Specification Refinement with System F

Jo Erskine Hannay

LFCS, Division of Informatics, University of Edinburgh, Scotland, U.K.
joh@dcs.ed.ac.uk

Abstract. Essential concepts of algebraic specification refinement are translated into a 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. We show the equivalence of the acquired type-theoretic notion of specification refinement with that from algebraic specification. To do this, a generic algebraic-specification strategy for behavioural refinement proofs is mirrored in the type-theoretic setting.

1 Introduction

This paper aims to express in type theory certain essential concepts of algebraic specification refinement. The benefit 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.

On the other hand, the benefit to type theory is to draw from the rich source of formalisms, development methodology and reasoning techniques in algebraic specification. See [7] for a survey and comprehensive bibliography. 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. In this paper we base ourselves on the description in [31, 30], and we highlight three essential concepts that make this account of specification refinement apt for real-life development. These are so-called *constructor implementations*, *behavioural equivalence* and *stability.* We will express this refinement framework in a type-theoretic environment comprised of System F and the assumption of relational parametricity in Reynolds' sense [27, 18], as expressed in Plotkin and Abadi's logic for parametric polymorphism [24]. Abstract data types are expressed in the type theory as existential types.

The above concepts of specification refinement fall out naturally in this setting. In this, relational parametricity plays an essential role. It gives the equivalence at first order of observational equivalence to equality at existential type.

In algebraic specification there is a generic proof strategy formalised in [6, 4, 5] for proving observational refinements. This considers axiomatisations of so-called behavioural (partial) congruences. As also observed in [25], Plotkin and Abadi's logic is not sufficient to accommodate this proof strategy. Inspired by [25], we choose the simple solution of adding axioms stating the existence of quotients and sub-objects. This is justified by the soundness of the axioms w.r.t. the parametric PER-model [3] for Plotkin and Abadi's logic. In this paper we import the proof strategy into type theory to show a correspondence between two notions of refinement. But this importation is also interesting in its own right, and our results complement those of [25] in that we consider also partial congruences.

Other work linking algebraic specification and type theory includes [17] encoding constructor implementations in ECC, [26] expressing module-algebra axioms in ECC, [23] encoding behavioural equalities in UTT, [2] treating the specification language ASL+, [35] using Nuprl as a specification language, and [34] promoting dependent types in specification. Only [25] utilises relational parametricity. There are also non-type-theoretic higher-order approaches using higher-order universal algebra [20], and other set-theoretic models [16].

The next section outlines algebraic specification refinement, highlighting the three essential concepts above. Then, the translation of algebraic specification refinement into a System F environment is presented, giving a type-theoretic notion of specification refinement. The main result of this paper 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.

2 Algebraic Specification Refinement

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 operator names. We write profiles $f\colon s_1 \times \cdots \times s_n \to s \in \Omega$, meaning $f \in \Omega_{s_1,\ldots,s_n,s}$. A Σ-algebra $A = \langle (A)_{s\in S}, F \rangle$ consists of an S-sorted set $(A)_{s\in S}$ of non-empty carriers and a set F containing a total function $f^A \in (A_{s_1} \times \cdots \times A_{s_n} \to A_s)$ for every $f\colon s_1 \times \cdots \times s_n \to s \in \Omega$. The class of Σ-algebras is denoted by $\Sigma\mathbf{Alg}$. 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. We consider sorted first-order logic with equality. A formula φ is a Σ-formula if all terms in φ are of sorts in S. Let Φ be a set of closed Σ-formulae. Then $SP = \langle \Sigma, \Phi \rangle$ is a basic algebraic specification, and its semantics $[\![SP]\!]$ is $Mod_\Sigma(\Phi)$, the class of Σ-algebras that are models of Φ.

Example 1. The following specification specifies stacks of natural numbers.

spec Stack **is**
  sorts $\mathsf{nat}, \mathsf{stack}$
  operators $\mathsf{empty} : \mathsf{stack},\ \mathsf{push} : \mathsf{nat} \times \mathsf{stack} \to \mathsf{stack},$
    $\mathsf{pop} : \mathsf{stack} \to \mathsf{stack},\ \mathsf{top} : \mathsf{stack} \to \mathsf{nat}$
  axioms $\Phi_{\mathsf{Stack}} : \mathsf{pop}(\mathsf{push}(x, s)) = s$
    $\mathsf{top}(\mathsf{push}(x, s)) = x$

We omit universal quantification over free variables in examples. The semantics of a data type (in a program) is an algebra. Wide-spectrum specification languages *e.g.* Extended ML [14], allow specifications and programs to be written in a uniform language, so that specifications are abstract descriptions of a data type or systems of data types, while program modules and programs are concrete executable descriptions of the same. A refinement process seeks to develop in a sound methodical way the latter from the former, and a program is then a *full refinement* or *realisation* of an abstract specification. The basic definition of refinement we adopt here is given by the following refinement relation $\rightsquigarrow$ on specifications of the same signature [30, 32]: $SP_j \rightsquigarrow SP_{j+1} \overset{def}{\iff} [\![SP_j]\!] \supseteq [\![SP_{j+1}]\!]$.

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

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* [9] with input interface SP_{j+1} and output interface SP_j. Given a program P that is a full refinement of SP_{j+1}, the instantiation $\kappa(P)$ is then a full refinement of SP_j. The semantics of a parameterised program is a function $[\![\kappa]\!] \in (\Sigma_{SP_{j+1}}\mathbf{Alg} \to \Sigma_{SP_j}\mathbf{Alg})$ called a *constructor. Constructor implementation* is then defined [30] as $SP_j \underset{\kappa}{\rightsquigarrow} SP_{j+1} \overset{def}{\iff} [\![SP_j]\!] \supseteq [\![\kappa]\!]([\![SP_{j+1}]\!])$. The parameterised program κ is the fully refined part of the system which is set aside, and SP_{j+1} specifies the remaining unresolved part that needs further refinement.

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. Two Σ-algebras A and B are *observationally equivalent w.r.t. Obs, In*, written $A \equiv_{Obs,In} B$, if every observable computation has equivalent denotations in A and B [29]. However, the semantics $[\![SP]\!]$ is not always closed under behavioural equivalence. For example, the stack-with-pointer implementation of stacks of natural numbers does not satisfy $\mathsf{pop}(\mathsf{push}(x, s)) = s$ and is not in $[\![\mathsf{Stack}]\!]$, but is behaviourally equivalent w.r.t. $Obs = In = \{\mathsf{nat}\}$ to an algebra that is. To capture this, one defines the semantics $[\![SP]\!]_{Obs,In} \overset{def}{=} \{B \mid \exists A \in [\![SP]\!] \,.\, B \equiv_{Obs,In} A\}$, and defines *refinement up to behavioural equivalence* [30] as $\langle SP_j, Obs, In\rangle \underset{\kappa}{\rightsquigarrow} \langle SP_{j+1}, Obs', In'\rangle \overset{def}{\iff} [\![SP_j]\!]_{Obs,In} \supseteq [\![\kappa]\!]([\![SP_{j+1}]\!]_{Obs',In'})$. Why do we want designated input sorts? One extremal view would be to say that all observable computations should be ground terms, *i.e.* proclaim $In = \emptyset$. 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 behavioural refinements as illustrated in the following example from [10].

Example 2. Consider the following specification of sets of natural numbers.

$$\begin{array}{l}
\textbf{spec } \mathsf{Set} \textbf{ is} \\
\quad \textbf{sorts } \mathsf{nat}, \mathsf{set} \\
\quad \textbf{operators } \mathsf{empty} : \mathsf{set},\ \mathsf{add} : \mathsf{nat} \times \mathsf{set} \to \mathsf{set} \\
\qquad\qquad\qquad \mathsf{in} : \mathsf{nat} \times \mathsf{set} \to \mathsf{bool},\ \mathsf{remove} : \mathsf{nat} \times \mathsf{set} \to \mathsf{set} \\
\quad \textbf{axioms } \mathsf{add}(x, \mathsf{add}(x, s)) = \mathsf{add}(x, s) \\
\qquad\qquad \mathsf{add}(x, \mathsf{add}(y, s)) = \mathsf{add}(y, \mathsf{add}(x, s)) \\
\qquad\qquad \mathsf{in}(x, \mathsf{empty}) = \mathsf{false} \\
\qquad\qquad \mathsf{in}(x, \mathsf{add}(y, s)) = \mathsf{if}\ x =_{\mathsf{nat}} y\ \mathsf{then\ true\ else}\ \mathsf{in}(x, s) \\
\qquad\qquad \mathsf{in}(x, \mathsf{remove}(x, s)) = \mathsf{false}
\end{array}$$

Consider the Σ_{Set}-algebra *ListImpl* (*LI*) whose carrier LI_{set} is the set of finite lists over the natural numbers; empty^{LI} gives the empty list, add^{LI} appends a given element to the end of a list only if the element does not occur already, in^{LI} is the occurrence function, and remove^{LI} removes the first occurrence of a given element. Being a Σ_{Set}-algebra, *LI* allows users only to build lists using empty^{LI} and add^{LI}, and on such lists the efficient remove^{LI} gives the intended result. However, $LI \notin [\![\mathsf{Set}]\!]_{Obs,In}$, for $Obs = \{\mathsf{bool}, \mathsf{nat}\}$ 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, $LI \in [\![\mathsf{Set}]\!]_{Obs,In}$ for $In = Obs = \{\mathsf{bool}, \mathsf{nat}\}$, since now the use of set-variables in observable computations is prohibited. ○

In this example, the correct choice was $In = Obs$. In fact $In = Obs$ is virtually always a sensible choice, and a very reasonable simplifying assumption.

Behavioural refinement steps are in general hard to verify. A helpful concept is *stability* [33]. 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 $\langle SP_j, Obs, In\rangle \underset{\kappa}{\rightsquigarrow} \langle SP_{j+1}, Obs', In'\rangle$, to show that $[\![SP_j]\!]_{Obs,In} \supseteq [\![\kappa]\!]([\![SP_{j+1}]\!])$. The following contrived but short example from [31] illustrates the point. See *e.g.* [33] for a more realistic example.

Example 3. Consider the specification

$$\begin{array}{l}
\textbf{spec } \mathsf{Triv} \textbf{ is} \\
\quad \textbf{sorts } \mathsf{nat} \\
\quad \textbf{operators } \mathsf{id} : \mathsf{nat} \times \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat} \\
\quad \textbf{axioms } \Phi_{\mathsf{Triv}} : \mathsf{id}(x, n, z) = x
\end{array}$$

Define the constructor $Tr \in (\Sigma_{\mathsf{Stack}}\mathbf{Alg} \to \Sigma_{\mathsf{Triv}}\mathbf{Alg})$ as follows. For $A \in \Sigma_{\mathsf{Stack}}\mathbf{Alg}$, define $multipush_A \in (\mathbb{N} \times \mathbb{N} \times A \to A)$ and $multipop_A \in (\mathbb{N} \times A \to A)$ by

$$multipush_A(n, z, a) = \begin{cases} a, & n = 0 \\ \mathsf{push}^A(z, multipush_A(n-1, z+1, a)), & n > 0 \end{cases}$$

$$multipop_A(n, a) = \begin{cases} a, & n = 0 \\ multipop_A(n-1, \mathsf{pop}^A(a)), & n > 0 \end{cases}$$

Then $Tr(A)$ is the Triv-algebra whose single operator is given by

$$id(x,n,z) = \mathsf{top}^A(\mathit{multipop}_A(n, \mathit{multipush}_A(n, z, \mathsf{push}^A(x, \mathsf{empty}^A)))).$$

We have $[\![\mathsf{Triv}]\!]_{\{\mathsf{nat}\},In} \supseteq \mathit{Tr}([\![\mathsf{Stack}]\!]_{\{\mathsf{nat}\},In})$, but to prove this only assuming membership in $[\![\mathsf{Stack}]\!]_{\{\mathsf{nat}\},In}$ is not straight-forward. However, Tr is in fact stable. So it suffices to show $[\![\mathsf{Triv}]\!]_{\{\mathsf{nat}\},In} \supseteq \mathit{Tr}([\![\mathsf{Stack}]\!])$, and the proof of this is easy [31]. In particular, one may now hold $\mathsf{pop}(\mathsf{push}(x,s)) = s$ among ones assumptions, although this formula is not valid for $[\![\mathsf{Stack}]\!]_{\{\mathsf{nat}\},In}$. ○

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.

Example 4 ([31]). Consider the constructor $\mathit{Tr}' \in (\Sigma_{\mathsf{Stack}}\mathbf{Alg} \to \Sigma_{\mathsf{Triv}}\mathbf{Alg})$ such that $\mathit{Tr}'(A)$ is the Triv-algebra whose single operator is given by

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

Then for A the array-with-pointer algebra, we get $\mathit{Tr}'(A) \notin [\![Triv]\!]_{\{\mathsf{nat}\},In}$ and so $[\![\mathsf{Triv}]\!]_{\{\mathsf{nat}\},In} \not\supseteq \mathit{Tr}'([\![\mathsf{Stack}]\!]_{\{\mathsf{nat}\},In})$. Tr' is not stable, and Tr' breaches the abstraction barrier by checking equality on the underlying implementation. ○

Algebraic specifications may be complex, built from basic specifications using specification building operators, *e.g.* [36, 32, 37]. But as a starting point for the translation into type theory, we only consider basic specifications.

3 The Type Theory

We now sketch the logic in [24, 19] for parametric polymorphism on System F. It is this accompanying logic that bears an extension rather than the type theory. See [1] for a more internalised approach. System F has types and terms as follows.

$$T ::= X \mid T \to T \mid \forall X.T \qquad\qquad t ::= x \mid \lambda x{:}T.t \mid tt \mid \Lambda X.t \mid tT$$

where X and x range over type and term variables resp. However, formulae are now built using the usual connectives from equations *and* relation symbols.

$$\phi ::= (t =_A u) \mid R(t,u) \mid \cdots \mid \forall R \subset A \times B.\phi \mid \exists R \subset A \times B.\phi$$

where R ranges over relation symbols. We write $\alpha[R, X, x]$ to indicate possible occurrences of R, X and x in α, and may write $\alpha[\rho, A, t]$ for the result of substitution, following the appropriate rules concerning capture.

Judgements for type and term formation and second-order environments with term environments depending on type environments, are as usual. But formula formation now involves relation symbols, so second-order environments are augmented with relation environments, *viz.* a finite sequence Υ of relational typings $R \subset A \times B$ of relation variables, depending on the type environment, and obeying standard conventions for environments. The formation rules for atomic formulae consists of the usual one for equations, and now also one for relations:

$$\frac{\Gamma \vdash t{:}A, \quad \Gamma \vdash u{:}B, \quad \Gamma \vdash \Upsilon, \quad \Upsilon \vdash R \subset A \times B}{\Gamma, \Upsilon \vdash R(t,u)\ \mathit{Prop} \qquad \textit{(also written tRu)}}$$

The other rules for formulae are as expected. Relation definition is accommodated:

$$\frac{\Gamma, x{:}A, y{:}B \ \vdash \ \phi\ \mathit{Prop}}{\Gamma \ \vdash \ (x{:}A, y{:}B)\ .\ \phi \quad \subset A \times B}$$

For example $\mathsf{eq}_A \overset{def}{=} (x{:}A, y{:}A).(x =_A y)$.

If $\rho \subset A \times B$, $\rho' \subset A' \times B'$ and $\rho''[R] \subset A[Y] \times B[Z]$, then complex relations are built by $\rho \to \rho' \subset (A \to A') \times (B \to B')$ where

$$(\rho \to \rho') \overset{def}{=} (f{:}A \to A', g{:}B \to B').(\forall x{:}A \forall x'{:}B.(x \rho x' \ \Rightarrow \ (fx)\rho'(gx')))$$

and $\forall(Y, Z, R \subset Y \times Z)\rho''[R] \subset (\forall Y.A[Y]) \times (\forall Z.B[Z])$ where

$$\forall(Y, Z, R \subset Y{\times}Z)\rho'' \overset{def}{=} (y{:}\forall Y.A[Y], z{:}\forall Z.B[Z]).(\forall Y \forall Z \forall R \subset Y{\times}Z.((yY)\rho''[R](zZ)))$$

One can now acquire further definable relations by substituting definable 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 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:

$$\frac{\Phi \vdash_{\Gamma, R \subset A \times B} \phi[R]}{\Phi \vdash_{\Gamma} \forall R \subset A \times B\ .\ \phi[R]} \qquad \frac{\Phi \vdash_{\Gamma} \forall R \subset A \times B.\phi[R], \quad \Gamma \vdash \rho \subset A \times B}{\Phi \vdash_{\Gamma} \phi[\rho]}$$

One also has axioms for equational reasoning and $\beta\eta$ equalities. Finally, the following parametricity axiom schema is asserted:

$$\textsc{Param} : \vdash_{\emptyset} \forall Y_1, \ldots, \forall Y_n \forall u{:}(\forall X.T[X, Y_1, \ldots, Y_n])\ .\ u(\forall X.T[X, \mathsf{eq}_{Y_1}, \ldots, \mathsf{eq}_{Y_n}])u$$

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

Fact 1 (Identity Extension Lemma [24]**).** *For any $T[\boldsymbol{Z}]$, the following sequent is derivable using* Param.

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

Encapsulation is provided by the following encoding of existential types and the following pack and unpack combinators.

$$\exists X.T[X] \overset{def}{=} \forall Y.(\forall X.(T[X] \to Y) \to Y)$$

$$\begin{aligned}&\mathsf{pack}_{T[X]}{:}\,\forall X.(T[X] \to \exists X.T[X])\\ &\mathsf{pack}_{T[X]}(A)(\mathit{impl}) \overset{def}{=} \Lambda Y.\lambda f{:}\forall X.(T[X] \to Y).f(A)(\mathit{impl})\end{aligned}$$

$$\begin{aligned}&\mathsf{unpack}_{T[X]}{:}\,(\exists X.T[X]) \to \forall Y.(\forall X.(T[X] \to Y) \to Y)\\ &\mathsf{unpack}_{T[X]}(\mathit{package})(B)(\mathit{client}) \overset{def}{=} \mathit{package}(B)(\mathit{client})\end{aligned}$$

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

Fact 2 (Characterisation by Simulation Relation [24]). *The following sequent schema is derivable using* PARAM.

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

The sequent in Fact 2 states the equivalence of equality at existential type with the existence of a simulation relation in the sense of [21]. From this we also get

$$\vdash_\emptyset \forall \boldsymbol{Z}.\forall u{:}\exists X.T[X,\boldsymbol{Z}].\exists A.\exists \mathfrak{a}{:}T[A,\boldsymbol{Z}]\ .\ u = (\mathsf{pack} A\ \mathfrak{a})$$

Weak versions of standard constructs such as products, initial and final (co-)algebras are encodable in System F [8]. With PARAM, these constructs are provably universal constructions. We can *e.g.* freely use product types. Given $\rho \subset A{\times}B$ and $\rho' \subset A'{\times}B'$, $(\rho{\times}\rho)$ is defined as the action $(X{\times}X')[\rho,\rho']$. One derives $\forall u{:}A{\times}A', v{:}B{\times}B'\ .\ u(\rho{\times}\rho')v \ \Leftrightarrow\ (\mathsf{fst}(u)\ \rho\ \mathsf{fst}(v) \wedge \mathsf{snd}(u)\ \rho\ \mathsf{snd}(v))$. We also use the abbreviations $\mathsf{bool} \overset{def}{=} \forall X.X \to X \to X$ and $\mathsf{nat} \overset{def}{=} \forall X.X \to (X \to X) \to X$; which are provably initial constructs.

Finally, this logic is sound w.r.t. to the parametric PER-model of [3].

4 The Translation

We now define a translation $\mathcal{T}$ giving an interpretation in the type theory and logic outlined in Sect. 3, of the concept of algebraic specification refinement $\langle SP_j, \mathit{Obs}, \mathit{In}\rangle \underset{\kappa}{\rightsquigarrow} \langle SP_{j+1}, \mathit{Obs}', \mathit{In}'\rangle$. We will use inhabitants of existential types as analogues to algebras, and then existentially quantified variables will correspond to non-observable (*behavioural*) sorts.

To keep things simple, we will at any one refinement stage assume a single behavioural sort b; 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 straight-forward to generalise to multiple existentially quantified variables [21].

In algebraic specification, there is no restraint on the choice of input sorts and observable sorts within the sorts S of a signature. In the type-theoretic setting, we will see that we have only one choice for the corresponding notion of *input types*, namely the collection of all types. Since a behavioural sort corresponds to an existentially quantified type variable, this automatically caters for situations which in algebraic specification correspond to crucially excluding the behavioural sort from the input sorts (Example 2). In algebraic specification, conforming to this type-theoretic insistence means assuming $In = S \setminus b$, which probably covers all reasonable examples of refinement. Thus, the type-theoretic formalisms inherently select a sensible choice.

For *observable types* on the other hand, we seem to have some choice. Our assumption of at most one behavioural sort means $Obs = S \setminus b$, hence $Obs = In$, in the algebraic specification setting. In type theory we could therefore let all types be observable types, as we must for input types. However, since 'observable' should mean 'printable', we limit the observable types by letting Obs denote also observable types; we assume that for every sort $s \in Obs$ there is an obvious closed type given the name s, for which the Identity Extension Lemma (Fact 1) gives $x\,(s[\rho])\,y \;\Leftrightarrow\; x =_s y$. Examples are bool and nat.

Note that the assumption of $Obs = In$ means that it suffices to write algebraic specification refinement as $\langle SP_j, Obs\rangle \underset{\kappa}{\rightsquigarrow} \langle SP_{j+1}, Obs'\rangle$.

In the following we use record type notation as a notational convenience.

Definition 1 (Translation and Type Theory Specification). *Let $SP = \langle \Sigma, \Phi\rangle$ where $\Sigma = \langle S, \Omega\rangle$. Define the translation $\mathcal{T}$ by*

$$\mathcal{T}\langle SP, Obs\rangle = \langle\langle Sig_{SP}, \Theta_{SP}\rangle, Obs\rangle$$

where $Sig_{SP} = \exists X.Prof_{SP}$,
where $Prof_{SP} = Record(f_1{:}s_{11}{\times}\cdots{\times}s_{1n_1} \to s_1,\ \ldots\ , f_k{:}s_{k1}{\times}\cdots{\times}s_{kn_k} \to s_k)[X/b]$,
for $f_i{:}s_{i1} \times \cdots \times s_{in_i} \to s_i \in \Omega$,
and where $\Theta_{SP}(u) = \exists X.\exists \mathfrak{x}{:}Prof_{SP}\ .\ u = (\mathsf{pack} X \mathfrak{x}) \ \wedge\ \Phi[X, \mathfrak{x}]$.

Here, $\Phi[X, \mathfrak{x}]$ indicates the conjunction of Φ, where X substitutes b, and every operator symbol in Φ belonging to Ω is prefixed with $\mathfrak{x}$. We call $\mathcal{T}\langle SP, Obs\rangle$ a type theory specification. *If $\Theta_{SP}(u)$ is derivable then u is a* realisation *of $\mathcal{T}\langle SP, Obs\rangle$.*

Example 5. For example, $\mathcal{T}\langle \mathsf{Stack}, \{\mathsf{nat}\}\rangle = \langle\langle Sig_{\mathsf{Stack}}, \Theta_{\mathsf{Stack}}\rangle, \{\mathsf{nat}\}\rangle$, where

$$Sig_{\mathsf{Stack}} = \exists X.Prof_{\mathsf{Stack}},$$
$$Prof_{\mathsf{Stack}} = Record(\mathsf{empty}{:}X, \mathsf{push}{:}\,\mathsf{nat} \times X \to X, \mathsf{pop}{:}X \to X, \mathsf{top}{:}X \to \mathsf{nat})$$
$$\begin{aligned}\Theta_{\mathsf{Stack}}(u) = \exists X.\exists \mathfrak{x}{:}Prof_{\mathsf{Stack}}\ .\ u = (\mathsf{pack} X \mathfrak{x})\ \wedge&\\ \forall x{:}\mathsf{nat}, s{:}X\ .\ \mathfrak{x}.\mathsf{pop}(\mathfrak{x}.\mathsf{push}(x, s)) = s\ \ \wedge&\\ \forall x{:}\mathsf{nat}, s{:}X\ .\ \mathfrak{x}.\mathsf{top}(\mathfrak{x}.\mathsf{push}(x, s)) = x&\end{aligned}$$

○

Henceforth, existential types arise from algebraic specifications as in Def. 1. We do not consider free type variables in existential types since this corresponds to parameterised algebraic specifications, which is outside this paper's scope.

The type theory specification of Def. 1 is essentially that of [17]. The important difference is that with parametricity, equality of data type inhabitants is inherently behavioural, so implementation is up to observational equivalence.

In algebraic specification we said that two Σ-algebras A and B are observationally equivalent w.r.t. Obs and In iff for any observable computation $t \in T_\Sigma(X^{In})_s$, $s \in Obs$ the interpretations t^A and t^B are equivalent. Analogously, and in the style of [21], we give the following definition of type-theoretic observational equivalence.

Definition 2 (Type Theory Observational Equivalence). *For any $u, v\colon \exists X.T[X]$, we say u and v are* observationally equivalent w.r.t. *Obs iff the following sequent is derivable.*

$$\vdash_\Gamma \exists A, B.\exists \mathfrak{a}\colon T[A], \mathfrak{b}\colon T[B] \;.\; u = (\mathsf{pack} A \mathfrak{a}) \;\wedge\; v = (\mathsf{pack} B \mathfrak{b}) \;\wedge \\ \textstyle\bigwedge_{C \in Obs} \forall f\colon \forall X.(T[X] \to C) \;.\; (f A\, \mathfrak{a}) = (f B\, \mathfrak{b})$$

Notice that there is nothing hindering having free variables in an observable computation $f\colon \forall X.(T[X] \to C)$. Importantly, though, these free variables can not be of the existentially bound type.

Example 6. Recalling Example 2, for *ListImpl* to be a behavioural implementation of Set, it was essential that the input sorts did not include set, as then the observable computation $\mathsf{in}(x, \mathsf{remove}(x, s))$ would not have the same denotation in *ListImpl* as in any algebra in $[\![\mathsf{Set}]\!]$. In our type-theoretic setting, the corresponding observable computation is $\Lambda X.\lambda \mathfrak{x}\colon Prof_{\mathsf{Set}} \;.\; \mathfrak{x}.\mathsf{in}(x, \mathfrak{x}.\mathsf{remove}(x, g))$. Here g must be a term of the bound type X. The typing rules insist that g can only be of the form $\mathfrak{x}.\mathsf{add}(\cdots \mathfrak{x}.\mathsf{add}(\mathfrak{x}.\mathsf{empty}) \cdots)$ and not a free variable. ○

Our first result is essential to understanding the translation.

Theorem 3. *Suppose $\exists X.T[X] = Sig_{SP}$ in $\mathcal{T}\langle SP, Obs\rangle$ for some basic algebraic specification SP and set of observable sorts Obs. Then, assuming* Param, *equality at existential type is derivably equivalent to observational equivalence, i.e. the following sequent is derivable in the logic.*

$$\vdash_\emptyset \forall u, v\colon \exists X.T[X] \;. \\ u =_{\exists X.T[X]} v \quad \Leftrightarrow \\ \exists A, B.\exists \mathfrak{a}\colon T[A], \mathfrak{b}\colon T[B] \;.\; u = (\mathsf{pack} A \mathfrak{a}) \;\wedge\; v = (\mathsf{pack} B \mathfrak{b}) \;\wedge \\ \textstyle\bigwedge_{C \in Obs} \forall f\colon \forall X.(T[X] \to C) \;.\; (f A\, \mathfrak{a}) = (f B\, \mathfrak{b})$$

Proof: This follows from Fact 2 and Lemma 4 below. □

Lemma 4. *Let $\exists X.T[X] = Sig_{SP}$ be as in Theorem 3. Then, assuming* Param, *the existence of a simulation relation is derivably equivalent to observational equivalence, i.e. the following sequent is derivable.*

$$\vdash_\emptyset \forall A, B.\forall \mathfrak{a}\colon T[A], \mathfrak{b}\colon T[B] \;. \\ \exists R \subset A \times B \;.\; \mathfrak{a}(T[R])\mathfrak{b} \quad \Leftrightarrow \quad \textstyle\bigwedge_{C \in Obs} \forall f\colon \forall X.(T[X] \to C) \;.\; (f A\, \mathfrak{a}) = (f B\, \mathfrak{b})$$

Proof: $\Rightarrow$: This follows from PARAM.

$\Leftarrow$: We must exhibit an R such that $\mathfrak{a}(T[R])\mathfrak{b}$. Semantically, [22, 33] define a relation between elements iff they are denotable by some common term. We mimic this: Give $R \overset{def}{=} (a{:}A, b{:}B).(\exists f{:}\forall X.(T[X] \to X).(f A\, \mathfrak{a}) = a \wedge (f B\, \mathfrak{b}) = b)$. We must now derive $\mathfrak{a}(T[R])\mathfrak{b}$, *i.e.* for every component $(g{:}\, s_1 \times \cdots \times s_n \to s)[X/b]$ in $T[X]$, we must show that

$$\begin{aligned} &\forall v_1{:}\, s_1[A], \dots, \forall v_n{:}\, s_n[A], \forall w_1{:}\, s_1[B], \dots, \forall w_n{:}\, s_n[B]\ .\\ &\qquad\qquad v_1\ s_1[R]\ w_1\ \wedge\ \cdots\ \wedge\ v_n\ s_n[R]\ w_n\\ &\qquad\qquad\qquad \Rightarrow\ \mathfrak{a}.g(v_1, \dots, v_n)\ s[R]\ \mathfrak{b}.g(w_1, \dots, w_n) \end{aligned}$$

Under our present assumptions, any s_j in the antecedent is either b or else an observable sort. If s_j is b then the antecedent says $v_j\ R\ w_j$ hence we may assume $\exists f_j{:}\forall X.(T[X] \to X).(f_j A\, \mathfrak{a}) = v_j\ \wedge\ (f_j B\, \mathfrak{b}) = w_j$. If s_j is an observable sort we may by Fact 1 assume $v_j = w_j$. Consider $f \overset{def}{=} \Lambda X.\lambda \mathfrak{x}{:}\, T[X]\ .\ \mathfrak{x}.g(u_1, \dots, u_n)$, where u_j is $(f_j X \mathfrak{x})$ if s_j is b, and $u_j = v_j$ otherwise.

Suppose now the co-domain sort s is an observable sort. Then by assumption we have $(f A\, \mathfrak{a}) = (f B\, \mathfrak{b})$ and by β-reduction we are done. Suppose the co-domain sort s is b. Then we need to derive $\mathfrak{a}.g(v_1, \dots, v_n)\ R\ \mathfrak{b}.g(w_1, \dots, w_n)$, *i.e.* that $\exists f{:}\forall X.(T[X] \to X).(f A\, \mathfrak{a}) = \mathfrak{a}.g(v_1, \dots, v_n)\ \wedge\ (f B\, \mathfrak{b}) = \mathfrak{b}.g(w_1, \dots, w_n)$. But then we exhibit our f above, and we are done. □

This proof does not in general generalise to higher order T. The problem lies in exhibiting a simulation relation R.

Given Theorem 3, $\Theta_{SP}(u)$ of translation $\mathcal{T}$ expresses "u is observationally equivalent to a package $(\mathsf{pack} X \mathfrak{x})$ that satisfies the axioms Φ". Therefore:

Definition 3 (Type Theory Specification Refinement). *A type theory specification $\mathcal{T}\langle SP', Obs'\rangle$ is a* refinement *of a type theory specification $\mathcal{T}\langle SP, Obs\rangle$,* with constructor $F{:}\, Sig_{SP'} \to Sig_{SP}$ *iff* $\vdash_\Gamma \forall u{:}\, Sig_{SP'}\ .\ \Theta_{SP'}(u)\ \Rightarrow\ \Theta_{SP}(Fu)$ *is derivable. We write $\mathcal{T}\langle SP, Obs\rangle \overset{\mathcal{T}}{\underset{F}{\rightsquigarrow}} \mathcal{T}\langle SP', Obs'\rangle$ for this fact.*

Any constructor $F{:}\, Sig_{SP'} \to Sig_{SP}$ is by Theorem 3 inherently stable under parametricity: Congruence gives $\forall u, v{:}\, Sig_{SP'}.u =_{Sig_{SP'}} v\ \Rightarrow\ F(u) =_{Sig_{SP}} F(v)$. But equality at existential type is of course observational equivalence.

Example 7. The constructor *Tr* of Example 3 is expressed in this setting as $\lambda u{:}\, Sig_{\mathsf{Stack}}.\mathsf{unpack}(u)(Sig_{\mathsf{Triv}})(\Lambda X.\lambda\mathfrak{x}{:}\, Prof_{\mathsf{Stack}}\ .\ (\mathsf{pack} X\ \mathit{record}(\mathsf{id} =$
$\lambda x, n, z{:}\, \mathsf{nat}\ .\ \mathfrak{x}.\mathsf{top}(\mathsf{multipop}(n, \mathsf{multipush}(n, z, \mathfrak{x}.\mathsf{push}(x, \mathfrak{x}.\mathsf{empty})))))))$ ○

Note that we automatically get the proof simplification due to stability that we have in algebraic specification. 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 4 is not expressible in the type theory, because $x =_X y$ *Prop* is not allowed in System F terms.

5 A Correspondence at First Order

We seek to establish a formal connection between the concept of algebraic specification refinement and its type-theoretic counterpart as defined in Def. 3, *i.e.*

$$\langle SP, Obs\rangle \underset{\kappa}{\rightsquigarrow} \langle SP', Obs'\rangle \;\Leftrightarrow\; \mathcal{T}\langle SP, Obs\rangle \overset{\mathcal{T}}{\underset{F_\kappa}{\rightsquigarrow}} \mathcal{T}\langle SP', Obs'\rangle$$

where κ and F_κ are constructors that correspond in a sense given below.

Now, that u is a realisation of a type theory specification $\langle\langle Sig_{SP}, \Theta_{SP}\rangle, Obs\rangle$ can in general only be proven by exhibiting an observationally equivalent package u' that satisfies Φ_{SP}. For any particular closed term $g\colon Sig_{SP}$, 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 $\mathfrak{a}$. We therefore need a universal method for exhibiting suitable observationally equivalent packages. It also defies the point of behavioural abstraction having to construe a literal implementation to justify a behavioural one.

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 [5], 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 [6, 4]. The quotients are of the form $dom_A(\approx_{Obs,In})/\approx_{Obs,In}$, where $dom_A(\approx_{Obs,In})_s \overset{def}{=} \{a \in A_s \mid a \approx_{Obs,In} a\}$.

This proof method is not available in the type theory and logic of [24]. One remedy would be to augment the type theory by quotient types, *e.g.* [11], and subset types. However, for its simplicity and because it complies to existing proof techniques in algebraic specification, we adapt an idea from [25] where the logic is augmented with an axiom schema postulating the existence of quotients (Def. 4). In addition, we need a schema asserting the existence of sub-objects (Def. 5) for dealing with partial congruences. The justification for these axioms lies in their soundness w.r.t. the parametric PER-model [3] that is one justification for the logic of [24]. These axioms are tailored to suit refinement proof purposes. One could alternatively derive them from more fundamental and general axioms.

Definition 4 (Existence of Quotients (Quot) [25]**).**

$$\begin{array}{l}
\vdash_{\emptyset} \forall X.\forall \mathfrak{x}{:}T[X].\forall R\subset X\times X\;.\quad (\mathfrak{x}\; T[R]\; \mathfrak{x} \;\wedge\; equiv(R)) \;\Rightarrow\\
\quad \exists Q.\exists \mathfrak{q}{:}T[Q].\exists epi{:}X\to Q\;.\; \forall x,y{:}X\;.\; xRy \;\Leftrightarrow\; (epi\,x) =_Q (epi\,y) \;\wedge\\
\qquad\qquad\qquad\qquad\qquad\qquad \forall q{:}Q.\exists x{:}X\;.\; q =_Q (epi\,x) \;\wedge\\
\qquad\qquad\qquad\qquad\qquad\qquad \mathfrak{x}\;(T[(x{:}X, q{:}Q).((epi\,x) =_Q q)])\;\mathfrak{q}
\end{array}$$

where $equiv(R)$ specifies R to be an equivalence relation.

Definition 5 (Existence of Sub-objects (Sub)**).**

$$\begin{array}{l}
\vdash_{\emptyset} \forall X.\forall \mathfrak{x}{:}T[X].\forall R\subset X\times X\;.\quad (\mathfrak{x}\; T[R]\; \mathfrak{x}) \;\Rightarrow\\
\exists S.\exists \mathfrak{s}{:}T[S].\exists R'\subset S\times S.\exists mono{:}S\to X\;.\mathfrak{x}\;(T[(x{:}X, s{:}S).(x =_X (mono\,s))])\;\mathfrak{s} \;\wedge\\
\qquad\qquad\qquad\qquad \forall s.s'{:}S\;.\; s\;R'\;s' \;\Leftrightarrow\; (mono\,s)\;R\;(mono\,s') \;\wedge\\
\qquad\qquad\qquad\qquad\qquad \forall s{:}S\;.\; s\;R'\;s
\end{array}$$

Algebraic specification uses classical logic, while the logic in [24] 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 [24]. For our comparison with algebraic specification, we shall do this.

We can now show our desired correspondence. We first do this for refinements without constructors. We must assume that specifications are *behaviourally closed w.r.t.* $\approx_{Obs,In}$, *i.e.* $\{dom_A(\approx_{Obs,In})/\!\approx_{Obs,In} \mid A \in [\![SP]\!]\} \subseteq [\![SP]\!]$. This is methodologically an obvious requirement for behavioural specification [6, 5].

Theorem 5. *Let* $SP = \langle \Sigma, \Phi \rangle$ *and* $SP' = \langle \Sigma, \Phi' \rangle$ *be basic algebraic specifications, with* $\Sigma = \langle S, \Omega \rangle$. *Assume one behavioural sort* b, *and assume* $Obs = In = S \setminus b$. *Assume behavioural closedness w.r.t.* $\approx_{Obs,In}$. *Then*

$$\langle SP, Obs \rangle \leadsto \langle SP', Obs \rangle \;\Leftrightarrow\; \mathcal{T}\langle SP, Obs \rangle \overset{\mathcal{I}}{\leadsto} \mathcal{T}\langle SP', Obs \rangle$$

Proof: $\Rightarrow$: We must show the derivability of $\vdash_\Gamma \forall u{:}Sig_{SP'} \,.\, \Theta_{SP'}(u) \;\Rightarrow\; \Theta_{SP}(u)$. We can obtain proof-theoretical information from $\langle SP, Obs \rangle \leadsto \langle SP', Obs \rangle$. By behavioural closedness, there exists a sound and complete calculus $\vdash_{\Pi_\leadsto}$ for behavioural refinement, based on a calculus $\vdash_{\Pi_S}$ for structured specifications [6]. By syntax directedness, we must have had $SP'/\!\approx_{Obs,In} \vdash_{\Pi_S} \Phi$, where the semantics of $SP'/\!\approx_{Obs,In}$ is $\{dom_A(\approx_{Obs,In})/\!\approx_{Obs,In} \mid A \in [\![SP']\!]\}$. For our basic specification case, this boils down to the predicate logic statement of

$$\Phi', Ax(\sim) \;\vdash\; \mathcal{L}(\Phi) \qquad (\dagger)$$

Here $\sim$ stands for a new symbol representing $\approx_{Obs,In}$ at the behavioural sort b, and $\mathcal{L}(\Phi) \overset{def}{=} \{\mathcal{L}(\phi) \mid \phi \in \Phi\}$, 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) $(u =_b v)^* \overset{def}{=} u \sim v$,
(b) $(\neg\phi)^* \overset{def}{=} \neg(\phi^*)$ and $(\phi \;\wedge\; \psi)^* \overset{def}{=} \phi^* \;\wedge\; \psi^*$,
(c) $(\forall x{:}b.\phi)^* \overset{def}{=} \forall x{:}b.(x \sim x \;\Rightarrow\; \phi^*)$,
(d) $\phi^* \overset{def}{=} \phi$, otherwise.

and $Ax(\sim) \overset{def}{=} \forall x, y{:}\, b.(x \sim y \;\Leftrightarrow\; Beh_b(x, y))$, where $Beh_b(x, y)$ is an axiomatisation of $\approx_{Obs,In}$ at b [4]. (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 T denote $Prof_{SP'} (= Prof_{SP})$. We must derive $\exists B.\exists \mathfrak{b}{:}T[B].(\mathsf{pack}B\mathfrak{b}) = u \wedge \Phi[B, \mathfrak{b}]$ assuming $\exists A.\exists \mathfrak{a}{:}T[A].(\mathsf{pack}A\mathfrak{a}) = u \wedge \Phi'[A, \mathfrak{a}]$. Let $\mathfrak{a}$ and A denote the witnesses projected out from that assumption.

Now, *Beh* is in general infinitary. However, with higher-order logic one gets a finitary Beh^* equivalent to *Beh* [12]. Thus we form $\sim$ type-theoretically by $\sim \overset{def}{=} (a{:}\, A, a'{:}\, A).(Beh^*_A(a, a'))$. Since $\sim$ is an axiomatisation of a partial congruence, we have $\mathfrak{a}\; T[\sim]\; \mathfrak{a}$. We use SUB to get S_A, $\mathfrak{s}_\mathfrak{a}$ and $\sim' \subset S_A \times S_A$ and $mono{:}S_A \to A$ s.t. we can derive

$(s1)$ $\mathfrak{a}\; (T[(a{:}A, s{:}S_A).(a =_A (mono\, s))])\; \mathfrak{s}_\mathfrak{a}$
$(s2)$ $\forall s.s'{:}S_A \,.\, s \sim' s' \;\Leftrightarrow\; (mono\, s) \sim (mono\, s')$
$(s3)$ $\forall s{:}S_A \,.\, s \;\sim'\; s$

By $(s2)$ we get $\mathfrak{s}_{\mathfrak{a}}\ T[\sim']\ \mathfrak{s}_{\mathfrak{a}}$. We also get $equiv(\sim')$ by $(s3)$. We now use QUOT to get Q and $\mathfrak{q}{:}T[Q]$ and $epi{:}S_A \to Q$ s.t.

$(q1)\ \forall s, s'{:}S_A\ .\ s \sim' s' \ \Leftrightarrow\ (epi\ s) =_Q (epi\ s')$
$(q2)\ \forall q{:}Q.\exists s{:}S_A\ .\ q =_Q (epi\ s)$
$(q3)\ \mathfrak{s}_{\mathfrak{a}}\ (T[(s{:}S_A, q{:}Q).((epi\ s) =_Q q)])\ \mathfrak{q}$

We exhibit Q for B, and $\mathfrak{q}$ for $\mathfrak{b}$; it remains to derive 1. $(\mathsf{pack}Q\mathfrak{q}) = (\mathsf{pack}A\mathfrak{a})$ and 2. $\Phi[Q, \mathfrak{q}]$. To show the derivability of (1), it suffices to observe that, through Fact 2, $(s1)$ 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 $\phi \in \Phi$. We induce on the structure of ϕ.

(a) 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. $(epi\ s_{q_i}) = q_i$. From (†) we can derive $\wedge_i((mono\ s_{q_i}) \sim (mono\ s_{q_i})) \Rightarrow u[\mathfrak{a}][\cdots(mono\ s_{q_i})\cdots] \sim v[\mathfrak{a}][\cdots(mono\ s_{q_i})\cdots]$, but by $(s2)$ and $(s1)$ 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 $(s3)$ 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 $(epi\ u[\mathfrak{s}_{\mathfrak{a}}][\cdots s_{q_i}\cdots]) =_Q (epi\ v[\mathfrak{s}_{\mathfrak{a}}][\cdots s_{q_i}\cdots])$. By $(q3)$ we then get $(epi\ u[\mathfrak{s}_{\mathfrak{a}}][\cdots s_{q_i}\cdots]) = u[\mathfrak{q}]$ and $(epi\ v[\mathfrak{s}_{\mathfrak{a}}][\cdots s_{q_i}\cdots]) = v[\mathfrak{q}]$.

(b) Suppose $\phi = \neg\phi'$. By negation n.f. convertibility it suffices to consider ϕ' an atomic formula. The case for ϕ' as (a) warrants a proof for $\neg\phi'$ similar to that of (a). Suppose $\phi = \phi' \ \wedge\ \phi''$. This is dealt with by i.h. on ϕ' and ϕ''.

(c) $\phi = \forall x{:}B.\phi'$. This is dealt with by i.h. on ϕ'.

(d) This covers the remaining cases. Proofs are similar to those above.

$\Leftarrow$: Observe that to show $\Phi[Q, \mathfrak{q}]$ we must either use $\Phi'[Q, \mathfrak{q}]$ and the definition of $\sim$, or else $\Phi[Q, \mathfrak{q}]$ was a tautology; in both cases we get (†). $\square$

We can easily extend Theorem 5 to deal with constructors. Dealing with constructors in full generality, requires specification building operators, which is outside the scope of this paper. However, consider *simple* type theory constructors $F{:}Sig_{SP'} \to Sig_{SP}$ of the form

$$\lambda u{:}Sig_{SP'}.\mathsf{unpack}(u)(Sig_{SP})(\Lambda X.\lambda\mathfrak{x}{:}Prof_{SP'}[X]\ .\ (\mathsf{pack}X\mathfrak{x}'))$$

for some $\mathfrak{x}' : Prof_{SP}[X]$. The concept of algebraic specification of algebras is extended in [28] to algebraic specifications of constructors. In the simple case, we can extend our translation in Def. 1 to this framework.

Example 8. An algebraic specification of Example 3's *Tr*, can be given by $\Pi S{:}\mathsf{Stack}.\mathsf{Triv}'[S]$ where $\mathsf{Triv}'[S]$ is

hide multipush, multipop **in**
operators multipush: nat × nat × S.stack → S.stack,
  multipop: nat × S.stack → S.stack, id: nat × nat × nat → nat
axioms Φ_{Tr} : multipop(n, multipush(n, z, s)) = s
  id(x, n, z) = S.top(multipop(n, multipush(n, z, S.push(x, S.empty))))

We can give a corresponding type theory specification $\mathcal{T}(\Pi S{:}\mathsf{Stack}.\mathsf{Triv}')$ by $\langle Sig_{\Pi S:\mathsf{Stack}.\mathsf{Triv}'}, \Theta_{\Pi S:\mathsf{Stack}.\mathsf{Triv}'}\rangle$, where $Sig_{\Pi S:\mathsf{Stack}.\mathsf{Triv}'} \overset{def}{=} Sig_{\mathsf{Stack}} \to Sig_{\mathsf{Triv}}$ and

$$\begin{aligned}\Theta_{\Pi S:\mathsf{Stack}.\mathsf{Triv}'}(u,v) \overset{def}{=} {}& \exists X.\exists \mathfrak{x}{:}\, \mathit{Prof}_{\mathsf{Stack}}.\exists Y.\exists \mathfrak{y}{:}\, \mathit{Prof}_{\mathsf{Triv}} \,.\\ & (\mathsf{pack}X\mathfrak{x}) = u \quad\wedge\quad (\mathsf{pack}Y\mathfrak{y}) = v \quad\wedge\ \ldots\ \wedge\\ \forall x,n,z{:}\,\mathsf{nat}\,.\ \mathfrak{y}.\mathsf{id}(x,n,z) = {}& \mathfrak{x}.\mathsf{top}(\mathsf{multipop}(n,\mathsf{multipush}(n,z,\mathfrak{x}.\mathsf{push}(x,\mathfrak{x}.\mathsf{empty}))))\end{aligned}$$

whereby F is a realisation of, or satisfies, $\mathcal{T}(\Pi S{:}\,\mathsf{Stack}.\mathsf{Triv}')$ if one can derive $\forall u{:}\, \mathit{Sig}_{\mathsf{Stack}}\,.\ \Theta_{\mathsf{Stack}}(u) \ \Rightarrow\ \Theta_{\Pi S:\mathsf{Stack}.\mathsf{Triv}'}(u, Fu)$. ○

We now want to show $\langle SP, \mathit{Obs}\rangle \underset{\kappa_F}{\rightsquigarrow} \langle SP', \mathit{Obs}'\rangle \ \Leftrightarrow\ \mathcal{T}\langle SP, \mathit{Obs}\rangle \overset{\mathcal{T}}{\underset{F}{\rightsquigarrow}} \mathcal{T}\langle SP', \mathit{Obs}'\rangle$ where κ_F is a realisation of a specification SP_F that maps to a specification $\mathcal{T}(SP_F)$ for which F, a simple constructor, is a realisation, and where the axioms of SP_F and $\mathcal{T}(SP_F)$ are given by Φ_F. We have to show the derivability of $\vdash_\Gamma\ \forall u : \mathit{Sig}_{SP'}\ .\ \Theta_{SP'}(u) \ \Rightarrow\ \Theta_{SP}(Fu)$, supposing $\langle SP, \mathit{Obs}\rangle \underset{\kappa_F}{\rightsquigarrow} \langle SP', \mathit{Obs}'\rangle$. Similarly to the proof of Theorem 5, we get $\Phi', Ax(\sim), \Phi_F \ \vdash\ \mathcal{L}(\Phi)$ (‡) We need to exhibit a B and $\mathfrak{b}$ s.t. 1. $\mathsf{pack}B\mathfrak{b} = F(\mathsf{pack}A\mathfrak{a})$ and 2. $\Phi[B, \mathfrak{b}]$. We construct Q and $\mathfrak{q}$ from $F(\mathsf{pack}A\mathfrak{a}) = \mathsf{pack}A\mathfrak{a}'$, for $\mathfrak{a}'{:}\, \mathit{Prof}_{SP}[A]$, as in the proof of Theorem 5, and (1) follows as before. Then for (2), to show $\Phi[Q, \mathfrak{q}]$, and also for the converse direction, use (‡) in place of (†).

Finally and importantly, the '⇒' direction of the proof of Theorem 5 displays a reasoning technique in its own right for type theory specification refinement. This extends the discussion in [25] to deal also with partial congruences.

6 Final Remarks

In this paper we have expressed an account of algebraic specification refinement in System F and the logic for parametric polymorphism of [24]. We have seen in Sect. 4 how the concepts of behavioural (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 3). We have shown a correspondence (Theorem 5) between refinement in the algebraic specification sense, and a notion of type theory specification refinement (Def. 3). We have seen how a proof technique from algebraic specification can be mirrored in type theory by extending the logic soundly with axioms Quot and Sub, the latter also extending the discussion in [25].

The stage is now set for type-theoretic development in at least two 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. This would also allow a full account of specifications of parameterised programs and also parameterised specifications [28].

Secondly, we can use our notion of type theory specification refinement and start looking at specification refinement for higher-order polymorphic functionals. In this context one must resolve what observational equivalence means, since the higher-order version of Theorem 3 is an open question. However, there are grounds to consider an alternative notion of simulation relation that would re-establish a higher-order version of Lemma 4. Operationally, the only way two concrete data types $(\mathsf{pack}A\mathfrak{a}) : T[A]$ and $(\mathsf{pack}B\mathfrak{b}) : T[B]$ can be utilised, is in clients of the form $\Lambda X.\lambda\mathfrak{x}{:}T[X]\ .\ \mathfrak{x}.t$. Such a client cannot incite the application of functionals $\mathfrak{a}.f$ and $\mathfrak{b}.f$ whose domain types involve the instantiations A and B of the existential type variable, to arbitrary terms of appropriate instantiated types, but only to terms definable in some sense by items in the respective implementations $\mathfrak{a}$ and $\mathfrak{b}$. However, the usual notion of simulation relation considers in fact arbitrary terms. At first order, this does not matter, because one can exhibit a relation that explicitly restricts arguments to be definable, namely the relation R in the proof of Lemma 4. At higher-order, one could try altering the relational proof criteria by incorporating explicit definability clauses. This is reminiscent of recent approaches on the semantical level [15, 13].

Acknowledgements Thanks are due to Martin Hofmann, Don Sannella, and the referees for helpful comments and suggestions. This research has been supported by EPSRC grant GR/K63795, and NFR (Norwegian Research Council) grant 110904/41.

References

1. M. Abadi, L. Cardelli, and P.-L. Curien. Formal parametric polymorphism. *Theoretical Computer Science*, 121:9–58, 1993.
2. D. Aspinall. *Type Systems for Modular Programs and Specifications.* PhD thesis, University of Edinburgh, 1998.
3. E.S. Bainbridge, P.J. Freyd, A. Scedrov, and P.J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70:35–64, 1990.
4. M. Bidoit and R. Hennicker. Behavioural theories and the proof of behavioural properties. *Theoretical Computer Science*, 165:3–55, 1996.
5. M. Bidoit, R. Hennicker, and M. Wirsing. Behavioural and abstractor specifications. *Science of Computer Programming*, 25:149–186, 1995.
6. M. Bidoit, R. Hennicker, and M. Wirsing. Proof systems for structured specifications with observability operators. *Theoretical Computer Sci.*, 173:393–443, 1997.
7. M. Bidoit, H.-J. Kreowski, P. Lescanne, F. Orejas, and D. Sannella (eds.). *Algebraic System Specification and Development: A Survey and Annotated Bibliography*, volume 501 of *LNCS*. Springer, 1991.
8. C. Böhm and A. Beraducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.
9. J.A. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, SE-10(5):528–543, 1984.
10. R. Hennicker. Structured specifications with behavioural operators: Semantics, proof methods and applications. Habilitationsschrift, LMU, München, 1997.
11. M. Hofmann. A simple model for quotient types. In *Proc. TLCA'95*, volume 902 of *LNCS*, pages 216–234. Springer, 1995.
12. M. Hofmann and D. Sannella. On behavioural abstraction and behavioural satisfaction in higher-order logic. *Theoretical Computer Science*, 167:3–45, 1996.
13. F. Honsell and D. Sannella. Pre-logical relations. In *Proc. CSL'99*, *LNCS*, 1999.
14. S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML: a gentle introduction. *Theoretical Computer Science*, 173:445–484, 1997.
15. 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 *Proceedings of TACS'97*, volume 1281 of *LNCS*, pages 191–212. Springer, 1997.

16. H. Kirchner and P.D. Mosses. Algebraic specifications, higher-order types, and set-theoretic models. In *Proc. AMAST'98*, volume 1548 of *LNCS*, pages 378–388. Springer, 1998.
17. Z. Luo. Program specification and data type refinement in type theory. *Math. Struct. in Comp. Sci.*, 3:333–363, 1993.
18. Q. Ma and J.C. Reynolds. Types, abstraction and parametric polymorphism, part 2. In *Proc. 7th MFPS*, volume 598 of *LNCS*, pages 1–40. Springer, 1991.
19. H. Mairson. Outline of a proof theory of parametricity. In *ACM Symposium on Functional Programming and Computer Architecture*, volume 523 of *LNCS*, pages 313–327. Springer, 1991.
20. K. Meinke. Universal algebra in higher types. *Theoretical Computer Science*, 100:385–417, 1992.
21. 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.
22. J.C. Mitchell. *Foundations for Programming Languages.* Foundations of Computing Series. MIT Press, 1996.
23. N. Mylonakis. Behavioural specifications in type theory. In *Recent Trends in Data Type Spec., 11th WADT*, volume 1130 of *LNCS*, pages 394–408. Springer, 1995.
24. G. Plotkin and M. Abadi. A logic for parametric polymorphism. In *Proc. of TLCA 93*, volume 664 of *LNCS*, pages 361–375. Springer, 1993.
25. E. Poll and J. Zwanenburg. A logic for abstract data types as existential types. In *Proc. TLCA'99*, volume 1581 of *LNCS*, pages 310–324, 1999.
26. B. Reus and T. Streicher. Verifying properties of module construction in type theory. In *Proc. MFCS'93*, volume 711 of *LNCS*, pages 660–670, 1993.
27. J.C. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing*, 83:513–523, 1983.
28. D. Sannella, S. Sokołowski, and A. Tarlecki. Toward formal development of programs from algebraic specifications: parameterisation revisited. *Acta Inform.*, 29:689–736, 1992.
29. D. Sannella and A. Tarlecki. On observational equivalence and algebraic specification. *Journal of Computer and System Sciences*, 34:150–178, 1987.
30. D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: Implementations revisited. *Acta Inform.*, 25(3):233–281, 1988.
31. D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9:229–269, 1997.
32. D. Sannella and M. Wirsing. A kernel language for algebraic specification and implementation. In *Proc. 1983 Intl. Conf. on Foundations of Computation Theory*, volume 158 of *LNCS*, pages 413–427. Springer, 1983.
33. O. Schoett. *Data Abstraction and the Correctness of Modular Programming.* PhD thesis, University of Edinburgh, 1986.
34. T. Streicher and M. Wirsing. Dependent types considered necessary for specification languages. In *Recent Trends in Data Type Spec.*, volume 534 of *LNCS*, pages 323–339. Springer, 1990.
35. J. Underwood. Typing abstract data types. In *Recent Trends in Data Type Spec., Proc. 10th WADT*, volume 906 of *LNCS*, pages 437–452. Springer, 1994.
36. M. Wirsing. Structured specifications: Syntax, semantics and proof calculus. In *Logic and Algebra of Specification*, pages 411–442. Springer, 1993.
37. M. Wirsing. Algebraic specification languages: An overview. In *Recent Trends in Data Type Specification*, volume 906 of *LNCS*, pages 81–115. Springer, 1994.