Submitted to MFPS 2024

Early Announcement: Parametricity for GADTs

Pierre Cagne[a,1] Patricia Johann[a,2]

[a] *Department of Computer Science*
Appalachian State University
Boone, NC, USA

Abstract

Relational parametricity was first introduced by Reynolds for System F. Although System F provides a strong model for the type systems at the core of modern functional programming languages, it lacks features of daily programming practice such as complex data types. In order to reason parametrically about such objects, Reynolds' seminal ideas need to be generalized to extensions of System F. Here, we explore such a generalization for the extension of System F by Generalized Algebraic Data Types (GADTs) as found in Haskell. Although GADTs generalize Algebraic Data Types (ADTs) — i.e., simple recursive types such as lists, trees, etc. — we show that naively extending the parametric treatment of these recursive types is not enough to tackle GADTs. We propose a tentative workaround for this issue, borrowing ideas from the categorical semantics of GADTs known as (functorial) completion. We discuss some applications, as well as some limitations, of this solution.

Keywords: Parametricity, Generalized Algebraic Data Types, Logical relations

1 Introduction

Relational parametricity [Rey83] is a key technique for reasoning about programs in strongly typed languages. It can be used to enforce invariants guaranteeing strong properties of programs, programming languages, and programming language implementations supporting parametric polymorphism. A polymorphic program is a program that can be applied to arguments and return results of different types; a parametric polymorphic program is a program that not only is polymorphic over *all* types, but is also defined by the same type-uniform algorithm regardless of the concrete type at which it is applied. Since parametric polymorphic programs cannot perform type-specific operations, the computational behaviors they can exhibit are actually quite constrained. Parametricity was originally put forth by Reynolds [Rey83] for System F [Gir72,Rey74], the formal calculus at the core of all polymorphic functional languages. It was later popularized as Wadler's "theorems for free" [Wad89], so-called because it allows the deduction of properties of programs in such languages solely from their types, i.e., with no knowledge whatsoever of the text of the programs involved. However, to get interesting free theorems, Wadler actually treats System F extended with built-in lists. Indeed, most of the free theorems in [Wad89] are essentially naturality properties for polymorphic list-processing functions. It is easy to extend the techniques developed there for handling lists to non-list algebraic data types (ADTs). Parametricity for such types can then be used to derive not just naturality (i.e., commutativity) properties, but also results — such as proofs of type inhabitance and correctness of the program optimization known as *short cut fusion* [GLP93] — that go beyond simple naturality.

[1] Email: cagnep@appstate.edu
[2] Email: johannp@appstate.edu

In his original formulation, Reynolds gives each type expression of System F a *relational interpretation* defined inductively. Each type expression Φ with type variables $\alpha_1, \alpha_2, \ldots, \alpha_n$ thus gives, for each tuple $\overline{R}$ of relations R_i between types A_i and B_i, a relation $\widehat{\Phi}\,\overline{R}$ between the type $\Phi[\overline{A}/\overline{\alpha}]$ and $\Phi[\overline{B}/\overline{\alpha}]$. To capture the intended type-uniformity of System F's polymorphic expressions, these relational interpretations are defined in such a way that every function $f : \forall\overline{\alpha}.\Phi \to \Psi$, where Φ and Ψ are two type expressions in the same type variables $\overline{\alpha}$, is *parametric* in the following sense: for each tuple of relations $\overline{R}$, the pairs related by $\widehat{\Phi}\,\overline{R}$ are sent by f to pairs related by $\widehat{\Psi}\,\overline{R}$.

Better approximations of realistic programming languages result from adding built-in data types to System F. Each such added data type induces a type constructor, and this type constructor must also be given a relational interpretation. Wadler [Wad89] considers the case of lists, which we review in detail in Section 2. To add a new inductive data type constructor T to an ambient parametric language in such a way that parametricity is preserved, the method is always the same: Define its relational interpretation as a (dependent) inductive family $\widehat{T}$ with one data constructor $\widehat{c}$ for each data constructor c of T expressing precisely that c is a parametric polymorphic function. The data constructors of such a data type's relational interpretation thus make formal the intuitive type-uniformity required of its data constructors by the grammars of languages such as Haskell. The relational interpretation $\widehat{T}$ captures the intuition that, if we regard data types as containers, then two data structures of (two instances of) T are related by $\widehat{T}\,R$ exactly when the data they store are related by R. This intuition also requires that $\widehat{T}$ preserves inclusion, i.e., that $\widehat{T}\,R \subseteq \widehat{T}\,S$ whenever $R \subseteq S$. Indeed, if two data structures are related by $\widehat{T}\,R$, then the data they store are related by R, and thus by S, so the two data structures must be related by $\widehat{T}\,S$. Fortunately, for lists and other ADTs, the relational interpretations defined in this way enjoy this crucial inclusion-preservation property.

Here, we report our ongoing efforts to add the generalization of ADTs known as *Generalized Algebraic Data Types (GADTs)* to System F in such a way that parametricity is preserved. In doing so, we insist on understanding GADTs as types of *data structures*, i.e., as types of containers that can be filled with data. Since this entails in particular that GADTs are inductive data type constructors, we might expect that following the method outlined above will suffice. In Section 2, we show that naively doing so results in relational interpretations of GADTs that do not satisfy the inclusion-preservation property identified at the end of the preceding paragraph. This is problematic: if we are to understand GADTs as types of data structures, then they should certainly satisfy all properties — among them the inclusion-preservation property — expected of such types. In Section 3, we explore a promising approach to overcoming this issue. This approach consists in defining the relational interpretation of a GADT through that of its *completion*, an ADT-like type constructor that contains the original GADT. In Section 4 we offer some applications of parametricity for GADTs obtained using our proposed approach. In Section 5 we discuss some issues that arise when making our proposed approach precise. Doing so requires defining a source language (an extension of System F that allows for GADTs), a target language (a dependent type theory strong enough to encode relations), and interpretations of each type of the source language as both a type and a relation in the target language. We point out some difficulties in the design of the target language, and also offer some thoughts on how to resolve them. Throughout the paper, we use an Agda-like syntax to write examples of types and terms of the anticipated target language. We note, however, that this language might end up being very different from Agda's type theory. In particular, this early announcement by no means reports on an attempt to formalize our work in a proof assistant.

We are not the first to consider parametricity for GADTs. Very recent progress on the subject has been presented in [SSSB24]. Sieczkowski *et al.* construct there a parametric model of an extension of System F supporting GADTs, with the aim of deriving free theorems and representation independence results. However, their work differs drastically from the line of research presented here in several ways. First, the semantics presented by Sieczkowski *et al.* targets normalization-by-evaluation. By contrast, our work is in no way concerned with such methods. Second, Sieczkowski et al. make essential use of guarded recursion through a universe of step-indexed propositions equipped with a later modality (as exists, e.g., in Iris). By contrast, we are concerned only with structural recursion in this work. Third, Sieczkowski *et al.* insist on the importance of two particular rules of their type system: discriminability and injectivity of type constructors. By contrast, we are agnostic about such rules, thus accommodating more diverse host languages. Finally, and most importantly, the semantics of Sieczkowski *et al.* models parametricity for GADTs only in those type indices that are *unconstrained*, i.e., that can be promoted to parameters.

In particular, their approach cannot handle free theorems such as the one presented in Section 4.1 for Seq, since pairing has a constrained instance of Seq as return type. By contrast, we not only recognize the non-uniformity of GADTs acknowledged by Sieczkowski *et al.*, but we also recognize that this break of uniformity is governed by uniform type constructors (namely, those constraining the instances of the return types of GADTs' data constructors), and that this uniformity must be captured by parametric models of the language at play.

2 Naive approach: The problem

In this section, we first review Wadler's relational interpretation of the standard built-in type constructor List for the ADT of lists, and then try to extend the method directly to GADTs. As noted in Section 1, the resulting relational interpretations for GADTs lack the desired inclusion-preservation property.

The type constructor List for the ADT of lists is given by:

$$\begin{aligned}&\mathsf{data}\,\mathsf{List} : \mathsf{Set} \to \mathsf{Set}\,\mathsf{where}\\ &\quad \mathsf{nil} : \forall\{\alpha\} \to \mathsf{List}\,\alpha\\ &\quad \mathsf{cons} : \forall\{\alpha\} \to \alpha \to \mathsf{List}\,\alpha \to \mathsf{List}\,\alpha\end{aligned} \tag{1}$$

Wadler effectively gave a relational interpretation for List informally when he declared two lists $[a_1, a_2, ..., a_n] : \mathsf{List}\,A$ and $[b_1, b_2, ..., b_n] : \mathsf{List}\,B$ to be related by $\widehat{\mathsf{List}}\,R$ for a relation R between the types A and B exactly when each pair of their corresponding elements is related by R, i.e., when he required

$$\widehat{\mathsf{List}}\,R\,[a_1, a_2, \ldots, a_n]\,[b_1, b_2, \ldots, b_n] \quad \text{if and only if} \quad \forall i = 1, \ldots, n,\ R\,a_i\,b_i$$

If we represent the type relation $\mathsf{Rel}\ A\,B$ of relations between A and B by the function type $A \to B \to \mathsf{Set}$, then we can formalize Wadler's relational interpretation for lists as the (dependent) inductive family represented by:

$$\begin{aligned}&\mathsf{data}\,\widehat{\mathsf{List}} : \forall\{\alpha\,\beta\} \to \mathsf{Rel}\ \alpha\,\beta \to \mathsf{Rel}\ (\mathsf{List}\,\alpha)\,(\mathsf{List}\,\beta)\,\mathsf{where}\\ &\quad \widehat{\mathsf{nil}} : \forall\{\alpha\,\beta\}(\mathsf{R} : \mathsf{Rel}\ \alpha\,\beta) \to \widehat{\mathsf{List}}\,\mathsf{R}\,\mathsf{nil}\,\mathsf{nil}\\ &\quad \widehat{\mathsf{cons}} : \forall\{\alpha\,\beta\}(\mathsf{R} : \mathsf{Rel}\ \alpha\,\beta)(\mathsf{a} : \alpha)(\mathsf{b} : \beta)(\mathsf{as} : \mathsf{List}\,\alpha)(\mathsf{bs} : \mathsf{List}\,\beta) \to\\ &\qquad \mathsf{R}\,\mathsf{a}\,\mathsf{b} \to \widehat{\mathsf{List}}\,\mathsf{R}\,\mathsf{as}\,\mathsf{bs} \to \widehat{\mathsf{List}}\,\mathsf{R}\,(\mathsf{cons}\,\mathsf{a}\,\mathsf{as})\,(\mathsf{cons}\,\mathsf{b}\,\mathsf{bs})\end{aligned}$$

Notice that only terms both constructed from the same data constructor can be related, and that the definition of $\widehat{\mathsf{List}}$ mimics the recursive structure of List's data type declaration. This ensures in particular that $\widehat{\mathsf{List}}$ preserves inclusions: Given $i : R \subseteq S$, define $\widehat{\mathsf{List}}\,i$ by mapping $\widehat{\mathsf{nil}}\,R$ to $\widehat{\mathsf{nil}}\,S$ and $\widehat{\mathsf{cons}}\,R\,a_h\,b_h\,a_t\,b_t\,w_h\,w_t$ to $\widehat{\mathsf{cons}}\,S\,a_h\,b_h\,a_t\,b_t\,(i\,w_h)\,(\widehat{\mathsf{List}}\,i\,w_t)$. Moreover, this definition has exactly the feature announced in Section 1, namely that $\widehat{\mathsf{nil}}$ and $\widehat{\mathsf{cons}}$ express that nil and cons are parametric, respectively.

The method we used to construct List can easily be extended to other ADTs, or even to more general inductive definitions. This is the approach explored by the authors of [BJP10] for generic inductive families, which encompass, in particular, GADTs. Although interesting in its own right, this approach fails to recognize GADTs as data types, in the sense that their relational interpretations do not necessarily preserve inclusion as is intuitively expected. To illustrate the issue, consider the GADT of sequences given by:

$$\begin{aligned}&\mathsf{data}\,\mathsf{Seq} : \mathsf{Set} \to \mathsf{Set}\,\mathsf{where}\\ &\quad \mathsf{inj} : \forall\{\alpha\} \to \alpha \to \mathsf{Seq}\ \alpha\\ &\quad \mathsf{pairing} : \forall\{\alpha_1\,\alpha_2\} \to \mathsf{Seq}\ \alpha_1 \to \mathsf{Seq}\ \alpha_2 \to \mathsf{Seq}\ (\alpha_1 \times \alpha_2)\end{aligned} \tag{2}$$

The same method used above yields the following relational interpretation $\widehat{\mathsf{Seq}}$ for Seq:

$$\begin{aligned}
&\mathsf{data}\,\widehat{\mathsf{Seq}} : \forall\{\alpha\,\beta\} \to \mathsf{Rel}\ \alpha\,\beta \to \mathsf{Rel}\ (\mathsf{Seq}\ \alpha)\,(\mathsf{Seq}\ \beta)\,\mathsf{where}\\
&\quad \widehat{\mathsf{inj}} : \forall\{\alpha\,\beta\}(\mathsf{R} : \mathsf{Rel}\ \alpha\,\beta)(\mathsf{a} : \alpha)(\mathsf{b} : \beta) \to \mathsf{R}\,\mathsf{a}\,\mathsf{b} \to \widehat{\mathsf{Seq}}\ \mathsf{R}\,(\mathsf{inj}\ \mathsf{a})\,(\mathsf{inj}\ \mathsf{b})\\
&\quad \widehat{\mathsf{pairing}} : \forall\{\alpha_1\,\alpha_2\,\beta_1\,\beta_2\}(\mathsf{R}_1 : \mathsf{Rel}\ \alpha_1\,\beta_1)(\mathsf{R}_2 : \mathsf{Rel}\ \alpha_2\,\beta_2) \to\\
&\qquad \forall(\mathsf{s}_1 : \mathsf{Seq}\ \alpha_1)(\mathsf{s}_2 : \mathsf{Seq}\ \alpha_2)(\mathsf{t}_1 : \mathsf{Seq}\ \beta_1)(\mathsf{t}_2 : \mathsf{Seq}\ \beta_2) \to\\
&\qquad \widehat{\mathsf{Seq}}\ \mathsf{R}_1\,\mathsf{s}_1\,\mathsf{t}_1 \to \widehat{\mathsf{Seq}}\ \mathsf{R}_2\,\mathsf{s}_2\,\mathsf{t}_2 \to \widehat{\mathsf{Seq}}\ (\mathsf{R}_1 \mathbin{\widehat{\times}} \mathsf{R}_2)\,(\mathsf{pairing}\,\mathsf{s}_1\,\mathsf{s}_2)\,(\mathsf{pairing}\,\mathsf{t}_1\,\mathsf{t}_2)
\end{aligned}$$

Here, $\_ \mathbin{\widehat{\times}} \_$ is the relational interpretation of the product type constructor $\_ \times \_$ defined on relations $R_1 : \mathsf{Rel}\ A_1\,B_1$ and $R_2 : \mathsf{Rel}\ A_2\,B_2$ by $(R_1 \mathbin{\widehat{\times}} R_2)\,(a_1, a_2)\,(b_1, b_2) = (R_1\,a_1\,b_1) \times (R_2\,a_2\,b_2)$ for all $a_1 : A_1$, $a_2 : A_2$, $b_1 : B_1$, $b_2 : B_2$.

Now, assuming extensionality for relations (i.e., R is equal to S when they relate the same elements), if R is the equality relation on $\mathsf{Bool} \times \mathsf{Bool}$, where Bool is the type of booleans, then $\widehat{\mathsf{Seq}}\ R$ is the equality relation on $\mathsf{Seq}\ (\mathsf{Bool} \times \mathsf{Bool})$. On the other hand, if S is the binary relation on $\mathsf{Bool} \times \mathsf{Bool}$ with only $S\,(\mathsf{false}, \mathsf{false})\,(\mathsf{true}, \mathsf{true})$ uninhabited, then $\widehat{\mathsf{Seq}}\ S\,(\mathsf{pairing}\,s_1\,s_2)\,(\mathsf{pairing}\,t_1\,t_2)$ is uninhabited for any s_1, s_2, t_1, and t_2 because S is not a product of relations. However, S contains R, so $\widehat{\mathsf{Seq}}\ S\,s\,t$ should be inhabited at least whenever $\widehat{\mathsf{Seq}}\ R\,s\,t$ is. That it is not violates the inclusion-preservation property expected of $\widehat{\mathsf{Seq}}$.

3 Completing GADTs: Toward a solution

To remedy the problem exposed in Section 2, we first observe that we can obtain an alternative relational interpretation for Seq by first embedding it into (a data type that is essentially) an ADT S and then constructing the relational interpretation of S as above. The data type S is given by:

$$\begin{aligned}
&\mathsf{data}\,\mathsf{S} : \mathsf{Set} \to \mathsf{Set}\,\mathsf{where}\\
&\quad \mathsf{i} : \forall\{\alpha\} \to \alpha \to \mathsf{S}\ \alpha\\
&\quad \mathsf{p} : \forall\{\alpha_1\,\alpha_2\,\alpha\} \to (\alpha_1 \times \alpha_2 \to \alpha) \to \mathsf{S}\ \alpha_1 \to \mathsf{S}\ \alpha_2 \to \mathsf{S}\ \alpha
\end{aligned} \tag{3}$$

We can compute the relational interpretation of S as we did above for ADTs. This gives:

$$\begin{aligned}
&\mathsf{data}\,\widehat{\mathsf{S}} : \forall\{\alpha\,\beta\} \to \mathsf{Rel}\ \alpha\,\beta \to \mathsf{Rel}\ (\mathsf{S}\ \alpha)\,(\mathsf{S}\ \beta)\,\mathsf{where}\\
&\quad \widehat{\mathsf{i}} : \forall\{\alpha\,\beta\}(\mathsf{R} : \mathsf{Rel}\ \alpha\,\beta)(\mathsf{a} : \alpha)(\mathsf{b} : \beta) \to \mathsf{R}\,\mathsf{a}\,\mathsf{b} \to \widehat{\mathsf{S}}\ \mathsf{R}\,(\mathsf{i}\ \mathsf{a})\,(\mathsf{i}\ \mathsf{b})\\
&\quad \widehat{\mathsf{p}} : \forall\{\alpha_1\,\alpha_2\,\beta_1\,\beta_2\,\alpha\,\beta\}(\mathsf{R}_1 : \mathsf{Rel}\ \alpha_1\,\beta_1)(\mathsf{R}_2 : \mathsf{Rel}\ \alpha_2\,\beta_2)(\mathsf{R} : \mathsf{Rel}\ \alpha\,\beta) \to\\
&\qquad \forall(\mathsf{s}_1 : \mathsf{S}\ \alpha_1)(\mathsf{s}_2 : \mathsf{S}\ \alpha_2)(\mathsf{t}_1 : \mathsf{S}\ \beta_1)(\mathsf{t}_2 : \mathsf{S}\ \beta_2)(\mathsf{f} : \alpha_1 \times \alpha_2 \to \alpha)(\mathsf{g} : \beta_1 \times \beta_2 \to \beta) \to\\
&\qquad ((\mathsf{R}_1 \mathbin{\widehat{\times}} \mathsf{R}_2) \mathbin{\widehat{\to}} \mathsf{R})\,\mathsf{f}\,\mathsf{g} \to \widehat{\mathsf{S}}\ \mathsf{R}_1\,\mathsf{s}_1\,\mathsf{t}_1 \to \widehat{\mathsf{S}}\ \mathsf{R}_2\,\mathsf{s}_2\,\mathsf{t}_2 \to \widehat{\mathsf{S}}\ \mathsf{R}\,(\mathsf{p}\ \mathsf{f}\,\mathsf{s}_1\,\mathsf{s}_2)\,(\mathsf{p}\ \mathsf{g}\,\mathsf{t}_1\,\mathsf{t}_2)
\end{aligned}$$

Here, $\_ \mathbin{\widehat{\to}} \_$ is the relational interpretation of the function type constructor $\_ \to \_$ defined for any $R : \mathsf{Rel}\ A\,B$ and $S : \mathsf{Rel}\ C\,D$ by $(R \mathbin{\widehat{\to}} S)\,f\,g = \forall(a : A)(b : B) \to R\,a\,b \to S\,(f\,a)\,(g\,b)$.

Note that there is an embedding ι of Seq into S, obtained by mapping a sequence of the form $\mathsf{inj}\ a$ to $\mathsf{i}\ a$ and one of the form $\mathsf{pairing}\,s_1\,s_2$ with $s_1 : \mathsf{Seq}\ A_1$ and $s_2 : \mathsf{Seq}\ A_2$ to $\mathsf{p}\ \mathsf{id}_{A_1 \times A_2}\,(\iota\,s_1)\,(\iota\,s_2)$. With ι and $\widehat{\mathsf{S}}$ in hand, we can now define the relational interpretation of Seq by $\widehat{\mathsf{Seq}}\ R\,s\,t = \widehat{\mathsf{S}}\ R\,(\iota\,s)\,(\iota\,t)$. It is easy to see that the relational interpretation $\widehat{\mathsf{Seq}}$ not only ensures that the constructors of Seq are parametric, but also satisfies the inclusion-preservation property. Indeed, if $i : R \subseteq S$ then we can define $\widehat{\mathsf{S}}\,i : \widehat{\mathsf{S}}\,R \subseteq \widehat{\mathsf{S}}\,S$ simply by mapping $\widehat{\mathsf{i}}\ R\,a\,b\,w$ to $\widehat{\mathsf{i}}\ S\,a\,b\,(i\,w)$ and $\widehat{\mathsf{p}}\,R_1\,R_2\,R\,s_1\,s_2\,t_1\,t_2\,f\,g\,w\,w_1\,w_2$ to $\widehat{\mathsf{p}}\,R_1\,R_2\,S\,s_1\,s_2\,t_1\,t_2\,f\,g\,(i \circ w)\,(\widehat{\mathsf{S}}\,i\,w_1)(\widehat{\mathsf{S}}\,i\,w_2)$.

The method described for Seq can easily be extended to any GADT. Indeed, given any GADT G, its completion G_c is obtained by first identifying each constructor $c : \forall\overline{\alpha}.\Phi \to G\,\overline{\Psi}$ whose return instance $\overline{\Psi}$ of G

is not simply $\overline{\alpha}$ and replacing it by $c_c : \forall \overline{\alpha}\,\overline{\beta}.(\overline{\Psi \to \beta}) \to \Phi \to G\,\overline{\beta}$. The name *completion* is justified by the embedding ι_G of G into G_c defined on each element of the form $c\,x$ by $c_c\,\overline{\mathsf{id}_\Psi}\,x$. The completion G_c is akin to an ADT in the sense that the return type of each of its constructors is a variable instance of G_c; however, it is not an ADT *per se* because each of its constructors $c_c : \forall \overline{\alpha}\,\overline{\beta}.(\overline{\Psi \to \beta}) \to \Phi \to G\,\overline{\beta}$ quantifies over a possibly non-empty vector $\overline{\alpha}$ of type variables as well as over the vector $\overline{\beta}$ of type variables appearing in c_c's return instance. Nevertheless, the relational interpretation $\widehat{G_c}$ of G_c can be constructed as in Section 2, and the relational interpretation $\widehat{G}$ of G is then defined by restriction as $\widehat{G}\,R\,g\,h = \widehat{G_c}\,R\,(\iota_G\,g)\,(\iota_G\,h)$. The resulting relational interpretation $\widehat{G}$ inherits the inclusion-preservation property of $\widehat{G_c}$ and also ensures that each of the constructors c of G is parametric.

When G is simply an ADT its completion G_c is just a copy of G itself (since then there is no constructor $c : \forall \overline{\alpha}.\Phi \to G\,\overline{\Psi}$ whose return instance $\overline{\Psi}$ is not $\overline{\alpha}$). In this case, the relational interpretation $\widehat{G}$ defined in this section is trivially the same as the relational interpretation associated to it by the method described in Section 2. The construction of $\widehat{G}$ in this section for an arbitrary GADT G thus produces true generalizations of the relational interpretations of ADTs, as introduced for lists by Wadler and subsequently developed for more general data types by others (see, e.g., [BJP10,JG07]).

4 Applications

4.1 Free theorems

The relational interpretations defined for GADTs in the previous section can be used to establish free theorems à la Wadler. We illustrate this with the data type Seq. Given a type A and an element $a : A$, we say that a sequence $s : \mathsf{Seq}\ A$ *contains only* a *as data* either when $s = \mathsf{inj}\ a$ or when $s = \mathsf{pairing}\,s_1\,s_2$ — which forces a to be of the form (a_1, a_2) — and s_1 contains only a_1 as data and s_2 contains only a_2 as data.

Proposition 4.1 *Let $f : \forall \alpha.\alpha \to \mathsf{Seq}\ \alpha$ be a parametric polymorphic function. For any type A and any element $a : A$, the sequence $f\,A\,a$ contains only a as data.*

Proof. Since the function f is parametric, for any types A and B, any relation $R : \mathsf{Rel}\ A\,B$, any $a : A$ and any $b : B$, the sequences $f\,A\,a$ and $f\,B\,b$ are related by $\widehat{\mathsf{Seq}}\,R$ whenever $R\,a\,b$ holds. Now, fix a type A and an element $a : A$ and consider the relation $\delta_a : \mathsf{Rel}\ A\,A$ that relates x and y only when both are a itself. Because $\delta_a\,a\,a$, we get a witness of $\widehat{\mathsf{Seq}}\ \delta_a\,(f\,A\,a)\,(f\,A\,a)$. It remains to show, by induction on $s : \mathsf{Seq}\ A$, that if $\widehat{\mathsf{Seq}}\ \delta_a\,s\,s$ then s contains only a as data. If $s = \mathsf{inj}\ x$ for some $x : A$, then the element we have in $\widehat{\mathsf{Seq}}\ \delta_a\,s\,s = \widehat{\mathsf{S}}\ \delta_a\,(\mathsf{i}\ x)\,(\mathsf{i}\ x)$ must be of the form $\widehat{\mathsf{i}}\ \delta_a\,x\,x\,w$ with $w : \delta_a\,x\,x$. By definition of δ_a, w entails that x is a itself, and thus $s = \mathsf{inj}\ a$ does indeed contain only a as data. If $s = \mathsf{pairing}\,s_1\,s_2$, then A is of the form $A_1 \times A_2$, a is of the form (a_1, a_2), and the element we have in $\widehat{\mathsf{Seq}}\ \delta_a\,s\,s = \widehat{\mathsf{S}}\ \delta_a\,(\mathsf{p}\ \mathsf{id}_{A_1\times A_2}\,(\iota\,s_1)\,(\iota\,s_2))\,(\mathsf{p}\ \mathsf{id}_{A_1\times A_2}\,(\iota\,s_1)\,(\iota\,s_2))$ must be of the form $\widehat{\mathsf{p}}\,R_1\,R_2\,\delta_a\,s_1\,s_2\,s_1\,s_2\,\mathsf{id}_{A_1\times A_2}\,\mathsf{id}_{A_1\times A_2}\,w\,w_1\,w_2$ with $R_1 : \mathsf{Rel}\ A_1\,A_1$, $R_2 : \mathsf{Rel}\ A_2\,A_2$, $w : ((R_1 \mathbin{\widehat{\times}} R_2) \mathbin{\widehat{\to}} \delta_a)\,\mathsf{id}_{A_1\times A_2}\,\mathsf{id}_{A_1\times A_2}$, $w_1 : \widehat{\mathsf{Seq}}\ R_1\,s_1\,s_1$, and $w_2 : \widehat{\mathsf{Seq}}\ R_2\,s_2\,s_2$. From w, we can prove that $R_1 \subseteq \delta_{a_1}$ and $R_2 \subseteq \delta_{a_2}$. Then from w_1 and w_2, inclusion-preservation of $\widehat{\mathsf{Seq}}$ gives witnesses v_1 and v_2 of $\widehat{\mathsf{Seq}}\ \delta_{a_1}\,s_1\,s_1$ and $\widehat{\mathsf{Seq}}\ \delta_{a_2}\,s_2\,s_2$, respectively. By the induction hypothesis, v_1 gives that s_1 has only a_1 as data and v_2 gives that s_2 contains only a_2 as data. That is, $s = \mathsf{pairing}\,s_1\,s_2$ contains only $a = (a_1, a_2)$ as data. □

4.2 Graph lemma

If R is the graph of a function $f : A \to B$, then $\widehat{G}\,R$ is of particular interest. When G is an ADT, then the graph lemma says that $\widehat{G}\,R$ is exactly the graph of the function $\mathsf{map}_G\,f : G\,A \to G\,B$, where map_G is the usual map function associated with the ADT G. However, when G is a more general GADT, then there can be no map function associated with G ([JP19,JC22]). In other words, the relation $\widehat{G}\,R$ is not necessarily the graph of a function from $G\,A$ to $G\,B$. Significantly, $\widehat{G}\,R$ can still be understood as the graph of a *partial* function from $G\,A$ to $G\,B$; see Proposition 4.2 below. The key observation is that the completion G_c of G has a map function $\mathsf{map}_{G_c} : (A \to B) \to G_c\,A \to G_c\,B$ whose application to f is defined for each constructor c_c by $\mathsf{map}_{G_c}\,f\,(c_c\,h\,x) = c_c\,(f \circ h)\,x$.

Proposition 4.2 *Let $f : A \to B$ be a function and let G be a GADT. If R is the graph of f, then $\widehat{G}\,R$ is the graph of a partial function, i.e., for any $x : G\,A$, there is at most one $y : G\,B$ such that $\widehat{G}\,R\,x\,y$ is inhabited.*

Proof. Let $x : G\,A$ and $y, y' : G\,B$ be such that $\widehat{G}\,R\,x\,y$ and $\widehat{G}\,R\,x\,y'$ are inhabited. Then $\widehat{G_c}\,R\,(\iota\,x)\,(\iota\,y)$ and $\widehat{G_c}\,R\,(\iota\,x)\,(\iota\,y')$ are inhabited as well. Since G_c has a map function, $\widehat{G_c}\,R$ is also the graph of a function, so $\iota\,y = \iota\,y'$, and thus the injectivity of the embedding ι implies $y = y'$. □

Using Proposition 4.2, we can *define* a "function mapping operation" m_G for G by declaring $\mathsf{m}_G\,f\,x$ to be y if $\widehat{G}\,R\,x\,y$ is inhabited and undefined if no such y exists. This definition of m_G generalizes the notion of mappability introduced in [JC22]: when f is mappable over x in the specification G in the sense of [JC22], then the partial function $\mathsf{m}_G\,f$ above is defined on x. The converse does not hold, however, as shown in Example 4.3.

Example 4.3 Consider the functions $f : \mathsf{Bool} \times \mathsf{Bool} \to \mathsf{Bool} \times \mathsf{Bool}$ and $g : \mathsf{Bool} \to \mathsf{Bool}$ defined by $f\,(x, y) = (y, x)$ and $g\,x = \neg x$. It should be fairly intuitive to the reader that we can map (for Seq) the function $f \times g$ over the sequence $s = \mathsf{pairing}\,(\mathsf{inj}\ (\mathsf{true}, \mathsf{false}))\,(\mathsf{inj}\ \mathsf{true})$, with result $s' = \mathsf{pairing}\,(\mathsf{inj}\ (\mathsf{false}, \mathsf{true}))\,(\mathsf{inj}\ \mathsf{false})$. More formally, according to the algorithm of [JC22], $f \times g$ is indeed mappable over s in the specification Seq, with result s'. In fact, writing R for the graph of $f \times g$, we can give an actual witness of $\widehat{\mathsf{Seq}}\ R\,s\,s'$, namely

$$\widehat{\mathsf{p}}\,R_1\,R_2\,R\,(\mathsf{i}\ (\mathsf{true}, \mathsf{false}))\,(\mathsf{i}\ \mathsf{true})\,(\mathsf{i}\ (\mathsf{false}, \mathsf{true}))\,(\mathsf{i}\ \mathsf{false})\,\mathsf{id}\,\mathsf{id}\,w\,(\widehat{\mathsf{i}}\ R_1\ (\mathsf{true}, \mathsf{false})\ (\mathsf{false}, \mathsf{true})\ w_1)\ (\widehat{\mathsf{i}}\ R_2\,\mathsf{true}\,\mathsf{false}\,w_2)$$

Here, R_1 is the graph of f, R_2 is the graph of g, w is a proof of inclusion (actually equality) of $R_1 \mathbin{\widehat{\times}} R_2$ in R, and w_1 and w_2 are witnesses of $f\,(\mathsf{true}, \mathsf{false}) = (\mathsf{false}, \mathsf{true})$ and $g\,\mathsf{true} = \mathsf{false}$, respectively. However, the partial function $\mathsf{m}_{\mathsf{Seq}}\,(f \times g)$ is also defined on elements on which the algorithm of [JC22] would not consider $f \times g$ to be mappable, such as $t = \mathsf{pairing}\,(\mathsf{pairing}\,(\mathsf{inj}\ \mathsf{true})\,(\mathsf{inj}\ \mathsf{false}))\,(\mathsf{inj}\ \mathsf{true})$. Indeed, the algorithm finds $f \times g$ to not be mappable over t because f is not of the form $f_1 \times f_2$ but the first argument of the outer $\mathsf{pairing}$ in t is again constructed from $\mathsf{pairing}$. Nevertheless, $\mathsf{m}_{\mathsf{Seq}}\,(f \times g)\,t$ still exists and equals $t' = \mathsf{pairing}\,(\mathsf{pairing}\,(\mathsf{inj}\ \mathsf{false})\,(\mathsf{inj}\ \mathsf{true}))\,(\mathsf{inj}\ \mathsf{false})$ because there is a witness of $\widehat{\mathsf{Seq}}\ R\,t\,t'$, namely

$$\widehat{\mathsf{p}}\,R_1\,R_2\,R\,(\mathsf{p}\ \mathsf{id}\ (\mathsf{i}\ \mathsf{true})\ (\mathsf{i}\ \mathsf{false}))\,(\mathsf{i}\ \mathsf{true})\,(\mathsf{p}\ \mathsf{id}\ (\mathsf{i}\ \mathsf{false})\ (\mathsf{i}\ \mathsf{true}))\,(\mathsf{i}\ \mathsf{false})\,\mathsf{id}\,\mathsf{id}\,w\,w_1\,(\widehat{\mathsf{i}}\ R_2\,\mathsf{true}\,\mathsf{false}\,w_2)$$

Here, R_2 is again the graph of g, and w_2 is again a witness of $g\,\mathsf{true} = \mathsf{false}$, but R_1 is the relation that only relates $(\mathsf{true}, \mathsf{false})$ with $(\mathsf{false}, \mathsf{true})$ (and nothing else), w is a witness that $R_1 \mathbin{\widehat{\times}} R_2$ is included (strictly!) in R, and w_1 is of the form

$$\widehat{\mathsf{p}}\,R_1'\,R_1''\,R_1\,(\mathsf{i}\ \mathsf{true})\,(\mathsf{i}\ \mathsf{false})\,(\mathsf{i}\ \mathsf{false})\,(\mathsf{i}\ \mathsf{true})\,\mathsf{id}\,\mathsf{id}\,z\,(\widehat{\mathsf{i}}\ R_1'\,\mathsf{true}\,\mathsf{false}\,w_1')\,(\widehat{\mathsf{i}}\ R_1''\,\mathsf{false}\,\mathsf{true}\,w_1'')$$

Here, R_1' relates true with false (and nothing else) with witness w_1', R_1'' relates false with true (and nothing else) with witness w_1'', and z is a witness of inclusion (actually equality) of $R_1' \mathbin{\widehat{\times}} R_1''$ in R_1.

5 Issues

The research programme outlined above seems eminently reasonable. However, to carry it out precisely, and thus to obtain results such as those in Section 4, we need to define a source language extending System F with built-in GADTs and a target (dependent) type theory strong enough to express the relational interpretations described in Sections 2 and 3. In trying to do so, the following issues concerning the target type theory arise:

(i) We have chosen to model relations between A and B as functions $A \to B \to \mathsf{Set}$, i.e., as proof-relevant relations. However, we have also freely used notions most naturally associated to proof-irrelevant relations, such as inclusion of relations. A rigorous treatment would either replace inclusion with an operation mapping witnesses to witnesses, or replace Set by a proof-irrelevant sort of propositions such as Coq's Prop. The former choice amounts to representing relations as spans and replacing inclusions of relations by morphisms between those spans, whereas the latter choice provides direct support for the proof-irrelevant notions used herein.

(ii) In Section 4.2, we investigated a graph lemma for GADTs. There again, proof-relevance plays an important role. Indeed, the graph of a function $f : A \to B$ is the relation $\mathsf{Gr}\, f : \mathsf{Rel}\ A\, B$ defined by $\mathsf{Gr}\, f\, a\, b = (f\, a \equiv b)$, where $\equiv$ is the equality type former of the prospective target type theory. Whether or not this relation $\mathsf{Gr}\, f$ is proof-relevant depends on whether or not the target type theory supports a version of Axiom K. Assuming Axiom K is, however, problematic: in a language like Agda, for example, Axiom K makes it possible to prove that the data constructors of inductive data types are injective. This is in direct opposition to the data constructors' expected parametric behavior.

(iii) To interpret GADTs correctly, the target type theory needs an impredicative universe that supports inductive constructions. The Calculus of Inductive Constructions offers such a universe (namely, Prop) at the bottom of its hierarchy. However, this universe is proof-irrelevant, and thus is not suitable for our purposes. Indeed, carrying out our constructions in such a universe would effectively identify all data structures of any given instance of a GADT. In addition, it is well-known that impredicativity is inconsistent with strong dependent sums, which eliminates some obvious candidates for the target type theory.

Resolving Point (i) may require new technical ideas, but we do not expect it to pose fundamental difficulties. Points (ii) and (iii) pose a different challenge: either we design a target type theory with the desired features, or we prove that we cannot. The former would provide a framework for understanding parametricity of (languages with) GADTs. The latter would definitively show that GADTs understood as types of *data structures* cannot be parametric, and thus the claim that GADTs *generalize* ADTs (implicit in the "GADT" terminology) is not justified.

Acknowledgment. This work was supported by NSF awards CCF1906388 and CCF2203217.

References

[BJP10] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Parametricity and dependent types. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, page 345–356. Association for Computing Machinery, 2010.

[Gir72] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, University of Paris, 1972.

[GLP93] Andy Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture*, pages 223–232, 1993.

[JC22] Patricia Johann and Pierre Cagne. Characterizing functions mappable over GADTs. In Ilya Sergey, editor, *Programming Languages and Systems*, volume 13658 of *Lecture Notes in Computer Science*, pages 135–154. Springer, 2022.

[JG07] Patricia Johann and Neil Ghani. Initial Algebra Semantics Is Enough! In *Typed Lambda Calculus and Applications*, volume 4583 of *Lecture Notes in Computer Science*, page 207–222. Springer Berlin Heidelberg, 2007.

[JP19] Patricia Johann and Andrew Polonsky. Higher-Kinded Data Types: Syntax and Semantics. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 1–13, 2019.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque Sur La Programmation*, pages 408—423. Springer-Verlag, 1974.

[Rey83] John C. Reynolds. Types, Abstraction and Parametric Polymorphism. In Richard Edward Allison Mason, editor, *Proceedings of the IFIP 9th World Computer Congress*, volume 83 of *Information Processing*, pages 513–523. North-Holland/IFIP, 1983.

[SSSB24] Filip Sieczkowski, Sergei Stepanenko, Jonathan Sterling, and Lars Birkedal. The Essence of Generalized Algebraic Data Types. In Michael Hicks, editor, *Proceedings of the ACM on Programming Languages*, volume 8 of *POPL*. Association for Computing Machinery, jan 2024.

[Wad89] Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, page 347–359. Association for Computing Machinery, 1989.