A Logic for Abstract Data Types as Existential Types

Erik Poll[1] and Jan Zwanenburg[2]

[1] E.Poll@ukc.ac.uk
Computing Lab, University of Kent at Canterbury, England
[2] janz@win.tue.nl
Eindhoven University of Technology, The Netherlands

Abstract. The second-order lambda calculus allows an elegant formalisation of abstract data types (ADT's) using existential types. Plotkin and Abadi's logic for parametricity [PA93] then provides the useful proof principle of *simulation* for ADT's, which can be used to show equivalence of data representations. However, we show that this logic is not sufficient for reasoning about specifications of ADT's, and we present an extension of the logic that does provide the proof principles for ADT's that we want.

1 Introduction

The second-order lambda calculus allows an elegant formalisation of abstract data types (ADT's), as shown in [MP88], using existential types. This description of ADT's provides a useful basis to investigate properties of ADT's. In particular, it has been successfully used to investigate a notion of equivalence of implementations of ADT's. [Mit91] considers a semantic notion of equivalence of data representations, which suggests a method for proving the equivalence of data representations, namely by showing that there exists a simulation relation between the representations. We will refer to this proof principle as **simulation**. Plotkin and Abadi's logic for parametricity [PA93] is a logic for reasoning about the second order lambda calculus (system F). It formalises the notion of *parametricity*, and for the existential types this logic does indeed provide the proof principle of simulation envisaged in [Mit91].

Unfortunately, it turns out that this proof principle of simulation for existential types is not enough for reasoning about specifications of ADT's, in particular specifications that use equality. We propose an extension of the logic of [PA93] (with axioms stating the existence of quotients, to be precise) that does provide all the proof principles one would like for reasoning about ADT's. The same PER model used in [PA93] as a semantics for their logic immediately justifies these additional axioms. (Indeed, in the PER model all types are "quotient types".)

The remainder of this introduction discusses one of the proof principles we want for ADT's. It is a very natural one, that immediately arises whenever an implementation of an ADT allows different concrete representations of the same abstract value. This example will be treated in more detail later in Section 4.

Suppose we implement an ADT for bags using lists to represents bags. Then there will be many different lists that represent the same bag: any two lists that are permutations represent the same bag. As a consequence, there are *different notions of equality* in play: equality of lists, equality of bags, and the relation $\sim_{perm}$ on lists that relates lists representing the same bag (i.e. that are permutations). A programmer implementing an ADT has to be aware of the fact that there are these different notions of equality. But a programmer using an ADT should only have to deal with equality of bags, and not have to know anything about an underlying relation $\sim_{perm}$ on lists. Indeed, this is precisely the *abstraction* that an *abstract* data type is supposed to provide. A consequence of all is that the programmer implementing an ADT and the programmer using an ADT may want to use a slightly different specification: the former in terms of the relation $\sim_{perm}$ on the concrete data type of lists, the latter in terms of equality on the abstract data type of bags. For instance, the programmer using the ADT might require that

$$\forall m, n : \mathit{Nat}, s : \mathit{Bag}.\ \mathit{add}(m, \mathit{add}(n, s)) = \mathit{add}(n, \mathit{add}(m, s)) \qquad \text{(i)}$$

and to meet this specification, the programmer implementing the ADT must ensure that

$$\forall m, n : \mathit{Nat}, s : \mathit{List}.\ \mathit{cons}(m, \mathit{cons}(n, s)) \sim_{perm} \mathit{cons}(n, \mathit{cons}(m, s)) \qquad \text{(ii)}$$

if *add* is implemented as *cons*. In a logic for reasoning with (specifications of) ADT's we should be able to relate statements such as (i) and (ii). In particular, here one would want to be able to prove that (ii) implies (i). We will refer to a proof principle that would allow us to deduce (i) from (ii) as **abstraction**.

The logic for parametricity of [PA93] does not quite provide this proof principle of abstraction for arbitrary ADT's and specifications. But extending the logic with axioms stating the existence of quotients solves this problem: we will show that then the proof principle of abstraction can be obtained from the proof principle of simulation, which is provided by the logic for parametricity of [PA93]. (For this particular example, we would want the existence of lists quotiented by $\sim_{perm}$.)

The organisation of this paper is as follows. Section 2 defines our notation for the second-order lambda calculus and gives a quick recap on how existential types can be used for ADT's. Section 3 discusses the logic for parametricity of [Tak97], which is a slightly different formulation of the logic as first introduced in [PA93]; in particular, we discuss the proof principle of simulation for proving equivalence of data representations that this logic provides. Section 4 then considers a simple example of a specification of an ADT for bags and illustrates the problem with reasoning about ADT's hinted at above. Section 5 then present our extension of the logic that does provide the power we want.

2 The second-order lambda calculus

We first give the definition of the second-order lambda calculus, and then illustrate how the existential types can be used for ADT's.

2.1 Definition of the second-order lambda calculus

The *terms* t and *types* T of the second-order lambda calculus are given by the grammar

$$t ::= x \mid \lambda x{:}T.\, t \mid tt \mid (t,t) \mid t.1 \mid t.2 \mid \lambda X.\, t \mid tT \mid \mathsf{pack}\ \langle T,t\rangle\ \mathsf{to}\ T \mid \mathsf{open}\ t\ \mathsf{as}\ \langle T,t\rangle\ \mathsf{in}\ t$$
$$T ::= X \mid T \times T \mid T \to T \mid \forall X.\, T \mid \exists X.\, T$$

Here x ranges over *term-variables*, X over *type-variables*. Free and bound variables are defined as usual. Terms and types equal up to the names of bound variables and permutation of fields are identified.

We use the following convention for our meta-variables: x, y, z range over term variables, X, Y, Z range over type variables, a, b, c, f range over terms (or programs), A, B, C range over types.

We include products and existentials as primitives here because they play an important role later, but of course they can be regarded as syntactic sugar for their usual encodings. (In fact, we will not even need the universal types in this paper.) Later on we will also use some base types, namely a type *Nat* of natural numbers and a type *List* of lists of natural numbers. These can be encoded in the usual way, too.

The type inference rules for judgements of the form $\Gamma \vdash t : T$, where Γ is a sequence of declarations $x_1 : T_1, \ldots, x_n : T_n$, are

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A}$$

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x{:}A.\, b : A \to B} \qquad \frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash fa : B}$$

$$\frac{\Gamma \vdash a_1 : A_1 \quad \Gamma \vdash a_2 : A_2}{\Gamma \vdash (a_1, a_2) : A_1 \times A_2} \qquad \frac{\Gamma \vdash a : A_1 \times A_2}{\Gamma \vdash a.i : A_i}\ i = 1, 2$$

$$\frac{\Gamma \vdash b : B}{\Gamma \vdash \lambda X.\, b : \forall X.\, B}\ X \text{ not free in } \Gamma \qquad \frac{\Gamma \vdash f : \forall X.\, B}{\Gamma \vdash fA : B[A/X]}$$

$$\frac{\Gamma \vdash c : A[C/X]}{\Gamma \vdash (\mathsf{pack}\ \langle C, c\rangle\ \mathsf{to}\ \exists X.\, A) : \exists X.\, A}\ X \text{ not free in } \Gamma$$

$$\frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash s : \exists X.\, A}{\Gamma \vdash (\mathsf{open}\ s\ \mathsf{as}\ \langle X, x\rangle\ \mathsf{in}\ b) : B}\ X \text{ not free in } B \text{ or } \Gamma$$

The reduction rules are

$$\begin{aligned}
(\lambda x{:}A.\, b)a &\rhd_\beta b[a/x] \\
(\lambda X.\, a)A &\rhd_\beta a[A/X] \\
(a_1, a_2).i &\rhd_\beta a_i \\
\mathsf{open}\ (\mathsf{pack}\ \langle C, c\rangle\ \mathsf{to}\ \exists X.\, A)\ \mathsf{as}\ \langle X, x\rangle\ \mathsf{in}\ b &\rhd_\beta b[C/X, c/x]
\end{aligned}$$

Notation. The notation for pairs is extended to n-tuples, which are simply nested pairs. E.g. we write $A \times B \times C$ for $A \times (B \times C)$ and (a, b, c) for $(a, (b, c))$. We typically omit the second type parameter of **pack**, writing $\mathsf{pack}\ \langle C, a\rangle$ for $(\mathsf{pack}\ \langle C, a\rangle\ \mathsf{to}\ \exists X.\, A)$, whenever this type is clear from the context. Finally, we will sometimes use a "pattern-matching" style notation for tuples, e.g. writing $\lambda(y, z){:}A \times B.\, c$ instead of $\lambda x{:}A \times B.\, c[x.1/y, x.2/z]$. □

2.2 Abstract Data Types as Existential Types

Existential types allow an elegant formalisation of abstract data types (ADT's), as shown in [MP88]. This formalisation provides a clean separation between using an ADT on the one hand and implementing an ADT on the other hand. Moreover, as is often the case with descriptions of notions from programming languages in terms of typed lambda calculus, this formalisation provides a more powerful notion than exists in most existing programming languages: existential types provide implementations of ADT's as "first-class citizens", i.e. as values that can be passed as parameters to functions or returned as results like any other value. This also means that we can talk about equality of implementations of ADT's just like we can talk about equality of other values. (This will be useful later, in Section 3, when we consider proof rules for ADT's.)

The remainder of this section briefly explains the use of existential types for ADT's (for a more extensive discussion see [MP88]), and introduces our running example of bags.

Example: bags

Our running example will be an ADT of bags, which provides a type *Bag* with three operations: the operation of adding an element to a bag, an operation to inspect how often a given element occurs in a bag, and the empty bag:

$$\begin{aligned} empty &: Bag, \\ add &: Nat \times Bag \to Bag, \\ card &: Nat \times Bag \to Nat. \end{aligned}$$

Tupling the three operations yields

$$(empty, add, card) : Bag \times (Nat \times Bag \to Bag) \times (Nat \times Bag \to Nat),$$

so the signature of the ADT can be given as

$$BagSig(X) \mathrel{\hat{=}} X \times (Nat \times X \to X) \times (Nat \times X \to Nat).$$

The existential type *BagImp*,

$$BagImp \mathrel{\hat{=}} \exists X.\, BagSig(X)$$

can be used as type of implementations of the ADT of bags, as we will now explain.

To implement the ADT of bags, we have to come up with some type *Rep* which will be used as representations of bags, and a 3-tuple of functions of type $BagSig(Rep)$ that implement the bag-operations for this representation. An obvious way to represent bags is to use lists. In this case *empty* can be implemented as the empty list $nil : List$, *add* as the operation $cons : Nat \times List \to List$ on lists, and *card* as a function $count : Nat \times List \to List$ that counts how often a given natural number occurs in a given list of natural numbers. These three operations have the right types, since

$$(nil, cons, count) : BagSig(List).$$

The introduction rule for existential types can be used to construct an element of type *BagImp* from the type *List* and the triple $(nil, cons, count)$:

$$imp1 \mathrel{\hat{=}} (\mathsf{pack}\ \langle List, (nil, cons, count)\rangle\ \mathsf{to}\ BagImp) : BagImp.$$

Now suppose we want to define some program b that uses the ADT of bags. Then in b we want to use the abstract operations *empty*, *add*, and *card*, and b has to be well-typed under the assumption that these three abstract operations have their correct types:

$$empty : Bag, add : Nat \times Bag \to Bag, card : Nat \times Bag \to Nat \vdash b : B$$

Here *Bag* is a type variable. The elimination rule for existential types now tells us how we can combine this program b with the implementation $imp1 : BagImp$ defined above:

$$\mathsf{open}\ imp1\ \mathsf{as}\ \langle Bag, (empty, add, card)\rangle\ \mathsf{in}\ b : B$$

It is easy to verify that this program behaves as expected:

$$\begin{array}{ll} & \mathsf{open}\ imp1\ \mathsf{as}\ \langle Bag, (empty, add, card)\rangle\ \mathsf{in}\ b \\ \rhd_\beta & \\ & b[List/Bag, nil/empty, cons/add, count/card]. \end{array}$$

So the concrete representation *List* gets substituted for the abstract type *Bag*, and the concrete implementations of the operations on *List*'s get substituted for the abstract operations on *Bag*'s.

The typing rules play a crucial role in hiding the concrete implementation of the ADT (using *List*'s) from the main program b. It is not possible to apply list operations to bags in b, because this would not be well-typed. The program b has to be typed under the assumptions that

$$empty : Bag, add : Nat \times Bag \to Bag, card : Nat \times Bag \to Nat,$$

where *Bag* is a type variable.

3 The logic for parametricity

Plotkin and Ababi's logic for parametric polymorphism [PA93] is a logic for reasoning about the second-order lambda calculus that exploits the notion of parametricity. We will use the somewhat different presentation of the logic given by Takeuti [Tak97].

We only describe the fragment of the logic that is of interest to us. This makes the description much simpler and this paper much easier to digest. (In particular, Definition 4 only deals with the type constructors $\to$ and $\times$, not $\forall$ and $\exists$ – which are more complex – and considers the parametricity property only for existential types $\exists X.T$ where T is a "first-order" signature built using $\times$ and $\to$. The small price we pay for this is that we can only consider ADT's with such signatures, but this covers most examples.)

Takeuti defines the logic for parametricity in two stages: first a base logic **L** which provides the standard logical connectives and their rules, and then a logic **Par** which extends **L** with axioms expressing parametricity.

3.1 The base logic L

L is a second-order predicate logic over the second-order lambda calculus, i.e. it provides predicates on the types of the second-order lambda calculus. **L** is a *typed* logic, with predicates – and also propositions – having types. The type of propositions is denoted by $*_p$. Predicates can be viewed as functions that return propositions, so $T \to *_p$ is the type of predicates over type T. Relations are binary predicates, so $T \to T \to *_p$ is the type of binary predicates – or relations – on T.

So the types of propositions and predicates are given by

$$\mathbb{P} ::= *_p \mid T \to \mathbb{P}.$$

The propositions and predicates are given by

$$P ::= P \Rightarrow Q \mid \forall x{:}T.\, P \mid \forall X.\, P \mid \forall P{:}\mathbb{P}.\, Q \mid \lambda x{:}T.\, P \mid P\, t.$$

The first four constructions provide ways to built propositions: namely implication $P \Rightarrow Q$, and three kinds of universal quantification, universal quantification over all elements of a type $\forall x{:}T.\, P$, universal quantification over all types $\forall X.\, P$, and (second-order) universal quantification over propositions and predicates $\forall P{:}\mathbb{P}.\, Q$. The last two constructs allow the definition of predicates $\lambda x{:}T.\, P$ and the application of predicates to terms $P\, t$.

Judgements in the logic **L** are of the form $\Gamma, \Delta \vdash P$ where Γ is a sequence of declarations $x_1 : T_1, \ldots, x_n : T_n$ as before, Δ is a sequence of assumptions $P_1, \ldots, P_m$, and P is a proposition. We have the standard structural rules, and the standard elimination and introduction for the logical connective $\Rightarrow$ and the quantifiers $\forall$ (for details see [Tak97]).

The second-order universal quantification over propositions and predicates enables the definition of the logical connectives $\vee$, $\wedge$ and $\exists$ in the usual way. It also enables *Leibniz' equality* for datatypes T to be defined in the standard way:

Definition 1 (Leibniz' equality). For any type T, Leibniz' equality of type T, $=_T : T \to T \to *_p$, is defined by

$$=_T \;\widehat{=}\; \lambda x, y{:}T.\;\; \forall P{:}(T \to *_p).\, (Px) \Rightarrow (Py).$$

The subscript of $=_T$ will sometimes be omitted when it is clear from the context. Leibniz' equality will be written infix. Other relations will sometimes also be written infix, and sometimes "postfix", i.e. $(t_1, t_2) \in P$ for Pt_1t_2. □

Remark 2. For readers familiar with *Pure Type Systems* (PTS's) [Bar92], we note that the logic **L** of Takeuti can be concisely described as a PTS, namely the PTS $(\mathsf{S}, \mathsf{A}, \mathsf{R})$ with

$$\begin{aligned}
\mathsf{S} &= \{*_s, \Box_s, *_p, \Box_p\} \\
\mathsf{A} &= \{(*_s : \Box_s)\;,\; (*_p : \Box_p)\} \\
\mathsf{R} &= \{\, (\Box_s, *_s), (*_s, *_s), \\
&\qquad\qquad (*_s, \Box_p), \\
&\quad (\Box_s, *_p), (*_s, *_p),\; (\Box_p, *_p), (*_p, *_p)\}
\end{aligned}$$

Here $*_s$ is the type of all datatypes, just like $*_p$ is the type of all propositions. The fact that **L** is a PTS is the main reason why we chose Takeuti's presentation of the logic rather than Plotkin & Abadi's; it enabled us to verify some examples using the theorem prover Yarrow [Zwa97] which implements arbitrary PTS's.

L is a subsystem of the logic $\boldsymbol{\lambda\omega_L}$ introduced in [Pol94] as a logic for reasoning about the higher-order typed lambda calculus (system F^ω). $\boldsymbol{\lambda\omega_L}$ includes a few more PTS rules, so that it includes the higher-order rather than the second order lambda calculus as "programming language" and allows more powerful abstractions in the logic (such as polymorphic predicates). □

3.2 The logic for parametricity

The logic **Par** extends **L** with an axiom for every type T which states that all elements of T satisfy a certain parametricity property. Since we are only interested in certain properties of existential types in **Par** – viz. the simulation principles - we simply introduce these properties as axioms here.

First, the constructions $\to$ and $\times$ for building types have to be "lifted" to constructions for building relations on types.

Definition 3. Let R_1 and R_2 be relations (i.e. binary predicates), with $R_i : A_i \to A'_i \to *_p$. Then the relations $R_1 \to R_2 : (A_1 \to A_2) \to (A'_1 \to A'_2) \to *_p$ and $R_1 \times R_2 : (A_1 \times A_2) \to (A'_1 \times A'_2) \to *_p$ are defined as follows

$$\begin{aligned} f(R_1 \to R_2)f' &\mathrel{\widehat{=}} \forall x : A_1, x' : A'_1.\, xR_1x' \Rightarrow (fx)R_2(f'x') \\ f(R_1 \times R_2)f' &\mathrel{\widehat{=}} (f.1)R_1(f'.1) \wedge (f.2)R_2(f'.2) \end{aligned}$$

□

Now we lift the type expressions $A(X)$ to relations:

Definition 4. Let $A(X)$ be a type expression built using $\to$ and $\times$ from X and closed type expressions. We write $A(B)$ for $A[B/X]$.

For any relation $\sim: B_1 \to B_2 \to *_p$ the relation $A(\sim) : A(B_1) \to A(B_2) \to *_p$ is defined by induction on the structure of A, as follows:

$$\begin{array}{lll} A(\sim) \mathrel{\widehat{=}} A_1(\sim) \to A_2(\sim) & , \text{if } A(X) \equiv A_1(X) \to A_2(X) \\ A(\sim) \mathrel{\widehat{=}} A_1(\sim) \times A_2(\sim) & , \text{if } A(X) \equiv A_1(X) \times A_2(X) \\ A(\sim) \mathrel{\widehat{=}} \sim & , \text{if } A(X) \equiv X \\ A(\sim) \mathrel{\widehat{=}} =_C & , \text{otherwise, i.e. } A(X) \equiv C \text{ and } X \notin FV(C) \end{array}$$

In the right-hand sides $\to$ and $\times$ denote the construction on relations defined in Definition 3, and $=_C$ is Leibniz' equality as defined in Definition 1. □

As an example, consider the interface of the ADT for bags. Suppose $\sim : B_1 \to B_2 \to *_p$. Then $\mathit{BagSig}(\sim) : \mathit{BagSig}(B_1) \to \mathit{BagSig}(B_2) \to *_p$ is the following relation on 3-tuples:

$$\begin{array}{l} \quad ((\mathit{empty}_1, \mathit{add}_1, \mathit{card}_1), (\mathit{empty}_2, \mathit{add}_2, \mathit{card}_2)) \in \mathit{BagSig}(\sim) \\ \Longleftrightarrow \\ \quad \mathit{empty}_1 \sim \mathit{empty}_2 \wedge \\ \quad \forall n : \mathit{Nat}, b_1 : B_1, b_2 : B_2.\, b_1 \sim b_2 \Rightarrow \mathit{add}_1(n, b_1) \sim \mathit{add}_2(n, b_2) \wedge \\ \quad \forall n : \mathit{Nat}, b_1 : B_1, b_2 : B_2.\, b_1 \sim b_2 \Rightarrow \mathit{card}_1(n, b_1) =_{Nat} \mathit{card}_2(n, b_2) \end{array}$$

Definition 5 (Par). The logic **Par** is the extension of **L** with the axioms

$$\begin{array}{l} \forall u_1, u_2{:}\exists X.A(X). \\ \qquad u_1 = u_2 \\ \quad \Longleftrightarrow \\ \qquad (\exists X_1, X_2.\, \exists x_1{:}A(X_1), x_2{:}A(X_2).\, \exists \sim{:}X_1 \to X_2 \to *_p. \\ \qquad\quad u_1 = \mathsf{pack}\,\langle X_1, x_1\rangle \wedge u_2 = \mathsf{pack}\,\langle X_2, x_2\rangle \wedge (x_1, x_2) \in A(\sim)) \end{array}$$

for all type expressions $A(X)$ built using $\to$ and $\times$ from X and closed type expressions. □

This axiom allows us to prove equivalence of different implementations of an ADT by showing there exists a simulation relation $\sim$ between them. We will refer to this proof principle as *simulation*.

Example: Equality of bag implementations.

We briefly illustrate how we can prove equivalence of different data representations in **Par**.

Recall the implementation $imp1 : \mathit{BagImp}$. Now consider another implementation of the ADT for bags, where we implement the *add*-operation not as the *cons*-operation on *List*'s, but as the *snoc*-operation on *List*'s, which adds a element to the end rather than the front of a list:

$$imp2 \mathrel{\widehat{=}} \mathsf{pack}\,\langle \mathit{List}, (\mathit{nil}, \mathit{snoc}, \mathit{count})\rangle : \mathit{BagImp}.$$

Intuitively, this should not make any difference, because the order of the list representing a bag is irrelevant. In **Par** we can prove $imp1 =_{BagImp} imp2$, namely by proving

$$((nil, cons, count), (nil, snoc, count)) \in BagSig(\sim_{perm}),$$

where $\sim_{perm}: List \to List \to *_p$ relates all lists that are permutations.

Of course, $imp1$ and $imp2$ use the same datatype to represent bags. But we can also prove equivalence of implementations that use different representation types. For example, consider the implementation $imp3$ below, which represents bags as functions of type $Nat \to Nat$:

$$imp3 \mathrel{\widehat{=}} \mathsf{pack}\ \langle Nat \to Nat, (const_0, addimp, app)\rangle : BagImp$$

where

$$\begin{aligned} const_0 &= \lambda n{:}Nat.\, 0 \\ addimp &= \lambda(n, f){:}(Nat \times (Nat \to Nat)).\, \lambda m{:}Nat. \begin{cases} 1 + (f\,m) & \text{if } m = n \\ f\,m & \text{otherwise} \end{cases} \\ app &= \lambda(n, f){:}(Nat \times (Nat \to Nat)).\, fn \end{aligned}$$

The principle of simulation can be used to prove $imp1 =_{BagImp} imp3$, namely by showing that from

$$((nil, cons, count), (const_0, addimp, app)) \in BagSig(\sim),$$

where $\sim: List \to (Nat \to Nat) \to *_p$ relates $l : List$ and $f : Nat \to Nat$ iff $\forall n.\, fn = count(n, l)$.

4 Insufficiency of Par

We will show that the principle of simulation that **Par** provides is not sufficient for reasoning over ADT's. To illustrate this, we consider a specification for the ADT of bags.

Naive Specification

A possible specification for the operations *empty*, *add*, and *card* could be:

$$\begin{aligned} \forall n : Nat.\ & card(n, empty) =_{Nat} 0\ \wedge \\ \forall m : Nat, s : Bag.\ & card(m, add(m, s)) =_{Nat} 1 + card(m, s)\ \wedge \\ \forall m, n : Nat, s : Bag.\ & m \neq_{Nat} n \Rightarrow card(m, add(n, s)) =_{Nat} card(m, s)\ \wedge \\ \forall m, n : Nat, s : Bag.\ & add(m, add(n, s)) =_{Bag} add(n, add(m, s)) \end{aligned}$$

We will consider a simple specification *Spec* giving only the last conjunct. This is the most interesting part of the specification, as it uses equality of bags. For any type *Bag* and any triple $(empty, add, card) : BagSig(Bag)$ we define

$$\begin{aligned} & Spec(Bag, (empty, add, card)) \\ & \mathrel{\widehat{=}} \forall m, n : Nat, s : Bag.\, add(m, add(n, s)) =_{Bag} add(n, add(m, s)). \end{aligned}$$

Spec can be turned into a predicate on *BagImp* as follows

$$\begin{aligned} & Spec^{\exists} : BagImp \to *_p \\ & \mathrel{\widehat{=}} \lambda imp{:}BagImp.\ \exists Rep, ops.\ imp =_{BagImp} \mathsf{pack}\ \langle Rep, ops\rangle\ \wedge\ Spec(Rep, ops) \end{aligned}$$

Note that here $\mathit{Spec}(\mathit{Rep}, \mathit{ops})$ uses Leibniz' equality on type Rep, i.e. $=_{Rep}$.

Clearly

$$\mathit{Spec}(\mathit{Rep}, \mathit{ops}) \;\Rightarrow\; \mathit{Spec}^{\exists}(\mathsf{pack}\ \langle \mathit{Rep}, \mathit{ops}\rangle).$$

(But beware that the reverse implication does not always hold. In fact, this would be inconsistent with parametricity, following the example given in Remark 7.)

Remark 6. It is tempting to extend the "$\mathsf{open}\ \ \mathsf{as}\ \langle\ \rangle\ \mathsf{in}$ " construction that we have for programs to predicates, c.f. the inductive types proposed in [CP90]. This so-called "strong" elimination principle is included in Coq [PM93]. It would mean having the rule

$$\frac{\Gamma, x : A \vdash P : *_p \qquad \Gamma \vdash s : \exists X.\, A}{\Gamma \vdash (\mathsf{open}\ s\ \mathsf{as}\ \langle X, x\rangle\ \mathsf{in}\ P) : *_p}\ X \notin \mathsf{FV}(\Gamma)$$

With this rule the specification *Spec* could be turned into a predicate on *BagImp* in a much more direct way:

$$\mathit{Spec}^{\exists}(\mathit{imp}) \;\widehat{=}\; \mathsf{open}\ \mathit{imp}\ \mathsf{as}\ \langle \mathit{Bag}, \mathit{ops}\rangle\ \mathsf{in}\ \mathit{Spec}(\mathit{Bag}, \mathit{ops})$$

and $\mathit{Spec}^{\exists}(\mathsf{pack}\ \langle \mathit{List}, (\mathit{nil}, \mathit{cons}, \mathit{count})\rangle)$ would then simply β-reduce to $\mathit{Spec}(\mathit{List}, (\mathit{nil}, \mathit{cons}, \mathit{count}))$, so these two propositions would be equivalent. Unfortunately, this is inconsistent with parametricity, as will be shown in Remark 7. $\square$

The problem with the naive specification

The specification $\mathit{Spec}^{\exists}$ might be what the user of the ADT wants, but it may be a problem for the implementor of the ADT to meet this specification. As an example we take the implementation $\mathit{imp}1$,

$$\mathit{imp}1 \,\widehat{=}\, \mathsf{pack}\ \langle \mathit{List}, (\mathit{nil}, \mathit{cons}, \mathit{count})\rangle : \mathit{BagImp},$$

and consider the following question: *Can we prove* $\mathit{Spec}^{\exists}(\mathit{imp}1)$ *?*

We could prove $\mathit{Spec}^{\exists}(\mathit{imp}1)$ by proving $\mathit{Spec}(\mathit{List}, (\mathit{nil}, \mathit{cons}, \mathit{count}))$, i.e. by proving

$$\forall m, n : \mathit{Nat}, s : \mathit{List}.\ \mathit{cons}(m, \mathit{cons}(n, s)) =_{List} \mathit{cons}(n, \mathit{cons}(m, s)).$$

But this is clearly not true! Note that the proposition above uses Leibniz' equality of lists, $=_{List}$, since *Spec* uses Leibniz' equality. The equality above makes sense for bags, but not for lists. We could only prove the proposition above for a weaker notion of equality for lists than $=_{List}$, e.g. $\sim_{perm}$.

We now discuss two ways to solve (or avoid) the problem above. Neither of these is really acceptable, which is why we then propose an extension of the logic **Par** to solve the problem in a more satisfactory way.

Solution 1: Finding another implementation

Recall that by the definition of $\mathit{Spec}^{\exists}$

$$\mathit{Spec}^{\exists}(\mathit{imp}1) \Longleftrightarrow \exists \mathit{Rep}, \mathit{ops}.\, \mathit{imp}1 =_{BagImp} \mathsf{pack}\ \langle \mathit{Rep}, \mathit{ops}\rangle\ \wedge\ \mathit{Spec}(\mathit{Rep}, \mathit{ops}).$$

So we can prove $\mathit{Spec}^{\exists}(\mathit{imp}1)$ by finding another implementation $\mathsf{pack}\ \langle \mathit{Rep}, \mathit{ops}\rangle$ of the ADT such that $\mathit{imp}1 =_{BagImp} \mathsf{pack}\ \langle \mathit{Rep}, \mathit{ops}\rangle$ for which we *can* prove $\mathit{Spec}(\mathit{Rep}, \mathit{ops})$.

It turns out that such an implementation exists, namely the implementation which represents bags as *sorted* lists. Let

$$imp_{sort} \mathrel{\widehat{=}} \mathsf{pack}\ \langle List, (nil, insert, count)\rangle,$$

where $insert : Nat \times List \to List$ inserts a natural number in a list and returns the list sorted. For this implementation we can prove it meets *Spec*, since

$$\forall m, n : Nat, s : List.\ insert(m, insert(n, s)) =_{List} insert(n, insert(m, s)). \qquad \text{(i)}$$

The reason we can prove *Spec* for this implementation is due to the fact that for this particular representation – bags are represented as sorted lists – equality of the concrete representation type, i.e. equality of lists, coincides with equality of the abstract type, i.e. equality of bags.

Using parametricity we can prove

$$imp1 =_{BagImp} imp_{sort}, \qquad \text{(ii)}$$

namely by showing that $\sim_{perm}$ is a simulation relation between the two implementations. Now $Spec^{\exists}(imp1)$ follows from (i) – i.e. $Spec(List, (nil, insert, count))$ – and (ii).

There are obvious drawbacks to this way of proving $Spec^{\exists}(imp1)$. Firstly, it is not acceptable that to prove correctness of our original implementation $imp1$ we have to come up with a second implementation imp_{sort}. Moreover, it may not always be possible to find a second implementation that does meet the specification, i.e. for which concrete and abstract equality coincide! For example, for a generic datatype $Bag(X)$ of bags over an arbitrary type X we would have a problem; there is no way to extend the implementation using sorted lists of natural numbers to lists of an arbitrary type, since there is no generic sorting algorithm for arbitrary types.

Remark 7. We can use imp_{sort} to show the inconsistency of the elimination scheme discussed in Remark 6. If $Spec^{\exists}$ were defined with this scheme, then $Spec^{\exists}(\mathsf{pack}\ \langle Rep, ops\rangle)$ would be β-equivalent with $Spec(Rep, ops)$, so then

$$\begin{aligned} Spec^{\exists}(imp1) &\Longleftrightarrow Spec(List, (nil, cons, count)) \\ Spec^{\exists}(imp_{sort}) &\Longleftrightarrow Spec(List, (nil, insert, count)) \end{aligned}$$

But $Spec(List, (nil, cons, count))$ is false (since *cons* is not "commutative"), whereas $Spec(List, (nil, insert, count))$ is true, (since *insert* is "commutative"). And by parametricity $imp1 = imp_{sort}$, so $Spec^{\exists}(imp1) \Longleftrightarrow Spec^{\exists}(imp_{sort})$, and we have a contradiction. $\square$

Solution 2: Using a weaker specification

The best we could prove for $imp1$ is that

$$\forall m, n : Nat, s : List.\ cons(m, cons(n, s)) \sim_{perm} cons(n, cons(m, s)).$$

Note that $\sim_{perm}$ is a bisimulation for the implementation, i.e.

$$((nil, cons, count), (nil, cons, count)) \in BagSig(\sim_{perm}), \qquad (*)$$

since

$$\begin{aligned} & nil \sim_{perm} nil \ \wedge \\ & \forall n : Nat, l, l' : List.\ l \sim_{perm} l' \Rightarrow cons(n, l) \sim_{perm} cons(n, l') \ \wedge \\ & \forall n : Nat, l, l' : List.\ l \sim_{perm} l' \Rightarrow count(n, l) =_{Nat} count(n, l'). \end{aligned}$$

Intuitively, (*) says that lists in the relation $\sim_{perm}$ cannot be distinguished using the bag-operations, so that lists in the relation $\sim_{perm}$ represent the same bag. With this in mind, one could propose a weaker specification for bags. First, we abstract the specification *Spec* over a notion of equality for bags, to get the following "generic" specification *GenSpec*:

$$\begin{array}{l} \mathit{GenSpec}(\mathit{Bag}, (\mathit{empty}, \mathit{add}, \mathit{card}), \sim) \\ \mathrel{\widehat{=}}\ \forall m, n : \mathit{Nat}, s : \mathit{Bag}.\ \mathit{add}(m, \mathit{add}(n, s)) \sim \mathit{add}(n, \mathit{add}(m, s)). \end{array}$$

(So $\mathit{Spec}(\mathit{Bag}, \mathit{ops}) = \mathit{GenSpec}(\mathit{Bag}, \mathit{ops}, =_{Bag})$.)

We can now consider the following weaker specification

$$\begin{array}{l} \mathit{WeakSpec}(\mathit{Bag}, \mathit{ops}) \\ \mathrel{\widehat{=}} \exists \sim : \mathit{Bag} \to \mathit{Bag} \to *_p. \\ \quad \mathit{GenSpec}(\mathit{Bag}, \mathit{ops}, \sim)\ \wedge\ (\mathit{ops}, \mathit{ops}) \in \mathit{BagSig}(\sim)\ \wedge\ \mathit{Equiv}(\sim), \end{array}$$

where $\mathit{Equiv}(\sim)$ says that $\sim$ is an equivalence relation.

Turning *WeakSpec* into a predicate $\mathit{WeakSpec}^{\exists}$ on *BagImp* we get

$$\begin{array}{l} \mathit{WeakSpec}^{\exists} : \mathit{BagImp} \to *_p \\ \mathrel{\widehat{=}}\ \lambda \mathit{imp}{:}\mathit{BagImp}. \\ \quad \exists \mathit{Rep}, \mathit{ops}.\ \mathit{imp} =_{BagImp} (\mathsf{pack}\ \langle \mathit{Rep}, \mathit{ops}\rangle)\ \wedge\ \mathit{WeakSpec}(\mathit{Rep}, \mathit{ops}). \end{array}$$

The implementor of the ADT will be happy with this weaker specification, as it is possible to prove $\mathit{WeakSpec}^{\exists}(\mathit{imp}1)$, simply by proving $\mathit{WeakSpec}(\mathit{List}, (\mathit{nil}, \mathit{cons}, \mathit{count}))$, taking $\sim_{perm}$ for $\sim$.

The user of the ADT on the other hand will be less happy with $\mathit{WeakSpec}^{\exists}$: rather than using the standard Leibniz' equality of bags, the user has to reason about bags using some bisimulation $\sim$ as notion of equality for bags. This seems an unnecessary complication: there is no reason why the user shouldn't use Leibniz' equality instead of $\sim$. Indeed, this is precisely the *abstraction* that the *abstract* data type is supposed to provide.

5 Our Solution: Extending the logic

Given that the two solutions discussed above are not really satisfactory, we now consider an extension of the logic **Par** that provides a satisfactory solution of the problem.

What we really want is a way to relate the two specifications, $\mathit{WeakSpec}^{\exists}$ and $\mathit{Spec}^{\exists}$, by proving

$$\forall \mathit{imp} : \mathit{BagImp}.\ \mathit{WeakSpec}^{\exists}(\mathit{imp}) \Rightarrow \mathit{Spec}^{\exists}(\mathit{imp}). \qquad (*)$$

Then the implementor of the ADT would only have to establish $\mathit{WeakSpec}^{\exists}$ – i.e. prove the specification up to some bisimulation $\sim$ – and the user of the ADT could assume the stronger specification $\mathit{Spec}^{\exists}$ – i.e. assume the specification with (Leibniz') equality –. Intuitively the property (*) seems OK. (Indeed, it is true in the PER model.)

It turns out that if we have *quotient types* then (*) could be proved. Quotient types are available in some type theories, e.g. Nuprl [Con86] and HOL [GM93], and have been proposed as extensions of other type theories, see e.g. [Hof95] [BG96].

We will first give the general idea of how quotient types could be used to prove the property above. Suppose $\mathit{WeakSpec}^{\exists}(\mathit{imp})$, i.e.

$$\mathit{GenSpec}(\mathit{Rep}, \mathit{ops}, \sim) \wedge (\mathit{ops}, \mathit{ops}) \in \mathit{BagSig}(\sim) \wedge \mathit{Equiv}(\sim)$$

for some $\mathsf{pack}\,\langle Rep, ops\rangle =_{BagImp} imp$ and some $\sim$. The trick to proving (*) is to consider the quotient type $Rep/{\sim}$, i.e. the type with $\sim$-equivalence classes of *Rep* as elements.

$$(ops, ops) \in BagSig(\sim)$$

says that *ops* respects $\sim$-equivalence classes, so *ops* induces a related function $ops/{\sim}$ on $\sim$-equivalence classes, $ops/{\sim} : BagSig(Rep/{\sim})$. And by the principle of simulation it follows that

$$\mathsf{pack}\,\langle Rep, ops\rangle = \mathsf{pack}\,\langle Rep/{\sim}, ops/{\sim}\rangle.$$

The interesting thing about $ops/{\sim}$ is that is satisfies the specification *up to Leibniz' equality*: it follows from $GenSpec(Rep, ops, \sim)$ that

$$GenSpec(Rep/{\sim}, ops/{\sim}, =_{Rep/\sim}),$$

i.e. $Spec(Rep/{\sim}, ops/{\sim})$!

(Note that the argument above goes along the lines as indicated in Solution 1. But the use of quotient types means that the additional work of finding another implementation of ADT is avoided, as this implementation is constructed as a quotient.)

We could consider adding quotient types to the syntax of the second-order lambda calculus. But we do not actually have to do this: it suffices if we add axioms to the logic stating that quotients exist:

Definition 8 (ParQuot). The logic **ParQuot** is the extension of **Par** with the axioms

$$\begin{array}{l}\forall X.\, \forall opsX : A(X).\, \forall \sim : X \to X \to *_p.\\ \quad (opsX, opsX) \in A(\sim) \wedge Equiv(\sim)\\ \quad\; \Rightarrow \exists Q.\, \exists opsQ{:}A(Q).\, isQuot(X, opsX, \sim, Q, opsQ)\end{array}$$

where

$$\begin{array}{ll} & isQuot(X, opsX, \sim, Q, opsQ)\\ \widehat{=} & \\ & \exists inj{:}X \to Q.\, \forall r, r'{:}X.\, r \sim r' \iff (inj\ r) =_Q (inj\ r') \wedge\\ & \qquad\qquad\qquad \forall q{:}Q.\, \exists r{:}X.\, q =_Q (inj\ r) \wedge\\ & \qquad\qquad\qquad (opsX, opsQ) \in A(\lambda r{:}X, q{:}Q.\, q =_Q (inj\ r))\end{array}$$

for all type expressions $A(X)$ built using $\to$ and $\times$ from X and closed type expressions. □

The same PER model used in [PA93] as a semantics for their logic, viz. [BFSS90], quite trivially justifies these additional axioms. Indeed, in a PER model all types are "quotient types"!

Theorem 9. *In the logic* **ParQuot** *it can be proved that*

$$\forall imp : BagImp.\ WeakSpec^{\exists}(imp) \Rightarrow Spec^{\exists}(imp).$$

Proof. Assume $WeakSpec^{\exists}(imp)$. Then there is a type *Rep* with $ops : BagSig(Rep)$ such that

$$imp =_{BagImp} \mathsf{pack}\,\langle Rep, ops\rangle$$

for which

$$GenSpec(Rep, ops, \sim) \wedge (ops, ops) \in BagSig(\sim) \wedge Equiv(\sim)$$

for some $\sim: Rep \to Rep \to *_p$.

By $(ops, ops) \in BagSig(\sim)$ and $Equiv(\sim)$ there then exist a type Q with $opsQ : BagSig(Q)$ and $inj{:}Rep \to Q$ such that

$$\forall r, r'{:}Rep.\, r \sim r' \iff (inj\ r) =_Q (inj\ r') \quad \text{(i)}$$

$$\forall q{:}Q.\, \exists r{:}Rep.\, q =_Q (inj\ r) \quad \text{(ii)}$$

$$(ops, opsQ) \in A(\lambda r{:}Rep, q{:}Q.\, q =_Q (inj\ r)) \quad \text{(iii)}$$

It follows from (iii) that

$$\mathsf{pack}\,\langle Q, opsQ\rangle =_{BagImp} \mathsf{pack}\,\langle Rep, ops\rangle.$$

Using the definition of $GenSpec$, we can prove

$$GenSpec(Q, opsQ, =_Q) \quad \text{(iv)}$$

using $GenSpec(Rep, ops, \sim)$ and (i), (ii), and (iii).

And (iv) is equivalent with $Spec(Q, opsQ)$, and since $\mathsf{pack}\langle Q, opsQ\rangle =_{BagImp}$ $\mathsf{pack}\,\langle Rep, ops\rangle =_{BagImp} imp$ it then follows that

$$Spec^{\exists}(imp).$$

□

Similar theorems can be proved for other ADT's and other (equational) specifications: For any other ADT and specification for it, a weak version of the specification using some relation $\sim$ (similar to $WeakSpec^{\exists}$) and the strong version using Leibniz' equality (similar to $Spec^{\exists}$) can be related in exactly the same way as in the theorem above.

6 Conclusion

In this paper we have explored the gap between the formal notion of parametricity of [PA93] and the important "folk" reasoning principle about ADT's, which we have called *abstraction*.

Roughly, this principle of abstraction says that elements of the concrete representation type of an ADT can be considered equal if they are not distinguishable using the ADT-operations. For example, if we implement bags as lists, then lists that are permutations cannot be distinguished using the bag-operations – they represent the same bag – and can hence be considered equal. To prove that such an implementation of bags satisfies an equational specification we may therefore use permutation of lists as the notion of equality. This principle of abstraction is a well-known reasoning principle for ADT's.

Parametricity provides the proof principle of *simulation* for existential types [Mit91] [PA93]. This is a useful proof principle if existential types are used for abstract data types: it provides a method to prove that different implementations of an ADT are equivalent, namely by showing that there exists a simulation relation between them.

However, we have shown that this principle of simulation alone is not enough to reason about ADT's, since in general it does not provide the proof principle of abstraction that one would want. This observation is new, as far as we know. However, extending the logic for parametricity of [PA93] with axioms stating the existence of quotients is enough to solve this problem. Like the original logic for parametricity of [PA93] these additional axioms can be justified by a PER model.

Proofs for the example of the specification for bags have all been verified using the interactive theorem prover Yarrow [Zwa97]. Indeed, it was only in the course of formalising specifications for ADT's in Yarrow that we noticed that more was needed than just the proof principle of simulation to reason about specifications of ADT's.

References

[Bar92] H.P. Barendregt. Lambda calculi with types. In D.M. Gabbai, S. Abramsky, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1. Oxford University Press, 1992.

[BFSS90] E.S. Bainbridge, P.J. Freyd, A. Scedrov, and P.J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(1):35–64, 1990.

[BG96] G. Barthe and J.H. Geuvers. Congruence types. In *Computer Science Logic'95*, volume 1092 of *Lecture Notes in Computer Science*, pages 36–51. Springer, 1996.

[Con86] R.L. Constable et al. *Implementing Mathematics in the Nuprl proof development system.* Prentice-Hall, 1986.

[CP90] Thierry Coquand and Christine Paulin. Inductively Defined Types. In P. Martin-Löf and G. Mints, editors, *COLOG-88*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1990.

[GM93] M. J. Gordon and T. F. Melham. *Introduction to HOL.* Cambridge, 1993.

[Hof95] Martin Hofmann. A simple model for quotient types. In *Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 216–234, 1995.

[Mit91] John C. Mitchell. On the equivalence of data representations. In *Artificial Intelligence and Mathematical Theory of Computation*, pages 305–330. Academic Press, 1991.

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. on Prog. Lang. and Syst.*, 10(3):470–502, 1988.

[PA93] Gordon Plotkin and Martin Abadi. A logic for parametric polymorphism. In *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375, 1993.

[PM93] Christine Paulin-Mohring. Inductive definitions in the system Coq. In *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993.

[Pol94] Erik Poll. *A Programming Logic based on Type Theory.* PhD thesis, Technische Universiteit Eindhoven, 1994.

[Tak97] Izumi Takeuti. An axiomatic system of parametricity. In *Typed Lambda Calculi and Applications*, volume 1130 of *Lecture Notes in Computer Science*, pages 354–372, 1997.

[Zwa97] Jan Zwanenburg. The proof assistant Yarrow. Submitted for publication. See also http://www.win.tue.nl/cs/pa/janz/yarrow/, 1997.