Gradual Algebraic Data Types

MICHAEL GREENBERG, Pomona College, USA
STEFAN MALEWSKI, University of Santiago of Chile, Chile
ÉRIC TANTER, University of Chile, Chile

Algebraic data types are a distinctive feature of statically typed functional programming languages. Existing gradual typing systems support algebraic data types with set-theoretic approaches, e.g., union and intersection types. What would it look like for a gradual typing system to support algebraic data types directly? We describe our early explorations of the design space of gradually typed algebraic data types using AGT [Garcia et al. 2016].

ACM Reference Format:
Michael Greenberg, Stefan Malewski, and Éric Tanter. 2020. Gradual Algebraic Data Types. In *Informal Proceedings of the first ACM SIGPLAN Workshop on Gradual Typing (WGT20).* ACM, New York, NY, USA, 12 pages.

1 INTRODUCTION

Algebraic data types (ADTs) are one of the defining features of statically typed functional programming languages: it is difficult to imagine writing OCaml, Haskell, or Coq code without making use of algebraic data types (via the `type`, `data`, and `Inductive` keywords, respectively). While they are a critical feature, existing work on gradual types has taken a set-theoretic rather than algebraic approach [Castagna and Lanvin 2017; Castagna et al. 2019; Siek and Tobin-Hochstadt 2016]. What would it look like to relax the typing discipline of a language with ADTs?

We present progress on an account of a gradual type system with support for ADTs, in three parts:

- A series of motivating examples identifying the kinds of programs that are challenging to write in existing static regimes but may be easier with gradual typing (Section 2);
- A relaxation of a statically typed system with ADTs (λ_{DT}, Section 3) to include the *unknown type* ($\lambda_{\mathrm{DT?}}$, Section 4); and
- A relaxation of λ_{DT} to include open data types and *unknown constructors*, possibly generated at runtime ($\lambda_{\mathrm{DT¿}}$, Section 5).

As a general approach, we follow AGT [Garcia et al. 2016], deriving gradual systems from static system by means of Galois connections. We have by no means exhausted the design space, and critical questions remain both within our work itself and in how our work relates to existing work; we discuss these issues in Section 6 and anticipate further, productive discussion at the workshop.

2 EXAMPLES

Early in the design of any gradual type system, it is critical to ask: what programs am I seeking to allow that were previously disallowed by more rigid static checking? We offer two examples, both drawn from Greenberg's challenge problems [Greenberg 2019]: the `flatten` function on arbitrarily nested lists (i.e., heterogenous tries); and XML processing. We give our examples in the concrete syntax of our web-based prototype.[1]

[1]https://agtadt.herokuapp.com

WGT20, January 25, 2020, New Orleans
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.

Lest we get mired in the "Turing tarpit" [Perlis 1982], we should make it very clear that existing languages of all stripes can more or less handle these examples: CDuce can handle them gracefully [Benzaken et al. 2003], while Haskell and OCaml do so with more difficulty (see Section 6 and Greenberg [2019]). Our question is not one of *whether* these programs can be written, but *how* we write them.

2.1 ADTs + gradual types = `flatten`

One of the canonical dynamic idioms is the `flatten` function [Fagan 1991], which amounts to a linearization of a heterogeneous list of lists (of lists, of lists, etc.). In Scheme:

```
(define (flatten l)
  (cond
    [(null? l) l]
    [(cons? l) (append (flatten (car l)) (flatten (cdr l)))]
    [else      (list l)]))
```

The `flatten` function is canonical because it is so simple, yet conventional static type systems cannot accommodate the type of the variable `l`.[2]

In our prototype, we can express `flatten` with only a minimum of additional annotation by giving `l` the unknown type:

```
let flatten (l:?) =
  match l with
    | Nil => Nil
    | Cons v l' => append (flatten v) (flatten l')
    | _ => Cons l Nil
  end
```

The definition of `flatten` relies on the definition of a list ADT where the values can be of any type, i.e., heterogeneous lists.[3]

```
data List = Nil | Cons ? List
```

The definition of append is conventional and entirely statically typed. When we evaluate a call to append, no dynamic checks will be necessary.

```
let append (l1:List) (l2:List) =
  match l1 with
    | Nil => l2
    | Cons v l1' => Cons v (append l1' l2)
  end
```

The most interesting part of the implementation is `flatten` itself. Scheme uses cond and predicates like `null?` and `cons?`; these predicates have (notional) type ? → bool. Our implementation, on the other hand, uses a conventional pattern match on `l`, a variable of unknown type ?. The code for `flatten` has three cases: one for each `List` constructor, and a catch-all case for values of any other type. We have reached a critical design question:

> Which constructors can/may/must we match on for dynamically typed scrutinees?

[2]Complicated circumlocutions using typeclasses or `Dynamic` allow Haskell to express `flatten`, but it is difficult to imagine actually using these approaches for the sort of ad hoc programming that calls for functions like `flatten`. In any case, `flatten` is merely emblematic of the *kinds* of programs we might want to write.

[3]Readers may expect polymorphic lists, but we've restricted ourselves to monomorphic types so far. See Section 6.

Our intention here is to have the branches of the match behave exactly like the corresponding cases of the Scheme cond, matching the empty list, a cons cell, and everything else, respectively. The precise choices we make for our static model of ADTs will yield different answers; see the discussion of the predicate complete in Section 3.1.

2.2 Open data types for XML processing

Dynamic languages can deal with semi-structured data—XML, JSON, etc.—with loosely structured data types, e.g., S-XML [Kiselyov 2002]. Statically typed languages tend to either use 'stringy' representations or fixed, static schemata. Can gradual typing help here?

Inspired by S-XML, we define XML data as either (a) textual CDATA or (b) tags represented as constructors. In our prototype, we might write:

```
data Attribute = ¿
data XML = Text String | ¿
parseXML : String -> XML
```

The `Text` constructor models CDATA. The constructors representing tags will be determined at runtime by the `parseXML` function. The technical means we use to accomplish this is the *unknown constructor*, written ¿. When a datatype is defined with ¿ in its constructor list, like `XML` or `Attribute`, then that ADT is *open*, i.e, it can include arbitrary other constructors.

The `parseXML` function takes advantage of `XML`'s openness by mapping tag names to constructors. Let us assume that, as its output contract, `parseXML` guarantees that each constructor will take two arguments: a list of attributes and a list of children. As a concrete example, consider the following XML:

```
<books>
  <book id="1" title="Solaris">
    <author name="Stanislaw Lem" age="98">
  </book>
  <book id="2" title="Foundation">
    <author name="Isaac Asimov">
    <review>Best of the series!</review>
  </book>
</books>
```

The output of `parseXML` ought to have the following form, using Haskell list notation for legibility:

```
Books [] [
  Book [Id 1, Title "Solaris"] [
    Author [Name "Stanislaw Lem", Age 98] []
  ],
  Book [Id 2, Title "Foundation"] [
    Author [Name "Isaac Asimov"] [],
    Review [] [Text "Best of the series!"]
  ]
]
```

Each attribute corresponds to a unary constructor whose name matches the attribute name and whose argument encodes the attribute value, e.g., `id="1"` corresponds to the `Id` constructor applied to the number `1`. Each tag corresponds to a binary constructor where (a) the name matches the tag name, (b) the first argument to the constructor is a list of attributes (of type `Attribute`), and

(c) the second argument to the constructor is a list of child elements (of type `XML`). (To be clear, we cannot yet implement `parseXML` in our prototype implementation; see Section 6.)

Now, let us say we want to collect all the `author` names in this structure, i.e., the child elements of every <author> tag. As with any standard ADT, pattern matching is our best option.

```
let collectAuthors (xml : XML) : List =
  match xml with
  | Text str => Nil
  | Author attrs elems => Cons attrs Nil
  | ¿ attrs elems => concat (map collectAuthors elems)
  end
```

The `XML` data type declares only one constructor, `Text`, which represents CDATA; other tags are represented using unknown constructors. The `collectAuthors` function first explicitly matches on `Text`, to ignore it. It also explicitly matches on the `Author` constructor (which models <author> tags), returning the attributes, `attrs`. The last case matches all other tags; the pattern `¿ attrs elems` matches any constructor with two arguments, i.e., any other tag.

Running `collectAuthors` on the example XML above yields the following list of attributes:[4]

```
[[Name "Stanislaw Lem", Age 98],
 [Name "Isaac Asimov"]]
```

Continuing with the example, we can continuously evolve the datatypes making them progressively more static, e.g. by adding constructors to them as we have certainty of their shape and name. After adding the constructors relevant to `collectAuthors` the data definitions may look as follows:

```
data Attribute = Age Int | Name String | ¿
data XML = Text String | Author List List | ¿
```

Eventually, we can make both types fully static adding every tag and attribute used in the XML as a constructor to the respective ADT and closing them (removing ¿ from the list of variants). Interestingly, the functions that use these datatypes shouldn't change in this process.

3 A STATIC MODEL OF ADTS

To use AGT [Garcia et al. 2016] to derive a gradually-typed language, one must start with a statically-typed language and then provide an interpretation of gradual types in terms of sets of static types. Our static system, λ_{DT}, is a monomorphic lambda calculus with support for algebraic data type definitions (Figure 1). At the term level, λ_{DT} extends the simply typed lambda calculus with constants k, type ascriptions $(e :: T)$, constructor applications $c\ e$, and pattern matching. We use the metavariable k for constants, c for constructors, and C for sets of constructors. As a notational convention, we put lines over vectors of terms (see, e.g., Match). We write $A = B$ to mean the proposition that A and B are equal; we write $A \doteq B$ to mean that A is defined to be B.

At the type level, λ_{DT} collects algebraic data types in a *datatype context* Δ, which maps each ADT A to a set of constructors C; the types of individual constructors are held in *constructor contexts*

[4]Sadly there is no way in our model of fully enforcing `parseXML`'s output contract: how can we ensure that each tag's constructor takes exactly two arguments, a list of attributes and a list of child elements? We haven't fully worked out the formal details, but we can imagine *partially* open ADTs that only accept unknown constructors with a specific shape, as in `data Attribute = #? String | #? Int`.

Further, a reviewer notes that our behavior on a `<text>` tag is ambiguous! A real implementation would have to somehow distinguish such unknown constructors from defined ones.

Datatypes			
Datatype names	A	$\in$	DTName
Constructor names	c	$\in$	CtorName
Constructor sets	C	$\in$	Ctors $\subseteq$ CtorName
Types			
Base types	B	$::=$	$\mathsf{int} \mid \ldots$
Types	T	$::=$	$B \mid T \to T \mid A$
Contexts			
Type contexts	Γ	$::=$	$\cdot \mid \Gamma, x : T$
Datatype contexts	Δ	$::=$	$\cdot \mid \Delta, A : C$
Constructor contexts	Ξ	$::=$	$\cdot \mid \Xi, c : T \times \cdots \times T$
Terms			
Expressions	e	$::=$	$x \mid k \mid e\ e \mid \lambda x{:}T.\ e \mid (e :: T) \mid c\ e \ldots e \mid$ $\mathsf{match}\ e\ \mathsf{with}\ \{c\ x \ldots x \mapsto e; \ldots; c\ x \ldots x \mapsto e\}$
Values	v	$::=$	$k \mid \lambda x{:}T.\ e \mid c\ v \ldots v$
Constants	k	$::=$	$0 \mid 1 \mid \ldots$ $+ \mid \ldots$

Fig. 1. λ_{DT} syntax

Ξ, which maps each c to a product of zero or more types.[5] Our model calculus assumes that you provide a single, global Δ and Ξ for all algebraic datatypes. Our prototype uses the syntax in Section 2.

The static semantics for λ_{DT} is more or less standard (Figure 2), though some care should be taken with the context well-formedness rules for Δ and Ξ. Well-formed datatype contexts Δ never assign the same constructor to two different datatypes. Well-formed constructor contexts Ξ assign constructors well-formed types to constructors that are already associated with an ADT A in Δ. We have omitted the type well-formedness rules, which demand that all referenced ADTs A be defined in Δ. Our formal model does not have recursion, but our prototype does.

The typing rules for constructors (Ctor) and pattern matching (Match) make use of some helper functions and predicates—making these helpers explicit is part of the AGT approach [Garcia et al. 2016]. The interesting helpers here do characterize: which type a constructor belongs to (cty_Δ), the arity and argument types of a given constructor (carg_Ξ), whether or not a match is complete ($\mathrm{complete}_\Delta$), and whether or not the branches of a match all return the same type (equate_n). To find the type of a constructor c, the helper $\mathrm{cty}_\Delta(c)$ does a reverse lookup in the datatype context, Δ. Looking up the arguments of a constructor c by $\mathrm{carg}_\Xi(c)$ merely looks up the given constructor in the constructor context Ξ. According to $\mathrm{complete}_\Delta(C, A)$, the constructors C are a complete match for A when C is exactly the set of A's constructors in Δ. And, finally, equate_n returns its identical inputs or is undefined on non-equal inputs.

[5] We could have instead added tuples and the unit type to the language and had each constructor take a single argument, but our approach seems more fundamental as we can then *derive* the unit and tuple types (at the cost of some metatheoretic complexity).

Context well-formedness rules $\boxed{\vdash \Delta}$ $\boxed{\Delta \vdash \Xi}$ $\boxed{\Delta; \Xi \vdash \Gamma}$

$$\frac{}{\vdash \cdot}\ \Delta\text{-Empty} \qquad \frac{\vdash \Delta \quad A \notin \mathrm{dom}(\Delta) \quad \forall A' \in \Delta,\ \Delta(A') \cap C = \emptyset}{\vdash \Delta, A : C}\ \Delta\text{-Ext}$$

$$\frac{\vdash \Delta}{\Delta \vdash \cdot}\ \Xi\text{-Empty} \qquad \frac{\Delta \vdash \Xi \quad \Delta; \Xi \vdash T_i \quad \exists A \in \mathrm{dom}(\Delta),\ \mathrm{cty}_\Delta(c) = A}{\Delta \vdash \Xi, c : T_1 \times \cdots \times T_n}\ \Xi\text{-Ext}$$

$$\frac{\Delta \vdash \Xi}{\Delta; \Xi \vdash \cdot}\ \Gamma\text{-Empty} \qquad \frac{\Delta; \Xi \vdash \Gamma \quad \Delta; \Xi \vdash T}{\Delta; \Xi \vdash \Gamma, x : T}\ \Gamma\text{-Ext}$$

Typing rules $\boxed{\Delta; \Xi; \Gamma \vdash e : T}$

$$\frac{\Delta; \Xi \vdash \Gamma \quad T \doteq \Gamma(x)}{\Delta; \Xi; \Gamma \vdash x : T}\ \text{Var} \qquad \frac{\Delta; \Xi \vdash \Gamma \quad T \doteq \mathrm{ty}(k)}{\Delta; \Xi; \Gamma \vdash k : T}\ \text{Const}$$

$$\frac{\Delta; \Xi; \Gamma \vdash e_1 : T_1 \quad \Delta; \Xi; \Gamma \vdash e_2 : T_2 \quad T_2 = \mathrm{dom}(T_1) \quad T \doteq \mathrm{cod}(T_1)}{\Delta; \Xi; \Gamma \vdash e_1\ e_2 : T}\ \text{App}$$

$$\frac{\Delta; \Xi; \Gamma, x : T_1 \vdash e : T_2}{\Delta; \Xi; \Gamma \vdash \lambda x{:}T_1.\ e : T_1 \to T_2}\ \text{Lam} \qquad \frac{\Delta; \Xi; \Gamma \vdash e : T}{\Delta; \Xi; \Gamma \vdash e :: T : T}\ \text{Ascribe}$$

$$\frac{(\forall i.1 \leq i \leq n) \quad \Delta; \Xi; \Gamma \vdash e_i : T_i \quad \mathrm{carg}_\Xi(c) = T_1 \times \cdots \times T_n \quad A \doteq \mathrm{cty}_\Delta(c)}{\Delta; \Xi; \Gamma \vdash c\ e_1\ \ldots\ e_n : A}\ \text{Ctor}$$

$$\frac{\begin{array}{c}(\forall i.1 \leq i \leq n) \quad \Delta; \Xi; \Gamma \vdash e : T \quad \mathrm{complete}_\Delta(\{c_1, \ldots, c_n\}, T) \quad T \doteq \mathrm{equate}_n(T_1', \ldots, T_n') \\ T_{i1} \times \cdots \times T_{im_i} \doteq \mathrm{carg}_\Xi(c_i) \quad \Delta; \Xi; \Gamma, x_{i1} : T_{i1}, \ldots, x_{im_i} : T_{im_i} \vdash e_i : T_i'\end{array}}{\Delta; \Xi; \Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ \{c_1\ x_{11} \ldots x_{1m_1} \mapsto e_1; \ldots; c_n\ x_{n1} \ldots x_{nm_n} \mapsto e_n\} : T}\ \text{Match}$$

Helpers

$$\mathrm{cty}_\Delta(c) = \begin{cases} A & c \in \Delta(A) \\ \bot & \text{otherwise} \end{cases} \qquad \mathrm{carg}_\Xi(c) = \Xi(c) \qquad \mathrm{complete}_\Delta(C, A) \Leftrightarrow \Delta(A) = C$$

$$\begin{array}{rcl} \mathrm{dom}(T_1 \to T_2) & = & T_1 \\ \mathrm{dom}(\_) & = & \bot \end{array} \quad \begin{array}{rcl} \mathrm{cod}(T_1 \to T_2) & = & T_2 \\ \mathrm{cod}(\_) & = & \bot \end{array} \quad \begin{array}{rcl} \mathrm{equate}_n(T, \ldots, T) & = & T \\ \mathrm{equate}_n(\_, \ldots, \_) & = & \bot \end{array}$$

Fig. 2. λ_{DT} static semantics

We omit the completely conventional operational semantics, but we have defined them formally as call-by-value small-step semantics using reduction frames. We have proved our static semantics sound using the standard, syntactic, progress/preservation-based approach [Wright and Felleisen 1994].

3.1 The design space of complete match expressions

The complete predicate is used in Match to determine whether a pattern match sufficiently covers the type of the scrutinee. If a pattern match's set of constructors do not pass complete's muster,

```
data A = A0 | A1 in
data B = B0 | B1 in

match A0 with
| A0 -> 0
end
```

(a) A *sober*-only program

```
data A = A0 | A1 in
data B = B0 | B1 in

match A0 with
| A0 -> 0
| A1 -> 1
end
```

(b) An *exact* program (that type checks in *sober* and *complete*, too)

```
data A = A0 | A1 in
data B = B0 | B1 in

match A0 with
| A0 -> 0
| A1 -> 1
| B0 -> 2
end
```

(c) A *complete*-only program

Fig. 3. Programs for each non-trivial complete$_\Delta$ semantics

then the program does not typecheck. We have identified four possible semantics for complete, each of which treats the set of constructors C forming the branches of the match differently in terms of the hypothetical ADT A it is matching against:

- *Sober:* $\Delta(A) \supseteq C$
  The set of constructors C need not be exhaustive, but only a single ADT's constructors can be used.
- *Exact:* $\Delta(A) = C$
  The set of constructors C exactly matches the constructors of an ADT A.
- *Complete:* $\Delta(A) \subseteq C$
  The set of constructors C matches all of the constructors of an ADT A, but may match others.
- *Whatever:* $\top$
  All matches are considered complete.

The general relationship between the four possible semantics is implied by the underlying relations: $\supseteq$ for sober; $=$ for exact; and $\subseteq$ for complete; and a trivial, total relation for whatever. We can expect pattern matches that type check in the exact regime to typecheck in the other two non-trivial ones, but the differences are perhaps best understood by example: three programs suffice to distinguish the non-trivial semantics (Figure 3). We ignore the fourth, trivial 'whatever' semantics in this comparison. The *sober* semantics allows incomplete matches—like Haskell without `-fwarn-incomplete-patterns`. Therefore this semantics cannot guarantee that a match does not get stuck. The first program (Figure 3a) only typechecks using the *sober* semantics, because the match expression only has cases for constructor from `A`. The second program (Figure 3b) typechecks with the *exact* (and all other) semantics. The third program (Figure 3c) only typechecks with the *complete* semantics. The match expression has a case for each constructor in `A`—and some additional ones.

For conventional notions of errors and soundness, sober is unsound (change the scrutinee to `A1` in Figure 3a) and both exact and complete are sound. To our knowledge, no statically typed language follows the complete semantics, presumably because such unreachable cases indicate some kind of logic error. But matching on constructors from more than one ADT seems like quite a useful feature in a gradually typed system. The exact and complete semantics are both appealing candidates, and both yield interesting gradually typed systems under AGT.

$$
\begin{array}{llll}
\textit{Types} & G & ::= & B \mid G \to G \mid A \mid ? \\
\textit{Terms} & e & ::= & v \mid x \mid \varepsilon e :: G \mid e\ e \mid c\ e \ldots e \mid \\
 & & & \mathsf{match}\ e\ \mathsf{with}\ \{c\ x \ldots x \mapsto e; \ldots; c\ x \ldots x \mapsto e\} \\
\textit{Patterns} & p & ::= & c \mid \_
\end{array}
$$

Fig. 4. $\lambda_{\mathrm{DT?}}$ syntax

4 GRADUAL ADTS

We use AGT to to lift λ_{DT} into $\lambda_{\mathrm{DT?}}$. The general idea is as follows: we extend the syntax of λ_{DT} to use gradual types G, which include the unknown type ? (Figure 4). We then define two functions: a concretization function γ, which maps gradual types G to sets of static λ_{DT} types; and an abstraction function α, which maps sets of static types back to gradual types. The core AGT methodology generates rules and helpers for the gradual calculus (here, $\lambda_{\mathrm{DT?}}$) by 'lifting' the corresponding parts of the static calculus (here, λ_{DT}) via γ and α (Figure 5).

The lifted versions of functions are marked with a tilde. While we generally refer those not familiar with the technique to the original paper [Garcia et al. 2016], the rule for $\widetilde{\mathsf{equate}}_n(G_1, \ldots, G_n)$ is exemplary: to apply T-type operator to a G-type, we use the original predicate on each type in $\gamma(G)$; to get back from the set of T-types to a G-type, use α. Some of our predicates use only one of γ or α—the other direction isn't necessary. For example, $\widetilde{\mathsf{complete}}_\Delta(C, G)$ only needs γ and $\widetilde{\mathsf{carg}}_\Xi(c)$ only needs α. We compute concise forms for each of our lifted predicates directly in Figure 5—though of course these concise forms must be proved correct, not merely sketched. A wildcard (_) can be used in a pattern match to match anything, even non constructors. This is crucial to allow expressions of an unknown type (?) to be matched in match expressions as seen in the example of Section 2.1.

We have developed a version of $\lambda_{\mathrm{DT?}}$ that uses *evidence* to derive an operational semantics with runtime checks. We omit it here beyond the syntactic term $\epsilon e :: G$, but suffice to say that the conventional approach generates a semantics that can appropriately combine ADTs and the unknown type such that there are no stuck states at runtime—only correct runs and evidence failures (which correspond to failed casts).

5 GRADUAL CONSTRUCTORS

While $\lambda_{\mathrm{DT?}}$ extends λ_{DT} by adding the unknown type ?, we can extend λ_{DT} on a different axis: by adding ¿, the *unknown constructor*, and $?^{\mathrm{A}}$, the type of *unknown open datatypes*. Where ? is interpreted by γ as the set of all possible types, ¿ is interpreted as the set of all possible constructors and $?^{\mathrm{A}}$ as the set of all open datatypes (Figure 6). The unknown constructor can be listed in $\Delta(A)$ as a constructor of A to indicate that A is an *open* datatype, where arbitrary new constructors may appear. The unknown constructor can be listed in a pattern match to match arbitrary constructors of a given arity (Match). The unknown constructor cannot, however, be used to actually construct data! One must actually name the constructor. Some sort of syntactic affordance may be needed to differentiate the three kinds of constructors: statically known and in Δ; statically known but not mentioned in Δ; and dynamically generated. (See Section 6 for more discussion.) Superscripts on constructors make arity explicit; explicit notation ensures that $\widetilde{\mathsf{carg}}_\Xi$ of an unknown constructor returns the correct number of types. We have made several particular choices here: according to $\widetilde{\mathsf{carg}}_\Xi(¿)$, arbitrary constructors take arguments of type ?, i.e., any type; complete matches are those where the statically listed constructors cover all of the possibilites of *some* datatype.

$$\begin{array}{rcl} P & \subseteq & \mathsf{CtorName} \cup \{\_\} \\ p & ::= & c \mid \_ \end{array}$$

Galois connection $\boxed{\gamma : \mathsf{GType} \to \mathcal{P}^*(\mathsf{Type})}$ $\boxed{\alpha : \mathcal{P}^*(\mathsf{Type}) \to \mathsf{GType}}$ $\boxed{\gamma_\times : \vec{\mathsf{GType}} \to \mathcal{P}^*(\vec{\mathsf{Type}})}$ $\boxed{\alpha_\times : \mathcal{P}^*(\vec{\mathsf{Type}}) \to \vec{\mathsf{GType}}}$

$$\begin{array}{rcl} \gamma(B) & = & \{B\} \\ \gamma(A) & = & \{A\} \\ \gamma(G_1 \to G_2) & = & \{T_1 \to T_2 \mid T_1 \in \gamma(G_1), T_2 \in \gamma(G_2)\} \\ \gamma(?) & = & \mathsf{Type} \end{array}$$

$$\begin{array}{rcl} \alpha(\{B\}) & = & B \\ \alpha(\{A\}) & = & A \\ \alpha(\left\{\overline{T_{i1} \to T_{i2}}\right\}) & = & \alpha(\left\{\overline{T_{i1}}\right\}) \to \alpha(\left\{\overline{T_{i2}}\right\}) \\ \alpha(\left\{\overline{T}\right\}) & = & ? \quad \text{otherwise} \end{array}$$

$$\gamma(G_1 \times \cdots \times G_n) \;=\; \{T_1 \times \cdots \times T_n \mid T_i \in \gamma(G_i)\}$$

$$\begin{array}{rcl} \alpha(\left\{\overline{T_{i1} \times \cdots \times T_{in}}\right\}) & = & \alpha(\left\{\overline{T_{i1}}\right\}) \times \cdots \times \alpha(\left\{\overline{T_{in}}\right\}) \\ \alpha(\left\{\overline{T}\right\}) & = & \{?, ? \times ?, ? \times ? \times ?, \ldots\} \end{array}$$

Static semantics $\boxed{\Delta; \Xi; \Gamma \vdash e : G}$

$$\frac{\begin{array}{c}(\forall i.1 \le i \le n) \qquad \Delta; \Xi; \Gamma \vdash e_i : G_i' \qquad \Delta; \Xi \vdash G_i \sim G_i' \\ G_1 \times \cdots \times G_n \doteq \widetilde{\mathsf{carg}}_\Xi(c) \qquad A \doteq \mathsf{cty}_\Delta(c)\end{array}}{\Delta; \Xi; \Gamma \vdash c\; e_1 \ldots e_n : A}\;\textsc{Ctor}$$

$$\frac{\begin{array}{c}(\forall i.1 \le i \le n) \qquad \Delta; \Xi; \Gamma \vdash e : G \qquad \widetilde{\mathsf{complete}}_\Delta(\{p_1, \ldots, p_n\}, G) \qquad G_{i1} \times \cdots \times G_{im_i} \doteq \widetilde{\mathsf{carg}}_\Xi(p_i) \\ \Delta; \Xi; \Gamma, x_{i1} : G_{i1}', \ldots, x_{im_i} : G_{im_i}' \vdash e_i : G_i' \qquad \Delta; \Xi \vdash G_{i1}' \sim G_{i1} \ldots \Delta; \Xi \vdash G_{im_i}' \sim G_{im_i}\end{array}}{\Delta; \Xi; \Gamma \vdash \mathsf{match}\; e\; \mathsf{with}\; \{p_1\; x_{11} \ldots x_{1m_1} \mapsto e_1; \ldots; p_n\; x_{n1} \ldots x_{nm_n} \mapsto e_n\} : \widetilde{\mathsf{equate}}_{\Delta n}(\{G_1', \ldots, G_n'\})}\;\textsc{Match}$$

Helpers

$$\widetilde{\mathsf{complete}}_\Delta(P, G) = \_ \in P \lor \forall T \in \gamma(G),\; \mathsf{complete}_\Delta(P \setminus \{\_\}, T)$$

$$\begin{aligned} \widetilde{\mathsf{equate}}_n(G_1, \ldots, G_n) &= \alpha(\left\{\mathsf{equate}_n(T_1, \ldots, T_n) \mid T_i \in \gamma(G_i)\right\}) \\ &= \textstyle\bigsqcap_{i=1}^n G_i \end{aligned}$$

$$\begin{array}{rcl} \widetilde{\mathsf{carg}}_\Xi(c) & = & \alpha(\{\Xi(c)\}) = \mathsf{carg}_\Xi(c) \\ \widetilde{\mathsf{carg}}_\Xi(\_) & = & \langle\rangle \end{array}$$

$$\begin{array}{rcl} \widetilde{\mathsf{dom}}(G_1 \to G_2) & = & G_1 \\ \widetilde{\mathsf{dom}}(?) & = & ? \\ \widetilde{\mathsf{dom}}(G) & = & \bot \end{array} \qquad \begin{array}{rcl} \widetilde{\mathsf{cod}}(G_1 \to G_2) & = & G_2 \\ \widetilde{\mathsf{cod}}(?) & = & ? \\ \widetilde{\mathsf{cod}}(G) & = & \bot \end{array}$$

Fig. 5. $\lambda_{\mathrm{DT?}}$ typing rules and predicates

6 DISCUSSION

What about OCaml's polymorphic and extensible variants, Haskell's `Dynamic`, and CDuce [Benzaken et al. 2003; Garrigue 2000; Peyton Jones et al. 2016; Zenger and Odersky 2001]? CDuce, in particular finds it easy to type a function like `flatten`, as shown in Greenberg [2019]. We are slightly embarrassed that we have not yet discovered things that our systems can do that these others cannot! Gradual algebraic data types surely smoothen the path from polymorphic variants to standard data types, but can they capture novel idioms? Perhaps the better question to ask is: how can we integrate CDuce's features with those in more conventional functional languages?

What does it look like to name a constructor not statically included in any datatype? OCaml uses backticks for polymorphic variants. What does it look like to generate constructors at runtime? Technically, we can simply say that there is some function constant $\mathsf{mkCtor} : \mathsf{String} \to ?$. But can traditional, efficient implementations of ADTs accommodate such generated constructors? Extensible variants in OCaml typically know all of the constructor names at link time, while an XML parser would not know the names until run time. Research on open data types and open functions is closely related [Löh and Hinze 2006].

Gradual type systems often talk about their "fully static" and "fully untyped" variants. A reviewer asks what a "fully untyped" program is in our model. Depending on our notion of $\mathsf{complete}_\Delta$, the fully untyped configuration may be $\Delta = \Xi = \cdot$, we revert to plain lambda calculus or it may be $\Delta = \mathtt{Any} : \{¿\}$.

What about models of nested matching? When should we communicate mismatched branch types to the programmer and when should they be coerced to the dynamic type?

Finally, a reviewer points out the connection between our work and New et al.'s work on gradual typing and parametricity [New et al. 2019]. Their dynamic type is a recursive open sum type which can accommodate runtime-generated types, while we are instead interested in runtime-generated constructor names. How might these ideas be related?

Acknowledgments

We heartily thank the WGT reviewers for their diligent reviews and insightful comments.

REFERENCES

Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. 2003. CDuce: an XML-centric general-purpose language. ACM, 51–63. https://doi.org/10.1145/944705.944711

Giuseppe Castagna and Victor Lanvin. 2017. Gradual Typing with Union and Intersection Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 41 (Aug. 2017), 28 pages. https://doi.org/10.1145/3110285

Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. 2019. Gradual Typing: A New Perspective. *Proc. ACM Program. Lang.* 3, POPL, Article 16 (Jan. 2019), 32 pages. https://doi.org/10.1145/3290329

Mike Fagan. 1991. *Soft typing: An approach to type checking for dynamically typed languages.* Ph.D. Dissertation. Rice University. https://scholarship.rice.edu/handle/1911/16439

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting gradual typing. 429–442. https://doi.org/10.1145/2837614.2837670

Jacques Garrigue. 2000. Code reuse through polymorphic variants. In *FOSE*.

Michael Greenberg. 2019. The Dynamic Practice and Static Theory of Gradual Typing. In *3rd Summit on Advances in Programming Languages (SNAPL 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 136. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 6:1–6:20. https://doi.org/10.4230/LIPIcs.SNAPL.2019.6

Oleg Kiselyov. 2002. SXML Specification. *SIGPLAN Not.* 37, 6 (June 2002), 52âĂŞ58. https://doi.org/10.1145/571727.571736

Andres Löh and Ralf Hinze. 2006. Open Data Types and Open Functions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP âĂŹ06)*. Association for Computing Machinery, New York, NY, USA, 133âĂŞ144. https://doi.org/10.1145/1140335.1140352

Max S. New, Dustin Jamner, and Amal Ahmed. 2019. Graduality and Parametricity: Together Again for the First Time. *Proc. ACM Program. Lang.* 4, POPL, Article Article 46 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371114

Alan J. Perlis. 1982. Special Feature: Epigrams on Programming. *SIGPLAN Not.* 17, 9 (Sept. 1982), 7âĂŞ13. https://doi.org/10.1145/947955.1083808

Simon Peyton Jones, Stephanie Weirich, Richard A. Eisenberg, and Dimitrios Vytiniotis. 2016. *A Reflection on Types.* Springer International Publishing, Cham, 292–317. https://doi.org/10.1007/978-3-319-30936-1_16

Jeremy G. Siek and Sam Tobin-Hochstadt. 2016. The Recursive Union of Some Gradual Types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. 388–410. https://doi.org/10.1007/978-3-319-30936-1_21

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115 (1994), 38–94. Issue 1.

Matthias Zenger and Martin Odersky. 2001. Extensible Algebraic Datatypes with Defaults. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01)*. ACM, New York, NY, USA, 241–252. https://doi.org/10.1145/507635.507665

$$\textit{Types} \quad G \;::=\; B \mid G \to G \mid A \mid ? \mid ?^{\mathsf{A}}$$

$$\begin{array}{rcl} O & \subseteq & \mathsf{GtorSet} = \mathsf{CtorName} \cup \{¿, \_\} \\ o & ::= & c \mid \_ \mid ¿ \end{array}$$

Galois connection

$\boxed{\gamma_\Delta : \mathsf{GType} \to \mathcal{P}^*(\mathsf{Type})}$ $\boxed{\alpha : \mathcal{P}^*(\mathsf{Type}) \to \mathsf{GType}}$

$$\begin{array}{rcl} \gamma_\Delta(B) & = & \{B\} \\ \gamma_\Delta(A) & = & \{A\} \\ \gamma_\Delta(G_1 \to G_2) & = & \{T_1 \to T_2 \mid T_1 \in \gamma_\Delta(G_1), T_2 \in \gamma_\Delta(G_2)\} \\ \gamma_\Delta(?^{\mathsf{A}}) & = & \{A \mid ¿ \in \Delta(A)\} \\ \gamma_\Delta(?) & = & \mathsf{Type} \end{array}$$

$$\begin{array}{rcl} \alpha(\{B\}) & = & B \\ \alpha(\{A\}) & = & A \\ \alpha(\left\{\overline{T_{i1} \to T_{i2}}\right\}) & = & \alpha(\left\{\overline{T_{i1}}\right\}) \to \alpha(\left\{\overline{T_{i2}}\right\}) \\ \alpha(\left\{\overline{A}\right\}) & = & ?^{\mathsf{A}} \\ \alpha(\left\{\overline{T}\right\}) & = & ? \quad \text{otherwise} \end{array}$$

$\boxed{\gamma_\Delta : \mathsf{GtorSet} \to \mathcal{P}(\mathsf{Ctors})}$ $\boxed{\alpha : \mathcal{P}(\mathsf{Ctors}) \to \mathsf{GtorSet}}$

$$\gamma_\Delta(O) = \begin{cases} \{O\} & \{¿, \_\} \not\subseteq O \\ \{(O \setminus \{¿, \_\}) \cup C \mid C \in \mathsf{Ctors} \setminus \Delta\} & \text{otherwise} \end{cases} \qquad \alpha(\overline{C}) = \begin{cases} C & \overline{C} = \{C\} \\ \{¿\} \cup \bigcap_{C \in \overline{C}} C & \text{otherwise} \end{cases}$$

Static semantics

$\boxed{\Delta; \Xi; \Gamma \vdash e : G}$

$$\frac{\begin{array}{c}(\forall i.1 \le i \le n) \qquad \Delta; \Xi; \Gamma \vdash e_i : G'_i \qquad \Delta; \Xi \vdash G_i \sim G'_i \\ G_1 \times \cdots \times G_n \doteq \widetilde{\mathsf{carg}}_\Xi(c^n) \qquad G \doteq \widetilde{\mathsf{cty}}_\Delta(c)\end{array}}{\Delta; \Xi; \Gamma \vdash c\ e_1 \ldots e_n : G}\ \textsc{Ctor}$$

$$\frac{\begin{array}{c}(\forall i.1 \le i \le n) \qquad \Delta; \Xi; \Gamma \vdash e : G \qquad \widetilde{\mathsf{complete}}_\Delta(\{o_1, \ldots, o_n\}, G) \qquad G_{i1} \times \cdots \times G_{im_i} \doteq \widetilde{\mathsf{carg}}_\Xi(o_i^{m_i}) \\ \Delta; \Xi; \Gamma, x_{i1} : G'_{i1}, \ldots, x_{im_i} : G'_{im_i} \vdash e_i : G'_i \qquad \Delta; \Xi \vdash G'_{i1} \sim G_{i1} \ldots \Delta; \Xi \vdash G'_{im_i} \sim G_{im_i}\end{array}}{\Delta; \Xi; \Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ \{o_1\ x_{11} \ldots x_{1m_1} \mapsto e_1; \ldots; o_n\ x_{n1} \ldots x_{nm_n} \mapsto e_n\} : \widetilde{\mathsf{equate}}_{\Delta n}(\left\{G'_1, \ldots, G'_n\right\})}\ \textsc{Match}$$

Helpers

$$\widetilde{\mathsf{cty}}_\Delta(c) = \alpha(\{A \mid c \in \gamma_\Delta(\Delta(A))\})$$

$$\begin{array}{rcl} \widetilde{\mathsf{carg}}_\Xi(c^n) & = & \alpha(\{\Xi(c)\}) = \mathsf{carg}_\Xi(c) \\ \widetilde{\mathsf{carg}}_\Xi(¿^n) & = & ?_1 \times \cdots \times ?_n \\ \widetilde{\mathsf{carg}}_\Xi(\_^n) & = & \langle\rangle \end{array}$$

$$\widetilde{\mathsf{complete}}_\Delta(O, G) = \_ \in O \lor \forall T \in \gamma_\Delta(G),\ \exists C \in \gamma_\Delta(O),\ \mathsf{complete}_\Delta(C, T)$$

$$\begin{array}{rl} \widetilde{\mathsf{equate}}_{\Delta n}(G_1, \ldots, G_n) & = \alpha(\left\{\mathsf{equate}_n(T_1, \ldots, T_n) \mid T_i \in \gamma_\Delta(G_i)\right\}) \\ & = \bigsqcap_{i=1}^{n} G_i \end{array}$$

Fig. 6. $\lambda_{\mathsf{DT}_¿}$ syntax and Galois connection