\\ \mathsf{map}_1 : (Z\to Z)\to X\to X,\\ \mathsf{map}_2 : (Z\to Z')\to X\to X',\\ \mathsf{map}_3 : (Z'\to Z)\to X'\to X,\\ \mathsf{map}_4 : (Z'\to Z')\to X'\to X')\}_{ZZ'}\end{gathered}$$

This representation resembles the style derisively mentioned at the beginning of this chapter. However, this representation is for the purpose of reasoning only, and not for programming. The justification lies in what we think observable computations are, namely that they are uniform in any given type instance $\boldsymbol{Z}$, as for example in $\Lambda\mathfrak{X} : *\to *.\lambda\mathfrak{x} : \mathfrak{T}[\mathfrak{X}]\ .\ \mathfrak{x}.\mathsf{top}Z(\mathfrak{x}.\mathsf{push}Z(z, \mathfrak{x}.\mathsf{empty}Z))$. If we show observational equivalence between corresponding members in two families, then this should constitute observational equivalence for the two represented packages.

Definition 7.5 below gives a generic representative for a representation family, rather than the family itself. Thus, the formal F_2 representation for any

$$(\mathsf{pack}\mathfrak{A}\mathfrak{a}) : \exists\mathfrak{X}{*}\to{*}.\mathfrak{T}_{\mathsf{polySTACK}}[\mathfrak{X}]$$

is obtained by $(\mathsf{pack}\,(\mathfrak{A}Z)\,(\mathfrak{A}Z')\,\mathfrak{a}_{F_2}) : \exists X, X'.\mathfrak{T}_{\mathsf{polySTACK}_{F_2}}[X, X']$, where

$$\mathfrak{a}_{F_2} \overset{def}{=} (\mathfrak{a}.\mathsf{empty}Z, \ldots, \mathfrak{a}.\mathsf{map}Z'Z')$$

$$\mathfrak{T}_{\mathsf{polySTACK}_{F_2}} \overset{def}{=} (\mathsf{empty}_1 : X, \ldots, \mathsf{map}_4 : (Z'\to Z')\to X'\to X')$$

as above. We now give the rather complicated formal definition. However, the intuition given so far will suffice to follow the discussion.

For a type constructor $\mathfrak{X}$, we write $\boldsymbol{\mathfrak{X}Z}$ for $\mathfrak{X}Z_1, \ldots, \mathfrak{X}Z_n$ for $\boldsymbol{Z} \overset{def}{=} Z_1, \ldots, Z_n$. We also write $T[\boldsymbol{X}/\boldsymbol{\mathfrak{X}Z}]$ to indicate replacing every occurrence of $\mathfrak{X}Z_i$ by X_i. We write $\boldsymbol{Z}_n$ to indicate the length of $\boldsymbol{Z}$.

Definition 7.5 (Representation in F_2) *For $\mathfrak{T}[\mathfrak{X}]$ and $\mathfrak{x} : \mathfrak{T}[\mathfrak{X}]$ we construct $\mathfrak{T}_{F_2}[\boldsymbol{X}, \boldsymbol{Z}]$ and $\mathfrak{x}_{F_2}$ as follows. Let n be the maximum number of outermost universally quantified variables for any operation profile in $\mathfrak{T}[\mathfrak{X}]$. Then for any operation profile $\forall \boldsymbol{Z}_{n_i}.T_i[\mathfrak{X}, \boldsymbol{Z}_{n_i}]$ in $\mathfrak{T}[\mathfrak{X}]$, we put n^{n_i} versions of $T_i[\boldsymbol{X}_{n_i}/\boldsymbol{\mathfrak{X}}\boldsymbol{Z}_{n_i}, \boldsymbol{Z}_{n_i}]$ in $\mathfrak{T}_{F_2}[\boldsymbol{X}, \boldsymbol{Z}]$ for every combination of types of $\boldsymbol{Z}_n$ for $\boldsymbol{Z}_{n_i}$. Now, $\mathfrak{x}_{F_2}$ contains n^{n_i} versions of $\mathfrak{x}.g_i\boldsymbol{Z}_{n_i}$ for every operation $\mathfrak{x}.g_i$ in $\mathfrak{x}$, also for every combination of types of $\boldsymbol{Z}_n$ for $\boldsymbol{Z}_{n_i}$.*

We now define observational equivalence by F_2 representation.

Definition 7.6 (Observational Equivalence ($\mathsf{PolyObsEq}F_2$)) *Define* observational equivalence $\mathsf{PolyObsEq}F_2$ w.r.t. $\mathfrak{T}[\mathfrak{X}]$ and observable types *Obs by*

$$\begin{aligned}&\mathsf{PolyObsEq}F_2{}^{Obs}_{\mathfrak{T}[\mathfrak{X}]} \stackrel{def}{=} (u{:}\exists\mathfrak{X}{:}*\to*.\mathfrak{T}[\mathfrak{X}], v{:}\exists\mathfrak{X}{:}*\to*.\mathfrak{T}[\mathfrak{X}]).\\ &\quad(\exists\mathfrak{A}{:}*\to*, \mathfrak{B}{:}*\to*.\exists\mathfrak{a}{:}\mathfrak{T}[\mathfrak{A}], \mathfrak{b}{:}\mathfrak{T}[\mathfrak{B}] \;.\; u = (\mathsf{pack}\mathfrak{A}\mathfrak{a}) \;\wedge\; v = (\mathsf{pack}\mathfrak{B}\mathfrak{b}) \;\wedge\\ &\quad\quad\forall\boldsymbol{Z}.\textstyle\bigwedge_{D\in Obs\cup\{\boldsymbol{Z}\}}\forall f{:}\forall\boldsymbol{X}{:}*.(\mathfrak{T}_{F_2}[\boldsymbol{X},\boldsymbol{Z}]\to D) \;.\; (f(\boldsymbol{\mathfrak{A}Z})\mathfrak{a}_{F_2}) =_D (f(\boldsymbol{\mathfrak{B}Z})\mathfrak{b}_{F_2}))\end{aligned}$$

Similarly, we define relatedness by simulation relation by F_2 representation.

Definition 7.7 (Simulation Relation ($\mathsf{PolySimRel}F_2$)) *Relatedness by* simulation relation *w.r.t. $\mathfrak{T}[\mathfrak{X}, \boldsymbol{Z'}]$ is expressed in the logic by*

$$\begin{aligned}&\mathsf{PolySimRel}F_{2\,\mathfrak{T}[\mathfrak{X},\boldsymbol{eq}_{\boldsymbol{Z'}}]} \stackrel{def}{=} (u{:}\exists\mathfrak{X}{:}*\to*.\mathfrak{T}[\mathfrak{X},\boldsymbol{Z'}], v{:}\exists\mathfrak{X}{:}*\to*.\mathfrak{T}[\mathfrak{X},\boldsymbol{Z'}]) \;.\\ &(\exists\mathfrak{A}{:}*\to*, \mathfrak{B}{:}*\to*.\exists\mathfrak{a}{:}\mathfrak{T}[\mathfrak{A},\boldsymbol{Z'}], \mathfrak{b}{:}\mathfrak{T}[\mathfrak{B},\boldsymbol{Z'}] \;.\; u = (\mathsf{pack}\mathfrak{A}\mathfrak{a}) \;\wedge\; v = (\mathsf{pack}\mathfrak{B}\mathfrak{b})\\ &\qquad\qquad\wedge\;\forall\boldsymbol{Z}.\exists\boldsymbol{R}\subset(\boldsymbol{\mathfrak{A}Z})\times(\boldsymbol{\mathfrak{B}Z}) \;.\; \mathfrak{a}_{F_2}(\mathfrak{T}_{F_2}[\boldsymbol{R}, \mathbf{eq}_{\boldsymbol{ZZ'}}])\mathfrak{b}_{F_2})\end{aligned}$$

With these notions of observational equivalence and simulation relation, we can lift the results of the previous chapter to polymorphism in F_3. First we lift the results for first-order profiles, *i.e.,* Theorems 4.17 (*p.* 87) and 4.22 (*p.* 90).

Theorem 7.8 *Suppose $\mathfrak{T}[\mathfrak{X}]$ adheres to $\mathtt{PADT}_{Obs}$ and only contains first-order subprofiles. With* PARAM *we derive that the existence of a simulation relation is equivalent to observational equivalence, i.e.,*

$$\forall\boldsymbol{Z'}.\forall u, v{:}\exists\mathfrak{X}.\mathfrak{T}[\mathfrak{X},\boldsymbol{Z'}] \;.\; u \;\mathsf{PolySimRel}F_{2\,\mathfrak{T}[\mathfrak{X},\mathbf{eq}_{\boldsymbol{Z'}}]}\; v \quad\Leftrightarrow\quad u \;\mathsf{PolyObsEq}F_2{}^{Obs}_{\mathfrak{T}[\mathfrak{X},\boldsymbol{Z'}]}\; v$$

Proof: It suffices to derive for arbitrary $\boldsymbol{Z}$,

$$\begin{aligned}&\forall A, B.\forall\mathfrak{a}{:}\mathfrak{T}[\mathfrak{A},\boldsymbol{Z'}], \mathfrak{b}{:}\mathfrak{T}[\mathfrak{B},\boldsymbol{Z'}] \;.\\ &\qquad\exists\boldsymbol{R}\subset(\boldsymbol{\mathfrak{A}Z})\times(\boldsymbol{\mathfrak{B}Z}) \;.\; \mathfrak{a}_{F_2}(\mathfrak{T}_{F_2}[\boldsymbol{R}, \mathbf{eq}_{\boldsymbol{ZZ'}}])\mathfrak{b}_{F_2}\\ &\Leftrightarrow\quad\textstyle\bigwedge_{D\in Obs\cup\{\boldsymbol{Z}\}}\forall f{:}\forall\boldsymbol{X}.(\mathfrak{T}_{F_2}[\boldsymbol{X},\boldsymbol{Z},\boldsymbol{Z'}]\to D) \;.\; (f(\boldsymbol{\mathfrak{A}Z})\,\mathfrak{a}_{F_2}) =_D (f(\boldsymbol{\mathfrak{B}Z})\,\mathfrak{b}_{F_2})\end{aligned}$$

But this follows from Theorem 4.17 (*p.* 87). □

Theorem 7.9 (Composability of Simulation Relations) *Suppose $\mathfrak{T}[\mathfrak{X}]$ adheres to PADT_{Obs} and only contains first-order subprofiles. Then we can derive*

$$\begin{aligned}
&\forall \boldsymbol{Z}.\forall \mathfrak{A}, \mathfrak{B}, \mathfrak{C}, \boldsymbol{R} \subset (\mathfrak{A}\boldsymbol{Z}) \times (\mathfrak{B}\boldsymbol{Z}), \boldsymbol{S} \subset (\mathfrak{B}\boldsymbol{Z}) \times (\mathfrak{C}\boldsymbol{Z}).\\
&\qquad \forall \mathfrak{a} : \mathfrak{T}[\mathfrak{A}, \boldsymbol{Z'}], \mathfrak{b} : \mathfrak{T}[\mathfrak{B}, \boldsymbol{Z'}], \mathfrak{c} : \mathfrak{T}[\mathfrak{C}, \boldsymbol{Z'}] \ .\\
&\qquad\qquad \mathfrak{a}_{F_2}(\mathfrak{T}_{F_2}[\boldsymbol{R}, \mathbf{eq}_{\boldsymbol{ZZ'}}])\mathfrak{b}_{F_2} \ \wedge \ \mathfrak{b}_{F_2}(\mathfrak{T}_{F_2}[\boldsymbol{S}, \mathbf{eq}_{\boldsymbol{ZZ'}}])\mathfrak{c}_{F_2}\\
&\qquad\qquad\qquad\qquad \Rightarrow \ \mathfrak{a}_{F_2}(\mathfrak{T}_{F_2}[\boldsymbol{S} \circ \boldsymbol{R}, \mathbf{eq}_{\boldsymbol{ZZ'}}])\mathfrak{c}_{F_2}
\end{aligned}$$

Proof: This follows from Theorem 4.22 (*p.* 90). □

We now similarly define the F_3 analogues to SimRelA (Def. 5.11, *p.* 123), ObsEqC (Def. 5.25, *p.* 134), and SimRelC (Def. 5.30, *p.* 137), in the same way by F_2 representation as we did for PolyObsEqF_2 and PolySimRelF_2 above. In all cases, one gets the coincidences of observational equivalence with relatedness by simulation relation, as well as composability of simulation relations. Recall that w.r.t. syntactic models it suffices to use SimRelA together with ObsEq, but if one wants to relate to models other than those in which one relies on term denotability, then one can use SimRelC together with ObsEqC. In the following we give the results, but the discourse is rather similar to the case above.

7.3.1 Abstraction Barrier-Observing Simulation Relation *I*

Definition 7.10 (*abo*-Simulation Relation (PolySimRelAF_2)) *Relatedness by* abstraction barrier observing (*abo*) simulation relation *w.r.t.* $\mathfrak{T}[\mathfrak{X}, \boldsymbol{Z'}]$ *is defined*

$$\begin{aligned}
&\mathsf{PolySimRelA}F_{2\,\mathfrak{T}[\mathfrak{X},\mathbf{eq}_{\boldsymbol{Z'}}]} \stackrel{def}{=} (u : \exists \mathfrak{X} : * \to *.\mathfrak{T}[\mathfrak{X}, \boldsymbol{Z'}], v : \exists \mathfrak{X} : * \to *.\mathfrak{T}[\mathfrak{X}, \boldsymbol{Z'}]) \ .\\
&\quad (\exists \mathfrak{A} : * \to *, \mathfrak{B} : * \to *.\exists \mathfrak{a} : \mathfrak{T}[\mathfrak{A}, \boldsymbol{Z'}], \mathfrak{b} : \mathfrak{T}[\mathfrak{B}, \boldsymbol{Z'}] \ . \ u = (\mathsf{pack}\mathfrak{A}\mathfrak{a}) \ \wedge \ v = (\mathsf{pack}\mathfrak{B}\mathfrak{b})\\
&\qquad\qquad \wedge \ \forall \boldsymbol{Z}.\exists \boldsymbol{R} \subset (\mathfrak{A}\boldsymbol{Z}) \times (\mathfrak{B}\boldsymbol{Z}) \ . \ \mathfrak{a}_{F_2}(\mathfrak{T}_{F_2}[\boldsymbol{R}, \mathbf{eq}_{\boldsymbol{ZZ'}}]^{\mathsf{abo}})\mathfrak{b}_{F_2})
\end{aligned}$$

where $\mathsf{abo} = \mathfrak{A}\boldsymbol{Z}, \mathfrak{B}\boldsymbol{Z}, \mathfrak{a}_{F_2}, \mathfrak{b}_{F_2}$.

We now lift Theorem 5.14 (*p.* 125) to polymorphism in F_3.

Theorem 7.11 *Let* $\mathfrak{T}[\mathfrak{X}, \boldsymbol{Z'}]$ *adhere to* PADT_{Obs}. *With* spPARAM *we derive that the existence of an abo-simulation relation is equivalent to observational equivalence,*

$$\forall \boldsymbol{Z'}.\forall u, v : \exists \mathfrak{X}.\mathfrak{T}[\mathfrak{X}, \boldsymbol{Z'}] \ . \ u \ \mathsf{PolySimRelA}F_{2\,\mathfrak{T}[\mathfrak{X},\mathbf{eq}_{\boldsymbol{Z'}}]} \ v \quad \Leftrightarrow \quad u \ \mathsf{PolyObsEq}F_{2\,\mathfrak{T}[\mathfrak{X},\boldsymbol{Z'}]}^{\,Obs} \ v$$

Proof: This follows from Theorem 5.14 (*p.* 125). □

We then lift Theorem 5.17 (*p.* 127) to polymorphism in F_3.

Theorem 7.12 (Composability of Simulation Relations) *Given* spParam, *for* $\mathfrak{T}[\mathfrak{X}, \boldsymbol{Z'}]$ *adhering to* $\mathtt{PADT}_{Obs}$, *we can derive*

$$\begin{aligned}
&\forall \boldsymbol{Z}.\mathfrak{A}, \mathfrak{B}, \mathfrak{C}, \boldsymbol{R} \subset (\mathfrak{A}\boldsymbol{Z}) \times (\mathfrak{B}\boldsymbol{Z}), \boldsymbol{S} \subset (\mathfrak{B}\boldsymbol{Z}) \times (\mathfrak{C}\boldsymbol{Z}). \\
&\qquad \forall \mathfrak{a}:\mathfrak{T}[\mathfrak{A}, \boldsymbol{Z'}], \mathfrak{b}:\mathfrak{T}[\mathfrak{B}, \boldsymbol{Z'}], \mathfrak{c}:\mathfrak{T}[\mathfrak{C}, \boldsymbol{Z'}] \ . \\
&\qquad\qquad \mathfrak{a}_{F_2}(\mathfrak{T}_{F_2}[\boldsymbol{R}, \mathbf{eq}_{\boldsymbol{ZZ'}}]^{\mathsf{abo}})\mathfrak{b}_{F_2} \ \wedge \ \mathfrak{b}_{F_2}(\mathfrak{T}_{F_2}[\boldsymbol{S}, \mathbf{eq}_{\boldsymbol{ZZ'}}]^{\mathsf{abo}})\mathfrak{c}_{F_2} \\
&\qquad\qquad\qquad \Rightarrow \ \mathfrak{a}_{F_2}(\mathfrak{T}_{F_2}[\boldsymbol{S} \circ \boldsymbol{R}, \mathbf{eq}_{\boldsymbol{ZZ'}}]^{\mathsf{abo}})\mathfrak{c}_{F_2}
\end{aligned}$$

Proof: This follows from Theorem 5.17 (*p.* 127). □

We can now repeat all this for the notion of abstraction barrier-observing simulation relation using closed computations.

7.3.2 Abstraction Barrier-Observing Simulation Relation *II*

Definition 7.13 (Closed Observational Equivalence (PolyObsEqCF_2)**)** *Define* closed observational equivalence PolyObsEqCF_2 w.r.t. *Obs by*

$$\begin{aligned}
&\mathsf{PolyObsEqC}F_2{}^{Obs}_{\mathfrak{T}[\mathfrak{X}]} \stackrel{def}{=} (u:\exists\mathfrak{X}:*\to*.\mathfrak{T}[\mathfrak{X}], v:\exists\mathfrak{X}:*\to*.\mathfrak{T}[\mathfrak{X}]). \\
&\quad (\exists\mathfrak{A}:*\to*, \mathfrak{B}:*\to*.\exists\mathfrak{a}:\mathfrak{T}[\mathfrak{A}], \mathfrak{b}:\mathfrak{T}[\mathfrak{B}] \ . \ u = (\mathsf{pack}\mathfrak{A}\mathfrak{a}) \ \wedge \ v = (\mathsf{pack}\mathfrak{B}\mathfrak{b}) \ \wedge \\
&\qquad\qquad \forall\boldsymbol{Z}. \textstyle\bigwedge_{D\in Obs\cup\{\boldsymbol{Z}\}} \forall f:\forall X:*.(\mathfrak{T}_{F_2}[X, \boldsymbol{Z}]\to D) \ . \\
&\qquad\qquad\qquad \mathsf{Closed}_{\Gamma^{In\cup\{\boldsymbol{Z}\}}}(f) \ \Rightarrow \ (f(\mathfrak{A}\boldsymbol{Z})\mathfrak{a}_{F_2}) =_D (f(\mathfrak{B}\boldsymbol{Z})\mathfrak{b}_{F_2}))
\end{aligned}$$

Definition 7.14 (*abo*-Simulation Relation PolySimRelCF_2**)** *Relatedness by* abstraction barrier observing closed computation (*abo*) simulation relation *w.r.t.* $\mathfrak{T}[\mathfrak{X}, \boldsymbol{Z'}]$ *is expressed in the logic by*

$$\begin{aligned}
&\mathsf{PolySimRelC}F_2{}_{\mathfrak{T}[\mathfrak{X},\mathbf{eq}_{\boldsymbol{Z'}}]} \stackrel{def}{=} (u:\exists\mathfrak{X}:*\to*.\mathfrak{T}[\mathfrak{X}, \boldsymbol{Z'}], v:\exists\mathfrak{X}:*\to*.\mathfrak{T}[\mathfrak{X}, \boldsymbol{Z'}]) \ . \\
&\quad (\exists\mathfrak{A}:*\to*, \mathfrak{B}:*\to*.\exists\mathfrak{a}:\mathfrak{T}[\mathfrak{A}, \boldsymbol{Z'}], \mathfrak{b}:\mathfrak{T}[\mathfrak{B}, \boldsymbol{Z'}] \ . \ u = (\mathsf{pack}\mathfrak{A}\mathfrak{a}) \ \wedge \ v = (\mathsf{pack}\mathfrak{B}\mathfrak{b}) \\
&\qquad\qquad \wedge \ \forall\boldsymbol{Z}.\exists\boldsymbol{R}\subset(\mathfrak{A}\boldsymbol{Z})\times(\mathfrak{B}\boldsymbol{Z}) \ . \ \mathfrak{a}_{F_2}(\mathfrak{T}_{F_2}[\boldsymbol{R}, \mathbf{eq}_{\boldsymbol{ZZ'}}]^{\mathsf{abo}}_{\mathsf{C}})\mathfrak{b}_{F_2})
\end{aligned}$$

where $\mathsf{abo} = (\mathfrak{A}\boldsymbol{Z}), (\mathfrak{B}\boldsymbol{Z}), \mathfrak{a}_{F_2}, \mathfrak{b}_{F_2}$.

Theorem 7.15 *Let* $\mathfrak{T}[\mathfrak{X}, \boldsymbol{Z'}]$ *adhere to* $\mathtt{PADT}_{Obs}$. *Extending the language with the predicates* Closed *of Def. 5.19, with* spParamC *we derive that the existence of an abo-simulation relation is equivalent to observational equivalence,*

$$\forall\boldsymbol{Z'}.\forall u, v:\exists\mathfrak{X}.\mathfrak{T}[\mathfrak{X}, \boldsymbol{Z'}] \ . \ u \ \mathsf{PolySimRelC}_{\mathfrak{T}[\mathfrak{X},\mathbf{eq}_{\boldsymbol{Z'}}]} \ v \quad\Leftrightarrow\quad u \ \mathsf{PolyObsEqC}^{Obs}_{\mathfrak{T}[\mathfrak{X},\boldsymbol{Z'}]} \ v$$

Proof: This follows from Theorem 5.33 (*p.* 138). □

Theorem 7.16 (Composability of Simulation Relations) *With* spParamC, *for* $\mathfrak{T}[\mathfrak{X}, \boldsymbol{Z'}]$ *adhering to* $\mathtt{PADT}_{Obs}$, *we can derive*

$$\begin{aligned}&\forall \boldsymbol{Z}.\mathfrak{A}, \mathfrak{B}, \mathfrak{C}, \boldsymbol{R}\subset(\mathfrak{A}\boldsymbol{Z})\times(\mathfrak{B}\boldsymbol{Z}), \boldsymbol{S}\subset(\mathfrak{B}\boldsymbol{Z})\times(\mathfrak{C}\boldsymbol{Z}).\\ &\qquad \forall \mathfrak{a}:\mathfrak{T}[\mathfrak{A}, \boldsymbol{Z'}], \mathfrak{b}:\mathfrak{T}[\mathfrak{B}, \boldsymbol{Z'}], \mathfrak{c}:\mathfrak{T}[\mathfrak{C}, \boldsymbol{Z'}]\ .\\ &\qquad\qquad \mathfrak{a}_{F_2}(\mathfrak{T}_{F_2}[\boldsymbol{R}, \mathsf{eq}_{\boldsymbol{ZZ'}}]^{\mathsf{abo}}_{\mathsf{C}})\mathfrak{b}_{F_2}\ \wedge\ \mathfrak{b}_{F_2}(\mathfrak{T}_{F_2}[\boldsymbol{S}, \mathsf{eq}_{\boldsymbol{ZZ'}}]^{\mathsf{abo}}_{\mathsf{C}})\mathfrak{c}_{F_2}\\ &\qquad\qquad\qquad\qquad \Rightarrow\ \mathfrak{a}_{F_2}(\mathfrak{T}_{F_2}[\boldsymbol{S}\circ\boldsymbol{R}, \mathsf{eq}_{\boldsymbol{ZZ'}}]^{\mathsf{abo}}_{\mathsf{C}})\mathfrak{c}_{F_2}\end{aligned}$$

Proof: This follows from Theorem 5.36 (*p.* 139). □

We have thus lifted the results concerning simulation relations and observational equivalence for System F to F_3.

Notice however that the clause

$$\exists \mathfrak{A}:*\to*, \mathfrak{B}:*\to*.\exists \mathfrak{a}:\mathfrak{T}[\mathfrak{A}], \mathfrak{b}:\mathfrak{T}[\mathfrak{B}]\ .\ u = (\mathsf{pack}\mathfrak{A}\mathfrak{a})\ \wedge\ v = (\mathsf{pack}\mathfrak{B}\mathfrak{b})$$

in the definitions above is not obviously derivable in the same way the analogue clause is in F_2 by Theorem 3.6. Theorem 3.6 hinges on parametricity, which we have not established for F_3. This is acceptable at the programming level, since the user will always supply packages $(\mathsf{pack}\mathfrak{A}\mathfrak{a})$ and $(\mathsf{pack}\mathfrak{B}\mathfrak{b})$. For reasoning purposes, we are restricted to reason in terms of variables $\mathfrak{X}$ and $\mathfrak{x}$, rather than variables of existential type.

To facilitate this, we can provide expressions of observational equivalence and simulation relations that speak directly in terms of package components instead of variables of existential type. These definitions will therefore not be in terms of a relation in the logic, but rather a meta-statement about a certain proposition.

Definition 7.17 (Observational Equivalence (*PolyObsEq*)**)** *For* $\mathfrak{A}$, $\mathfrak{a}:\mathfrak{T}[\mathfrak{A}]$, $\mathfrak{B}$, $\mathfrak{b}:\mathfrak{T}[\mathfrak{B}]$, *we say that* $(\mathsf{pack}\mathfrak{A}\mathfrak{a})$ *and* $(\mathsf{pack}\mathfrak{B}\mathfrak{b})$ *are* observationally equivalent w.r.t. $\mathfrak{T}[\mathfrak{X}]$ and observable types *Obs if the following proposition is derivable.*

$$\begin{aligned}&\mathit{PolyObsEq}^{Obs}_{\mathfrak{T}[\mathfrak{X}]}(\mathfrak{A}, \mathfrak{B}, \mathfrak{a}, \mathfrak{b}) \stackrel{def}{=}\\ &\quad \forall \boldsymbol{Z}.\textstyle\bigwedge_{D\in Obs\cup\{\boldsymbol{Z}\}}\forall f:\forall \boldsymbol{X}:*.(\mathfrak{T}_{F_2}[\boldsymbol{X}, \boldsymbol{Z}]\to D)\ .\ (f(\mathfrak{A}\boldsymbol{Z})\mathfrak{a}_{F_2}) =_D (f(\mathfrak{B}\boldsymbol{Z})\mathfrak{b}_{F_2})\end{aligned}$$

We define relatedness by simulation relation by meta-statement in F_2 as follows.

Definition 7.18 (Simulation Relation (*PolySimRel*)**)** *For any* $\mathfrak{A}$, $\mathfrak{B}$, $\mathfrak{a}:\mathfrak{T}[\mathfrak{A}]$, $\mathfrak{b}:\mathfrak{T}[\mathfrak{B}]$ *we say that* $(\mathsf{pack}\mathfrak{A}\mathfrak{a})$ *and* $(\mathsf{pack}\mathfrak{B}\mathfrak{b})$ *are* related by simulation relation w.r.t. $\mathfrak{T}[\mathfrak{X}, \boldsymbol{Z'}]$, *if the following proposition is derivable.*

$$\begin{aligned}&\mathit{PolySimRel}_{\mathfrak{T}[\mathfrak{X}, \mathsf{eq}_{\boldsymbol{Z'}}]}(\mathfrak{A}, \mathfrak{B}, \mathfrak{a}, \mathfrak{b}) \stackrel{def}{=}\\ &\qquad\qquad \forall \boldsymbol{Z}.\exists \boldsymbol{R}\subset(\mathfrak{A}\boldsymbol{Z})\times(\mathfrak{B}\boldsymbol{Z})\ .\ \mathfrak{a}_{F_2}(\mathfrak{T}_{F_2}[\boldsymbol{R}, \mathsf{eq}_{\boldsymbol{ZZ'}}])\mathfrak{b}_{F_2}\end{aligned}$$

Notice that the formulae involved in Definitions 7.17 and 7.18 have no F_3 syntax at all. Of course, we have

$$PolyObsEq^{Obs}_{\mathfrak{T}[\mathfrak{X}]}(\mathfrak{A},\mathfrak{B},\mathfrak{a},\mathfrak{b}) \;\Leftrightarrow\; (\mathsf{pack}\mathfrak{A}\mathfrak{a})\ \mathsf{PolyObsEq}^{Obs}\ (\mathsf{pack}\mathfrak{B}\mathfrak{b})$$

$$PolySimRel^{Obs}_{\mathfrak{T}[\mathfrak{X}]}(\mathfrak{A},\mathfrak{B},\mathfrak{a},\mathfrak{b}) \;\Leftrightarrow\; (\mathsf{pack}\mathfrak{A}\mathfrak{a})\ \mathsf{PolySimRel}^{Obs}\ (\mathsf{pack}\mathfrak{B}\mathfrak{b})$$

This gives

$$\forall \boldsymbol{Z'}\ .\ PolySimRel_{\mathfrak{T}[\mathfrak{X},\mathbf{eq}_{\boldsymbol{Z'}}]}(\mathfrak{A},\mathfrak{B},\mathfrak{a},\mathfrak{b}) \quad\Leftrightarrow\quad PolyObsEq^{Obs}_{\mathfrak{T}[\mathfrak{X},\boldsymbol{Z'}]}(\mathfrak{A},\mathfrak{B},\mathfrak{a},\mathfrak{b})$$

This can be done for the other notions of simulation relation and observational equivalence as well.

7.4 Specification Refinement Represented in F_2

The ultimate reason for representing notions in F_2 is that we want to use SUB and QUOT, or SUBG and QUOTG, for proving refinements in F_3. In this respect, a problem with the definitions in the previous section is that we will for any type constructor $\mathfrak{A}$ be required to exhibit a type constructor $\mathfrak{B}$ that gives a subobject or quotient for every $\mathfrak{B}A$. This requires an axiom of choice that most likely does not hold in the models one is likely to consider. For methodological purposes, we can get around this by avoiding variables of F_3 existential types altogether.

Definition 7.19 (ADT Specification Represented in F_2) *An* abstract data type specification *SP in F_3 represented in F_2 is a tuple $\langle\langle Sig_{SP}, \Theta_{SP\,F_2}\rangle, \mathfrak{T}^e_{SP}, Obs_{SP}\rangle$ where*

$$\begin{aligned}
&Sig_{SP} \stackrel{def}{=} \exists \mathfrak{X}\!:\!*\!\to\!*.\mathfrak{T}_{SP}[\mathfrak{X}],\\
&\Theta_{SP\,F_2}(\mathfrak{U},\mathfrak{u}) \stackrel{def}{=} \forall \boldsymbol{Z}\ .\ \exists \boldsymbol{X}\!:\!*.\exists \mathfrak{x}\!:\!\mathfrak{T}^e_{SP\,F_2}[\boldsymbol{X}]\ .\\
&\qquad\qquad (\mathsf{pack}\boldsymbol{\mathfrak{U}}\boldsymbol{Z}\,\mathfrak{u}_{F_2})\ \mathsf{ObsEq}X^{Obs_{SP}\cup\boldsymbol{Z}}_{\mathfrak{T}_{SP\,F_2}[\boldsymbol{X}]}\ (\mathsf{pack}\boldsymbol{X}\,\mathfrak{x}|_{\mathfrak{T}_{SP\,F_2}})\ \wedge\ Ax_{SP\,F_2}[\boldsymbol{X},\mathfrak{x}]
\end{aligned}$$

for $\mathsf{ObsEq}X$ either of ObsEq and ObsEqC, and where $Ax_{SP\,F_2}[\mathfrak{X},\mathfrak{x}]$ is a conjunction of formulae in the logic. If $\Theta_{SP}(\mathfrak{U},\mathfrak{u})$ is derivable, then $(\mathsf{pack}\mathfrak{U}\mathfrak{u})$ is said to be a realisation of *SP. We assume that Obs_{SP} contains*

- *possibly closed inductive types, such as* Bool *or* Nat*,*
- *parameters $\boldsymbol{Z'}$, in case $\mathfrak{T}[\mathfrak{X}]$ has free $\boldsymbol{Z'}$ other than $\mathfrak{X}$*

Here the specifier supplies Sig_{SP} and Ax_{SP}. The realisation predicate Θ_{SP} then employs $Ax_{SP\,F_2}$ which is equivalent to Ax_{SP}. Specification refinement now goes:

Definition 7.20 (Specification Refinement in F_3 Represented in F_2) *Let SP and SP' be abstract data type specifications in F_3 represented in F_2. Then SP' is a* refinement *of SP* via constructor $F : Sig_{SP'} \to Sig_{SP}$, *if for all $\mathfrak{U}$ and $\mathfrak{u} : \mathfrak{T}[\mathfrak{U}]$,*

$$\Theta_{SP'}(\mathfrak{U}, \mathfrak{u}) \;\Rightarrow\; \Theta_{SP}(\mathfrak{U}', \mathfrak{u}')$$

where $(\mathsf{pack}\mathfrak{U}'\mathfrak{u}') = F(\mathsf{pack}\mathfrak{U}\mathfrak{u})$, *is derivable. We write $SP \underset{F}{\rightsquigarrow} SP'$ for this fact.*

Example 7.21 We refine polymorphic stacks to polymorphic arrays with top pointer. Recall the specification of the polymorphic stack ADT from Example 7.1:

$$\mathsf{polySTACK} \stackrel{def}{=} \langle\langle Sig_{\mathsf{polySTACK}}, \Theta_{\mathsf{polySTACK}}\rangle, \mathfrak{T}^e_{\mathsf{polySTACK}}, \{\}\rangle, \text{ where}$$

$$Sig_{\mathsf{polySTACK}} = \exists \mathfrak{X} : * \to *. \mathfrak{T}_{\mathsf{polySTACK}}[\mathfrak{X}],$$

$$\begin{aligned}\mathfrak{T}_{\mathsf{polySTACK}}[\mathfrak{X}] = (&\mathsf{empty} : \forall Z . \mathfrak{X}Z,\ \mathsf{push} : \forall Z . Z \to \mathfrak{X}Z \to \mathfrak{X}Z,\\ &\mathsf{pop} : \forall Z . \mathfrak{X}Z \to \mathfrak{X}Z,\ \mathsf{top} : \forall Z . \mathfrak{X}Z \to Z \to Z,\\ &\mathsf{map} : \forall Z, Z' . (Z \to Z') \to \mathfrak{X}Z \to \mathfrak{X}Z'\ ),\end{aligned}$$

$$\mathfrak{T}_{\mathsf{polySTACK}}[\mathfrak{X}]^e = \mathfrak{T}_{\mathsf{polySTACK}}[\mathfrak{X}] \times \mathsf{multipop} : \forall Z . \mathsf{Nat} \to \mathfrak{X}Z \to \mathfrak{X}Z$$

$$\begin{aligned}&\Theta_{\mathsf{polySTACK}}(u) = \exists \mathfrak{X} : * \to *. \exists \mathfrak{x} : \mathfrak{T}^e_{\mathsf{polySTACK}}[\mathfrak{X}] \,.\, u\ \mathsf{PolyObsEqC}F_2{}^{\{\}}\ (\mathsf{pack}\mathfrak{X}\,\mathfrak{x}|_{\mathfrak{T}_{\mathsf{polySTACK}}}) \wedge\\ &\ \forall Z . \forall x : Z, s : \mathfrak{X}Z \,.\, (\mathfrak{x}.\mathsf{pop}Z(\mathfrak{x}.\mathsf{push}Z\ x\ s)) = s \ \wedge\\ &\ \forall Z . \forall x, z : Z, s : \mathfrak{X}Z \,.\, (\mathfrak{x}.\mathsf{top}Z(\mathfrak{x}.\mathsf{push}Z\ x\ s)\ z) = x \ \wedge\\ &\ \forall Z . \forall s : \mathfrak{X}Z \,.\, (\mathfrak{x}.\mathsf{multipop}Z\ 0\ s) = s \ \wedge\\ &\ \forall Z . \forall n : \mathsf{Nat}, s : \mathfrak{X}Z \,.\, (\mathfrak{x}.\mathsf{multipop}Z\ (\mathsf{succ}\, n\ s)) = (\mathfrak{x}.\mathsf{multipop}Z\ n\ (\mathfrak{x}.\mathsf{pop}Z\ s)) \ \wedge\\ &\ \forall Z, Z' . \forall n : \mathsf{Nat} . \forall s : \mathfrak{X}Z . \forall g : Z \to Z' \,.\\ &\qquad (\mathfrak{x}.\mathsf{top}Z'\ (\mathfrak{x}.\mathsf{multipop}Z'\ n\ (\mathfrak{x}.\mathsf{map}ZZ'gs))) = g(\mathfrak{x}.\mathsf{top}Z(\mathfrak{x}.\mathsf{multipop}Z\ ns))\end{aligned}$$

We could specify a polymorphic array ADT, also with a map operation, like this:

$$\mathsf{polyARRAY} \stackrel{def}{=} \langle\langle Sig_{\mathsf{polyARRAY}}, \Theta_{\mathsf{polyARRAY}}\rangle, \{\}\rangle, \text{ where}$$

$$Sig_{\mathsf{polyARRAY}} = \exists \mathfrak{X} : * \to *. \mathfrak{T}_{\mathsf{polyARRAY}}[\mathfrak{X}],$$

$$\begin{aligned}\mathfrak{T}_{\mathsf{polyARRAY}}[\mathfrak{X}] = (&\mathsf{empty} : \forall Z . \mathfrak{X}Z,\ \mathsf{update} : \forall Z . \mathsf{Nat} \to Z \to \mathfrak{X}Z \to \mathfrak{X}Z,\\ &\mathsf{access} : \forall Z . \mathsf{Nat} \to \mathfrak{X}Z \to Z,\\ &\mathsf{map} : \forall Z, Z' . (Z \to Z') \to \mathfrak{X}Z \to \mathfrak{X}Z'\ ),\end{aligned}$$

$$\begin{aligned}\Theta_{\mathsf{polyARRAY}}(u) = \exists \mathfrak{X} : * \to *. \exists \mathfrak{x} : \mathfrak{T}_{\mathsf{polyARRAY}}[\mathfrak{X}] \,.\, &u\ \mathsf{PolyObsEqC}F_2{}^{\{\}}\ (\mathsf{pack}\mathfrak{X}\mathfrak{x}) \ \wedge\\ &\forall Z . \forall i, j : \mathsf{Nat} . \forall x : Z, a : \mathfrak{X}Z \,.\, \mathfrak{x}.\mathsf{access}Z(i)(\mathfrak{x}.\mathsf{update}Z(j)(x)(a)) =\\ &\qquad\qquad \mathsf{if}\ i = j\ \mathsf{then}\ x\ \mathsf{else}\ \mathfrak{x}.\mathsf{access}Z(i)(a) \ \wedge\\ &\forall Z, Z' . \forall i : \mathsf{Nat} . \forall a : \mathfrak{X}Z . \forall g : Z \to Z' \,.\\ &\quad \mathfrak{x}.\mathsf{access}Z'(i)(\mathfrak{x}.\mathsf{map}ZZ'(g)(a)) = g(\mathfrak{x}.\mathsf{access}Z(i)(a))\end{aligned}$$

For representation in F_2 we get, for $\mathsf{polySTACK}$:

$$\begin{array}{l}\Theta_{\mathsf{polySTACK}_{F_2}}(\mathfrak{U},\mathfrak{u}) = \forall Z_1, Z_2 \,.\, \exists X_1, X_2 {:} *.\exists \mathfrak{x} {:} \mathfrak{T}^e_{\mathsf{polySTACK}_{F_2}}[X_1, X_2] \,. \\ \qquad (\mathsf{pack}(\mathfrak{U}Z_1)(\mathfrak{U}Z_2)\mathfrak{u}_{F_2}) \;\mathsf{ObsEqC}^{\{Z_1,Z_2\}}_{\mathfrak{T}_{\mathsf{polySTACK}_{F_2}}[X_1,X_2]}\; (\mathsf{pack}X_1 X_2\, \mathfrak{x}|_{\mathfrak{T}_{\mathsf{polySTACK}_{F_2}}}) \;\wedge \\ \quad \forall x {:} Z_i, s {:} X_i \,.\, (\mathfrak{x}_{F_2}.\mathsf{pop}_i(\mathfrak{x}_{F_2}.\mathsf{push}_i\, x\; s)) = s \;\wedge \\ \quad \forall x {:} Z_i, s {:} X_i \,.\, (\mathfrak{x}_{F_2}.\mathsf{top}_i(\mathfrak{x}_{F_2}.\mathsf{push}_i\, x\; s)) = x \;\wedge \\ \quad \forall s {:} X_i \,.\, (\mathfrak{x}_{F_2}.\mathsf{multipop}_i\, 0\; s) = s \;\wedge \\ \quad \forall n {:} \mathsf{Nat}, s {:} X_i \,.\, (\mathfrak{x}_{F_2}.\mathsf{multipop}_i(\mathsf{succ}\, n)\; s) = (\mathfrak{x}_{F_2}.\mathsf{multipop}_i\, n\; (\mathfrak{x}_{F_2}.\mathsf{pop}_i\, s)) \;\wedge \\ \quad \forall n {:} \mathsf{Nat}.\forall s {:} X_i.\forall g {:} Z_i \to Z_j \,. \\ \qquad (\mathfrak{x}_{F_2}.\mathsf{top}_j(\mathfrak{x}_{F_2}.\mathsf{multipop}_j(n)(\mathfrak{x}_{F_2}.\mathsf{map}_l g s))) = g(\mathfrak{x}_{F_2}.\mathsf{top}_i(\mathfrak{x}_{F_2}.\mathsf{multipop}_i(n)(s)))\end{array}$$

and the representation in F_2 of $\mathsf{polyARRAY}$ is:

$$\begin{array}{l}\Theta_{\mathsf{polyARRAY}_{F_2}}(\mathfrak{U},\mathfrak{u}) = \forall Z_1, Z_2 \,.\, \exists X_1, X_2 {:} *.\exists \mathfrak{x} {:} \mathfrak{T}_{\mathsf{polyARRAY}_{F_2}}[X_1, X_2] \,. \\ \qquad (\mathsf{pack}X_1 X_2 \mathfrak{x}) \;\mathsf{ObsEqC}^{\{Z_1,Z_2\}}_{\mathfrak{T}_{\mathsf{polyARRAY}_{F_2}}[X_1,X_2]}\; (\mathsf{pack}(\mathfrak{U}Z_1)(\mathfrak{U}Z_2)\mathfrak{u}_{F_2}) \;\wedge \\ \qquad \forall n, m {:} \mathsf{Nat}.\forall x {:} Z_i, a {:} X_i \,.\, \mathfrak{x}_{F_2}.\mathsf{access}_i(n)(\mathfrak{x}_{F_2}.\mathsf{update}_i(m)(x)(a)) = \\ \qquad\qquad \mathsf{if}\; n = m \;\mathsf{then}\; x \;\mathsf{else}\; \mathfrak{x}_{F_2}.\mathsf{access}_i(n)(a) \;\wedge \\ \qquad \forall n {:} \mathsf{Nat}.\forall a {:} X_i.\forall g {:} Z_i \to Z_j \,. \\ \qquad\quad \mathfrak{x}_{F_2}.\mathsf{access}_j(n)(\mathfrak{x}_{F_2}.\mathsf{map}_l(g)(a)) = g(\mathfrak{x}_{F_2}.\mathsf{access}_i(n)(a))\end{array}$$

We can now show $\mathsf{polySTACK} \underset{F}{\rightsquigarrow} \mathsf{polyARRAY}$ for

$$\begin{array}{l}F \stackrel{def}{=} \lambda u {:} Sig_{\mathsf{polyARRAY}} \,. \\ \quad \mathsf{unpack}(u)(Sig_{\mathsf{polySTACK}})(\Lambda\mathfrak{X}.\lambda\mathfrak{x} {:} \mathfrak{T}_{\mathsf{polyARRAY}}[\mathfrak{X}] \,.\, (\mathsf{pack}\,(\lambda Z {:} *.\mathfrak{X}Z \times \mathsf{Nat})\; \mathfrak{x}'))\end{array}$$

where

$$\begin{array}{l}\mathfrak{x}' \stackrel{def}{=} (\mathsf{empty} = \Lambda Z.\mathsf{pair}(\mathfrak{x}.\mathsf{empty}Z)(0), \\ \quad \mathsf{push} = \Lambda Z.\lambda z {:} Z.\lambda p {:} \mathfrak{X}Z \times \mathsf{Nat} \,.\, \mathsf{pair}(\mathfrak{x}.\mathsf{update}Z(\mathsf{snd}\, p)(z)(\mathsf{fst}\, p))(\mathsf{succ}(\mathsf{snd}\, p)), \\ \quad \mathsf{pop} = \Lambda Z.\lambda p {:} \mathfrak{X}Z \times \mathsf{Nat} \,.\, \mathsf{ifzero}(\mathsf{snd}\, p)(p)(\mathsf{pair}(a)(\mathsf{pred}(\mathsf{snd}\, p))), \\ \quad \mathsf{top} = \Lambda Z.\lambda p {:} \mathfrak{X}Z \times \mathsf{Nat}.\lambda z {:} Z \,.\, \mathsf{ifzero}(\mathsf{snd}\, p)(z)(\mathfrak{x}.\mathsf{access}(\mathsf{pred}(\mathsf{snd}\, p))(\mathsf{fst}\, p)) \\ \quad \mathsf{map} = \Lambda Z.\Lambda Z'.\lambda g {:} Z \to Z'.\lambda p {:} \mathfrak{X}Z \times \mathsf{Nat} \,.\, \cdots)\end{array}$$

Consider arbitrary $\mathfrak{U}$ and $\mathfrak{u} : \mathfrak{T}[\mathfrak{U}]$. Let $(\mathsf{pack}\mathfrak{U}'\mathfrak{u}')$ denote $F(\mathsf{pack}\mathfrak{U}\mathfrak{u})$ We must show the derivability of

$$\Theta_{\mathsf{polyARRAY}_{F_2}}(\mathfrak{U},\mathfrak{u}) \;\Rightarrow\; \Theta_{\mathsf{polySTACK}_{F_2}}(\mathfrak{U}',\mathfrak{u}')$$

that is, assuming

$$\begin{array}{l}\forall Z_1, Z_2 \,.\, \exists A_1, A_2 {:} *.\exists \mathfrak{a} {:} \mathfrak{T}_{\mathsf{polyARRAY}_{F_2}}[A_1, A_2] \,. \\ \qquad (\mathsf{pack}(\mathfrak{U}Z_1)(\mathfrak{U}Z_2)\,\mathfrak{u}_{F_2}) \;\mathsf{ObsEqC}^{\{Z_1,Z_2\}}_{\mathfrak{T}_{\mathsf{polyARRAY}_{F_2}}[X_1,X_2]}\; (\mathsf{pack}A_1 A_2 \mathfrak{a}) \\ \qquad\qquad \wedge\; Ax_{\mathsf{polyARRAY}_{F_2}}[A_1, A_2, \mathfrak{a}]\end{array}$$

we should be able to derive

$$\begin{aligned}&\forall Z_1, Z_2 \,.\, \exists B_1, B_2{:}{*}.\exists \mathfrak{b}{:}\mathfrak{T}^e_{\mathsf{polySTACK}\,F_2}[B_1,B_2]\,.\\ &\quad(\mathsf{pack}(\mathfrak{U}'Z_1)(\mathfrak{U}'Z_2)\,\mathfrak{u}'_{F_2})\ \mathsf{ObsEqC}^{\{Z_1,Z_2\}}_{\mathfrak{T}_{\mathsf{polySTACK}\,F_2}}[X_1,X_2]\ (\mathsf{pack}B_1B_2\,\mathfrak{b}|_{\mathfrak{T}_{\mathsf{polySTACK}\,F_2}})\\ &\qquad\qquad\qquad\qquad\wedge\ Ax_{\mathsf{polySTACK}\,F_2}[B_1,B_2,\mathfrak{b}]\end{aligned}$$

We postulate that F is stable, and replace $\mathsf{pack}(\mathfrak{U}Z_1)(\mathfrak{U}Z_2)\,\mathfrak{u}_{F_2})$ by $(\mathsf{pack}A_1A_2\mathfrak{a})$ for our purposes. Let $(\mathsf{pack}A'_1A'_2\mathfrak{a}')$ denote $F(\mathsf{pack}A_1A_2\mathfrak{a})$. We can now use the technology for System F. In particular we would like to use QUOTG and SUBG. The first thing we must do is axiomatise a suitable partial congruence. The congruence must relate array with pointers that represent the same stack. Thus, we define

$$\begin{aligned}\sim_i\ \stackrel{def}{=}\ (a{:}A'_i, a'{:}A'_i)\,.\,((\mathsf{snd}a) = (\mathsf{snd}a')\ \wedge\ \forall n{:}\mathsf{Nat}\,.\,0 \le n\ \wedge\ n < (\mathsf{snd}a)\\ \Rightarrow\ \mathfrak{a}.\mathsf{access}_i(n)(\mathsf{fst}a) =_{Z_i} \mathfrak{a}.\mathsf{access}_i(n)(\mathsf{fst}a'))\end{aligned}$$

Here $\sim$ is total, so we only need QUOTG. The rest of the verification goes in a standard manner. ○

—

We have now expressed specification refinement for F_3 by representation in F_2. There are two gaps that remain to be bridged. The first is the question whether or not the notion of observational equivalence $\mathsf{PolyObsEq}$ in F_3 (Def. 7.4) is equivalent to the representation $\mathsf{PolyObsEq}F_2$ in F_2 (Def. 7.6). It is easy to show

$$u\ \mathsf{PolyObsEq}\ v\ \Rightarrow\ u\ \mathsf{PolyObsEq}F_2\ v$$

but the converse implication seems to require the axiom of choice. This also applies to the closed-computation versions of $\mathsf{PolyObsEq}$ and $\mathsf{PolyObsEq}F_2$.

Secondly, we want that the F_3 notion of specification refinement (Def. 7.2) is equivalent to the F_2 representation version (Def. 7.19). Here we must consider packages, rather than variables of existential type. It is easy to see that

$$\Theta_{SP}(\mathsf{pack}\mathfrak{U}\mathfrak{u})\ \Rightarrow\ \Theta_{SP\,F_2}(\mathfrak{U},\mathfrak{u})$$

Again the axiom of choice seems necessary for obtaining the converse direction. It might be easier to show an equivalence between the notions of specification refinement (Definitions 7.3 and 7.20), rather than between the notions of specification and between the notions of observational equivalence. In the absence of these formal equivalences, one has to take the F_2 representations as definitional, rather than derived.

—

7.5 True F_3 Formalisms

The main bulk of this chapter has been devoted to representing polymorphic notions in F_2 in order that we may use the technology developed for System F. The idea is that the specifier can use F_3 expressions to specify the desired abstract data types, and then that these specifications get translated, ultimately automatically, into an F_2 representation together with F_2 verification conditions.

On the other hand, it would be even better if we could have true F_3 formalisms for everything we have developed for System F. From the beginning this would demand that we decide what the action of type constructors on relations are, that we assert relational parametricity in the context of type constructors, and that we find a model for this new logic. Then we should reformulate the notions of observational equivalence and simulation relation in this context. The axiom schemata SUB, QUOT and SUBG, QUOTG, should also be reformulated.

We think that this endeavour is compelling and raises some intriguing issues. Nevertheless, we leave this for future study.

7.6 Summary

This chapter addressed polymorphism inside data types. It is necessary to use the polymorphic lambda calculus F_3, rather than System F, in order to express that the data representation of abstract types depends on other types. However, we were able to express the notions of observational equivalence, and relatedness by simulation relation, in terms of System F technology developed in earlier chapters. This frees us from the burden of considering relational parametricity for F_3, and at the same time gives us earlier results for System F lifted to F_3, virtually for free. However, the formal equivalence between the F_3 notions and their F_2 representations is not clear, due to an apparent call for the axiom of choice.

Chapter 8

Conclusions

8.1 Summary

This thesis has expressed specification refinement in type theory. The type-theoretic setting consisted of the polymorphic lambda calculus together with extensions of Plotkin and Abadi's logic for parametric polymorphism asserting relational parametricity.

The motivation to do this lies in the desire to generalise successful concepts from the field of algebraic specification refinement to beyond inherent first-order formalisms. We concentrated on generalisations to higher-order operation profiles and polymorphism. The reason for doing this specifically in type theory is firstly that type theory is an adequately powerful syntactic formalism with several well-defined semantics. Secondly, the expressivity of type theory allows one to internalise semantic notions into syntax. This then ultimately links the practice of specification refinement to a multitude of powerful reasoning tools.

At first order, we showed a correspondence between algebraic specification refinement and type-theoretic refinement. The correspondence encompasses normal-form specifications and constructors. This then applies to structured specifications in light of the specification normalisation results and the view that the abstractor operator that in our discussion gives specification up to observational equivalence, should only be applied outermost. At first order, we utilised the fact that the existence of simulation relations coincides with observational equivalence, which then therefore corresponds to equality at existential type.

For data types with higher-order operations, observational equivalence loses its two-way correspondence to simulation relations. Furthermore, simulation relations do not in general compose at higher order. This compromises the use of these notions for stepwise refinement. We argued that this situation could in fact be rectified, because the standard notion of simulation relation fails to observe a crucial aspect of the information-hiding abstraction barrier promoted by existential types. The proposed solution is to modify the notion of simulation relation by weakening the arrow-type relation according to how clients may actually use data type operations. The connection to observational equivalence and the composability of simulation relations is thus restored. The correspondence to observational equivalence suggests that the notion of abstraction barrier-observing simulation relation is exactly the desired relation for explaining refinement.

A universal proof strategy for proving observational refinements was also imported into the type-theoretic setting. This was done by introducing axioms SUB and QUOT asserting the existence of subobjects and quotients. While validating these axioms w.r.t. the parametric PER model at first order is easy, the situations for higher-order operations is more complex. Nonetheless, we can validate these axioms at any order using a setoid-based model. We then get a scenario for observational refinement at any order depicted as follows.

$$\begin{tikzcd} \textsc{Quot} \arrow[dr] & \\ & \mathsf{SimRel} \quad \Rightarrow \quad \mathsf{ObsEq} \\ \textsc{Sub} \arrow[ur] & \end{tikzcd}$$

However, it may not be possible to use standard simulation relations at higher order. We therefore supplied axioms SUBG and QUOTG that incorporate the abstraction barrier-observing notion of simulation relation. This gives the scenario for observational refinement at any order depicted as follows.

$$\begin{tikzcd} \textsc{QuotG} \arrow[dr] & \\ & \mathsf{SimRelC} \quad \Rightarrow \quad \mathsf{ObsEqC} \\ \textsc{SubG} \arrow[ur] & \end{tikzcd}$$

When attempting to validate SUBG and QUOTG, one can observe that the parametric interpretation into the PER structure ignores the same crucial aspect of the information-hiding abstraction barrier promoted by existential types, as was in question for simulation relations. Prompted by this, we proposed an abstraction barrier-observing interpretation into the PER structure.

The main bulk of the discussion focuses on the higher-order aspect, *i.e.,* data types with higher-order operations. To deal properly with polymorphism inside data types, we used the third-order polymorphic lambda calculus for expressivity, and then reduced the formalisms to second-order ones. This allowed us to use the results obtained earlier also for data types with polymorphic operations.

8.2 Further Research

During our main discussion, we mentioned several ideas for further work.

For Ch. 4 we mentioned expressing specification building operators in System F and the logic, in order to reflect specificational structure in proofs. We also mentioned providing a fuller account of specifications of parameterised programs as well as parameterised specifications.

In Ch. 5, we established the link between observational equivalence and the existence of simulation relations at higher order. However, the logic is not strong enough to show the existence of simulation relations for certain examples. We mentioned that heuristics from semantical reasoning should give rise to a stronger logic. Alternatively, one could operate with assumptions in the logic, and employ semantic reasoning outside the logic to informally discharge those assumptions. This would however be less beneficial to the specifier, unless the specifier is a mathematician of the appropriate sort.

In Ch. 6, one should investigate further if the setoid semantics can fully replace the rôle of the data type semantics and annotated types.

For Ch. 7 we would very much like to see pure F_3 formalisms all the way through, indeed one could envision a generalisation of this whole discussion to any F_i, and thence F_ω.

We now suggest in the following, the most promising lines of further investigation that in our view could build on the results of this thesis.

8.2.1 Extending to Object Orientation

Having lifted the concepts of algebraic specification refinement into the realm of type theory and lambda calculi, one now has the potential to extend this fruitful idea to various programming concepts. The field that this task constitutes is vast, and in this light, this thesis only begins to cover some modest ground. An immediate aim would be to extend the idea of specification refinement to object orientation. This could be done in the same type-theoretic framework with the logic for parametric polymorphism that underlies the work in this thesis and

(Poll and Zwanenburg, 1999; Zwanenburg, 1999). The reason for the anticipated success of this approach is the results in (Reddy, 1999, 1997, 1988) establishing a clarifying link between classes and objects and abstract data types. Classes determine not only what the operations of its objects should do, but may also to some degree determine the implementations of these operations. Moreover, classes are seen type-theoretically to be of abstract types similar to the abstract types accompanying abstract data types. Recall that abstract types are coded in the polymorphic lambda calculus as existential types. Thus classes are seen to sub-classify algebras of the same abstract type. There are various other views on this, see (Hofmann and Pierce, 1996, 1995), (Bruce et al., 1997), (Fischer and Mitchell, 1997; Fisher and Mitchell, 1996), (Abadi and Cardelli, 1996), and although we find the one above most apt for our purposes, the other alternatives should come into consideration. Relating to component-based development, a certain flexibility is called for in light of the ongoing discussion of what in fact constitutes a component.

8.2.2 Clarifying Simulation Relations

The exact correspondence between the alternative semantic notions of refinement relations (Honsell et al., 2000; Honsell and Sannella, 1999; Kinoshita and Power, 1999; Kinoshita et al., 1997; Plotkin et al., 2000), and the syntactic notion of abstraction barrier-observing simulation relation in Ch. 5 has not been clarified. This task is important for several reasons. First, the ongoing research on alternative refinement relations is of wide-spread interest, and has resolved a series of hitherto unclarified issues, *e.g.,* the lack of composability precluding stepwise refinement, and the failure of correspondence with observational equivalence at higher order. It is therefore important to connect the semantic and syntactic notions for reasoning purposes. Secondly, refinement relations also explain abstraction in object-oriented approaches. Again, this only works satisfactorily for first-order operations. Therefore, the recent research on abstraction barrier-observing simulation relations is highly relevant for the object-oriented approach. It seems particularly fruitful to look at the connection between the semantic notion of *pre-logical relations* (Honsell and Sannella, 1999; Honsell et al., 2000), and the syntactic notions developed in this thesis.

One should also try expressing pre-logical relations directly in the polymorphic lambda-calculus and logic. For example, in System F and the logic, semantic notions such as applicative structures and combinatory algebras can be internalised in syntax. The semantic notion of pre-logical relations could therefore also be internalised. The problem posed by an infinite family of relations might be solved by using the more lenient notion of definability w.r.t. the relevant ADT according to *Abs-Bar*, as developed and used in this thesis, rather than more absolute forms of definability.

8.2.3 Reasoning within Abstraction Barriers

The encapsulation aspect of information hiding provides vital protection of both the data and the operations of a module. For example, if sets are represented by built-in lists, one might for performance reasons want the set constructors to build sorted lists without duplicates, and then, say, a *remove* operation would benefit from this sortedness. However, if users are allowed to use arbitrary lists to build sets, one will get inconsistencies since the operations assume sorted lists, *e.g.,* the predicate $\forall x, s.in(x, remove(x, s)) = \mathit{false}$ will fail for wrongfully created sets with multiple occurrences if the remove operator only removes first occurrences.

Encapsulation solves these issues for programming by raising appropriate abstraction barriers. However, for reasoning about data types, one must in general take into account the internal representation, and then one must take action so as to avoid inconsistencies. It is argued in (Hannay, 1998) that one should reflect abstraction barriers in proofs as well. This is demonstrated for simple equational logic. In fact, when introducing the universal proof strategy for observational refinement proofs in Sect.2.7, we also mentioned a possibility for proof simplification using the referentially opaque equational calculus from (Hannay, 1998). This simplification carries over to the type-theoretic setting as well. Furthermore, it might be fruitful to implement abstraction barriers of various kinds in logic to reflect information hiding in general, and then the information hiding captured by *Abs-Bar* in particular. This might in turn present an alternative approach to that represented by the modified parametric PER-model interpretation of Ch. 6.

8.2.4 Using Tools

There is an obvious thing which we have not done, and that is to use our formalisms in practice in tools such as HOL, LEGO, Coq, Isabelle. See (Zwanenburg, 1999) for proofs done in Yarrow. We are fairly convinced of the usefulness of the formalisms in this thesis, but putting them to use will inevitably disclose any shortcomings or modifications that ought to be addressed.

This thesis is full of type-theoretic elaborations intended to aid the specifier. However, these formalistic ideas are meant to be accessible to the specifier only through a sensible interface, such as a specification language. The type-theoretic elaborations typically express proof obligations to be submitted to the tools just mentioned. Therefore, machinery must be implemented that compiles what the specifier writes, into proof obligations as devised in this thesis.

8.2.5 Other Models and Other Type Systems

We have added axioms to the logic, but we have not made any model-theoretical deliberations outside relating the discussion mainly to interpretations into the parametric PER-model. It is for example highly relevant to investigate the general power of these axioms in restricting the class of models. This would especially come into play if one were to consider consistency w.r.t. related type systems. We have not looked into whether or not the logic extended with the axioms Quot and Sub, or QuotG and SubG is complete w.r.t. the parametric models.

8.2.6 Semantic Reasoning

We generalised notions of specification refinement in type theory partly because this brings the subject closer to mechanical reasoning, an absolute necessity for any real-life software development process based on formal methods. However, we also recognise the fact that semantic reasoning is often easier and quicker to perform in the absence of adequate tools and in the presence of experts. Indeed syntactic formal reasoning can seem unnecessarily cumbersome at times, and some people might find this exactly the case in this thesis.

We would like to see the development in this thesis inspire reasoning on the semantic level as well as on the syntactic level explicitly treated in this thesis. The proposal to look at pre-logical relations in conjunction with our notions of simulation relation is one such endeavour.

Related to this, one should conduct research emphasising the semantic significance of the formalisms developed in this thesis. We have primarily used models to show the soundnes of extensions of the logic with various axioms, but it is also relevant to see what sense the formalisms make semantically. This is most fruitful and interesting when done w.r.t. non-syntactic models. That is why we made the effort to develop the results in this thesis also referring to such models.

—

This concludes the main body of this thesis. We hope you have enjoyed at least some of what has been described. Thank you for your attention.

If you have an apple and I have an apple and we exchange apples then you and I will still each have one apple. But if you have an idea and I have an idea and we exchange these ideas, then each of us will have two ideas. GEORGE BERNARD SHAW

Appendix A

Logical Deduction Rules

This appendix contains full sets of logical deduction rules, many of which are omitted from the main discussion.

A.1 Referentially Opaque Equational Calculi

Here we cite the referentially opaque equational calculi $\vdash^{\mathrm{FI}}$ and $\vdash^{\mathrm{FRI}}$ of (Hannay, 1998). We referred to these calculi in Sect. 2.8.

First, for any signature Σ, a Σ-*context* $c[\Box]$ is a term $c \in T_{\Sigma\cup\{\Box\}}(X)$. If $c[\Box]$ is a Σ-context, we will write $c \in T_{\Sigma}(X)$ instead of $c \in T_{\Sigma\cup\{\Box\}}(X)$.

A.1.1 Congruence Induced by a Set of Equations

Recall that for a set of Σ-equations $E \subseteq T_{\Sigma}(X) \times T_{\Sigma}(X)$, the *congruence* $\sim^A_E$ *induced by* E on any Σ-algebra A is defined as the least Σ-congruence containing $\{\langle \phi(l), \phi(r)\rangle \mid \langle l, r\rangle \in E,\ \phi : T_{\Sigma}(X) \to A\}$. This definition is equivalent to demanding the least *equivalence relation* containing $\{\langle \phi(c[l]), \phi(c[r])\rangle \mid \langle l, r\rangle \in E, c \in T_{\Sigma}(X),\ \phi : T_{\Sigma}(X) \to A\}$, *i.e.*, the relation inductively defined by

$$\textit{induce} : \frac{}{\phi(c[l]) \sim^A_E \phi(c[r])};\quad \langle l, r\rangle \in E, c \in T_{\Sigma}(X),\ \phi : T_{\Sigma}(X) \to A$$

$$\textit{refl} : \frac{}{a \sim^A_E a} \qquad \textit{sym} : \frac{a \sim^A_E a'}{a' \sim^A_E a} \qquad \textit{trans} : \frac{a \sim^A_E a'', a'' \sim^A_E a'}{a \sim^A_E a'}$$

The quotient w.r.t. to $\sim^A_E$ is written A/E. Of course, one usually writes $s \sim^{T_{\Sigma}(X)}_E t$ as $E \vdash s = t$. This defines the *equational calculus* over equations E.

A.1.2 Abstraction Barrier by FI Structure

We are focusing on equational instances of FRI and FI specificational structures. We start with the simpler FI-structure. The particular structure of interest is

$$\textbf{quotient}\ \ (\textbf{derive}\ \langle\Sigma, E\rangle\ \textbf{by}\ \mathit{incl} : \Sigma^{\mathit{export}} \hookrightarrow \Sigma)\ \ \textbf{by}\ \ E' \qquad (\dagger)$$

with semantics

$$\{(A|_{\Sigma^{\mathit{export}}})/E' \mid A \in \mathit{Mod}_\Sigma(E)\}$$

for $\mathit{Mod}_\Sigma(E)$ the class of Σ-algebras satisfying all equations in E. If $\Sigma^{\mathit{export}} \subsetneq \Sigma$ then the signature fragment $\Sigma^h = \Sigma \setminus \Sigma^{\mathit{export}}$ is outside the image of incl, so the reduct construct hides the interpretations of operator symbols and sorts in Σ^h. Structure is the essential abstraction barrier here: It is crucial that the hiding **derive** step is done before quotienting, since quotienting in the presence of hidden operators might yield inconsistency as illustrated in Example 2.5 (*p.* 35).

A.1.3 Calculus $\vdash^{\mathrm{FI}}$

The calculus $\vdash^{\mathrm{FI}}$ will be a generalisation in a certain sense of the equational calculus for the flat basic case, as explained in the following. For a basic specification $\mathit{SP} = \langle\Sigma, E\rangle$ we have by Birkhoff for the equational calculus $\vdash$

$$[\![\mathit{SP}]\!] \models s = t \ \Leftrightarrow\ T_\Sigma(X)/E \models s = t \ \Leftrightarrow\ E \vdash s = t \qquad *$$

Here $T_\Sigma(X)/E$ is a *classifying* model of $[\![\mathit{SP}]\!]$. Now let K^{FI} be the semantics of the FI structure ($\dagger$). We shall obtain a calculus $\vdash^{\mathrm{FI}}$ and a classifying model $T_{K^{\mathrm{FI}}}$ such that

$$K^{\mathrm{FI}} \models s = t \ \Leftrightarrow\ T_{K^{\mathrm{FI}}} \models s = t \ \Leftrightarrow\ \vdash^{\mathrm{FI}} s = t$$

Algebras in K^{FI} are of the form $(A|_{\Sigma^{\mathit{export}}})/E'$ where A is a model for E. The classifying model is $(T_\Sigma(X)/E|_{\Sigma^{\mathit{export}}})/E'$ (Theorem A.1 below). Viewing for the moment $T_\Sigma(X)/E|_{\Sigma^{\mathit{export}}}$ as a "term-algebra" T, we directly get an *abstract calculus* for K^{FI} by considering $\sim^T_{E'}$ on T and the classifying model T/E'. This is a generalisation of the basic case above where $E \vdash$ is given directly by $\sim^{T_\Sigma(X)}_E$. The abstract calculus thus operates on elements of T, *i.e.*, congruence classes q of $T_\Sigma(X)/E|_{\Sigma^{\mathit{export}}}$. Notice that each q has the form $[t]_E$ for $t \in T_\Sigma(X)|_{\Sigma^{\mathit{export}}}$, and recall that in general $T_{\Sigma^{\mathit{export}}}(X) \not\supseteq T_\Sigma(X)|_{\Sigma^{\mathit{export}}}$, because for any $s \in S^{\mathit{export}}$, ${T_\Sigma(X)|_{\Sigma^{\mathit{export}}}}_s = T_\Sigma(X)_s$.

Of course, instead of this abstract calculus we would rather have a calculus operating on terms. We obtain this by "opening up" the congruence classes q and then building a calculus over E' on $T_\Sigma(X)|_{\Sigma^{export}}$. Opening up the congruence classes necessitates importing the calculus $E \vdash$. We thus quite naturally get the following calculus $\vdash^{\mathrm{FI}}$. For all $u, v \in T_\Sigma(X)|_{\Sigma^{export}}$,

$$importE : \frac{E \vdash u = v}{\vdash^{\mathrm{FI}} u = v}$$

$$induceE' : \frac{}{\vdash^{\mathrm{FI}} \phi(c[l]) = \phi(c[r])}; \quad \begin{array}{l} \langle l, r\rangle \in E', c \in T_{\Sigma^{export}}(X), \\ \phi : T_{\Sigma^{export}}(X) \rightarrow T_\Sigma(X)|_{\Sigma^{export}} \end{array}$$

$$refl : \frac{}{\vdash^{\mathrm{FI}} u = u} \qquad sym : \frac{\vdash^{\mathrm{FI}} u = v}{\vdash^{\mathrm{FI}} v = u} \qquad trans : \frac{\vdash^{\mathrm{FI}} u = w, \vdash^{\mathrm{FI}} w = v}{\vdash^{\mathrm{FI}} u = v}$$

In rule $induceE'$, the contexts are $T_{\Sigma^{export}}(X)$-contexts, giving referential opacity w.r.t. $T_\Sigma(X)|_{\Sigma^{export}}$. This is a direct consequence of the definition of congruence on a Σ^{export}-algebra ($T_\Sigma(X)|_{\Sigma^{export}}$) induced by a set of Σ^{export}-equations (E'). In this way the abstraction barrier provided by the reduct construct in the semantics K^{FI}, gets its counterpart in the calculus in the form of referential opacity. Notice that in fact $\phi(c[\Box]) \in T_\Sigma(X)|_{\Sigma^{export}}$. However, the fact that $c[\Box]$ is a $T_{\Sigma^{export}}(X)$-context, ensures the essential property that all operator symbols in the path from $\Box$ to the root of $\phi(c[\Box])$ (seen as a tree of subterms) are from Ω^{export}.

Theorem A.1 (Soundness and completeness) *Let K^{FI} be the semantics $\{(A|_{\Sigma^{export}})/E' \mid A \in Mod_\Sigma(E)\}$ of (†). For all $u, v \in T_{\Sigma^{export}}(X)$,*

$$K^{\mathrm{FI}} \models u = v \;\Leftrightarrow\; T_\Sigma(X)/E|_{\Sigma^{export}}/E' \models u = v \;\Leftrightarrow\; \vdash^{\mathrm{FI}} u = v$$

Proof: This follows from Theorem A.5 by observations A.3 and A.4 below. □

A.1.4 Abstraction Barrier by FRI-structure

The FRI approach is in our context represented by the specification structure

quotient (**restrict** (**derive** $\langle\Sigma, E\rangle$ **by** $incl : \Sigma^{export} \hookrightarrow \Sigma$) **on** S') **by** E' (‡)

where $\Sigma^{export} = \langle S^{export}, \Omega^{export}\rangle$, $S' \subseteq S^{export}$. Its semantics is

$$\{R_{S'}(A|_{\Sigma^{export}})/E' \mid A \in Mod_\Sigma(E)\}$$

As mentioned in the main discussion, the input sorts In are now $S^{export} \setminus S'$. There is a range of model classes according to the choice of S'. The case $S' = \emptyset$ gives $R_\emptyset(A|_{\Sigma^{export}}) = A|_{\Sigma^{export}}$, and corresponds to FI.

A.1.5 Calculus $\vdash^{\mathrm{FRI}}$

Again we relate to the standard equational calculus. For the basic equational specification $SP = \langle \Sigma, E \rangle$ we have for $\vdash^{\omega}$, *i.e.,* the equational calculus augmented with the ω-rule,

$$Reach([\![SP]\!]) \models s = t \;\Leftrightarrow\; G_{\Sigma}/E \models s = t \;\Leftrightarrow\; \vdash^{\omega} s = t \qquad **$$

where $Reach([\![SP]\!])$ is the subclass of $[\![SP]\!]$ consisting of all algebras reachable on the sorts S of Σ, *i.e.,* ground term denotable algebras, or computation structures. Now, in the FRI approach we are interested in classes of reachable subalgebras, rather than subclasses of reachable algebras. However in a flat equational setting these two are the same: Let $R_{S'}(Mod_{\Sigma}(E)) = \{R_{S'}(A) \mid A \in Mod_{\Sigma}(E)\}$ and $Reach_{S'}(Mod_{\Sigma}(E)) = \{A \in Mod_{\Sigma}(E) \mid A \text{ is reachable on } S'\}$.

Fact A.2 $R_{S'}(Mod_{\Sigma}(E)) = Reach_{S'}(Mod_{\Sigma}(E))$

This correspondence means that we can utilise the ω-rule also when considering $R_{S'}([\![SP]\!])$. In fact it is follows that for arbitrary $S' \subseteq S$,

$$R_{S'}([\![SP]\!]) \models s = t \;\Leftrightarrow\; R_{S'}(T_{\Sigma}(X)/E) \models s = t \;\Leftrightarrow\; \vdash^{\omega}_{S'} s = t$$

where $\vdash^{\omega}_{S'}$ denotes the standard equational calculus augmented by the following parameterised ω-rule.

$$\frac{\forall \tau : T_{\Sigma}(X) \to R_{S'}(T_{\Sigma}(X)) \;.\; \vdash \tau(s) = \tau(t)}{\vdash s = t}$$

The special case $S' = \emptyset$ is simply $*$. The case $S' = S$ is $**$, in which case $R_{S'}(T_{\Sigma}(X)/E) \cong G_{\Sigma}/E$ is the initial object of $Mod_{\Sigma}(E)$.

Now let $K^{\mathrm{FRI}}_{S'}$ be the semantics the FRI structure (‡). By analogy to the basic case we will devise a calculus $\vdash^{\mathrm{FRI}}_{S'}$ with a parameterised ω-rule and classifying model $T_{K^{\mathrm{FRI}}_{S'}}$ such that

$$K^{\mathrm{FRI}}_{S'} \models s = t \;\Leftrightarrow\; T_{K^{\mathrm{FRI}}_{S'}} \models s = t \;\Leftrightarrow\; \vdash^{\mathrm{FRI}}_{S'} s = t$$

By the way, also analogously to the basic case, the classifying model $T_{K^{\mathrm{FRI}}_{S'}}$ will in the case $S' = S$ be the initial object of K^{FI}.

Observation A.3 *For* $S' = \emptyset$, $K^{\mathrm{FRI}}_{S'} = K^{\mathrm{FI}}$.

Algebras in $K^{\mathrm{FRI}}_{S'}$ are of the form $R_{S'}(A|_{\Sigma^{export}})/E'$ where A is a model for E. The classifying model is $R_{S'}(T_{\Sigma}(X)/E|_{\Sigma^{export}})/E'$ (Theorem A.5 below). As we did for the FI case, viewing for the moment $R_{S'}(T_{\Sigma}(X)/E|_{\Sigma^{export}})$ as a "term-algebra" T', we directly get an abstract calculus for K^{FRI} by considering $\sim^{T'}_{E'}$ on T' and the classifying model T'/E'. The abstract calculus thus operates on elements of T', *i.e.*, congruence classes q of $R_{S'}(T_\Sigma(X)/E|_{\Sigma^{export}})$, and each q has the form $[t]_E$ for $t \in R_{S'}(T_\Sigma(X)|_{\Sigma^{export}})$. Again we obtain a term calculus by opening up the congruence classes and importing $E \vdash$. This calculus is defined over $R_{S'}(T_\Sigma(X)|_{\Sigma^{export}})$ and is given by the congruence on $R_{S'}(T_\Sigma(X)|_{\Sigma^{export}})$ induced by the set of Σ^{export}-equations

$$E^{\mathrm{FRI}} = (\sim_E^{T_\Sigma(X)}) \restriction_{R_{S'}(T_\Sigma(X)|_{\Sigma^{export}})} \;\cup\; E'$$

Remember now that as K^{FRI} consists of Σ^{export}-algebras, we are interested in satisfiability of Σ^{export} statements, *i.e.*, Σ^{export}-equations. However, depending on S' it may be the case that $T_{\Sigma^{export}}(X) \not\subseteq R_{S'}(T_\Sigma(X)|_{\Sigma^{export}})$, in which case the calculus will not respond to all Σ^{export}-equations. Hence, we supply an ω-rule dependent on S'.

The calculus $\vdash^{\mathrm{FRI}}_{S'}$ is given by the following single rule. For all $u, v \in T_{\Sigma^{export}}(X)$,

$$\omega_{S'} : \frac{\forall \tau : T_{\Sigma^{export}}(X) \to R_{S'}(T_\Sigma(X)|_{\Sigma^{export}})\ .\ \tau(u) \sim_{E^{\mathrm{FRI}}}^{R_{S'}(T_\Sigma(X)|_{\Sigma^{export}})} \tau(v)}{\vdash^{\mathrm{FRI}}_{S'}\ u = v}$$

To spell that out, let $\vdash^{\mathrm{FI}}_{S'}$ be the following calculus. For all $u, v \in R_{S'}(T_\Sigma(X)|_{\Sigma^{export}})$,

$$importE : \frac{E \vdash u = v}{\vdash^{\mathrm{FI}}_{S'}\ u = v}$$

$$induceE' : \frac{}{\vdash^{\mathrm{FI}}_{S'}\ \phi(c[l]) = \phi(c[r])};\quad \begin{array}{l}\langle l, r\rangle \in E', c \in T_{\Sigma^{export}}(X),\\ \phi : T_{\Sigma^{export}}(X) \to R_{S'}(T_\Sigma(X)|_{\Sigma^{export}})\end{array}$$

$$refl : \frac{}{\vdash^{\mathrm{FI}}_{S'}\ u = u} \qquad sym : \frac{\vdash^{\mathrm{FI}}_{S'}\ u = v}{\vdash^{\mathrm{FI}}_{S'}\ v = u} \qquad trans : \frac{\vdash^{\mathrm{FI}}_{S'}\ u = w, \vdash^{\mathrm{FI}}_{S'}\ w = v}{\vdash^{\mathrm{FI}}_{S'}\ u = v}$$

Now $\vdash^{\mathrm{FRI}}_{S'}$ is given by the following rule. For all $u, v \in T_{\Sigma^{export}}(X)$,

$$\omega_{S'} : \frac{\forall \tau : T_{\Sigma^{export}}(X) \to R_{S'}(T_\Sigma(X)|_{\Sigma^{export}})\ .\ \vdash^{\mathrm{FI}}_{S'}\ \tau(u) = \tau(v)}{\vdash^{\mathrm{FRI}}_{S'}\ u = v}$$

Observation A.4 *$\vdash^{\mathrm{FRI}}_{S'}$ subsumes $\vdash^{\mathrm{FI}}$: If $S' = \emptyset$ then the rule $\omega_{S'}$ adds nothing to $\vdash^{\mathrm{FI}}_{S'}$, so for $u, v \in T_{\Sigma^{export}}(X)$, $\vdash^{\mathrm{FRI}}_{\emptyset}\ u = v \;\Leftrightarrow\; \vdash^{\mathrm{FI}}_{\emptyset}\ u = v \;\Leftrightarrow\; \vdash^{\mathrm{FI}}\ u = v$.*

Theorem A.5 (Soundness and completeness) *Let $K^{\mathrm{FRI}}_{S'}$ be the semantics of (‡), namely $\{R_{S'}(A|_{\Sigma^{export}})/E' \mid A \in Mod_\Sigma(E)\}$. For any $u, v \in T_{\Sigma^{export}}(X)$,*

$$K^{\mathrm{FRI}}_{S'} \models u = v \;\Leftrightarrow\; R_{S'}(T_\Sigma(X)/E|_{\Sigma^{export}})/E' \models u = v \;\Leftrightarrow\; \vdash^{\mathrm{FRI}}_{S'}\ u = v$$

Proof: See (Hannay, 1998). □

A.2 Logic for Parametric Polymorphism

The inference rules for the logic for parametric polymorphism (Plotkin and Abadi, 1993) is natural deduction, intuitionistic style, extended over formulae involving relation symbols, together with the axiom schema of relational parametricity. We use the sequent notation $\Phi \vdash_\Gamma \phi$, where every term, type, and relation symbol in ϕ and in the formulae in Φ are well-formed given Γ. For negation, when we write $\neg\phi$, we really mean $\phi \;\Rightarrow\; \bot$.

$$\wedge\textit{-intro}: \frac{\Phi \vdash_\Gamma \phi \quad \Phi \vdash_\Gamma \varphi}{\Phi \vdash_\Gamma \phi \;\wedge\; \varphi} \qquad \wedge\textit{1-elim}: \frac{\Phi \vdash_\Gamma \phi \;\wedge\; \varphi}{\Phi \vdash_\Gamma \phi} \qquad \wedge\textit{2-elim}: \frac{\Phi \vdash_\Gamma \phi \;\wedge\; \varphi}{\Phi \vdash_\Gamma \varphi}$$

$$\Rightarrow\textit{-intro}: \frac{\Phi, \phi \vdash_\Gamma \varphi}{\Phi \vdash_\Gamma \phi \;\Rightarrow\; \varphi} \qquad \Rightarrow\textit{-elim}: \frac{\Phi \vdash_\Gamma \phi \;\Rightarrow\; \varphi \quad \Phi \vdash_\Gamma \phi}{\Phi \vdash_\Gamma \varphi}$$

$$\textit{ty-}\forall\textit{-intro}: \frac{\Phi \vdash_{\Gamma,X} \phi[X]}{\Phi \vdash_\Gamma \forall X.\phi[X]}, X \text{ not free in } \Phi \qquad \textit{ty-}\forall\textit{-elim}: \frac{\Phi \vdash_\Gamma \forall X.\phi[X] \quad \Gamma \rhd T}{\Phi \vdash_\Gamma \phi[T/X]}$$

$$\textit{te-}\forall\textit{-intro}: \frac{\Phi \vdash_{\Gamma,x:T} \phi[x]}{\Phi \vdash_\Gamma \forall x\!:\!T.\phi[x]}, x \text{ not free in } \Phi \qquad \textit{te-}\forall\textit{-elim}: \frac{\Phi \vdash_\Gamma \forall x\!:\!T.\phi[x] \quad \Gamma \rhd t\!:\!T}{\Phi \vdash_\Gamma \phi[t/x]}$$

$$\bot\textit{-elim}: \frac{\Phi \vdash_\Gamma \bot}{\Phi \vdash_\Gamma \phi}$$

$$\vee\textit{1-intro}: \frac{\Phi \vdash_\Gamma \phi}{\Phi \vdash_\Gamma \phi \vee \varphi} \qquad \vee\textit{2-intro}: \frac{\Phi \vdash_\Gamma \varphi}{\Phi \vdash_\Gamma \phi \vee \varphi}$$

$$\vee\textit{-elim}: \frac{\Phi \vdash_\Gamma \phi \vee \varphi \quad \Phi, \phi \vdash_\Gamma \psi \quad \Phi, \varphi \vdash_\Gamma \psi}{\Phi \vdash_\Gamma \psi}$$

$$\textit{ty-}\exists\textit{-intro}: \frac{\Phi \vdash_\Gamma \phi[T] \quad \Gamma \rhd T}{\Phi \vdash_\Gamma \exists X.\phi[X]}$$

$$\textit{ty-}\exists\textit{-elim}: \frac{\Phi \vdash_\Gamma \exists X.\phi[X] \quad \Phi, \phi[X] \vdash_{\Gamma,X} \varphi}{\Phi \vdash_\Gamma \varphi}, X \text{ not free in } \Phi$$

$$\textit{te-}\exists\textit{-intro}: \frac{\Phi \vdash_\Gamma \phi[t] \quad \Gamma \rhd t\!:\!T}{\Phi \vdash_\Gamma \exists x\!:\!T.\phi[x]}$$

$$\textit{te-}\exists\textit{-elim}: \frac{\Phi \vdash_\Gamma \exists x\!:\!T.\phi[x] \quad \Phi, \phi[x] \vdash_{\Gamma,x:T} \varphi}{\Phi \vdash_\Gamma \varphi}, x \text{ not free in } \Phi$$

—

$$re\text{-}\forall\text{-}intro: \ \frac{\Phi \vdash_{\Gamma, R\subset A\times B} \phi[R]}{\Phi \vdash_\Gamma \forall R \subset A \times B \ . \ \phi[R]}, R \text{ not free in } \Phi$$

$$re\text{-}\forall\text{-}elim: \ \frac{\Phi \vdash_\Gamma \forall R \subset A \times B . \phi[R] \quad \Gamma \rhd \rho \subset A \times B}{\Phi \vdash_\Gamma \phi[\rho]}$$

$$re\text{-}\exists\text{-}intro: \ \frac{\Phi \vdash_\Gamma \phi[\rho] \quad \Gamma \rhd \rho \subset A \times B}{\Phi \vdash_\Gamma \exists R . \phi[R]}$$

$$re\text{-}\exists\text{-}elim: \ \frac{\Phi \vdash_\Gamma \exists R . \phi[R] \quad \Phi, \phi[R] \vdash_{\Gamma, R\subset A\times B} \varphi}{\Phi \vdash_\Gamma \varphi}, R \text{ not free in } \Phi$$

—

$$refl: \quad \forall X . \forall x : X . x =_X x$$

$$\begin{aligned} subst: \quad & \forall X . \forall Y . \forall R \subset X \times Y . \forall x : X . \forall x' : X . \forall y : Y . \forall y' : Y \ . \\ & R(x, y) \ \wedge \ x =_X x' \ \wedge \ y =_Y y' \ \Rightarrow \ R(x', y') \end{aligned}$$

$$ty\text{-}cong: \quad (\forall X \ . \ t[X] =_Y u[X]) \ \Rightarrow \ (\Lambda X . t) =_{\forall X . Y} (\Lambda X . u)$$

$$te\text{-}cong: \quad (\forall x : X \ . \ t[x] =_Y u[x]) \ \Rightarrow \ (\lambda x : X . t) =_{X \to Y} (\lambda x : X . u)$$

$$ty\text{-}\beta eq: \quad \forall X . (\Lambda X . t) X = t$$

$$te\text{-}\beta eq: \quad \forall x : T . (\Lambda x : T . t) x = t$$

$$ty\text{-}\eta eq: \quad \forall f : \forall X . T (\Lambda X . f X) =_{\forall X . T} f$$

$$te\text{-}\eta eq: \quad \forall X . \forall Y . \forall x : X \to Y . (\lambda x : X . f x) =_{X \to Y} f$$

—

$$weak: \ \frac{\Phi \vdash_\Gamma \phi}{\Phi, \varphi \vdash_\Gamma \phi}$$

—

$$\textsc{Param}: \ \forall Y_1, \ldots, \forall Y_n \forall u : (\forall X . T[X, Y_1, \ldots, Y_n]) \ . \ u (\forall X . T[X, \mathsf{eq}_{Y_1}, \ldots, \mathsf{eq}_{Y_n}]) u$$

A.3 Excluded Middle

The constructive logic for parametric polymorphism in (Plotkin and Abadi, 1993) can be made classical by adding the rule of excluded middle. This is required for the correspondence in Sect. 4.6.

$$\textsc{Excluded Middle}: \ \Phi \vdash_\Gamma \phi \vee \neg\phi$$

A.4 Axiom of Choice

The logic can be augmented with the axiom of choice.

$$\mathrm{AC}: \forall X, Y\ .\ (\forall x\!:\!X.\exists y\!:\!Y\ .\ \phi[x,y]) \ \Rightarrow\ (\exists f\!:\!X\!\to\! Y.\forall x\!:\!X\ .\ \phi[x,(f x)])$$

AC does not hold in the parametric PER-model. The axiom of choice enters the discussion in Sect. 4.4 when importing the universal proof strategy for proving refinement into type theory, and in Sect. 6.2 when discussing the setoid semantics.

A.5 Universal Proof Strategy

In Sect. 4.4, for importing the universal proof strategy for proving refinement into type theory, it suffices for data types with first-order profiles to assert:

$$\begin{array}{l}
\textsc{Sub}: \ \forall X\,.\,\forall \mathfrak{x}\!:\!\mathfrak{T}[X,\boldsymbol{Z}]\,.\,\forall R\!\subset\! X\!\times\! X\ .\quad (\mathfrak{x}\ \mathfrak{T}[R,\mathbf{eq}_{\boldsymbol{Z}}]\ \mathfrak{x})\ \Rightarrow\\
\qquad \exists S\,.\,\exists \mathfrak{s}\!:\!\mathfrak{T}[S,\boldsymbol{Z}]\,.\,\exists R'\!\subset\! S\!\times\! S\,.\,\exists \mathsf{mono}\!:\!S\!\to\! X\ .\\
\qquad\qquad \forall s\!:\!S\ .\ s\ R'\ s \hfill \wedge\\
\qquad\qquad \forall s,s'\!:\!S\ .\ s\ R'\ s' \ \Leftrightarrow\ (\mathsf{mono}\, s)\ R\ (\mathsf{mono}\, s') \quad \wedge\\
\qquad\qquad \mathfrak{x}\ (\mathfrak{T}[(x\!:\!X, s\!:\!S).(x =_X (\mathsf{mono}\, s)), \mathbf{eq}_{\boldsymbol{Z}}])\ \mathfrak{s}
\end{array}$$

$$\begin{array}{l}
\textsc{Quot}: \ \forall X\,.\,\forall \mathfrak{x}\!:\!\mathfrak{T}[X,\boldsymbol{Z}]\,.\,\forall R\!\subset\! X\!\times\! X\ .\quad (\mathfrak{x}\ \mathfrak{T}[R,\mathbf{eq}_{\boldsymbol{Z}}]\ \mathfrak{x}\ \wedge\ \mathit{equiv}(R))\ \Rightarrow\\
\qquad \exists Q\,.\,\exists \mathfrak{q}\!:\!\mathfrak{T}[Q,\boldsymbol{Z}]\,.\,\exists \mathsf{epi}\!:\!X\!\to\! Q\ .\\
\qquad\qquad \forall x,y\!:\!X\ .\ xRy\ \Leftrightarrow\ (\mathsf{epi}\, x) =_Q (\mathsf{epi}\, y) \quad \wedge\\
\qquad\qquad \forall q\!:\!Q.\exists x\!:\!X\ .\ q =_Q (\mathsf{epi}\, x) \quad \wedge\\
\qquad\qquad \mathfrak{x}\ (\mathfrak{T}[(x\!:\!X, q\!:\!Q).((\mathsf{epi}\, x) =_Q q), \mathbf{eq}_{\boldsymbol{Z}}])\ \mathfrak{q}
\end{array}$$

These axiom schemata hold also for higher-order profiles if one relates to the setoid model, Sect. 6.2 and Sect. 6.3, but then SUB has to be modified to:

$$\begin{aligned}
\textsc{Sub}: \quad & \forall X\,.\,\forall \mathfrak{r}{:}\mathfrak{T}[X,\boldsymbol{Z}]\,.\,\forall R\subset X\times X\ .\ \ (\mathfrak{r}\ \mathfrak{T}[R,\mathbf{eq}_{\boldsymbol{Z}}]\ \mathfrak{r})\ \ \wedge\ \ (\mathfrak{r}\ \mathfrak{T}[P_R,\mathbf{eq}_{\boldsymbol{Z}}]\ \mathfrak{r})\ \Rightarrow \\
& \quad \exists S\,.\,\exists \mathfrak{s}{:}\mathfrak{T}[S,\boldsymbol{Z}]\,.\,\exists R'\subset S\times S\,.\,\exists \mathsf{mono}{:}S\to X\ . \\
& \qquad \forall s{:}S\ .\ s\ R'\ s \qquad\qquad \wedge \\
& \qquad \forall s,s'{:}S\ .\ s\ R'\ s'\ \Leftrightarrow\ (\mathsf{mono}\,s)\ R\ (\mathsf{mono}\,s') \quad \wedge \\
& \qquad \mathfrak{r}\ (\mathfrak{T}[(x{:}X,s{:}S)\ .\ (x=_X(\mathsf{mono}\,s)),\mathbf{eq}_{\boldsymbol{Z}}])\ \mathfrak{s}
\end{aligned}$$

The axiom schemata SUB and QUOT do not admit important examples with higher-order profiles. For data types with first-order or higher-order profiles we therefore assert:

$$\begin{aligned}
\textsc{SubG}: \quad & \forall X\,.\,\forall \mathfrak{r}{:}\mathfrak{T}[X,\boldsymbol{Z}]\,.\,\forall R\subset X\times X\ .\ \ (\mathfrak{r}\ \mathfrak{T}[R,\mathbf{eq}_{\boldsymbol{Z}}]_{\mathsf{C}}^{X,X,\mathfrak{r},\mathfrak{r}}\ \mathfrak{r})\ \Rightarrow \\
& \quad \exists S\,.\,\exists \mathfrak{s}{:}\mathfrak{T}[S,\boldsymbol{Z}]\,.\,\exists R'\subset S\times S\,.\,\exists\,\mathsf{mono}{:}S\to X\ . \\
& \qquad \forall s{:}S\ .\ s\ R'\ s \qquad\qquad \wedge \\
& \qquad \forall s,s'{:}S\ .\ s\ R'\ s'\ \Leftrightarrow\ (\mathsf{mono}\,s)\ R\ (\mathsf{mono}\,s') \quad \wedge \\
& \qquad \mathfrak{r}\ (\mathfrak{T}[(x{:}X,s{:}S).(x=_X(\mathsf{mono}\,s)),\mathbf{eq}_{\boldsymbol{Z}}]_{\mathsf{C}}^{\mathsf{abo}})\ \mathfrak{s}
\end{aligned}$$

$$\begin{aligned}
\textsc{QuotG}: \quad & \forall X\,.\,\forall \mathfrak{r}{:}\mathfrak{T}[X,\boldsymbol{Z}]\,.\,\forall R\subset X\times X\ .\ \ (\mathfrak{r}\ \mathfrak{T}[R,\mathbf{eq}_{\boldsymbol{Z}}]_{\mathsf{C}}^{X,X,\mathfrak{r},\mathfrak{r}}\ \mathfrak{r}\ \wedge\ \mathit{equiv}(R)) \\
& \quad \Rightarrow\ \exists Q\,.\,\exists \mathfrak{q}{:}\mathfrak{T}[Q,\boldsymbol{Z}]\,.\,\exists\,\mathsf{epi}{:}X\to Q\ . \\
& \qquad \forall x,y{:}X\ .\ x\ R\ y\ \Leftrightarrow\ (\mathsf{epi}\,x)=Q(\mathsf{epi}\,y) \quad \wedge \\
& \qquad \forall q{:}Q.\exists x{:}X\ .\ q=_Q(\mathsf{epi}\,x) \quad \wedge \\
& \qquad \mathfrak{r}\ (\mathfrak{T}[(x{:}X,q{:}Q).((\mathsf{epi}\,x)=_Q q),\mathbf{eq}_{\boldsymbol{Z}}]_{\mathsf{C}}^{\mathsf{abo}})\ \mathfrak{q}
\end{aligned}$$

These axiom schemata hold in the data type semantics of Sect. 6.4.

A.6 ω-Rule

The following infinitary ω-rule is never a part of the logic, but holds in the parametric minimal model due to (Hasegawa, 1991), mentioned in Sect. 3.4.2. The ω-rule is used in semantic arguments in the first sections of Ch. 5.

$$\omega\text{-RULE}: \ \frac{\Phi\vdash_\Gamma \phi[t],\quad \text{for all closed terms } \Gamma\rhd t{:}T}{\Phi\vdash_\Gamma \forall x{:}T\ .\ \phi[x]},\quad \text{T is inhabited}$$

Appendix B

Proofs

This Appendix contains proofs that may hold some interest, and that are omitted from the main discussion for clarity of discourse.

B.1 Simulation Relations in General

Proof of Theorem 3.4: $\Rightarrow$: Suppose $u\ (\exists X.T[X,\boldsymbol{\rho}])\ v$. Substitution on the result of Lemma B.2 using the result of Lemma B.1, gives the desired result.

$\Leftarrow$: Suppose $u\ \mathsf{SimRel}_{T[X,\boldsymbol{\rho}]}\ v$, *i.e.*,

$$\begin{array}{l}\exists A,B.\exists a{:}T[A,\boldsymbol{U}],b{:}T[B,\boldsymbol{V}].\\ \qquad\qquad\exists R\subset A\times B\ .\ u=(\mathsf{pack}Aa)\ \wedge\ v=(\mathsf{pack}Bb)\ \wedge\ a(T[R,\boldsymbol{\rho}])b\end{array}$$

By assumption we may substitute $(\mathsf{pack}Aa)$ and $(\mathsf{pack}Bb)$ for u and v, so it suffices to show that $(\mathsf{pack}Aa)\ (\exists X.T[X,\boldsymbol{\rho}])\ (\mathsf{pack}Bb)$, *i.e.,* that

$$\begin{array}{l}\forall Y.\forall Z.\forall S\subset Y\times Z.\\ \quad\forall f{:}\forall X.T[X,\boldsymbol{U}]\to Y.\ \forall g{:}\forall X.T[X,\boldsymbol{V}]\to Z\ .\\ \quad[\forall A'.\forall B'.\forall S'\subset A'\times B'\ .\ \forall a'{:}T[A',\boldsymbol{U}].\forall b'{:}T[B',\boldsymbol{V}].\\ \qquad\qquad\qquad a'\ (T[S',\boldsymbol{\rho}])\ b'\ \Rightarrow\ (f\,A'\,a')\ S\ (g\,B'\,b')\,]\\ \qquad\qquad\qquad\qquad\qquad\Rightarrow\ ((\mathsf{pack}Aa)\ Y\ f)\ \ S\ \ ((\mathsf{pack}Bb)\ Z\ g)\end{array}$$

The conclusion is β-equal to $(f\,A\,a)\ S\ (g\,B\,b)$. Instantiating the antecedent with A, B, a, b, and the R given by the assumption, gives exactly what we want. $\square$

The two following lemmata were reconstructed in all essence by Martin Hofmann.

Lemma B.1 *The following is derivable.*

$$\forall\boldsymbol{Z}.\forall u{:}\exists X.T[X,\boldsymbol{Z}]\ .\ u=_{\exists X.T[X,\boldsymbol{Z}]}(u\ \exists X.T[X,\boldsymbol{Z}]\ \mathsf{pack})$$

Proof: By the Identity Extension Lemma (Theorem 3.2) we must show

$$
\begin{array}{l}
\forall Y.\forall Z.\forall S\subset Y\times Z. \\
\quad \forall f{:}\forall X.T[X,\boldsymbol{Z}]\to Y.\ \forall g{:}\forall X.T[X,\boldsymbol{Z}]\to Z. \\
\quad\quad [\forall A'.\forall B'.\forall S'\subset A'\times B'\ .\ \forall a'{:}T[A',\boldsymbol{Z}].\forall b'{:}T[B',\boldsymbol{Z}]\ . \\
\quad\quad\quad\quad a'\ (T[S',\mathbf{eq}_{\boldsymbol{Z}}])\ b' \ \Rightarrow\ (f\,A'\,a')\ S\ (g\,B'\,b')\,] \\
\quad\quad\quad\quad\quad\quad \Rightarrow\ (u\,Y\,f)\ \ S\ \ ((u\ \exists X.T[X,\boldsymbol{Z}]\ \mathsf{pack})\ Z\ g)
\end{array}
$$

Let $\tilde{R} \stackrel{def}{=} (y:Y, v:\exists X.T[X,\boldsymbol{Z}])\ .\ (y\ S\ (v\,Z\,g))$. To get the conclusion it suffices to show $(u\,Y\,f)\ \ \tilde{R}\ \ (u\ \exists X.T[X,\boldsymbol{Z}]\ \mathsf{pack})$. This follows from the PARAM instance $u\ (\forall Y.\forall X.(T[X,\mathbf{eq}_{\boldsymbol{Z}}]\to Y)\to Y)\ u$, if $f\ (\forall X.(T[X,\mathbf{eq}_{\boldsymbol{Z}}]\to\tilde{R}))\ \mathsf{pack}$ is derivable. So we must derive

$$
\begin{array}{lll}
\multicolumn{3}{l}{\forall A'.\forall B'.\forall S\subset A'\times B'\ .\ \forall a'{:}T[A',\boldsymbol{Z}].\forall b'{:}T[B',\boldsymbol{Z}]\ .} \\
\quad a'\ (T[S',\mathbf{eq}_{\boldsymbol{Z}}])\ b' & \Rightarrow & (f\,A'\,a')\ \tilde{R}\ (\mathsf{pack}\,B'\,b') \\
 & & \quad\quad \shortparallel \quad \text{by def. of } \tilde{R} \\
 & & (f\,A'\,a')\ S\ ((\mathsf{pack}\,B'\,b')\ Z\ g) \\
 & & \quad\quad \shortparallel \quad \text{by } \beta \text{ red.} \\
 & & (f\,A'\,a')\ S\ (g\ B'\ b')
\end{array}
$$

which is exactly the antecedent above. □

Lemma B.2 *The following is derivable.*

$$
\begin{array}{l}
\forall u{:}\exists X.T[X,\boldsymbol{U}], v{:}\exists X.T[X,\boldsymbol{V}]\ . \\
\quad\quad u\ (\exists X.T[X,\boldsymbol{\rho}])\ v\ \ \Rightarrow\ \ (u\ \exists X.T[X,\boldsymbol{U}]\ \mathsf{pack})\ \hat{R}\ (v\ \exists X.T[X,\boldsymbol{V}]\ \mathsf{pack})
\end{array}
$$

where

$$
\begin{array}{l}
\hat{R} \stackrel{def}{=} (u{:}\exists X.T[X,\boldsymbol{U}], v{:}\exists X.T[X,\boldsymbol{V}])\ . \\
\quad\quad\quad\quad (\exists A,B.\exists a{:}T[A,\boldsymbol{U}], b{:}T[B,\boldsymbol{V}].\exists R\subset A\times B\ . \\
\quad\quad\quad\quad\quad\quad u = (\mathsf{pack}A a)\ \wedge\ v = (\mathsf{pack}B b)\ \wedge\ a\ (T[R,\boldsymbol{\rho}])\ b)
\end{array}
$$

Proof: The assumption $u\ (\exists X.T[X,\boldsymbol{\rho}])\ v$ says

$$\forall Y.\forall Z.\forall S\subset Y\times Z\ .\ uY\ \ (\forall X.(T[X,\boldsymbol{\rho}]\to S)\to S)\ vZ$$

which instantiates to $(u\ \exists X.T[X,\boldsymbol{U}])\ (\forall X.(T[X,\boldsymbol{\rho}]\to\hat{R})\to\hat{R})\ (v\ \exists X.T[X,\boldsymbol{V}])$. This in turns says

$$
\begin{array}{l}
\forall f{:}\forall X.(T[X,\boldsymbol{U}]\to\exists X.T[X,\boldsymbol{U}]).\forall g{:}\forall X.(T[X,\boldsymbol{V}]\to\exists X.T[X,\boldsymbol{V}])\ . \\
\quad f\ (\forall X.(T[X,\boldsymbol{\rho}]\to\hat{R}))\ g\ \ \Rightarrow\ \ (u\ \exists X.T[X,\boldsymbol{U}]\ f)\ \ \hat{R}\ \ (v\ \exists X.T[X,\boldsymbol{V}])\ g)
\end{array}
$$

which instantiates to

$$\mathsf{pack}_{T[X,\boldsymbol{U}]}\ (\forall X.(T[X,\boldsymbol{\rho}]\to\hat{R}))\ \mathsf{pack}_{T[X,\boldsymbol{V}]}\ \Rightarrow$$
$$(u\ \exists X.T[X,\boldsymbol{U}]\ \mathsf{pack}_{T[X,\boldsymbol{U}]})\ \ \hat{R}\ \ (v\ \exists X.T[X,\boldsymbol{V}]\ \mathsf{pack}_{T[X,\boldsymbol{V}]})$$

Thus we need only show that $\mathsf{pack}_{T[X,\boldsymbol{U}]}\ (\forall X.(T[X,\boldsymbol{\rho}]\to\hat{R}))\ \mathsf{pack}_{T[X,\boldsymbol{V}]}$ is derivable. To do this, we must derive

$$\forall A'.\forall B'.\forall S'\subset A'\times B'.\forall a'{:}T[A',\boldsymbol{U}].\forall b'{:}T[B',\boldsymbol{V}]\ .$$
$$a'\ (T[S',\boldsymbol{\rho}])\ b'\ \Rightarrow\ (\mathsf{pack}_{T[X,\boldsymbol{U}]}\ A'\ a')\ \ \hat{R}\ \ (\mathsf{pack}_{T[X,\boldsymbol{V}]}\ B'\ b')$$

But indeed this follows from the definition of $\hat{R}$. □

B.2 Simulation Relations at Higher Order

Proof of Lemma 5.12: We are to show, for $\mathfrak{T}[X,\boldsymbol{Z}]$ adhering to HADT_{Obs}, for $U[X,\boldsymbol{Z}]$ having no occurrences of universal types other than those in *Obs*, and whose only free variables are among X and $\boldsymbol{Z}$, for $f:\forall X.(\mathfrak{T}[X]\to U[X])$ whose only free variables are term variables of types in *In*, the derivability of

$$f\ (\forall X.\mathfrak{T}[X,\mathbf{eq}_{\boldsymbol{Z}}]^{\epsilon}\to U[X,\mathbf{eq}_{\boldsymbol{Z}}]^{\epsilon})\ f$$

Lemma B.5 below establishes

$$f\ (\forall X.\mathfrak{T}[X,\mathbf{eq}_{\boldsymbol{Z}}]^{f\epsilon}\to U[X,\mathbf{eq}_{\boldsymbol{Z}}]^{f\epsilon})\ f$$

Since $\mathfrak{T}[X,\boldsymbol{Z}]$ adheres to HADT_{Obs}, and since $U[X,\boldsymbol{Z}]$ has no occurrences of universal types other than those in *Obs*, we have, via Lemma B.4 and Lemma 5.10, for any $A,B,R\subset A\times B$, $\mathfrak{a}{:}\mathfrak{T}[A]$, $\mathfrak{b}{:}\mathfrak{T}[B]$, and $\mathsf{abo}=A,B,\mathfrak{a},\mathfrak{b}$, that by definition, $\mathfrak{T}[R,\mathbf{eq}_{\boldsymbol{Z}}]^{f\mathsf{abo}}=\mathfrak{T}[R,\mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}}$ and $U[R,\mathbf{eq}_{\boldsymbol{Z}}]^{f\mathsf{abo}}=U[R,\mathbf{eq}_{\boldsymbol{Z}}]^{\mathsf{abo}}$. Thus the result follows. □

Recall from the comments following Def. 5.8 (p. 120) that, according to *Abs-Bar* (p. 54), actual computations arising from a virtual computation can have applications of polymorphic terms of two kinds. If the instantiating type in the virtual computation does not contain the virtual data representation, then the instantiating type will appear identically in actual computations. On the other hand, if the instantiating type in the virtual computation does contain the virtual data representation, then the instantiating type in each actual computation will differ according to the actual data representations. To capture this in the present type theory and logic, we can for the second case, relate to a given virtual computation and make a finite conjunction over all types involving the virtual data representation that actually occur in the given virtual computation. This is what we do in Def. B.3 below, where an *abo*-relation is defined w.r.t. a given computation f. Although this is not appropriate for the top-level discussion, this is suitable for the purpose of proving Lemma 5.12.

We point out that it is essential that we do capture type application as described by ***Abs-Bar***. It is not adequate to handle universal types by writing

$$
\begin{aligned}
U = \forall Y.U'[\boldsymbol{Y}, Y, X] \quad &: \quad U[\boldsymbol{\rho}, R]^{\mathsf{abo}} \stackrel{\textit{def}}{=} \\
&\forall(E_{k+1}, F_{k+1}, \rho_{k+1} \subset E_{k+1} \times F_{k+1})(U'[\boldsymbol{\rho}, \rho_{k+1}, R]^{\mathsf{abo}, E_{k+1}, F_{k+1}}) \\
&\qquad \text{where } \mathsf{abo}, E_{k+1}, F_{k+1} \stackrel{\textit{def}}{=} \boldsymbol{E}, E_{k+1}, \boldsymbol{F}, F_{k+1}, A, B, \mathfrak{a}, \mathfrak{b}
\end{aligned}
$$

in Def 5.8.

Definition B.3 (*abo*-Relation) *Relative to* $\mathfrak{T}[X]$, *for* $\boldsymbol{Y}$, A, B, $R \subset A \times B$, $\mathfrak{a} : \mathfrak{T}[A]$, $\mathfrak{b} : \mathfrak{T}[B]$, *let* $\boldsymbol{\rho} \subset \boldsymbol{E} \times \boldsymbol{F}$ *be s.t.* $\rho_i = \mathsf{eq}_{Y_j} \subset Y_j \times Y_j$ *or* $\rho_i = R \subset A \times B$. *Given* $f : \forall X.(\mathfrak{T}[X] \to U[X])$, *whose only free variables are term variables of types in In, define the abo*-relation $U[\boldsymbol{\rho}, R]^{f\mathsf{abo}} \subset U[\boldsymbol{E}, A] \times U[\boldsymbol{F}, B]$ relative to f, *for the list* $\mathsf{abo} = A, B, \mathfrak{a}, \mathfrak{b}$, *inductively on* $U[\boldsymbol{Y}, X]$ *by*

$$
\begin{aligned}
&U = X &&: \quad U[\boldsymbol{\rho}, R]^{f\mathsf{abo}} \stackrel{\textit{def}}{=} R \\
&U = Y_i &&: \quad U[\boldsymbol{\rho}, R]^{f\mathsf{abo}} \stackrel{\textit{def}}{=} \rho_i \\
&U = \forall Y.U'[\boldsymbol{Y}, Y, X] &&: \quad U[\boldsymbol{\rho}, R]^{f\mathsf{abo}} \stackrel{\textit{def}}{=} (g : \forall Y.U'[\boldsymbol{E}, Y, A],\ h : \forall Y.U'[\boldsymbol{F}, Y, B]) \ . \\
& && \quad (\ (\forall Y.gY \ \ (U'[\boldsymbol{\rho}, \mathsf{eq}_Y, R]^{f\mathsf{abo}}) \ hY) \ \wedge \\
& && \qquad \textstyle\bigwedge_{D[\boldsymbol{Y},X]} (gD[\boldsymbol{E}, A] \ \ (U'[\boldsymbol{\rho}, D[\boldsymbol{\rho}, R], R]^{f\mathsf{abo}}) \ \ hD[\boldsymbol{F}, B])\ ) \\
& && \qquad \textit{where the conjunction ranges over all types } D[\boldsymbol{Y}, X] \textit{ containing } X\textit{, that instantiate polymorphic terms in } f. \\
&U = U' \to U'' &&: \quad U[\boldsymbol{\rho}, R]^{f\mathsf{abo}} \stackrel{\textit{def}}{=} \\
&\quad (g : U'[\boldsymbol{E}, A] \to U''[\boldsymbol{E}, A],\ h : U'[\boldsymbol{F}, B] \to U''[\boldsymbol{F}, B]) \ . \\
&\qquad (\forall x : U'[\boldsymbol{E}, A], \forall y : U'[\boldsymbol{F}, B] \ . \\
&\qquad\qquad (x \ U'[\boldsymbol{\rho}, R]^{f\mathsf{abo}} \ y \ \ \wedge \ \ \mathsf{Dfnbl}^{f\mathsf{abo}}_{U'[\boldsymbol{Y},X]}(x, y)) \ \Rightarrow \ \ (gx) \ U''[\boldsymbol{\rho}, R]^{f\mathsf{abo}} \ (hy))
\end{aligned}
$$

where,

$$
\mathsf{Dfnbl}^{f\mathsf{abo}}_{U'[\boldsymbol{Y},X]}(x, y) \stackrel{\textit{def}}{=} \exists f_{U'} : \forall X.(\mathfrak{T}[X] \to U'[\boldsymbol{I}, X]) \ . \ (f_{U'} A \, \mathfrak{a}) = x \ \wedge \ (f_{U'} B \, \mathfrak{b}) = y
$$

where I_i *is* Y_j *or* X *as appropriate. To avoid clutter, we omitted parameters* $\boldsymbol{Z}$ *of* $\mathfrak{T}[X, \boldsymbol{Z}]$ *in the above presentation. Of course,* ${\mathbf{eq}_{\boldsymbol{Z}}}^{f\mathsf{abo}} \stackrel{\textit{def}}{=} \mathbf{eq}_{\boldsymbol{Z}}$.

We usually omit the type subscript on the $\mathsf{Dfnbl}^{f\mathsf{abo}}$ clause.

Lemma B.4 *We can derive the following in the logic.*

$$\forall g, h : D \;.\; g =_D h \;\Leftrightarrow\; g(D^{f\mathsf{abo}})h, \qquad \textit{for } D \in \mathit{Obs}$$

Proof: We illustrate with the inductive type Nat. So consider $g(\mathsf{Nat}^{f\mathsf{abo}})h$, *i.e.,*

$$\begin{array}{l}
(\forall Y.\forall y, y' : Y.\forall s, s' : Y \to Y \;. \\
\qquad y =_Y y' \;\wedge\; \mathsf{Dfnbl}^{f\mathsf{abo}}(y, y') \;\wedge\; s(\mathsf{eq}_Y \to \mathsf{eq}_Y)^{f\mathsf{abo}} s' \;\wedge\; \mathsf{Dfnbl}^{f\mathsf{abo}}(s, s') \\
\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow\; (gYys) =_Y (hYy's')) \;\wedge \\
\bigwedge_{D[\boldsymbol{Y}, X]} (\forall a : D[\boldsymbol{E}, A], b : D[\boldsymbol{F}, B].\forall s : D[\boldsymbol{E}, A] \to D[\boldsymbol{E}, A], s' : D[\boldsymbol{F}, B] \to D[\boldsymbol{F}, B] \;. \\
\quad a\; D[\boldsymbol{\rho}, R]\; b \;\wedge\; \mathsf{Dfnbl}^{f\mathsf{abo}}(a, b) \;\wedge\; s(D[\boldsymbol{\rho}, R] \to D[\boldsymbol{\rho}, R])^{f\mathsf{abo}} s' \;\wedge\; \mathsf{Dfnbl}^{f\mathsf{abo}}(s, s') \\
\qquad\qquad\qquad\qquad\qquad \Rightarrow\; (gD[\boldsymbol{E}, A]as)\; D[\boldsymbol{\rho}, R]\; (hD[\boldsymbol{F}, B]bs')))
\end{array}$$

The clause $\mathsf{Dfnbl}^{f\mathsf{abo}}(y, y')$ says

$$\exists f : \forall X.(\mathfrak{T}[X] \to Y) \;.\; (f A\, \mathfrak{a}) = y \;\wedge\; (f B\, \mathfrak{b}) = y'$$

Since we have $y =_Y y'$, this is derivable by exhibiting $\Lambda X.\lambda\mathfrak{x} : \mathfrak{T}[X].y$. Similarly, we derive $\mathsf{Dfnbl}^{f\mathsf{abo}}(s, s')$ by exhibiting $\Lambda X.\lambda\mathfrak{x} : \mathfrak{T}[X].s : \forall X.(\mathfrak{T}[X] \to (Y \to Y))$. This suffices because $s(\mathsf{eq}_Y \to \mathsf{eq}_Y)^{f\mathsf{abo}} s'$ is equivalent to $s(\mathsf{eq}_Y \to \mathsf{eq}_Y)s'$, which by the Identity Extension Lemma gives $s =_{Y \to Y} s'$. Thus, these $\mathsf{Dfnbl}^{f\mathsf{abo}}$ clauses are vacuous. So this part of the definition of $g(\mathsf{Nat}^{f\mathsf{abo}})h$ is equivalent to

$$\forall Y.\forall y, y' : Y.\forall s, s' : Y \to Y \;.\; y =_Y y' \wedge s(\mathsf{eq}_Y \to \mathsf{eq}_Y)s' \wedge \quad \Rightarrow\; (gYys) =_Y (hYy's')$$

or $\forall Y \;.\; gY(\mathsf{eq}_Y \to (\mathsf{eq}_Y \to \mathsf{eq}_Y) \to \mathsf{eq}_Y)hY$, *i.e.,* $\forall Y \;.\; gY = hY$. By the congruence axiom schema, we get $\Lambda Y.gY = \Lambda Y.hY$, which by η-equality yields $g =_{\mathsf{Nat}} h$.

Furthermore, let $\hat{R} \stackrel{def}{=} (a : A, b : B) \;.\; (a \;R\; b \;\wedge\; \mathsf{Dfnbl}^{f\mathsf{abo}}(a, b))$. We have that $s(R \to R)^{f\mathsf{abo}} s' \;\wedge\; \mathsf{Dfnbl}^{f\mathsf{abo}}(s, s') \;\Rightarrow\; s(\hat{R} \to \hat{R})s'$. This is easily seen; $s(R \to R)^{f\mathsf{abo}} s'$ says $\forall a : A, b : B \;.\; a \;R\; b \;\wedge\; \mathsf{Dfnbl}^{f\mathsf{abo}}(a, b) \;\Rightarrow\; sa \;R\; s'b$, *i.e.,* $\forall a : A, b : B \;.\; a \;\hat{R}\; b \;\Rightarrow\; sa \;R\; s'b$. Since $\mathsf{Dfnbl}^{f\mathsf{abo}}(a, b)$ and $\mathsf{Dfnbl}^{f\mathsf{abo}}(s, s')$ give $\mathsf{Dfnbl}^{f\mathsf{abo}}(sa, s'b)$, we get $s(\hat{R} \to \hat{R})s'$. This gives

$$\begin{array}{l}
(\forall a : A, b : B.\forall s : A \to A, s' : B \to B \;. \\
\qquad\qquad\qquad a\; \hat{R}\; b \;\wedge\; s(\hat{R} \to \hat{R})s' \;\Rightarrow\; (gAas)\; \hat{R}\; (hBbs'))) \\
\quad \Rightarrow\; (\forall a : A, b : B.\forall s : A \to A, s' : B \to B \;. \\
\qquad\quad a\; R\; b \;\wedge\; \mathsf{Dfnbl}^{f\mathsf{abo}}(a, b) \;\wedge\; s(R \to R)^{f\mathsf{abo}} s' \;\wedge\; \mathsf{Dfnbl}^{f\mathsf{abo}}(s, s') \\
\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow\; (gAas)\; R\; (hBbs')))
\end{array}$$

i.e.,

$$gA\; (\hat{R} \to (\hat{R} \to \hat{R}) \to \hat{R})\; hB \;\Rightarrow\; gA\; (R \to (R \to R) \to R)^{f\mathsf{abo}}\; hB$$

This argument generalises to any $D[\boldsymbol{\rho}, R] \subset D[\boldsymbol{E}, A] \times D[\boldsymbol{F}, B]$.

From $g =_{\mathsf{Nat}} h$ we then get $g(\mathsf{Nat}^{f\mathsf{abo}})h$ by instantiating

$$\forall E, F, \rho \subset E \times F \ . \ gE \ (\rho \to (\rho \to \rho) \to \rho) \ hF$$

first by Y and eq_Y, and then by $D[\boldsymbol{E}, A]$, $D[\boldsymbol{F}, B]$ and suitable $\hat{R}$. $\square$

Due to induction purposes, Lemma B.5 below is quite general. Note however that we cannot transfer the full generality to the axiom schema spParam, because some instances of $\forall X.(\mathfrak{T}[X] \to U[\boldsymbol{I}, X])$ are not inhabited by desired terms, in particular, there is no $f : \forall X.(\mathfrak{T}[X] \to Y)$ for free Y different from any parameter Z of $\exists X.\mathfrak{T}[X]$, whose only free variables are term variables of types in *In*. This is directly relevant to the discussion in Sect. 5.4.1 (p. 140), where this excludes an erroneous instance that would otherwise yield inconsistency.

Lemma B.5 *For any term $f : \forall X.(\mathfrak{T}[X] \to U[\boldsymbol{I}, X])$, where I_i is Y_j or X, and where the free variables of f are term variables of types in In, we can derive*

$$f \ (\forall X.\mathfrak{T}[X, \mathbf{eq}_{\boldsymbol{Z}}]^{f\epsilon} \to U[\boldsymbol{\rho}, X, \mathbf{eq}_{\boldsymbol{Z}}]^{f\epsilon}) \ f$$

where ρ_i is eq_{Y_j} or X.

Proof: Unravelling the definitions, this becomes

$$\forall A, B, R \subset A \times B \ . \ \forall \mathfrak{a} : \mathfrak{T}[A, \boldsymbol{Z}], \mathfrak{b} : \mathfrak{T}[B, \boldsymbol{Z}] \ . \\ \mathfrak{a} \ \mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}]^{f\mathsf{abo}} \ \mathfrak{b} \ \Rightarrow \ (f A \, \mathfrak{a}) \ U[\boldsymbol{\rho}, R, \mathbf{eq}_{\boldsymbol{Z}}]^{f\mathsf{abo}} \ (f B \, \mathfrak{b})$$

where ρ_i is eq_{Y_j} or R. By assumption, f has the form $\Lambda X.\lambda \mathfrak{x} : \mathfrak{T}[X, \boldsymbol{Z}].t$, where the free term variables of t are among $\mathfrak{x} : \mathfrak{T}[X, \boldsymbol{Z}]$ and variables of types in *In*.

We will prove this lemma by induction on the structure of t. This is however complicated by the fact that the $\mathsf{Dfnbl}^{f\mathsf{abo}}$ clause is not stable over λ-abstraction. For example, for $t \stackrel{def}{=} \lambda x : X.x$, where X is the quantified type variable in f the corresponding $\mathsf{Dfnbl}^{f\mathsf{abo}}$ clause is indeed fulfilled by $f_t \stackrel{def}{=} \Lambda X.\lambda \mathfrak{x} : \mathfrak{T}[X].\lambda x : X.x$, but this cannot be asserted based on an induction hypothesis on subterm x, because x as a free variable of bound type X cannot occur in any $\mathsf{Dfnbl}^{f\mathsf{abo}}$ clause, *i.e.*, $f_x \stackrel{def}{=} \Lambda X.\lambda \mathfrak{x} : \mathfrak{T}[X, \boldsymbol{Z}].x$ is ill-typed in this context.

We will simplify matters via a combinatory approach, since this eliminates λ-abstraction, and it is easy to assert $\mathsf{Dfnbl}^{f\mathsf{abo}}$ clauses for the combinators. First, for any types U, V, W, let

$$K_{U,V} \stackrel{def}{=} \lambda x : U.\lambda y : V.x,$$

$$S_{U,V,W} \stackrel{def}{=} \lambda x : U \to V \to W.\lambda y : U \to V.\lambda z : U.(xz)(yz).$$

Any *simply-typed* term $\lambda x{:}T.t$ is $\beta\eta$-equal to a term with SK combinators without any occurrence of λ (other than those in the combinators). However, our terms involve type abstraction as well. We do not have to eliminate Λ-abstraction as we must λ-abstraction. But we must account for second-order terms when eliminating λ-abstraction. For this we use the following (non-algebraic) combinators. (Algebraic combinators can be found in (Bruce et al., 1990).)

$$P_{U,V} \stackrel{def}{=} \lambda f{:}\forall X.U \to V.\lambda x{:}U\ .\ \Lambda Y.(fY)x, \qquad X \textit{ not free in } U.$$
$$Q_{D,U,V} \stackrel{def}{=} \lambda f{:}U \to \forall X.V.\lambda x{:}U\ .\ (fx)D$$

Then, any System F term $\lambda x{:}T.t$ is $\beta\eta$-equal to a term with $SKPQ$ combinators without any occurrence of λ (other than those in the combinators). We indicate the translation cl_F of System F terms into $SKPQ$-combinator terms. The parts marked with $(*)$ comprise the extension to the otherwise standard (e.g. (Mitchell, 1996)) translation of simply typed terms into SK-combinator terms.

$$cl_F(x) = x, \quad \text{for } x \text{ a variable,}$$
$$cl_F(\lambda x{:}U.t) = \langle x{:}U\rangle\, cl_F(t),$$
$$cl_F(ut) = cl_F(u)\, cl_F(t),$$
$$cl_F(\Lambda X.t) = \Lambda X.cl_F(t), \qquad (*)$$
$$cl_F(tD) = cl_F(t)D. \qquad (*)$$

where for any combinatory term t, $\langle x{:}U\rangle t$ is defined by

$$\langle x{:}U\rangle x = SKK,$$
$$\langle x{:}U\rangle y = Ky, \quad \text{for } x \neq y,$$
$$\langle x{:}U\rangle c = Kc, \quad \text{for } c \text{ a combinator,}$$
$$\langle x{:}U\rangle ut = S(\langle x{:}U\rangle u)(\langle x{:}U\rangle t),$$
$$\langle x{:}U\rangle \Lambda X.t = P(\Lambda X.\langle x{:}U\rangle t), \qquad (*)$$
$$\langle x{:}U\rangle tD = Q_D(\langle x{:}U\rangle t), \qquad (*)$$

For example, $cl_F(\lambda z{:}U.\Lambda X.\lambda x{:}X.z) = P(\Lambda X.S(KK)(SKK))$, and then internally applying type D, $cl_F(\lambda z{:}U.(\Lambda X.\lambda x{:}X.z)D) = Q_D(P(\Lambda X.S(KK)(SKK)))$. One needs to verify firstly that cl_F maps any System F term to some combinator term without λ-abstraction, and secondly for any System F term t, that $cl_F(t)$ is $\beta\eta$-equal to t. This is straightforward.

—

We can thus do induction on the structure of t, omitting λ-abstraction, but considering four more base cases corresponding to the four combinators. So, assuming $\mathfrak{a}\ \mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}]^{f\mathsf{abo}}\ \mathfrak{b}$, we must derive $(f A\, \mathfrak{a})\ U[\boldsymbol{\rho}, R, \mathbf{eq}_{\boldsymbol{Z}}]^{f\mathsf{abo}}\ (f B\, \mathfrak{b})$:

t is a variable x: Well-typing means either (1) x is $\mathfrak{x}$, or (2) x is of a type D in *In*. For (1) we must prove the derivability of $\mathfrak{a}\ \mathfrak{T}[R, \mathbf{eq}_{\boldsymbol{Z}}]^{f\mathsf{abo}}\ \mathfrak{b}$. But this is simply the antecedent. For (2), if U is some Z_i we must show $x =_{Z_i} x$ which is immediate, and if U is some closed $D \in In$, PARAM gives $x(D)x$ and then Lemma 5.10 gives $x(D^{f\mathsf{abo}})x$.

For the remainder of the proof we omit $\boldsymbol{Z}$ and $\mathbf{eq}_{\boldsymbol{Z}}$ to avoid excessive clutter.

t is $K_{U,V}$: We must demonstrate the derivability of

$$K_{U[\boldsymbol{E},A],V[\boldsymbol{E},A]}(U[\boldsymbol{\rho}, R] \to (V[\boldsymbol{\rho}, R] \to U[\boldsymbol{\rho}, R]))^{f\mathsf{abo}} K_{U[\boldsymbol{F},B],V[\boldsymbol{F},B]}$$

which is trivial, since this expands to

$$\begin{aligned}
&\forall a : U[\boldsymbol{E}, A], a' : V[\boldsymbol{E}, A], b : U[\boldsymbol{F}, B], b' : V[\boldsymbol{F}, B]\ .\\
&\qquad a(U[\boldsymbol{\rho}, R]^{f\mathsf{abo}})b\ \wedge\ \mathsf{Dfnbl}^{f\mathsf{abo}}(a, b)\ \wedge\\
&\qquad\qquad a'(V[\boldsymbol{\rho}, R]^{f\mathsf{abo}})b'\ \wedge\ \mathsf{Dfnbl}^{f\mathsf{abo}}(a', b')\\
&\qquad\qquad\qquad \Rightarrow\ a(U[\boldsymbol{\rho}, R]^{f\mathsf{abo}})b
\end{aligned}$$

t is $S_{U,V,W}$: We must show the derivability of

$$\begin{aligned}
&\forall a : U[\boldsymbol{E}, A] \to V[\boldsymbol{E}, A] \to W[\boldsymbol{E}, A], a' : U[\boldsymbol{E}, A] \to V[\boldsymbol{E}, A], a'' : U[\boldsymbol{E}, A],\\
&\forall b : U[\boldsymbol{F}, B] \to V[\boldsymbol{F}, B] \to W[\boldsymbol{F}, B], b' : U[\boldsymbol{F}, B] \to V[\boldsymbol{F}, B], b'' : U[\boldsymbol{F}, B]\ .\\
&\quad [\, a\ (U[\boldsymbol{\rho}, R] \to V[\boldsymbol{\rho}, R] \to W[\boldsymbol{\rho}, R])^{f\mathsf{abo}}\ b\ \wedge\ \mathsf{Dfnbl}^{f\mathsf{abo}}(a, b)\ \wedge\\
&\quad a'(U[\boldsymbol{\rho}, R] \to V[\boldsymbol{\rho}, R])^{f\mathsf{abo}} b'\ \wedge\ \mathsf{Dfnbl}^{f\mathsf{abo}}(a', b')\ \wedge\\
&\quad a''(U[\boldsymbol{\rho}, R]^{f\mathsf{abo}})b''\ \wedge\ \mathsf{Dfnbl}^{f\mathsf{abo}}(a'', b'')]\ \Rightarrow\ (aa'')(a'a'')(W[\boldsymbol{\rho}, R]^{f\mathsf{abo}})(bb'')(b'b'')
\end{aligned}$$

First, $a''(U[\boldsymbol{\rho}, R]^{f\mathsf{abo}})b'' \wedge \mathsf{Dfnbl}^{f\mathsf{abo}}(a'', b'')$ gives $(a'a'')\ (V[\boldsymbol{\rho}, R]^{f\mathsf{abo}})\ (b'b'')$. Secondly, $\mathsf{Dfnbl}^{f\mathsf{abo}}(a'', b'')$ and $\mathsf{Dfnbl}^{f\mathsf{abo}}(a', b')$ give $\mathsf{Dfnbl}^{f\mathsf{abo}}((a'a''), (b'b''))$. The combination then gives $(aa'')(a'a'')\ (W[\boldsymbol{\rho}, R]^{f\mathsf{abo}})\ (bb'')(b'b'')$.

t is $P_{U,V}$: We must show the derivability of

$$\begin{aligned}
&\forall f : \forall X.(U[\boldsymbol{E}, A] \to V[\boldsymbol{E}, X, A]), a : U[\boldsymbol{E}, A],\\
&\forall g : \forall X.(U[\boldsymbol{F}, B] \to V[\boldsymbol{F}, X, B]), b : U[\boldsymbol{F}, B]\ .\\
&\quad [\, f\ (\forall X.(U[\boldsymbol{\rho}, R] \to V[\boldsymbol{\rho}, X, R]))^{f\mathsf{abo}}\ g\ \wedge\ \mathsf{Dfnbl}^{f\mathsf{abo}}(f, g)\ \wedge\\
&\quad a(U[\boldsymbol{\rho}, R]^{f\mathsf{abo}})b\ \wedge\ \mathsf{Dfnbl}^{f\mathsf{abo}}(a, b)]\ \Rightarrow\ \Lambda Y.(fY)a\ (\forall Y.V[\boldsymbol{\rho}, Y, R])^{f\mathsf{abo}}\ \Lambda Y.(fY)b
\end{aligned}$$

The goal is thus

$$\begin{aligned}
&\forall Y\ .\ (\Lambda Y.(fY)a)Y\ \ (V[\boldsymbol{\rho}, \mathsf{eq}_Y, R]^{f\mathsf{abo}})\ (\Lambda Y.(gY)b)Y\ \ \wedge\\
&\quad \textstyle\bigwedge_{D[\boldsymbol{Y},X]}(\Lambda Y.(fY)a)D[\boldsymbol{E}, A]\ \ (V[\boldsymbol{\rho}, D[\boldsymbol{\rho}, R], R]^{f\mathsf{abo}})\ \ (\Lambda Y.(gY)b)D[\boldsymbol{F}, B]
\end{aligned}$$

First, $f\ (\forall X.(U[\boldsymbol{\rho}, R] \to V[\boldsymbol{\rho}, X, R]))^{f\mathsf{abo}}\ g$ gives

$$\begin{aligned}&\forall Y\ .\ fY\ (U[\boldsymbol{\rho}, R] \to V[\boldsymbol{\rho}, \mathsf{eq}_Y, R])^{f\mathsf{abo}}\ gY\ \ \wedge\\ &\qquad\qquad \textstyle\bigwedge_{D[\boldsymbol{Y},X]} fD[\boldsymbol{E}, A]\ (U[\boldsymbol{\rho}, R] \to V[\boldsymbol{\rho}, D[\boldsymbol{\rho}, R], R])^{f\mathsf{abo}}\ gD[\boldsymbol{F}, B]\end{aligned}$$

And then $a\ (U[\boldsymbol{\rho}, R]^{f\mathsf{abo}})\ b\ \wedge\ \mathsf{Dfnbl}^{f\mathsf{abo}}(a, b)$ gives

$$\begin{aligned}&(\forall Y\ .\ fYa\ (V[\boldsymbol{\rho}, \mathsf{eq}_Y, R])^{f\mathsf{abo}}\ gYb)\ \ \wedge\\ &\qquad\qquad\qquad \textstyle\bigwedge_{D[\boldsymbol{Y},X]} fD[\boldsymbol{E}, A]a\ (V[\boldsymbol{\rho}, D[\boldsymbol{\rho}, R], R])^{f\mathsf{abo}}\ gD[\boldsymbol{F}, B]b\end{aligned}$$

which is β-equal to the goal.

t is $Q_{D,U,V}$: We must show the derivability of

$$\begin{aligned}&\forall f{:}U[\boldsymbol{E}, A] \to \forall X.V[\boldsymbol{E}, X, A], a{:}U[\boldsymbol{E}, A],\\ &\forall g{:}U[\boldsymbol{F}, B] \to \forall X.V[\boldsymbol{F}, X, B], b{:}U[\boldsymbol{F}, B]\ .\\ &\ [\,f\ (U[\boldsymbol{\rho}, R] \to (\forall X.V[\boldsymbol{\rho}, X, R]))^{f\mathsf{abo}}\ g\ \wedge\ \mathsf{Dfnbl}^{f\mathsf{abo}}(f, g)\ \wedge\\ &\qquad\qquad\qquad\qquad a(U[\boldsymbol{\rho}, R]^{f\mathsf{abo}})b\ \wedge\ \mathsf{Dfnbl}^{f\mathsf{abo}}(a, b)]\\ &\qquad\qquad\qquad\qquad\quad \Rightarrow\ (fa)D[\boldsymbol{E}, A]\ (V[\boldsymbol{\rho}, D[\boldsymbol{\rho}, R], R])^{f\mathsf{abo}}\ (fb)D[\boldsymbol{F}, B]\end{aligned}$$

From $a(U[\boldsymbol{\rho}, R]^{f\mathsf{abo}})b\ \wedge\ \mathsf{Dfnbl}^{f\mathsf{abo}}(a, b)$ we get $fa\ (\forall X.V[\boldsymbol{\rho}, X, R])^{f\mathsf{abo}}\ gb$, *i.e.,*

$$\begin{aligned}&(\forall Y\ .\ (fa)Y\ (V[\boldsymbol{\rho}, \mathsf{eq}_Y, R]^{f\mathsf{abo}})(fb)Y)\ \ \wedge\\ &\qquad\qquad\qquad \textstyle\bigwedge_{D[\boldsymbol{Y},X]}(fa)D[\boldsymbol{E}, A]\ (V[\boldsymbol{\rho}, D[\boldsymbol{\rho}, R], R]^{f\mathsf{abo}})\ (fb)D[\boldsymbol{F}, B]\end{aligned}$$

Instantiated with $D[\boldsymbol{E}, A]$, $D[\boldsymbol{F}, B]$, and $D[\boldsymbol{\rho}, R]$, this gives the goal.

t is (ut'): We must prove derivability of $(ut')[A, \mathfrak{a}](U[\boldsymbol{\rho}, R]^{f\mathsf{abo}})(ut')[B, \mathfrak{b}]$, where for some type U', we have $u : U' \to U$ and $t' : U'$. From t we get the terms $f_u \stackrel{def}{=} \Lambda X.\lambda\mathfrak{x}.u$ and $f_{t'} \stackrel{def}{=} \Lambda X.\lambda\mathfrak{x}.t'$, such that $f_u A\mathfrak{a} = u[A, \mathfrak{a}]$ and $f_u B\mathfrak{b} = u[B, \mathfrak{b}]$, and $f_{t'} A\mathfrak{a} = t'[A, \mathfrak{a}]$ and $f_{t'} B\mathfrak{b} = t'[A, \mathfrak{a}]$. Then by induction hypothesis, we can derive $u[A, \mathfrak{a}]\ (U'[\boldsymbol{\rho}, R] \to U[\boldsymbol{\rho}, R])^{f\mathsf{abo}}\ u[B, \mathfrak{b}]$ and $t'[A, \mathfrak{a}]\ (U'[\boldsymbol{\rho}, R]^{f\mathsf{abo}})\ t'[B, \mathfrak{b}]$. Using $f_{t'}$ to derive the necessary $\mathsf{Dfnbl}^{f\mathsf{abo}}(t'[A, \mathfrak{a}], t'[B, \mathfrak{b}])$ clause, we then get $u[A, \mathfrak{a}]t'[A, \mathfrak{a}]\ (U[\boldsymbol{\rho}, R]^{f\mathsf{abo}})\ u[B, \mathfrak{b}]t'[B, \mathfrak{b}]$.

t is $\Lambda Y.t'$: Given $\Lambda Y.t' : \forall Y.U'[\boldsymbol{Y}, Y, X]$, we must show the derivability of $\Lambda Y.t'[\boldsymbol{E}, Y, A, \mathfrak{a}]\ (\forall Y.U'[\boldsymbol{\rho}, Y, R])^{f\mathsf{abo}}\ \Lambda Y.t'[\boldsymbol{F}, Y, B, \mathfrak{b}]$, which expands to

$$\begin{aligned}&(\forall Y\ .\ t'[\boldsymbol{E}, Y, A, \mathfrak{a}]\ \ (U[\boldsymbol{\rho}, \mathsf{eq}_Y, R]^{f\mathsf{abo}})\ \ t'[\boldsymbol{F}, Y, B, \mathfrak{b}]\ \ \wedge\\ &\qquad \textstyle\bigwedge_{D[\boldsymbol{Y},X]} t'[\boldsymbol{E}, D[\boldsymbol{E}, A], A, \mathfrak{a}]\ (U[\boldsymbol{\rho}, D[\boldsymbol{\rho}, R], R]^{f\mathsf{abo}})\ t'[\boldsymbol{F}, D[\boldsymbol{F}, B], B, \mathfrak{b}]\end{aligned}$$

This follows by induction hypothesis on $\Lambda X.\lambda\mathfrak{x}{:}\mathfrak{T}[X].t'$.

t is $t'D$: To derive: $t'[\boldsymbol{E}, A, \mathfrak{a}]D[\boldsymbol{E}, A]\ (U[\boldsymbol{\rho}, D[\boldsymbol{\rho}, R], R]^{f\mathsf{abo}})\ t'[\boldsymbol{F}, B, \mathfrak{b}]D[\boldsymbol{F}, B]$. By induction hypothesis, we derive $t'[\boldsymbol{E}, A, \mathfrak{a}]\ (\forall Y.U'[\boldsymbol{\rho}, Y, R])^{f\mathsf{abo}}\ t'[\boldsymbol{F}, B, \mathfrak{b}]$, *i.e.,*

$$(\forall Y\,.\, t'[\boldsymbol{E}, A, \mathfrak{a}]Y \quad (U[\boldsymbol{\rho}, \mathsf{eq}_Y, R]^{f\mathsf{abo}}) \quad t'[\boldsymbol{F}, B, \mathfrak{b}]Y \quad \wedge$$
$$\textstyle\bigwedge_{D[\boldsymbol{Y},X]} t'[\boldsymbol{E}, A, \mathfrak{a}]D[\boldsymbol{E}, A] \; (U[\boldsymbol{\rho}, D[\boldsymbol{\rho}, R], R]^{f\mathsf{abo}}) \; t'[\boldsymbol{F}, B, \mathfrak{b}]D[\boldsymbol{F}, B]$$

We get what we want by instantiating with $D[\boldsymbol{E}, A]$, $D[\boldsymbol{F}, B]$, and $D[\boldsymbol{\rho}, R]$. □

Proof of Lemma 5.24: Let γ be any environment on Γ.

1. $\models_{\Gamma,\gamma} \mathsf{ClosedS}_{\urcorner,X}(X)$. We have X in $\urcorner, X$. Thus we choose $\widehat{\Gamma}$ to include X, $\widehat{X}$ to be X, and $\widehat{\gamma}$ such that $[X \mapsto \gamma(X)]$.

2. $\models_{\Gamma,\gamma} \mathsf{ClosedS}_{\urcorner}(U) \;\wedge\; \mathsf{ClosedS}_{\urcorner}(V) \;\Rightarrow\; \mathsf{ClosedS}_{\urcorner}(U \to V)$. From the antecedent, we have $\widehat{\Gamma}_U$, $\widehat{U} = U[\boldsymbol{T}/\boldsymbol{X}]$, and $\widehat{\gamma}_U$, as well as $\widehat{\Gamma}_V$, $\widehat{V} = V[\boldsymbol{T'}/\boldsymbol{X'}]$, and $\widehat{\gamma}_V$. Since $\widehat{\gamma}_U$ and $\widehat{\gamma}_V$ are made to agree with γ, in particular on common variables, we may choose $\widehat{\Gamma} = \widehat{\Gamma}_U \cup \widehat{\Gamma}_V$, $\widehat{U \to V} = \widehat{U} \to \widehat{V}$, and $\widehat{\gamma} = \widehat{\gamma}_U \cup \widehat{\gamma}_V$.

3. $\models_{\Gamma,\gamma} \mathsf{ClosedS}_{\urcorner,X}(U) \;\Rightarrow\; \mathsf{ClosedS}_{\urcorner}(\forall X.U)$. The interesting case is when X occurs freely in U. From the antecedent, we have $\widehat{\Gamma}$ and $\widehat{U} = U[\boldsymbol{T}/\boldsymbol{X}, T/X]$, where $[\![\widehat{\Gamma} \rhd T_i]\!]_{\widehat{\gamma}} = \gamma(X_i)$. Furthermore, since X is in $\urcorner, X$, we have that T is X with $\widehat{\gamma} = \widehat{\gamma}'[X \mapsto \gamma(X)]$, for some $\widehat{\gamma}'$. Note that X does not occur in any T_i, because T_i is either exactly X_i or a closed type. Then we can choose $\forall X.\widehat{U}$ for $\widehat{\forall X.U}$, and choose $\widehat{\gamma}'$ for the required environment, since for any value $\mathcal{A}$, we have $[\![\Gamma \rhd U[\boldsymbol{X}, X]]\!]_{\gamma[X \mapsto \mathcal{A}]} = [\![\widehat{\Gamma} \rhd U[\boldsymbol{T}/\boldsymbol{X}, X]]\!]_{\widehat{\gamma}'[X \mapsto \mathcal{A}]}$.

4. $\models_{\Gamma,\gamma} \mathsf{ClosedS}^{U}_{\urcorner,\boldsymbol{X},U}(x)$, for variable x. Since $\boldsymbol{X}$ and U is in $\urcorner$, we can choose $\widehat{\Gamma}$ to include $\boldsymbol{X}$ and $x\!:\!U$. By the same token, we must choose $\widehat{U} = U$ and $\widehat{x} = x$ together with an environment $\widehat{\gamma}$ such that $[x \mapsto \gamma(x)]$.

5. $\models_{\Gamma,\gamma} \mathsf{ClosedS}_{\urcorner,\boldsymbol{X}}(U) \;\wedge\; \mathsf{ClosedS}^{V}_{\urcorner,\boldsymbol{X},U}(v) \;\Rightarrow\; \mathsf{ClosedS}^{U\to V}_{\urcorner,\boldsymbol{X}}(\lambda x\!:\!U.v)$. The interesting case is when $x : U$ occurs freely in v. From the antecedent, we firstly get $\widehat{\Gamma}_U$, $\widehat{U} = U[\boldsymbol{T}/\boldsymbol{X}]$, and $\widehat{\gamma}_U$, where $\widehat{U} = U$, since $\boldsymbol{X}$ is in $\urcorner, \boldsymbol{X}$. We also get $\widehat{\Gamma}_V$, $\widehat{V} = V[\boldsymbol{T'}/\boldsymbol{X'}]$, $\widehat{v} = v[\boldsymbol{T'}/\boldsymbol{X'}, \boldsymbol{t}/\boldsymbol{x}, t/x] : \widehat{V}$, and $\widehat{\gamma}_V$, such that $[\![\widehat{\Gamma}_V \rhd \widehat{V}]\!]_{\widehat{\gamma}_V} = [\![\Gamma \rhd V]\!]_\gamma$, and $[\![\widehat{\Gamma}_V \rhd \widehat{v}\!:\!\widehat{V}]\!]_{\widehat{\gamma}_V} = [\![\Gamma \rhd v\!:\!V]\!]_\gamma$. Since U is in $\urcorner, \boldsymbol{X}, U$, we have that t is x in v, with $\widehat{\gamma}_V(x) = \gamma(x)$. By assuming an appropriate renaming of term variables, and since $\widehat{\gamma}_U$ and $\widehat{\gamma}_V$ are made to agree with γ, we may set $\widehat{\Gamma} = \widehat{\Gamma}_U \cup \widehat{\Gamma}_V$ and $\widehat{\gamma} = \widehat{\gamma}_U \cup \widehat{\gamma}_V$. Let $\mathcal{A} \stackrel{\mathit{def}}{=} [\![\widehat{\Gamma} \rhd \widehat{U}]\!]_{\widehat{\gamma}} = [\![\Gamma \rhd U]\!]_\gamma$. For any $a \in \mathcal{A}$, we get $[\![\widehat{\Gamma} \rhd \widehat{v}[x] : \widehat{V}]\!]_{\widehat{\gamma}[x \mapsto a]} = [\![\Gamma \rhd v[x] : V]\!]_{\gamma[x \mapsto a]}$, and we can choose $\lambda x\!:\!\widehat{U}.\widehat{v}$ and $\widehat{U} \to \widehat{V}$, respectively, for $\widehat{\lambda x\!:\!U.v}$ and $\widehat{U \to V}$.

6. $\models_{\Gamma,\gamma} \mathsf{ClosedS}^{U\to V}_{\urcorner}(g) \;\wedge\; \mathsf{ClosedS}^{U}_{\urcorner}(u) \;\Rightarrow\; \mathsf{ClosedS}^{V}_{\urcorner}(gu)$. From the antecedent, we have $\widehat{\Gamma}$, $\widehat{U \to V} = (U \to V)[\boldsymbol{T}/\boldsymbol{X}]$, $\widehat{g} = g[\boldsymbol{T}/\boldsymbol{X}, \boldsymbol{t}/\boldsymbol{x}]$, and $\widehat{\gamma}$. We

also have $\widehat{\Gamma}'$, $\widehat{U}' = U[\boldsymbol{T'}/\boldsymbol{X}]$, $\widehat{u} = u[\boldsymbol{T'}/\boldsymbol{X}, \boldsymbol{t'}/\boldsymbol{x'}]$, and $\widehat{\gamma}'$. We now replace $\widehat{u}$ with a $\widehat{u}' : U[\boldsymbol{T}/\boldsymbol{X}]$ such that $[\![\widehat{\Gamma} \triangleright \widehat{u}']\!]_{\widehat{\gamma}} = [\![\widehat{\Gamma}' \triangleright \widehat{u}]\!]_{\widehat{\gamma}'}$. Such a $\widehat{u}'$ exists, because $U[\boldsymbol{T}/\boldsymbol{X}]$ and $U[\boldsymbol{T'}/\boldsymbol{X}]$ differ only in the closed types T_i and T'_i substituted in for those X_i that are not in $\daleth$, and T_i and T'_i have equal denotations. From this we get the desired $\widehat{V} \stackrel{def}{=} V[\boldsymbol{T}/\boldsymbol{X}]$, and $\widehat{gu} \stackrel{def}{=} \widehat{g}\widehat{u}'$.

7. $\models_{\Gamma,\gamma} \mathsf{ClosedS}^{U}_{\daleth,X}(t) \;\Rightarrow\; \mathsf{ClosedS}^{\forall X.U}_{\daleth}(\Lambda X.t)$. The interesting case is when X is free in U and t. As for (3), we get the desired $\widehat{\forall X.U}$ by $\forall X.\widehat{U}$, and the desired environment by using $\widehat{\gamma}'$ from the environment $\widehat{\gamma} = \widehat{\gamma}'[X \mapsto \gamma(X)]$ on $\widehat{\Gamma}$. It remains to deal with the term $\Lambda X.t$. From the antecedent, we have $\widehat{t} = t[\boldsymbol{T}/\boldsymbol{X}, T/X, \boldsymbol{t}/\boldsymbol{x}]$, where $T = X$. We choose $\Lambda X.\widehat{t}$ for $\widehat{\Lambda X.t}$.

8. $\models_{\Gamma,\gamma} \mathsf{ClosedS}^{\forall X.U[X]}_{\daleth}(f) \;\wedge\; \mathsf{ClosedS}_{\daleth}(A) \;\Rightarrow\; \mathsf{ClosedS}^{U[A]}_{\daleth}(fA)$. Here we can use $\widehat{f}$, $\widehat{A}$, and $\widehat{U}$ supplied by the antecedent, and form the desired $\widehat{fA}$ and $\widehat{U[A]}$, by $\widehat{f}\widehat{A}$ and $\widehat{U}[\widehat{A}]$.

9. $\models_{\Gamma,\gamma} \mathsf{ClosedS}_{\daleth}(U) \;\Rightarrow\; \mathsf{ClosedS}_{\daleth'}(U)$, for $\daleth'$ as stated. Immediate.

10. $\models_{\Gamma,\gamma} \mathsf{ClosedS}^{U}_{\daleth}(u) \;\Rightarrow\; \mathsf{ClosedS}^{U}_{\daleth'}(u)$, for $\daleth'$ as stated. Immediate.

$\square$

Bibliography

M. Abadi and L. Cardelli. An imperative object calculus. *Theory and Practice of Object Systems*, 1(3):151–166, 1996.

M. Abadi, L. Cardelli, and P.-L. Curien. Formal parametric polymorphism. *Theoretical Computer Science*, 121:9–58, 1993.

S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Information and Computation*, 1996. To appear.

D. Aspinall. *Type Systems for Modular Programs and Specifications*. PhD thesis, Laboratory for Foundations of Computer Science (LFCS), University of Edinburgh, 1997.

I. Attali, D. Caromel, H. Nilsson, and M. Russo. From executable formal specification to Java property verification. In *Formal Techniques for Java Programs. Proceedings of the ECOOP'2000 Workshop, Cannes (France)*, 2000.

I. Attali, D. Caromel, and M. Russo. A formal executable semantics for Java. In *Formal Underpinnings of Java. Proceedings of the OOPSLA'98 Workshop, Vancouver (Canada)*, 1998.

R.-J. Back and J. Wright. *Refinement Calculus, A Systematic Introduction.* Graduate Texts in Computer Science. Springer Verlag, 1998.

E.S. Bainbridge, P.J. Freyd, A. Scedrov, and P.J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70:35–64, 1990.

H.P. Barendregt. Lambda calculi with types. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 118–309. Oxford University Press, 1992.

F.L. Bauer, M. Broy, W. Dosch, R. Gnatz, F. Geiselbrechtinger, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, H. Partsch, P. Pepper,

K. Samelson, M. Wirsing, and H. Wösner. Report on a wide spectrum language for program specification and development. Technical Report TUM-I8104, Technische Universität München, 1981.

M. Bidoit and R. Hennicker. Behavioural theories and the proof of behavioural properties. *Theoretical Computer Science*, 165:3–55, 1996.

M. Bidoit, R. Hennicker, and M. Wirsing. Behavioural and abstractor specifications. *Science of Computer Programming*, 25:149–186, 1995.

M. Bidoit, R. Hennicker, and M. Wirsing. Proof systems for structured specifications with observability operators. *Theoretical Computer Science*, 173:393–443, 1997.

M. Bidoit, H.-J. Kreowski, P. Lescanne, F. Orejas, and D. Sannella, editors. *Algebraic System Specification and Development. Survey and Annotated Bibliography*, volume 501 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.

M. Bidoit, D. Sannella, and A. Tarlecki. Behavioural encapsulation. CoFI Language Design Study Note. Available at `ftp://ftp.brics.dk/Projects/CoFI/StudyNotes/Lang/MB+DTS+AT-1.ps.Z`, 1996.

C. Böhm and A. Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

J.A. Borror. *Java Principles of Object-Oriented Programming: Build a Working Financial Application with Java 2*. Miller Freeman, Inc., 1999.

V. Breazu-Tannen and T. Coquand. Extensional models for polymorphism. *Theoretical Computer Science*, 59:85–114, 1988.

K.C. Bruce, L. Cardelli, and B.C. Pierce. Comparing object encodings. In M. Abadi and T. Ito, editors, *Theoretical Aspects of Computer Software. Proceedings of the 3rd International Symposium, TACS'97, Sendai (Japan)*, volume 1281 of *Lecture Notes in Computer Science*, pages 191–212. Springer Verlag, 1997.

K.C. Bruce, A. Meyer, and J.C. Mitchell. The semantics of second-order lambda calculus. *Information and Computation*, 85(1):76–134, 1990.

M.V. Cengarle. *Formal Specification with Higher-Order Parameterization*. PhD thesis, Fakultät für Mathematik, Ludwig-Maximillians-Universität, München, 1995.

M. Cerioli, M. Gogolla, H. Kirchner, B. Krieg-Brückner, Z. Qian, and M. Wolf, editors. *Algebraic System Specification and Development. Survey and Annotated Bibliography, 2nd Ed.*, volume 3 of *Monographs of the Bremen Institute of Safe Systems*. Shaker Verlag, 1997. 1st edition available in (Bidoit et al., 1991).

A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:56–68, 1940.

A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, NJ (USA), 1941.

CIP-L. *The Wide Spectrum Language CIP-L*, volume 183 of *Lecture Notes in Computer Science*. Springer Verlag, 1985.

A. Coglio, A. Goldberg, and Z. Qian. Towards a provably-correct implementation of the JVM bytecode verifier. In *DARPA Information Survivability Conference and Exposition. Proceedings of DISCEX'00*, pages 403–410. IEEE Computer Society, 2000.

Coq. The Coq Proof Assistant web page, 2000. `http://www.pauillac.inria.fr/coq/assis-eng.html`.

O.-J. Dahl. *Verifiable Programming, Revised version 1993*. Prentice Hall International Series in Computer Science; C.A.R. Hoare, Series Editor. Prentice-Hall, UK, 1992.

O.-J. Dahl and B. Kristoffersen. On introducing higher-order functions in ABEL. Research Report 210, Institutt for informatikk, Universitetet i Oslo, 1995.

O.-J. Dahl, B. Myhrhaug, and K. Nygaard. Simula 67 Common Base Language. Report N. S-22, Norsk Regnesentral, Oslo, Norway, 1970.

O.-J. Dahl and K. Nygaard. An Algol-based simulation language. *Communications of the ACM*, 9(9):671–678, September 1966.

O.-J. Dahl and K. Nygaard. The development of the SIMULA language. In R. Wexelblat, editor, *History of Programming Languages*, chapter 9, pages 439–493. Academic Press, 1981.

O.-J. Dahl and O. Owe. Formal development with ABEL. In S. Prehn and W.J. Toetene, editors, *Formal Software Development. Proceedings of the 4th International Symposium of VDM Europe, VDM'91, Noordwijkerhout (The Netherlands)*, volume 552 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.

O.-J. Dahl and O. Owe. On the use of subtypes in ABEL. Research Report 206, Institutt for informatikk, Universitetet i Oslo, 1995.

S. Drossopoulou, S. Eisenbach, and S. Khurshid. Is the java type system sound. *Theory and Practice of Object Systems*, 7(1):3–24, 1999.

K. Engelhardt and W.-P. de Roever. *Data refinement : Model-Oriented Proof Methods and their Comparison Refinement Calculus, A Systematic Introduction.* Number 47 in Cambridge tracts in theoretical computer science. Cambridge University Press, 1998.

J. Farrés-Casals. Proving correctness w.r.t. specifications with hidden parts. In H. Kirchner and W. Wechler, editors, *Algebraic and Logic Programming. Proceedings of the 2nd International Conference, ALP'90, Nancy (France)*, volume 463 of *Lecture Notes in Computer Science*, pages 25–39. Springer Verlag, 1990.

J. Farrés-Casals. *Verification in ASL and Related Specification Languages, Report CST-92-92.* PhD thesis, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1992.

K. Fischer and J.C. Mitchell. On the relationship between classes, objects, and data abstraction. In M. Broy and B. Schieder, editors, *Mathematical Methods in Program Development*, volume 158 of *NATO ASI Series F: Computer and System Sciences*, pages 371–407. Springer Verlag, 1997.

K. Fisher and J.C. Mitchell. Classes = Objects + Data Abstraction. Technical Report STAN-CS-TN-96-31, Stanford University, 1996.

D. Flanagan and P. Ferguson. *Java in a Nutshell: A Desktop Quick Reference, Third Edition.* O'Reilly & Associates, Inc., 1999.

W. Wayt Gibbs. Software's chronic crisis. *Scientific American (International Edition)*, 271(3):86–95, September 1994.

J.-Y. Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J.E. Fenstad, editor, *Proceedings of the 2nd Scandinavian Logic Symposium, Universitetet i Oslo (Norway)*, volume 63 of *Studies in logic and the foundations of mathematics*, pages 63–92. North-Holland, 1971.

J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

J.A. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, SE-10(5):528–543, 1984.

J.A. Goguen and R. Burstall. CAT, a system for the structured elaboration of correct programs from structured specifications. Technical Report CSL-118, SRI International, 1980.

J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, corrctness, and implementation of abstract data types. In R.T. Yeh, editor, *Current Trends in Programming Methodology*, volume 4: Data Structuring, pages 80–149. Prentice-Hall, 1978. Originally in IBM Research Report RC6487 (1976).

J.V. Guttag. *The Specification and Application to Programming of Abstract Data Types*. PhD thesis, Department of Computer Science, University of Toronto, 1975. Technical Report CSRG-59.

J.E. Hannay. Abstraction barriers in equational proof. In A.M. Haeberer, editor, *Algebraic Methodology and Software Technology. Proceedings of the 7th International Conference, AMAST'98, Amazonas (Brasil)*, volume 1548 of *Lecture Notes in Computer Science*, pages 196–213. Springer Verlag, 1998.

J.E. Hannay. Specification refinement with System F. In J. Flum and M. Rodríguez-Artalejo, editors, *Computer Science Logic. Proceedings of the 13th International Workshop, CSL'99, Madrid (Spain)*, volume 1683 of *Lecture Notes in Computer Science*, pages 530–545. Springer Verlag, 1999a.

J.E. Hannay. Specification refinement with System F, the higher-order case. In D. Bert, C. Choppy, and P.D. Mosses, editors, *Recent Trends in Algebraic Development Techniques. Selected Papers from the 14th International Workshop, WADT'99, Chateau de Bonas (France)*, volume 1827 of *Lecture Notes in Computer Science*, pages 162–181. Springer Verlag, 1999b.

J.E. Hannay. A higher-order simulation relation for System F. In J. Tiuryn, editor, *Foundations of Software Science and Computation Structures. Proceedings of the 3rd International Conference, FOSSACS 2000, Berlin (Germany)*, volume 1784 of *Lecture Notes in Computer Science*, pages 130–145. Springer Verlag, 2000.

R. Hasegawa. Parametricity of extensionally collapsed term models of polymorphism and their categorical properties. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Software. Proceedings of the International Conference,* *TACS'91, Sendai (Japan)*, volume 526 of *Lecture Notes in Computer Science*, pages 495–512. Springer Verlag, 1991.

R. Hennicker. Structured specifications with behavioural operators: Semantics, proof methods and applications. Habilitationsschrift, Institut für Informatik, Ludwig-Maximillians-Universität, München, 1997.

C.A.R. Hoare. An Axiomatic Approach to Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

C.A.R. Hoare. Proofs of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

B. Hoffmann and B. Krieg-Brückner, editors. *Program Development by Specification and Transformation, The PROSPECTRA Methodology, Language Family, and System*, volume 680 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.

M. Hofmann. *Extensional Concepts in Intensional Type Theory, Report CST-117-95 and Technical Report ECS-LFCS-95-327.* PhD thesis, Laboratory for Foundations of Computer Science (LFCS), University of Edinburgh, 1995a.

M. Hofmann. A simple model for quotient types. In M. Dezani-Ciancaglini and G.D. Plotkin, editors, *Typed Lambda Calculi and Applications. Proceedings of the 2nd International Conference, TLCA'95, Edinburgh (UK)*, volume 902 of *Lecture Notes in Computer Science*, pages 216–234. Springer Verlag, 1995b.

M. Hofmann and B. Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, 1995.

M. Hofmann and B. Pierce. Positive subtyping. *Information and Computation*, 126(1):186–197, 1996.

M. Hofmann and D. Sannella. On behavioural abstraction and behavioural satisfaction in higher-order logic. *Theoretical Computer Science*, 167:3–45, 1996.

F. Honsell, J. Longley, D. Sannella, and A. Tarlecki. Constructive data refinement in typed lambda calculus. In J. Tiuryn, editor, *Foundations of Software Science and Computation Structures. Proceedings of the 3rd International Conference, FOSSACS 2000, Berlin (Germany)*, volume 1784 of *Lecture Notes in Computer Science*, pages 161–176. Springer Verlag, 2000.

F. Honsell and D. Sannella. Pre-logical relations. In J. Flum and M. Rodríguez-Artalejo, editors, *Computer Science Logic. Proceedings of the 13th International Workshop, CSL'99, Madrid (Spain)*, volume 1683 of *Lecture Notes in Computer Science*, pages 546–561. Springer Verlag, 1999.

Isabelle. The Isabelle Theorem Prover web page, 2000. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle`.

B. Jacobs. On cubism. *Journal of Functional Programming*, 6(3):379–391, 1996.

Java Semantics. The Java Semantics home page, 2000. `http://www-sop.inria.fr/croap/java/index.html`.

S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML. Technical Report ECS-LFCS-94-300, Laboratory for Foundations of Computer Science (LFCS), University of Edinburgh, 1994.

S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML: a gentle introduction. *Theoretical Computer Science*, 173:445–484, 1997.

Y. Kinoshita, P.W. O'Hearn, A.J. Power, M. Takeyama, and R.D. Tennent. An axiomatic approach to binary logical relations with applications to data refinement. In M. Abadi and T. Ito, editors, *Theoretical Aspects of Computer Software. Proceedings of the 3rd International Symposium, TACS'97, Sendai (Japan)*, volume 1281 of *Lecture Notes in Computer Science*, pages 191–212. Springer Verlag, 1997.

Y. Kinoshita and A.J. Power. Data refinement for call-by-value programming languages. In J. Flum and M. Rodríguez-Artalejo, editors, *Computer Science Logic. Proceedings of the 13th International Workshop, CSL'99, Madrid (Spain)*, volume 1683 of *Lecture Notes in Computer Science*, pages 562–576. Springer Verlag, 1999.

H. Kirchner and P.D. Mosses. Algebraic specifications, higher-order types, and set-theoretic models. In A.M. Haeberer, editor, *Algebraic Methodology and Software Technology. Proceedings of the 7th International Conference, AMAST'98, Amazonas (Brasil)*, volume 1548 of *Lecture Notes in Computer Science*, pages 378–388. Springer Verlag, 1998.

B.R. Kirkerud. *Object-Oriented Programming with SIMULA*. Addison-Wesley, 1989.

B. Krieg-Brückner. PROgram development by SPECification and TRAnsformation. *Technique et Science Informatiques, Special Issue on Advanced Software Engineering in ESPRIT*, pages 134–149, 1990.

B. Krieg-Brückner, E. Karlsen, J. Liu, and O. Traynor. The PROSPECTRA methodology and system: Uniform transformational (meta-) development. In S. Prehn and W.J. Toetene, editors, *Formal Software Development. Proceedings of the 4th International Symposium of VDM Europe, VDM'91, Noordwijkerhout (The Netherlands)*, volume 552 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.

P. Landin. A correspondence between ALGOL60 and Church's lambda notation. *Communications of the ACM*, 8(2), 1965.

P. Landin. The next 700 programming languages. *Communications of the ACM*, 9, 1966.

LEGO. The LEGO Proof Assistant web page, 2000. `http://www.dcs.ed.ac.uk/home/lego`.

N.G. Leveson and C.S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

B.H. Liskov and S.N. Zilles. Programming with abstract data types. *ACM SIGPLAN Notices*, pages 50–59, 1974.

Z. Luo. Program specification and data type refinement in type theory. *Mathematical Structures in Computer Science*, 3:333–363, 1993.

Q. Ma and J.C. Reynolds. Types, abstraction and parametric polymorphism, part 2. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics, Proceedings of the 7th International Conference, MFPS, Pittsburgh, PA (USA)*, volume 598 of *Lecture Notes in Computer Science*, pages 1–40. Springer Verlag, 1991.

D.B. MacQueen. Modules for Standard ML. *Polymorphism*, 2(2), 1985.

D.B. MacQueen. Using dependent types to express modular structure. In *ACM Symposium on Principles of Programming Languages. Proceedings of the 13th ACM Symposium, St. Peterburg Beach, FL (USA)*, pages 277–286. ACM, 1986.

H. Mairson. Outline of a proof theory of parametricity. In J. Hughes, editor, *Functional Programming and Computer Architecture. Proceedings of the 5th acm*

Conference, Cambridge, MA (USA), volume 523 of *Lecture Notes in Computer Science*, pages 313–327. Springer Verlag, 1991.

K. Meinke. Universal algebra in higher types. *Theoretical Computer Science*, 100: 385–417, 1992.

R. Milner. An algebraic definition of simulation between programs. In *Joint Conferences on Artificial Intelligence, Proceedings of the 2nd International Conference, JCAI, London (UK)*, pages 481–489. Morgan Kaufman Publishers, 1971.

R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (revised).* MIT Press, 1997.

J.C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 18, pages 367–457. Elsevier, 1990.

J.C. Mitchell. On the equivalence of data representations. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 305–330. Academic Press, 1991.

J.C. Mitchell. *Foundations for Programming Languages.* Foundations of Computing. MIT Press, 1996.

J.C. Mitchell and G.D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

E. Moggi and R. Statman. The maximum consistent theory of the second order lambda calculus. Available at `ftp://ftp.disi.unige.it/person/MoggiE/papers/maxcons`, 1986.

C. Morgan. *Programming from Specifications, 2nd ed.* Prentice Hall International Series in Computer Science; C.A.R. Hoare, Series Editor. Prentice-Hall, UK, 1994.

N. Mylonakis. Behavioural specifications in type theory. In M. Haveraaen, O. Owe, and O.-J. Dahl, editors, *Recent Trends in Data Type Specification. Selected Papers from the 11th Workshop on Specification of Abstract Data Types, WADT, joint with 8th COMPASS Workshop, Oslo (Norway)*, volume 1130 of *Lecture Notes in Computer Science*, pages 394–408. Springer Verlag, 1995.

G. Nelson, editor. *System Programming with Modula-3.* Prentice Hall, 1991.

T. Nipkow, D. von Oheimb, and C. Pusch. μJava: Embedding a programming language in a theorem prover. In F.L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation. Proceedings International Summer School Marktoberdorf 1999*, pages 117–144. IOS Press, 2000.

M.P. Nivela and F. Orejas. Initial behaviour semantics for algebraic specifications. In H. Reichel, editor, *Recent Trends in Data Type Specification. Selected Papers from the 5th Workshop on Specification of Abstract Data Types, WADT, S. Margherita (Italy)*, volume 332 of *Lecture Notes in Computer Science*, pages 184–207. Springer Verlag, 1988.

D. Normann. The continuous functionals of finite types over the reals. Technical Report 19, Institutt for matematikk, Universitetet i Oslo, 1998.

P.W. O'Hearn and R.D. Tennent. Relational parametricity and local variables. In *20th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Proceedings*, pages 171–184. ACM Press, 1993.

D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 119–156. Springer Verlag, 1999.

O. Owe and I. Ryl. The Oslo University Notation: A formalism for open, object-oriented, distributed systems. Research Report 270, Institutt for informatikk, Universitetet i Oslo, 1999.

L.C. Paulson. *ML for the Working Programmer, 2nd edition.* Cambridge University Press, 1996.

B. Pierce, S. Dietzen, and S. Michaylov. Programming in higher-order typed lambda-calculi. Technical Report CMU-CS-89-111, also ERGO-89-075, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (USA), March 1989.

A.M. Pitts. Parametric polymorphism and operational equivalence. In *Higher-Order Operational Techniques in Semantics. Proceedings of the 2nd Workshop, HOOTS II, Stanford University, CA (USA)*, volume 10 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1997.

A.M. Pitts. Existential types: Logical relations and operational equivalence. In K. Guldstrand Larsen, S. Skyum, and G. Winskel, editors, *Automata, Languages and Programming. Proceedings of the 25th International Colloquium,* *ICALP'98, Aalborg (Denmark)*, volume 1443 of *Lecture Notes in Computer Science*, pages 309–326. Springer Verlag, 1998.

G.D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

G.D. Plotkin and M. Abadi. A logic for parametric polymorphism. In M. Bezem and J.F. Groote, editors, *Typed Lambda Calculi and Applications. Proceedings of the International Conference, TLCA'93, Utrecht (The Netherlands)*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375. Springer Verlag, 1993.

G.D. Plotkin, A.J. Power, D. Sannella, and R.D. Tennent. Lax logical relations. In U. Montanari, J.D.P. Rolim, and E. Welzl, editors, *Automata, Languages and Programming. Proceedings of the 27th International Colloquium, ICALP 2000, Geneva (Switzerland)*, volume 1853 of *Lecture Notes in Computer Science*, pages 85–102. Springer Verlag, 2000.

D.B. Plume. A calculator for exact real number computation. B.Sc. Project Report, Laboratory for Foundations of Computer Science (LFCS), University of Edinburgh, `ftp://ftp/tardis.ed.ac.uk/users/dbp/report.ps.gz`, 1998.

E. Poll and J. Zwanenburg. A logic for abstract data types as existential types. In J.-Y. Girard, editor, *Typed Lambda Calculus and Applications. Proceedings of the 4th International Conference, TLCA'99, L'Aquila (Italy)*, volume 1581 of *Lecture Notes in Computer Science*, pages 310–324. Springer Verlag, 1999.

R.J. Pooley. *An Introduction to Programming in SIMULA*. Oxford, Blackwell Scientific Publications, 1987.

R.J. Pooley and P. Stevens. *Using UML, Software Engineering with Objects and Components, Second Edition*. Object Technology Series. Addison-Wesley, 1999.

Precise UML. The Precise UML Group web page, 2000. `http://www.cs.york.ac.uk/puml`.

Proof General. The Proof General generic interface for proof assistants, 2000. `http://zermelo.dcs.ed.ac.uk/~proofgen`.

X. Qian and A. Goldberg. Referential opacity in nondeterministic data refinement. *ACM LoPLaS*, 2(1–4):233–241, 1993.

Z. Qian. A formal specification of Java Virtual Machine instructions for objects, methods and subroutines. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer Verlag, 1999.

U.S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *LISP and Functional Programming. Proceedings of the 1988 ACM conference, LFP, Snowbird, UT (USA)*, pages 289–297. ACM digital library, 1988.

U.S. Reddy. Global state considered unnecessary: An introduction to object-based semantics. In P.W. O'Hearn and R.D. Tennent, editors, *Algol-Like Languages*, chapter 19. Birkhäuser, 1997.

U.S. Reddy. Objects and classes in Algol-like languages. *Theory and Practice of Object Systems*, 1999.

H. Reichel. *Initial Computability, Algebraic Specifications, and Partial Algebras.* Number 2 in International Series of Monographs in Computer Science, Oxford. Clarendon Press, 1987.

B. Reus and T. Streicher. Verifying properties of module construction in type theory. In A.M. Borzyszkowski and S. Sokolowski, editors, *Mathematical Foundations of Computer Science. Proceedings of the 18th International Symposium, MFCS'93, Gdansk (Poland)*, volume 711 of *Lecture Notes in Computer Science*, pages 660–670. Springer Verlag, 1993.

J.C. Reynolds. Towards a theory of type structure. In G. Goos and J. Hartmanis, editors, *Programming Symposium. Proceedings of Colloque sur la Programmation, Paris (France)*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer Verlag, 1974.

J.C. Reynolds. *The Craft of Programming.* Prentice-Hall International, 1981.

J.C. Reynolds. Types, abstraction and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris (France)*, pages 513–523. Elsevier Science Publishers B.V. (North-Holland), 1983.

J.C. Reynolds. *Theories of Programming Languages.* Cambridge University Press, 1998.

Risks Digest. The Risks Digest. Web page of the Forum on Risks to the Public in Computers and Related Systems, the ACM Committee on Computers and Public Policy, 2000. `http://catless.ncl.ac.uk/Risks`.

D. Sannella. Formal program development in Extended ML for the working programmer. In C. Morgan and J.C.P. Woodcock, editors, *Proceedings of the 3rd BCS/FACS Workshop on Refinement, Hursley Park (UK)*, Workshops in Computing, pages 99–130. Springer Verlag, 1991.

D. Sannella, S. Sokołowski, and A. Tarlecki. Toward formal development of programs from algebraic specifications: parameterisation revisited. *Acta Informatica*, 29:689–736, 1992.

D. Sannella and A. Tarlecki. On observational equivalence and algebraic specification. *Journal of Computer and System Sciences*, 34(2/3):150–178, 1987.

D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76:165–210, 1988a.

D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: Implementations revisited. *Acta Informatica*, 25(3):233–281, 1988b.

D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9:229–269, 1997.

D. Sannella and M. Wirsing. A kernel language for algebraic specification and implementation. In Marek Karpinski, editor, *Fundamentals of Computation Theory. Proceedings of the 1983 International FCT-Conference, Borgholm (Sweden)*, volume 158 of *Lecture Notes in Computer Science*, pages 413–427. Springer Verlag, 1983.

D. Sannella and M. Wirsing. Specification languages. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, chapter 8. Springer Verlag, 1999.

O. Schoett. *Data Abstraction and the Correctness of Modular Programming*. PhD thesis, Laboratory for Foundations of Computer Science (LFCS), University of Edinburgh, 1986.

O. Schoett. Behavioural correctness of data representations. *Science of Computer Programming*, 14:43–57, 1990.

SIAM. Inquiry board traces Ariane 5 failure to overflow error. *SIAM News*, 29 (8), October 1996. `http://www.siam.org/siamnews/general/ariane.htm`.

A.K. Simpson. Lazy functional algorithms for exact real functionals. In *Mathematical Foundations of Computer Science. Proceedings of the 23rd International Symposium, MFCS'98, Brno (Czech Republic)*, volume 1450 of *Lecture Notes in Computer Science*, pages 456–464. Springer Verlag, 1998.

H. Søndergaard and P. Sestoft. Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27(6):505–517, 1990.

C. Strachey. Fundamental concepts in programming languages. Lecture notes from the International Summer School in Programming Languages, Copenhagen (Denmark), 1967.

T. Streicher and M. Wirsing. Dependent types considered necessary for specification languages. In H. Ehrig, K.P. Jantke, F. Orejas, and H. Reichel, editors, *Recent Trends in Data Type Specification. Proceedings of the 7th Workshop on Specification of Abstract Data Types, WADT, Wusterhausen/Dosse (Germany)*, volume 534 of *Lecture Notes in Computer Science*, pages 323–339. Springer Verlag, 1990.

R.D. Tennent. Correctness of data representations in Algol-like languages. In A.W. Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*. Prentice Hall International, 1997.

I. Traore, D.B. Aredo, and K. Stølen. Formal development of open distributed systems: towards an integrated framework. In *Object-Oriented Specification Techniques for Distributed Systems and Behaviours. Proceedings of the Workshop, OOSDS'99, Paris, (France)*, 1999.

A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics, An Introduction*, volume 121 of *Studies in Logic and The Foundations of Mathematics*. North Holland, 1988.

J. Underwood. Typing abstract data types. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification. Selected Papers from the 10th Workshop on Specification of Abstract Data Types, WADT, joint with the 5th COMPASS Workshop, S. Margherita (Italy)*, volume 906 of *Lecture Notes in Computer Science*, pages 437–452. Springer Verlag, 1994.

René Vestergaard and James Brotherston. A formalised first-order confluence proof for the λ-calculus using one-sorted variable names (*Barendregt was right after all ... almost*). In Aart Middeldorp, editor, *Rewriting Techniques and Applications. Proceedings of the 12th International Conference RTA, Utrecht (The Netherlands)*, volume 2051 of *LNCS*, pages 306–321. Springer-Verlag, 2001a.

René Vestergaard and James Brotherston. The mechanisation of Barendregt-style equational proofs (the residual perspective). In Roy Crole and Simon Ambler, editors, *Mechanized Reasoning about Languages with Variable Binding. Proceedings of the workshop MERLIN 2001, Sienna (Italy)*, volume 58/1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001b.

P. Wadler. Theorems for free. In *Functional Programming Languages and Computer Architecture. Proceedings of the 4th International Symposium FPLCA, London (UK)*. ACM, 1989.

M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 13, pages 675–788. Elsevier, 1990.

M. Wirsing. Structured specifications: Syntax, semantics and proof calculus. In F.L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of *NATO ASI Series F: Computer and System Sciences*, pages 411–442. Springer Verlag, 1993.

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules: Formalization in Type Theory and Implementation in Yarrow*. PhD thesis, Technische Universiteit Eindhoven, 1999.

Index

Author Index

Subject Index