Gradual Compound Data Types

Pedro Ângelo
Faculdade de Ciências & LIACC,
Universidade do Porto
up201207861@fc.up.pt

Mário Florido
Faculdade de Ciências & LIACC,
Universidade do Porto
amf@dcc.fc.up.pt

Pedro Vasconcelos
Faculdade de Ciências & LIACC,
Universidade do Porto
pbv@dcc.fc.up.pt

Abstract

Gradual type systems have been proposed as a sound mechanism to integrate static and dynamic typing in programming languages. Several gradual type systems have been described in the literature: gradual typing for the lambda calculus, parametric polymorphic languages and extensions to deal with recursive functions and compound data types such as list, sum and product types. Although dynamic type annotations are enabled explicitly in function arguments and list elements, elements of general compound types cannot be explicitly annotated as dynamic. In this paper we present an extension to the original gradual type system which enables to declare compound type elements as static or dynamic and show its expressiveness by applying it to several working examples.

CCS Concepts **•Theory of computation → Type structures; •Software and its engineering → Functional languages;**

Keywords gradual typing, type systems, dynamic types, algebraic data type

1 Introduction

The support of both dynamic and static types in the same language has been an active area of research [1, 3–9, 11–15]. Gradual typing, as described in [11, 12] and extended with the contributions from [4–6], allows different parts of programs to be either dynamically or statically typed. Statically typed programs are type checked before runtime, and all type errors are caught before the program is executed, while dynamically typed programs are inserted with runtime checks that halt program execution once a runtime error is caught. In order to shift from static to dynamic typing in a part of a program, the dynamic type Dyn must be inserted in an abstraction's type annotation. Consider the following example:

$$(\lambda x : \mathsf{Dyn} \ . \ 1 + x) \ \mathsf{true}$$

ICFP '17, Oxford, United Kingdom
© 2017 Copyright held by the owner/author(s). 978-x-xxxx-xxxx-x/YY/MM...$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

The lambda abstraction is annotated with the dynamic type. Therefore, this program will type check. However, since true cannot be added to an integer, a runtime error will occur (called *blame error* [2, 4, 5]) when the program is executed.

Now consider the data type definition (Haskell syntax) of the instance of Maybe with elements of type Int:

$$data\ MaybeInt = Nothing \mid Just\ Int$$

Using algebraic type constructors [10], we can represent *MaybeInt* as the type Unit + Int using a sum type. We may use the standard operations over sum types, such as case, inl and inr to build expressions that interact with values of type *MaybeInt*, define values of type *MaybeInt* with explicit type annotations and apply the expressions to the values.

However, how could we implement the instance of Maybe when the element being contained is of type Dyn? We cannot just annotate inr with the type Unit + Dyn, because the type system and cast insertion procedure [4] do not deal with the dynamic type in an explicit type annotation, unless it is in an abstraction. Besides, the cast insertion procedure would not insert the necessary casts to Dyn that would lead to *blame errors* [2, 5]. In summary, dynamic types can only be inserted in type annotations for abstractions.

Now consider the following example:

$$data\ ListInt = Nil \mid Cons\ Int\ ListInt$$

Here we have two types (*Int* and *ListInt*) in the alternative on the right. Product types are used to represent the combination of *Int* and *ListInt*. Product types have 3 standard terms: pairs, fst and snd. However, there is no way to annotate the pair term with types. Therefore, we could not specify that a certain element of the product type has type Dyn. So, in order to allow elements of a data type to be dynamically typed, we must also allow elements of product and sum types to be dynamically typed.

This paper makes the following main contributions:

1. Extension of the gradual type system and cast insertion rules of [4] to allow elements of product or sum types to be typed with the dynamic type (Section 4).
2. Generalization of products and sums to tuples and variants. The generalizations of product and sum types are necessary to build data types (Section 5).
3. Experimental results from an implementation (Section 8).

The paper is structured as following. In Section 2, we review the current work towards gradual typing, and explain why dynamic data types cannot be currently supported. In

Section 3 we review gradual typing as well as the dynamic type and the consistency relation. In Section 4 and 5 we present our contributions, as stated previously. In Section 6 we briefly discuss recursive types. In Section 7 we present our extension to the type inference algorithm. Section 8 will provide several implementations details and an overview of the system. Finally, in Section 9 we will provide some examples.

2 Motivation

Gradual typing disciplines [4, 5] extend a static type system with a dynamic type and cast insertion rules that insert runtime type casts into the original expression. The later checks types during evaluation, producing *blame errors* in the event of a (runtime) type error. Consider the example given in Section 1:

$$(\lambda x : \mathsf{Dyn} \,.\, 1 + x)\ \mathsf{true}$$

This expression type checks and admits type Int because the lambda abstraction has type Dyn → Int and true has type Bool which is admissible for a dynamically typed argument. The cast insertion procedure produces the expression[1]:

$$(\lambda x : \mathsf{Dyn} \,.\, 1 + (x : \mathsf{Dyn} \Rightarrow \mathsf{Int}))\ (\mathsf{true} : \mathsf{Bool} \Rightarrow \mathsf{Dyn})$$

which evaluates to

$$1 + ((\mathsf{true} : \mathsf{Bool} \Rightarrow \mathsf{Dyn}) : \mathsf{Dyn} \Rightarrow Int)$$

The cast ((true : Bool ⇒ Dyn) : Dyn ⇒ Int) evaluates to a *blame error*. This example shows that, for a value to be treated dynamically, it must be wrapped in a cast from its type to the dynamic type Dyn.

Lets assume we would want to adapt the data type *MaybeInt* (given as example in Section 1) to be able to contain elements of any type:

$$data\ MaybeDyn = Nothing \mid Just\ Dyn$$

Using the type constructors we can represent *MaybeDyn* as Unit + Dyn. Then we can build expressions that interact with, and values of, type *MaybeDyn*:

$$\begin{aligned}
Err_1 &\triangleq Error\ \text{“}Maybe.fromJust : Nothing\text{''}\\
isJust &\triangleq \lambda x.\ \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\ n \Rightarrow \mathsf{false} \mid \mathsf{inr}\ v \Rightarrow \mathsf{true}\\
fromJust &\triangleq \lambda x.\ \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\ n \Rightarrow Err_1 \mid \mathsf{inr}\ v \Rightarrow v\\
just4 &\triangleq \mathsf{inr}\ 4\ \mathsf{as}\ \mathsf{Unit} + \mathsf{Dyn}\\
nothing &\triangleq \mathsf{inl}\ \mathsf{unit}\ \mathsf{as}\ \mathsf{Unit} + \mathsf{Dyn}\\
justTrue &\triangleq \mathsf{inr}\ \mathsf{true}\ \mathsf{as}\ \mathsf{Unit} + \mathsf{Dyn}
\end{aligned}$$

The flaw in this approach is when these values go through cast insertion. According to the cast insertion rules *Inl* and *Inr*, no casts are inserted. The cast insertion produces the same value expressions (those defined in let) as above. Unlike the example above, where true was wrapped around a cast from it's type to dynamic, the elements contained in tags are not wrapped in casts. Therefore, they are not truly dynamically typed. The evaluation of the expression

$$fromJust\ justTrue\ + 1$$

would halt at this point

$$\begin{gathered}
\mathsf{true} : \mathsf{Dyn} \Rightarrow \mathsf{Int} : \mathsf{Int} \Rightarrow \mathsf{Int} : \mathsf{Int} \Rightarrow \mathsf{Int}\\
+\\
1 : \mathsf{Int} \Rightarrow \mathsf{Int}
\end{gathered}$$

instead of returning the expected *blame error*. This is due to the fact that true : Dyn ⇒ Int is not a value, and there is no cast that allows the reduction.

Considering the previous example, what failed was that true had no cast. Therefore a possible solution is to insert that cast directly. Simulating dynamic elements in product or sum types can be accomplished with a beta expansion whose abstraction is annotated with the dynamic type. The abstraction annotated with the dynamic type will then insert the necessary cast.

$$justTrue \triangleq \mathsf{inr}\ ((\lambda x : \mathsf{Dyn}\,.\, x)\ \mathsf{true})\ \mathsf{as}\ \mathsf{Unit} + \mathsf{Dyn}$$

This way, true will have cast true : Bool ⇒ Dyn. Therefore, the evaluation would reduce to

$$\begin{gathered}
\mathsf{true} : \mathsf{Bool} \Rightarrow \mathsf{Dyn} : \mathsf{Dyn} \Rightarrow \mathsf{Int} : \mathsf{Int} \Rightarrow \mathsf{Int} : \mathsf{Int} \Rightarrow \mathsf{Int}\\
+\\
1 : \mathsf{Int} \Rightarrow \mathsf{Int}
\end{gathered}$$

and that would result in a *blame error*. This approach also works with products, since we would only need to add the abstraction to dynamically type the elements of the pair. However, this approach is not very elegant, since the program would be flooded with beta expansions. There is also the question of blame tracking. The "blame" would be assigned to the beta expansions, rather than the algebraic data constructors.

Our goal is to extend the type system and cast insertion rules so that algebraic data type elements can be dynamically typed by adding explicit type annotations. The cast insertion procedure then adds casts to ensure those elements are dynamically typed.

3 Gradual Typing

Gradual typing is a typing discipline that combines static and dynamic typing in a single program, allowing some terms to be statically typed while others are dynamically typed. The gradually typed lambda calculus (GTLC) in Figure 1, first proposed by [11, 12], was developed with the purpose of achieving gradual typing.

The key aspects of this system are the Dyn (dynamic) type, that represents the unknown type, and the ~ (consistency) relation, which checks whether the static parts of the types are equal between two types. For example, in the GTLC, when typing an application between t_1 and t_2, these expressions may have types which are unknown. Therefore, the

[1]The expression is missing casts (those who cast a type to the same type) that are irrelevant to what we are trying to demonstrate.

t ::= terms:
- x variable
- $\lambda x : T.t$ abstraction
- $t\ t$ application

T ::= types:
- Dyn dynamic type
- $T \rightarrow T$ type of functions

$\boxed{\Gamma \vdash_G t : T}$ Typing

$$\frac{x : T \in \Gamma}{\Gamma \vdash_G x : T}\ \textsc{Var} \qquad \frac{\Gamma, x : T_1 \vdash_G t : T_2}{\Gamma \vdash_G \lambda x : T_1 \,.\, t : T_1 \rightarrow T_2}\ \textsc{Abs}$$

$$\frac{\begin{array}{cc}\Gamma \vdash_G t_1 : PM_1 & PM_1 \rhd T_1 \rightarrow T_2 \\ \Gamma \vdash_G t_2 : T_1' & T_1' \sim T_1\end{array}}{\Gamma \vdash_G t_1\ t_2 : T_2}\ \textsc{App}$$

$\boxed{T \sim T}$ Consistency

$$\frac{}{T \sim \mathsf{Dyn}} \qquad \frac{}{\mathsf{Dyn} \sim T} \qquad \frac{T_1 \sim T_3 \qquad T_2 \sim T_4}{T_1 \rightarrow T_2 \sim T_3 \rightarrow T_4}$$

$\boxed{PM \rhd T_1 \rightarrow T_2}$ Arrow Pattern Matching

$$\frac{}{(T_1 \rightarrow T_2) \rhd T_1 \rightarrow T_2} \qquad \frac{}{\mathsf{Dyn} \rhd \mathsf{Dyn} \rightarrow \mathsf{Dyn}}$$

Figure 1. Gradually Typed Lambda Calculus ($\lambda^{?}_{\rightarrow}$)

type system compares, using consistency, the domain of the type of t_1 with the type of t_2. If one of these types is Dyn, it type checks. If not, the types must be equal.

In the example from Section 1:

$$(\lambda x : \mathsf{Dyn} \,.\, 1 + x)\ \mathsf{true}$$

$(\lambda x : \mathsf{Dyn} \,.\, 1 + x)$ has type $\mathsf{Dyn} \rightarrow \mathsf{Int}$ and true has type Bool. Consistency is used to compare the domain type of the expression on the left, Dyn, with the type of the expression on the right, Bool, and since $\mathsf{Dyn} \sim \mathsf{Bool}$ (reads type dynamic is consistent with type boolean), type checking succeeds and the expression is typed with Int.

However, since we used consistency to compare types, we must check if these types are correct during runtime. After type checking, casts are inserted in the expression to be evaluated. A cast $t : T_1 \Rightarrow T_2$ is a runtime check, where the term t has the static type T_1 and the target type T_2. These casts are then evaluated and may produce *blame errors* or be reduced to values that are the result of the evaluation.

4 Dynamic Products and Sums

Algebraic data types are built from product and sum types. As such, in order to allow dynamic data types, product and sum types must also be allowed to contain dynamic elements. Here we extend the system in [4] to allow this.

4.1 Syntax

We must provide a way to explicitly annotate values of sum and product types (pairs and the tags inl and inr). These type annotations will later be used to derive the type of the expression. As inl and inr already have explicit type annotations (necessary to ensure uniqueness of types) in [4], only the remaining value, pair, has to be altered to allow a type annotation. Figure 2 presents the syntax for these terms. The new syntax allows to specify which element is to be dynamically typed by inserting the appropriate type annotation in the value expression.

4.2 Type System

We must also make extensions to the type system described in [4], so that the value is typed with the type in the annotation. These extensions enable the term, inside a pair or a tag, to be typed with the dynamic type. Figure 2 shows the extended type system.

Only the rules *Pair*, *Inl* and *Inr* differ from the (standard) gradual type system [4], because these are the rules that type values. In the case of the rule *Pair*, we now type the pair with the type present in the annotation. However, we must still make sure that the type in the annotation, and therefore the final type of the pair, is consistent with the elements being contained in the pair. Therefore, we need the consistency relations $T_1 \sim T_1'$ and $T_2 \sim T_2'$ to ensure the types are consistent. By having consistency ($\sim$) instead of equality ($=$) relations we allow the dynamic type to be inserted in the pair's type annotation. The same goes for inl and inr.

With these extensions, the values contained in a pair or in a tag can be typed according to the type annotation. Therefore, we can type these values with the dynamic type. For example, consider the following expression:

$$just4 \triangleq \mathsf{inr}\ 4\ \mathsf{as}\ \mathsf{Unit} + \mathsf{Dyn}$$

Without these extensions, this term could not be expressed simply because the dynamic type could not be added to a type annotation. However, with the extensions to the syntax and type system, this term is now valid and has type Unit + Dyn.

To show how these extensions work, a typing derivation for the expression

$$((\lambda x \,.\, \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\ n \Rightarrow Err_1 \mid \mathsf{inr}\ v \Rightarrow v)$$
$$(\mathsf{inr}\ \mathsf{true}\ \mathsf{as}\ \mathsf{Unit} + \mathsf{Dyn}))\ +\ 1$$

is presented in Figure 3. The expression has type Int and evaluation will result in a *blame error*.

$t ::= \ldots$	terms:
(t, t) as T	pair
fst t	first projection
snd t	second projection
case t of inl $x \Rightarrow t$ | inr $x \Rightarrow t$	case
inl t as T	tagging (left)
inr t as T	tagging (right)

$T ::= \ldots$	types:
$T \times T$	product type
$T + T$	sum type

$\boxed{\Gamma \vdash_G t : T}$ Typing

$$\frac{\Gamma \vdash_G t_1 : T_1' \qquad T_1 \sim T_1' \qquad \Gamma \vdash_G t_2 : T_2' \qquad T_2 \sim T_2'}{\Gamma \vdash_G (t_1, t_2) \text{ as } T_1 \times T_2 : T_1 \times T_2} \text{ PAIR}$$

$$\frac{\Gamma \vdash_G t : PM \qquad PM \rhd T_1 \times T_2}{\Gamma \vdash_G \mathsf{fst}\ t : T_1} \text{ FIRST}$$

$$\frac{\Gamma \vdash_G t : PM \qquad PM \rhd T_1 \times T_2}{\Gamma \vdash_G \mathsf{snd}\ t : T_2} \text{ LAST}$$

$$\frac{\begin{array}{c}\Gamma \vdash_G t : PM \qquad PM \rhd T_1' + T_2' \\ \Gamma, x_1 : T_1' \vdash_G t_1 : T_1 \qquad \Gamma, x_2 : T_2' \vdash_G t_2 : T_2 \\ T_1 \sqcup T_2 = T_J\end{array}}{\Gamma \vdash_G \mathsf{case}\ t \text{ of } \mathsf{inl}\ x_1 \Rightarrow t_1 \mid \mathsf{inr}\ x_2 \Rightarrow t_2 \ : T_J} \text{ CASE}$$

$$\frac{\Gamma \vdash_G t : T_1' \qquad T_1 \sim T_1'}{\Gamma \vdash_G \mathsf{inl}\ t \text{ as } T_1 + T_2 : T_1 + T_2} \text{ INL}$$

$$\frac{\Gamma \vdash_G t : T_2' \qquad T_2 \sim T_2'}{\Gamma \vdash_G \mathsf{inr}\ t \text{ as } T_1 + T_2 : T_1 + T_2} \text{ INR}$$

Figure 2. Gradual Type System (Products and Sums)

4.3 Cast Insertion

Finally, the cast insertion [4] rules must be extended to allow the insertion of key casts that will allow the expression to be evaluated in a correct manner. Figure 4 presents the cast insertion rules.

These rules must ensure the necessary casts are inserted in the sub-terms of pairs, inl and inr. The rule *Pair* inserts casts in both sub-terms of pair. For each sub-term, a cast from the type of the sub-term to the type specified in the annotation is inserted. The rules *Inl* and *Inr* are similar, and the only difference is what type is chosen as the target type of the cast, according to the tag. Although the main reason for these casts to be inserted is to enable values to be dynamically typed, these same casts are inserted even if the type annotations are not dynamic. If the type annotation is not consistent with the type of the sub-term, the type system should have caught the type error earlier. If the type annotation is equal to the type of the sub-term, we then end up with a cast from a type to the same type, that will be eliminated during evaluation using the evaluation rule ID-BASE from [5].

5 Dynamic Tuples and Variants

Here we generalize the previous system to deal with tuples and variants. The rules described in this section are very similar to those in Section 4.

Tuples may have an arbitrary number of elements. In the case of tuple projection, we define a projection primitive for any size of a tuple, and the projection requires the size of the tuple. The syntax π_i^n t (sometimes also written as *proj i n t*) means that we are projecting from a term t, and that the projection is expecting a tuple of size n, and we want to project the i^{th} element of that tuple. Similarly, the term case may now have an arbitrary number of alternatives.

Then we allow the description of structures of arbitrary size. $(t_i\ ^{i \in 1..n})$ for a tuple with n terms and $(T_i\ ^{i \in 1..n})$ for it's type. $\langle l_i : T_i\ ^{i \in 1..n} \rangle$ stands for the variant type with n alternatives, where t_i are terms, l_i are labels and T_i are types.

5.1 Syntax

An explicit type annotation is needed in a tuple (as was needed in a pair) and tag, so that the types of the elements can be specified. In the case of tag, there is already a type annotation, so only the tuple term changes. Figure 5 presents the syntax.

5.2 Type System

As with products and sums, extensions are necessary to the type system of tuples and variants. These are similar to those in Figure 2, however, we must take into account that these generalizations do not have a fixed structure, but have an arbitrary number of terms. The type system is presented in Figure 5.

Some subtleties that are present in this type system are worth mentioning. In the case of tuple projection, we need to indicate the size of the tuple we expect to project from, as stated previously. This is a necessary approach, because when we pattern match ($\rhd$) the type PM (in the Projection rule), we must ensure it has the expected size. If PM is the dynamic type, then we must be able to construct the tuple type of size m. The information about the size of the tuple is necessary to guide the gradual type system. If we were to define a different projection term for each different size that the tuple can have (fst3, fst4, fst5, ..., possibly up to a upper limit), then this information would be implicitly present, because each term would be supposed to project from a tuple of a certain size.

$$1\quad \dfrac{\dfrac{\dfrac{}{x : \mathsf{Unit} + \mathsf{Dyn} \vdash x : \mathsf{Unit} + \mathsf{Dyn}}\,\textsc{T-Var} \quad \dfrac{}{x : \mathsf{Unit} + \mathsf{Dyn}, n : \mathsf{Unit} \vdash Err_1 : \mathsf{Dyn}}\,\textsc{T-Error} \quad \dfrac{}{x : \mathsf{Unit} + \mathsf{Dyn}, v : \mathsf{Dyn} \vdash v : \mathsf{Dyn}}\,\textsc{T-Var} \quad \mathsf{Unit} + \mathsf{Dyn} \rhd \mathsf{Unit} + \mathsf{Dyn} \quad \mathsf{Dyn} \sqcup \mathsf{Dyn} = \mathsf{Dyn}}{x : \mathsf{Unit} + \mathsf{Dyn} \vdash \mathsf{case}\ x\ \mathsf{of\ inl}\ n \Rightarrow Err_1 \mid \mathsf{inr}\ v \Rightarrow v : \mathsf{Dyn}}\,\textsc{T-Case}}{\vdash \lambda x\ .\ \mathsf{case}\ x\ \mathsf{of\ inl}\ n \Rightarrow Err_1 \mid \mathsf{inr}\ v \Rightarrow v : (\mathsf{Unit} + \mathsf{Dyn}) \longrightarrow \mathsf{Dyn}}\,\textsc{T-Abs}$$

$$2\quad \dfrac{\dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\,\textsc{T-True} \qquad \mathsf{Dyn} \sim \mathsf{Bool}}{\vdash \mathsf{inr\ true\ as\ Unit} + \mathsf{Dyn} : \mathsf{Unit} + \mathsf{Dyn}}\,\textsc{T-Inr}$$

$$\dfrac{\mathsf{Dyn} \rhd \mathsf{Int} \qquad \dfrac{\begin{array}{l} 1\ \dfrac{}{\vdash \lambda x\ .\ \mathsf{case}\ x\ \mathsf{of\ inl}\ n \Rightarrow Err_1 \mid \mathsf{inr}\ v \Rightarrow v : (\mathsf{Unit} + \mathsf{Dyn}) \longrightarrow \mathsf{Dyn}}\,\textsc{T-Abs} \\ 2\ \dfrac{}{\vdash \mathsf{inr\ true\ as\ Unit} + \mathsf{Dyn} : \mathsf{Unit} + \mathsf{Dyn}}\,\textsc{T-Inr} \\ (\mathsf{Unit} + \mathsf{Dyn}) \longrightarrow \mathsf{Dyn} \rhd (\mathsf{Unit} + \mathsf{Dyn}) \longrightarrow \mathsf{Dyn} \qquad \mathsf{Unit} + \mathsf{Dyn} \sim \mathsf{Unit} + \mathsf{Dyn} \end{array}}{\vdash (\lambda x\ .\ \mathsf{case}\ x\ \mathsf{of\ inl}\ n \Rightarrow Err_1 \mid \mathsf{inr}\ v \Rightarrow v)\ (\mathsf{inr\ true\ as\ Unit} + \mathsf{Dyn}) : \mathsf{Dyn}}\,\textsc{T-App} \qquad \dfrac{}{\vdash 1 : \mathsf{Int}}\,\textsc{T-Int} \qquad \mathsf{Int} \rhd \mathsf{Int}}{\vdash (\lambda x\ .\ \mathsf{case}\ x\ \mathsf{of\ inl}\ n \Rightarrow Err_1 \mid \mathsf{inr}\ v \Rightarrow v)\ (\mathsf{inr\ true\ as\ Unit} + \mathsf{Dyn}) + 1 : \mathsf{Int}}\,\textsc{T-Add}$$

Figure 3. Typing Derivation

$\boxed{\Gamma \vdash_{CC} t \rightsquigarrow t' : T}$ Cast Insertion

$$\dfrac{\Gamma \vdash_{CC} t_1 \rightsquigarrow t_1' : T_1' \qquad \Gamma \vdash_{CC} t_2 \rightsquigarrow t_2' : T_2'}{\Gamma \vdash_{CC} (t_1, t_2)\ \mathsf{as}\ T_1 \times T_2 \rightsquigarrow (t_1' : T_1' \Rightarrow T_1, t_2' : T_2' \Rightarrow T_2)\ \mathsf{as}\ T_1 \times T_2 : T_1 \times T_2}\,\textsc{Pair}$$

$$\dfrac{\Gamma \vdash_{CC} t \rightsquigarrow t' : PM \qquad PM \rhd T_1 \times T_2}{\Gamma \vdash_{CC} \mathsf{fst}\ t \rightsquigarrow \mathsf{fst}\ (t' : PM \Rightarrow T_1 \times T_2) : T_1}\,\textsc{First} \qquad \dfrac{\Gamma \vdash_{CC} t \rightsquigarrow t' : PM \qquad PM \rhd T_1 \times T_2}{\Gamma \vdash_{CC} \mathsf{snd}\ t \rightsquigarrow \mathsf{snd}\ (t' : PM \Rightarrow T_1 \times T_2) : T_2}\,\textsc{Last}$$

$$\dfrac{\Gamma \vdash_{CC} t \rightsquigarrow t' : PM \qquad PM \rhd T_1' + T_2' \qquad \Gamma, x : T_1' \vdash_{CC} t_1 \rightsquigarrow t_1' : T_1 \qquad \Gamma, x : T_2' \vdash_{CC} t_2 \rightsquigarrow t_2' : T_2 \qquad T_1 \sqcup T_2 = T_J}{\begin{array}{c}\Gamma \vdash_{CC} \mathsf{case}\ t\ \mathsf{of\ inl}\ x_1 \Rightarrow t_1 \mid \mathsf{inr}\ x_2 \Rightarrow t_2 \rightsquigarrow \\ \mathsf{case}\ (t' : PM \Rightarrow T_1' + T_2')\ \mathsf{of\ inl}\ x_1 \Rightarrow (t_1' : T_1 \Rightarrow T_J) \mid \mathsf{inr}\ x_2 \Rightarrow (t_2' : T_2 \Rightarrow T_J)\ \ : T_J\end{array}}\,\textsc{Case}$$

$$\dfrac{\Gamma \vdash_{CC} t \rightsquigarrow t' : T_1'}{\Gamma \vdash_{CC} \mathsf{inl}\ t\ \mathsf{as}\ T_1 + T_2 \rightsquigarrow \mathsf{inl}\ (t' : T_1' \Rightarrow T_1)\ \mathsf{as}\ T_1 + T_2 : T_1 + T_2}\,\textsc{Inl}$$

$$\dfrac{\Gamma \vdash_{CC} t \rightsquigarrow t' : T_2'}{\Gamma \vdash_{CC} \mathsf{inr}\ t\ \mathsf{as}\ T_1 + T_2 \rightsquigarrow \mathsf{inr}\ (t' : T_2' \Rightarrow T_2)\ \mathsf{as}\ T_1 + T_2 : T_1 + T_2}\,\textsc{Inr}$$

Figure 4. Cast Insertion (Products and Sums)

This is also the case in the rule *Case*, but with slight differences. While in the *Projection* rule, we must ensure *PM* has the expected size, in the case of variants we must ensure that *PM* has the same labels as the labels present in the alternatives.

5.3 Cast Insertion

The cast insertion rules for tuples and variants are similar to those for products and sums. In the case for tuples, instead of of adding casts to the two elements of pair, we now must add casts for each element of the tuple.

$$\begin{array}{lll}
t ::= \ldots & & \text{terms:} \\
& (t_i \,^{i \in 1..n}) \textit{ as } T & \text{tuple} \\
& \pi_i^n \; t & \text{projection} \\
& \text{case } t \text{ of } \langle l_i = x_i \rangle \Rightarrow t_i \,^{i \in 1..n} & \text{case} \\
& \langle l = t \rangle \text{ as } T & \text{tagging}
\end{array}$$

$$\begin{array}{lll}
T ::= \ldots & & \text{types:} \\
& (T_i \,^{i \in 1..n}) & \text{tuple type} \\
& \langle l_i : T_i \,^{i \in 1..n} \rangle & \text{variant type}
\end{array}$$

$\boxed{\Gamma \vdash_G t : T}$ Typing

$$\frac{\text{for each } i \qquad \Gamma \vdash_G t_i : T_i' \qquad T_i \sim T_i'}{\Gamma \vdash_G (t_i \,^{i \in 1..n}) \text{ as } (T_i \,^{i \in 1..n}) : (T_i \,^{i \in 1..n})} \; \textsc{Tuple}$$

$$\frac{\Gamma \vdash_G t : PM \qquad PM \rhd (T_j \,^{j \in 1..n})}{\Gamma \vdash_G \pi_i^n \; t : T_i} \; \textsc{Projection}$$

$$\frac{\begin{array}{c} \Gamma \vdash_G t : PM \qquad PM \rhd \langle l_i : T_i' \,^{i \in 1..n} \rangle \\ \text{for each } i \qquad \Gamma, x_i : T_i' \vdash_G t_i : T_i \\ T_1 \sqcup \ldots \sqcup T_n = T_J \end{array}}{\Gamma \vdash_G \text{case } t \text{ of } \langle l_i = x_i \rangle \Rightarrow t_i \,^{i \in 1..n} : T_J} \; \textsc{Case}$$

$$\frac{\Gamma \vdash_G t_j : T_j' \qquad T_j \sim T_j'}{\Gamma \vdash_G \langle l_j = t_j \rangle \text{ as } \langle l_i : T_i \,^{i \in 1..n} \rangle : \langle l_i : T_i \,^{i \in 1..n} \rangle} \; \textsc{Tag}$$

Figure 5. Gradual Type System (Tuples and Variants)

The rule *Tag* is similar to the rules *Inl* and *Inr*. The difference is that instead of choosing the left or right side, the rule must obtain the type, that will be the cast destination, from the type annotation by comparing the label in the tag expression with the labels in the type annotation. The rules are presented in Figure 6.

6 Recursive Types

The type constructors referred so far are the building blocks for the creation of new types. However, there is another building block to the creation of recursive data types. The fixed point operator (μ) [10] is the type constructor that allows recursive type definitions. Although the treatment of this type constructor does not require any extension in order to allow dynamic data types, in accordance with the scope of this paper, we thought it would be helpful to describe the type system and cast insertion rules. Although we chose to represent the iso-recursive approach to recursive types, we could have chosen the equi-recursive approach [10]. However, the gradual type system for iso-recursive approach is already defined [4].

6.1 Syntax

Although no changes to the syntax of recursive types is required, Figure 7 presents the syntax.

6.2 Type System

The standard terms for the iso-recursive type are fold and unfold. These terms are used to guide the type system in type checking expressions with recursive types. The type system rules are similar to those of the static type system (such as in [10]), however they have some differences. In the *Unfold* rule, a pattern matching relation is needed, because we expect the term t to be typed with an recursive type. The term t may be typed with the dynamic type, and if that's the case, we must convert the dynamic type to the recursive type. Furthermore, we require the consistency relation between the type of the sub-term t and the expected type for the sub-term, which in *Fold* is a one-step unfolding of the type in the annotation in fold and in *Unfold* is the recursive type present in the annotation of unfold. The type rules are shown in Figure 7.

6.3 Cast Insertion

The cast insertion rules are a relatively straightforward transformation from the type system rules. In the *Fold* rule, no cast is inserted while in the *Unfold* rule only the cast pertaining the pattern matching relation is inserted. The cast insertion rules for the recursive type are shown in Figure 8.

7 Type Inference

Type inference for gradual typing was previously defined in [6], where type inference is extended to a solver which deals with consistency constraints. In this section we extend the type inference algorithm to deal with dynamic algebraic data types (products, sums, tuples and variants). These rules are an extension to the rules in [6], and as such follow the same conventions. The constrain generation judgment $\Gamma \vdash t : T \mid_\chi C$ means that given a context Γ and a term t, t can be given type T if the constraints C can be satisfied, and χ denotes the variables used to express the constraints. The symbols $\dot{\sim}$ and $\doteq$ are used to represent the consistency and the equality constraints, respectively. Constraint generation rules for products and sums are presented in Figure 9, for tuples and variants in Figure 11 and for the recursive type in Figure 13. The constraint unification judgment $C \upsilon S$ means that the constraints C are unified producing a set of substitutions S. The constraint unification rules are relatively simple, since these rules share a fixed structure with the rules in [6]. For each type constructor, there are 3 different rules that must be derived: two rules for unifying equality and consistency constraints between two types with the same type constructor and a rule for unifying a consistency constraint between a type variable and a type. Constraint unification rules for products and sums are present in Figure 10, for tuples and variants in Figure 12 and for the recursive

$\boxed{\Gamma \vdash_{CC} t \rightsquigarrow t' : T}$ Cast Insertion

$$\frac{\text{for each } i \qquad \Gamma \vdash_{CC} t_i \rightsquigarrow t_i' : T_i'}{\Gamma \vdash_{CC} (t_i\ ^{i\in 1..n}) \text{ as } (T_i\ ^{i\in 1..n}) \rightsquigarrow (t_i : T_i' \Rightarrow T_i\ ^{i\in 1..n}) \text{ as } (T_i\ ^{i\in 1..n}) : (T_i\ ^{i\in 1..n})} \text{ TUPLE}$$

$$\frac{\Gamma \vdash_{CC} t \rightsquigarrow t' : PM \qquad PM \rhd (T_j\ ^{j\in 1..n})}{\Gamma \vdash_{CC} \pi_i^n\ t \rightsquigarrow \pi_i^n\ (t' : PM \Rightarrow (T_j\ ^{j\in 1..n})) : T_i} \text{ PROJECTION}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{CC} t \rightsquigarrow t' : PM \qquad PM \rhd \langle l_i : T_i'\ ^{i\in 1..n}\rangle \\ \text{for each } i \qquad \Gamma, x_i : T_i' \vdash_G t_i \rightsquigarrow t_i' : T_i \qquad T_1 \sqcup \ldots \sqcup T_n = T_J\end{array}}{\Gamma \vdash_{CC} \text{case } t \text{ of } \langle l_i = x_i\rangle \Rightarrow t_i\ ^{i\in 1..n} \rightsquigarrow \text{case } (t' : PM \Rightarrow \langle l_i : T_i'\ ^{i\in 1..n}\rangle) \text{ of } \langle l_i = x_i\rangle \Rightarrow (t_i : T_i \Rightarrow T_J)\ ^{i\in 1..n} : T_J} \text{ CASE}$$

$$\frac{\Gamma \vdash_{CC} t_j \rightsquigarrow t_j' : T_j'}{\Gamma \vdash_{CC} \langle l_j = t_j\rangle \text{ as } \langle l_i : T_i\ ^{i\in 1..n}\rangle \rightsquigarrow \langle l_j = (t_j : T_j' \Rightarrow T_j)\rangle \text{ as } \langle l_i : T_i\ ^{i\in 1..n}\rangle : \langle l_i : T_i\ ^{i\in 1..n}\rangle} \text{ TAG}$$

Figure 6. Cast Insertion (Tuples and Variants)

t ::= ...	terms:
fold $[T]$ t	folding
unfold $[T]$ t	unfolding

T ::= ...	types:
$\mu X\ .\ T$	recursive type

$\boxed{\Gamma \vdash_G t : T}$ Typing

$$\frac{U = \mu X\ .\ T \qquad \Gamma \vdash_G t : T' \qquad [X \mapsto U]T\ \sim\ T'}{\Gamma \vdash_G \text{fold } [U]\ t : U} \text{ FOLD}$$

$$\frac{\begin{array}{c} U = \mu X\ .\ T \\ \Gamma \vdash_G t : PM \qquad PM \rhd \mu X'\ .\ T' \qquad U\ \sim\ \mu X'\ .\ T'\end{array}}{\Gamma \vdash_G \text{unfold } [U]\ t : [X \mapsto U]T} \text{ UNFOLD}$$

Figure 7. Gradual Type System (Recursive types)

type in Figure 14. There are some aspects that are not so straightforward and thus require additional explanation.

The meet relation between types ($\sqcap$) is defined in [6], and although the rules for product, tuple, sum and variant types are not presented in that paper, they can be derived in a straightforward manner. The meet relation is similar to the join relation ($\sqcup$), that is defined in [4] because both return the type that is the least upper bound w.r.t. the precision relation.

Pattern matching ($\rhd$) is paramount in order to convert a dynamic type to it's expected type. Types whose structures are of arbitrary size, complicate pattern matching. For example, consider the following pattern matching judgment:

$$PM \rhd T_1 + T_2 \mid_\chi C$$

This means that we will pattern match the type PM to some type whose structure is that of a sum (+) type. The pattern matching relation in a sense takes the input type and a type constructor and returns a type or fails. However, when dealing with types constructors of arbitrary size, things complicate. Consider the pattern matching judgment for tuple type:

$$PM \rhd (T_i\ ^{i\in 1..n}) \mid_\chi C$$

Pattern matching requires the size of the tuple. In the type inference rule *Projection*, we can observe that the size of the tuple constructor is provided by the projection term (in π_i^n t, n denotes the size of the tuple). The same is true for pattern matching variants:

$$PM \rhd \langle l_i : T_i\ ^{i\in 1..n}\rangle \mid_\chi C$$

Instead of providing the size, the pattern matching relation requires the labels that form the variant type constructor. Variants with different labels are considered different types. These labels are provided by the labels in the alternatives in the case term. In the case of pattern matching the recursive type, the pattern matching only requires the recursive variable (X).

$$PM \rhd \mu X\ .\ T \mid_\chi C$$

Pattern matching for the type constructors defined in this paper had to be derived, since there were no definitions available. The design was inspired by the Constraint Codomain Judgment and Constraint Domain Consistency Judgment in [6]. Deriving pattern matching judgments for a specific type constructor is somewhat straightforward. Like in the

$\boxed{\Gamma \vdash_{CC} t \rightsquigarrow t' : T}$ Cast Insertion

$$\frac{U = \mu X\ .\ T \qquad \Gamma \vdash_{CC} t \rightsquigarrow t' : T' \qquad [X \mapsto U]\ T \mathrel{\dot\sim} T'}{\Gamma \vdash_{CC} \mathsf{fold}\ [U]\ t \rightsquigarrow \mathsf{fold}\ [U]\ t' : U}\ \textsc{Fold}$$

$$\frac{U = \mu X\ .\ T \qquad \Gamma \vdash_{CC} t \rightsquigarrow t' : PM \qquad PM \rhd \mu X'\ .\ T' \qquad U \mathrel{\dot\sim} \mu X'\ .\ T'}{\Gamma \vdash_{CC} \mathsf{unfold}\ [U]\ t \rightsquigarrow \mathsf{unfold}\ [U]\ (t' : PM \Rightarrow \mu X'\ .\ T') : [X \mapsto U]T}\ \textsc{Unfold}$$

Figure 8. Cast Insertion (Recursive types)

$\boxed{\Gamma \vdash t : T \mid_\chi C}$ Constraint Generation

$$\frac{\Gamma \vdash t_1 : T_1' \mid_{\chi_1} C_1 \qquad \Gamma \vdash t_2 : T_2' \mid_{\chi_2} C_2}{\Gamma \vdash (t_1, t_2)\ \mathsf{as}\ T_1 \times T_2 : T_1 \times T_2 \mid_{\chi_1 \cup \chi_2} C_1 \cup C_2 \cup \{T_1 \mathrel{\dot\sim} T_1', T_2 \mathrel{\dot\sim} T_2'\}}\ \textsc{Pair}$$

$$\frac{\Gamma \vdash t : PM \mid_\chi C \qquad PM \rhd T_1 \times T_2 \mid_{\chi_1} C_1}{\Gamma \vdash \mathsf{fst}\ t : T_1 \mid_{\chi \cup \chi_1} C \cup C_1}\ \textsc{First} \qquad \frac{\Gamma \vdash t : PM \mid_\chi C \qquad PM \rhd T_1 \times T_2 \mid_{\chi_1} C_1}{\Gamma \vdash \mathsf{snd}\ t : T_2 \mid_{\chi \cup \chi_1} C \cup C_1}\ \textsc{Second}$$

$$\frac{\begin{array}{c}\Gamma \vdash\ t : PM \mid_\chi C \qquad PM \rhd T_1' + T_2' \mid_{\chi'} C' \\ \Gamma, x_1 : T_1' \vdash t_1 : T_1 \mid_{\chi_1} C_1 \qquad \Gamma, x_2 : T_2' \vdash t_2 : T_2 \mid_{\chi_2} C_2 \qquad T_1 \sqcap T_2 = T_J \mid_{\chi_3} C_3\end{array}}{\Gamma \vdash \mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}\ x_1 \Rightarrow t_1 \mid \mathsf{inr}\ x_2 \Rightarrow t_2\ : T_J \mid_{\chi \cup \chi' \cup \chi_1 \cup \chi_2 \cup \chi_3} C \cup C' \cup C_1 \cup C_2 \cup C_3}\ \textsc{Case}$$

$$\frac{\Gamma \vdash t : T_1' \mid_\chi C}{\Gamma \vdash \mathsf{inl}\ t\ \mathsf{as}\ T_1 + T_2 : T_1 + T_2 \mid_\chi C \cup \{T_1 \mathrel{\dot\sim} T_1'\}}\ \textsc{Inl} \qquad \frac{\Gamma \vdash t : T_2' \mid_\chi C}{\Gamma \vdash \mathsf{inr}\ t\ \mathsf{as}\ T_1 + T_2 : T_1 + T_2 \mid_\chi C \cup \{T_2 \mathrel{\dot\sim} T_2'\}}\ \textsc{Inr}$$

$\boxed{PM \rhd T_1 \times T_2 \mid_\chi C}$ Product Pattern Matching Judgment

$$\frac{}{X \rhd X_1 \times X_2 \mid_{\{X_1, X_2\}} \{X \doteq X_1 \times X_2\}} \qquad \frac{}{T_1 \times T_2 \rhd T_1 \times T_2 \mid_\emptyset \emptyset} \qquad \frac{}{\mathsf{Dyn} \rhd \mathsf{Dyn} \times\ \mathsf{Dyn} \mid_\emptyset \emptyset}$$

$\boxed{PM \rhd T_1 + T_2 \mid_\chi C}$ Sum Pattern Matching Judgment

$$\frac{}{X \rhd X_1 + X_2 \mid_{\{X_1, X_2\}} \{X \doteq X_1 + X_2\}} \qquad \frac{}{T_1 + T_2 \rhd T_1 + T_2 \mid_\emptyset \emptyset} \qquad \frac{}{\mathsf{Dyn} \rhd \mathsf{Dyn} + \mathsf{Dyn} \mid_\emptyset \emptyset}$$

Figure 9. Constraint Generation (Products and Sums)

Constraint Codomain Judgment and Constraint Domain Consistency Judgment, we must take into account that the judgment expects a certain type constructor and can receive as input three different forms of types: a type variable, a type whose constructor is the expected and the dynamic type. Anything other than that leads the judgment to fail. When we try to pattern match a type variable, we must create a type whose structure is the expected type constructor and it's elements are (fresh) type variables. We then return that type, and a equality constraint between the newly created type and the original type variable. If we try to pattern match a type whose type constructor is the expected, then we can return that type and no constraints are needed. When we try to pattern match the dynamic type, we must return a type whose structure is that of the type constructor and it's elements are the dynamic type. The reason pattern matching judgments expect a type variable is because we are considering a type inference style approach. In a type system, a pattern matching relation only expects either the correct type constructor or the dynamic type.

$\boxed{C\ \upsilon\ S}$ Constraint Unification

$$\frac{C \cup \{T_{11} \mathrel{\dot\sim} T_{21}, T_{12} \mathrel{\dot\sim} T_{22}\}\ \upsilon\ S}{C \cup \{T_{11} \times T_{12} \mathrel{\dot\sim} T_{21} \times T_{22}\}\ \upsilon\ S} \qquad \frac{C \cup \{T_{11} \mathrel{\dot\sim} T_{21}, T_{12} \mathrel{\dot\sim} T_{22}\}\ \upsilon\ S}{C \cup \{T_{11} + T_{12} \mathrel{\dot\sim} T_{21} + T_{22}\}\ \upsilon\ S} \qquad \frac{\begin{array}{c}\{X_1, X_2\} fresh \qquad X \notin Vars(T_1 \times T_2) \\ C \cup \{X \doteq X_1 \times X_2, X_1 \mathrel{\dot\sim} T_1, X_2 \mathrel{\dot\sim} T_2\}\ \upsilon\ S\end{array}}{C \cup \{X \mathrel{\dot\sim} T_1 \times T_2\}\ \upsilon\ S}$$

$$\frac{\begin{array}{c}\{X_1, X_2\} fresh \qquad X \notin Vars(T_1 + T_2) \\ C \cup \{X \doteq X_1 + X_2, X_1 \mathrel{\dot\sim} T_1, X_2 \mathrel{\dot\sim} T_2\}\ \upsilon\ S\end{array}}{C \cup \{X \mathrel{\dot\sim} T_1 + T_2\}\ \upsilon\ S} \qquad \frac{C \cup \{T_{11} \doteq T_{21}, T_{12} \doteq T_{22}\}\ \upsilon\ S}{C \cup \{T_{11} \times T_{12} \doteq T_{21} \times T_{22}\}\ \upsilon\ S} \qquad \frac{C \cup \{T_{11} \doteq T_{21}, T_{12} \doteq T_{22}\}\ \upsilon\ S}{C \cup \{T_{11} + T_{12} \doteq T_{21} + T_{22}\}\ \upsilon\ S}$$

Figure 10. Constraint Unification (Products and Sums)

$\boxed{\Gamma \vdash t : T \mid_{\chi} C}$ Constraint Generation

$$\frac{\text{for each } i \qquad \Gamma \vdash t_i : T_i' \mid_{\chi_i} C_i}{\Gamma \vdash (t_i\ ^{i\in 1..n}) \text{ as } (T_i\ ^{i\in 1..n}) : (T_i\ ^{i\in 1..n}) \mid_{\chi_1 \cup \ldots \cup \chi_n} C_1 \cup \ldots \cup C_n \cup \{T_1 \mathrel{\dot\sim} T_1', \ldots, T_n \mathrel{\dot\sim} T_n'\}}\ \textsc{Tuple}$$

$$\frac{\Gamma \vdash t : PM \mid_{\chi} C \qquad PM \rhd (T_j\ ^{j\in 1..n}) \mid ^{\chi'} C'}{\Gamma \vdash \pi_i^n\ t : T_i \mid_{\chi \cup \chi'} C \cup C'}\ \textsc{Projection}$$

$$\frac{\begin{array}{c}\Gamma \vdash t : PM \mid_{\chi} C \qquad PM \rhd \langle l_i : T_i'\ ^{i\in 1..n}\rangle \mid_{\chi'} C' \\ \text{for each } i \qquad \Gamma, x_i : T_i' \vdash t_i : T_i \mid_{\chi_i} C_i \qquad T_1 \sqcap \ldots \sqcap T_n = T_J \mid_{\chi''} C''\end{array}}{\Gamma \vdash \text{case } t \text{ of } \langle l_i = x_i\rangle \Rightarrow t_i\ ^{i\in 1..n} : T_J \mid_{\chi \cup \chi' \cup \chi_1 \cup \ldots \cup \chi_n \ldots \cup \chi''} C \cup C' \cup C_1 \cup \ldots \cup C_n \cup C''}\ \textsc{Case}$$

$$\frac{\Gamma \vdash t_j : T_j' \mid_{\chi} C}{\Gamma \vdash \langle l_j = t_j\rangle \text{ as } \langle l_i : T_i\ ^{i\in 1..n}\rangle : \langle l_i : T_i\ ^{i\in 1..n}\rangle \mid_{\chi} C \cup \{T_j \mathrel{\dot\sim} T_j'\}}\ \textsc{Tag}$$

$\boxed{PM \rhd (T_i\ ^{i\in 1..n}) \mid_{\chi} C}$ Tuple Pattern Matching Judgment

$$\frac{}{X \rhd (X_i\ ^{i\in 1..n}) \mid_{\{X_1, \ldots, X_n\}} \{X \doteq (X_i\ ^{i\in 1..n})\}} \qquad \frac{}{(T_i\ ^{i\in 1..n}) \rhd (T_i\ ^{i\in 1..n}) \mid_{\emptyset} \emptyset} \qquad \frac{}{\mathsf{Dyn} \rhd (\mathsf{Dyn}\ ^{i\in 1..n}) \mid_{\emptyset} \emptyset}$$

$\boxed{PM \rhd \langle l_i : T_i\ ^{i\in 1..n}\rangle \mid_{\chi} C}$ Variant Pattern Matching Judgment

$$\frac{}{X \rhd \langle l_i : X_i\ ^{i\in 1..n}\rangle \mid_{\{X_1, \ldots, X_n\}} \{X \doteq \langle l_i : X_i\ ^{i\in 1..n}\rangle\}} \qquad \frac{}{\langle l_i : T_i\ ^{i\in 1..n}\rangle \rhd \langle l_i : T_i\ ^{i\in 1..n}\rangle \mid_{\emptyset} \emptyset} \qquad \frac{}{\mathsf{Dyn} \rhd \langle l_i :\ \mathsf{Dyn}\ ^{i\in 1..n}\rangle \mid_{\emptyset} \emptyset}$$

Figure 11. Constraint Generation (Tuples and Variants)

8 Implementation

The type system [4], or type inference [6], rules and cast insertion rules [4] with the extensions provided in this paper, along with the evaluation rules [5] can be used to implement a language with dynamic data type. The architecture of our implementation is presented in Figure 15. The syntax is a simplified Haskell syntax. A parser is needed to read the code and translate it to an intermediate code (extended version of the lambda calculus). Then that intermediate code goes through type inference. This phase would either fail, most likely producing a type error, or would return the type of the expression and a fully annotated version of the intermediate code, where all terms would be annotated with their type. Then the fully annotated code is passed to the cast insertion procedure, that returns the cast calculus, which is

$\boxed{C\ \upsilon\ S}$ Constraint Unification

$$\frac{C \cup \{T_{11} \mathrel{\dot\sim} T_{21}, \ldots, T_{1n} \mathrel{\dot\sim} T_{2n}\}\ \upsilon\ S}{C \cup \{(T_{1i}{}^{i\in1..n}) \mathrel{\dot\sim} (T_{2i}{}^{i\in1..n})\}\ \upsilon\ S} \qquad \frac{C \cup \{T_{11} \mathrel{\dot\sim} T_{21}, \ldots, T_{1n} \mathrel{\dot\sim} T_{2n}\}\ \upsilon\ S}{C \cup \{\langle l_{1i} : T_{1i}{}^{i\in1..n}\rangle \mathrel{\dot\sim} \langle l_{2i} : T_{2i}{}^{i\in1..n}\rangle\}\ \upsilon\ S}$$

$$\frac{\begin{array}{c}\{X_1, \ldots, X_n\} fresh \qquad X \notin Vars((T_i{}^{i\in1..n})) \\ C \cup \{X \doteq (X_i{}^{i\in1..n}), X_1 \mathrel{\dot\sim} T_1, \ldots, X_n \mathrel{\dot\sim} T_n\}\ \upsilon\ S\end{array}}{C \cup \{X \mathrel{\dot\sim} (T_i{}^{i\in1..n})\}\ \upsilon\ S} \qquad \frac{\begin{array}{c}\{X_1, \ldots, X_n\} fresh \qquad X \notin Vars(\langle l_i : T_i{}^{i\in1..n}\rangle) \\ C \cup \{X \doteq \langle l_i : X_i{}^{i\in1..n}\rangle, X_1 \mathrel{\dot\sim} T_1, \ldots, X_n \mathrel{\dot\sim} T_n\}\ \upsilon\ S\end{array}}{C \cup \{X \mathrel{\dot\sim} \langle l_i : T_i{}^{i\in1..n}\rangle\}\ \upsilon\ S}$$

$$\frac{C \cup \{T_{11} \doteq T_{21}, \ldots, T_{1n} \doteq T_{2n}\}\ \upsilon\ S}{C \cup \{(T_{1i}{}^{i\in1..n}) \doteq (T_{2i}{}^{i\in1..n})\}\ \upsilon\ S} \qquad \frac{C \cup \{T_{11} \doteq T_{21}, \ldots, T_{1n} \doteq T_{2n}\}\ \upsilon\ S}{C \cup \{\langle l_{1i} : T_{1i}{}^{i\in1..n}\rangle \doteq \langle l_{2i} : T_{2i}{}^{i\in1..n}\rangle\}\ \upsilon\ S}$$

Figure 12. Constraint Unification (Tuples and Variants)

$\boxed{\Gamma \vdash t : T \mid_\chi C}$ Constraint Generation

$$\frac{U = \mu X\,.\,T \qquad \Gamma \vdash t : T' \mid_\chi C}{\Gamma \vdash \mathsf{fold}\ [U]\ t : U \mid_\chi C \cup \{[X \mapsto U]T \mathrel{\dot\sim} T'\}}\ \textsc{Fold} \qquad \frac{U = \mu X\,.\,T \qquad \Gamma \vdash t : PM \mid_\chi C \qquad PM \rhd \mu X'\,.\,T' \mid_{\chi'} C'}{\Gamma \vdash \mathsf{unfold}\ [U]\ t :\ [X \mapsto U]T \mid_{\chi\cup\chi'} C \cup C' \cup \{U \mathrel{\dot\sim} \mu X'\,.\,T'\}}\ \textsc{Unfold}$$

$\boxed{PM \rhd \mu X\,.\,T \mid_\chi C}$ Recursive Type Pattern Matching Judgment

$$\frac{}{X' \rhd \mu X\,.\,X_1 \mid_{\{X_1\}} \{X' \doteq \mu X\,.\,X_1\}} \qquad \frac{}{\mu X\,.\,T \rhd \mu X\,.\,T \mid_\emptyset \emptyset} \qquad \frac{}{\mathsf{Dyn} \rhd \mu X\,.\,\mathsf{Dyn} \mid_\emptyset \emptyset}$$

Figure 13. Constraint Generation (Recursive types)

$\boxed{C\ \upsilon\ S}$ Constraint Unification

$$\frac{C \cup \{T_1 \mathrel{\dot\sim} T_2\}\ \upsilon\ S}{C \cup \{\mu X\,.\,T_1 \mathrel{\dot\sim} \mu X\,.\,T_2\}\ \upsilon\ S} \qquad \frac{\begin{array}{c}\{X_1\} fresh \qquad X \notin Vars(\mu X'\,.\,T) \\ C \cup \{X \doteq \mu X'\,.\,X_1, X_1 \mathrel{\dot\sim} T\}\ \upsilon\ S\end{array}}{C \cup \{X \mathrel{\dot\sim} \mu X'\,.\,T\}\ \upsilon\ S} \qquad \frac{C \cup \{T_1 \doteq T_2\}\ \upsilon\ S}{C \cup \{\mu X\,.\,T_1 \doteq \mu X\,.\,T_2\}\ \upsilon\ S}$$

Figure 14. Constraint Unification (Recursive types)

the intermediate code with casts. Then this code is evaluated to produce the resulting value, which can be either a value such as Int or Bool or a *blame error*, indicating that there was a runtime type error.

Programmers can define data types similar to Haskell, using type constructors to build expressions instead of the syntax presented in this paper (Figures 2 and 5). This is just syntactic sugar, since the internal representation of the code (intermediate code) follows the syntax defined in Figures 2 and 5.

9 Working Examples

The extensions discussed so far aim to allow the building of dynamic data types. In order to demonstrate how the type system and cast insertion rules will work, we present a few examples.

Example 1: Dynamic Maybe

Let's first take a look at how a maybe containing a dynamic element can be built. We start with the data type definition:

$$data\ MaybeDyn = Nothing \mid Just\ Dyn$$

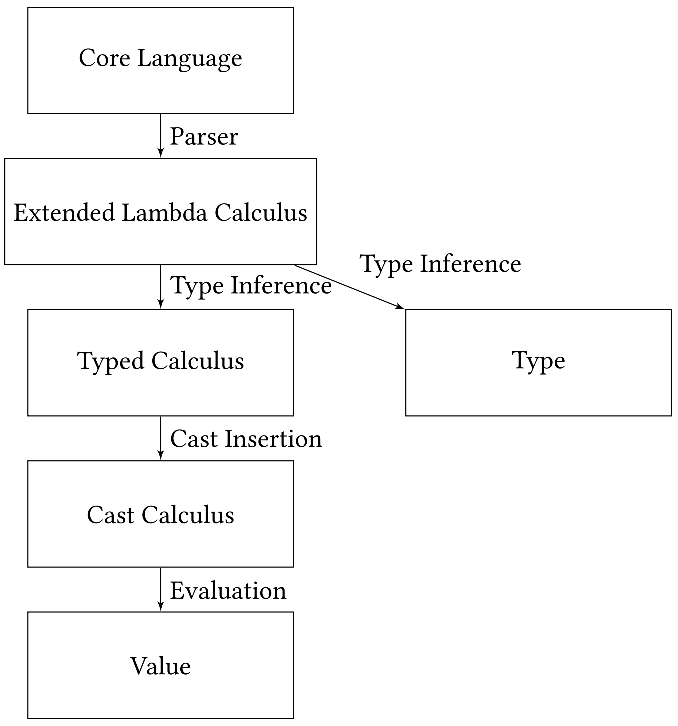

Figure 15. Architecture

This means the type *MaybeDyn* can either contain nothing or a element whose type is unknown. The representation of *MaybeDyn* with the type constructors approached in this paper is:

$$\langle Nothing : Unit, Just : Dyn\rangle$$

Now that we have the correct type, we can build expressions that interact with, and values of, type *MaybeDyn*.

```
let isJust = \maybe . case maybe of
    <Nothing=x> => false
  | <Just=v> => true in
let fromJust = \maybe . case maybe of
   <Nothing=x> => Error "Maybe.fromJust: Nothing"
  | <Just=v> => v in
let just4 =
  <Just=4> as <Nothing:Unit, Just:Dyn> in
let nothing =
  <Nothing=unit> as <Nothing:Unit, Just:Dyn> in
let justTrue =
  <Just=True> as <Nothing:Unit, Just:Dyn> in
fromJust justTrue + 1
```

The values that the variables *just*4, *nothing* and *justTrue* bind have type *MaybeDyn* due to the type annotations. As explained previously, the type annotation in values is what decides the type of the value. Therefore, when we interpret this expression the execution does not halt, like it does with the analogous example (with sum types) in Section 2, but results in a *blame error*. By comparing the execution of this example

```
((((True : Bool => Dyn) : Dyn => Int): Int => Int)
: Int => Int) + (1 : Int => Int)
```

with the execution of the example in Section 2

```
(((True : Dyn => Int) : Int => Int)
: Int => Int) + (1 : Int => Int)
```

we can see that, with the proposed cast insertion rules, the necessary cast is present in the expression. Therefore, the cast

```
((True : Bool => Dyn) : Dyn => Int)
```

reduces to a *blame error*, which is then pushed to the top level and returned as the result of the evaluation.

The expressions *isJust* and *fromJust* may also be applied to terms of type *MaybeInt* ($\langle Nothing : Unit, Just : Int\rangle$). The resulting expression will then be statically typed and evaluated thereafter. In order to choose between static and dynamic typing, we must now insert the appropriate type in the tag's type annotation.

Example 2: Dynamic List

Let's now consider building lists whose elements are of unknown type. The data type

$$data\ ListDyn = Nil \mid Cons\ Dyn\ ListDyn$$

can be used to define lists as a recursive type [10] using a equi-recursive approach. However, we take the iso-recursive approach and define lists as:

$$\mu L.\langle Nil : Unit, Cons : (Dyn, L)\rangle$$

Now, like the previous example, we need expressions to build and interact with terms of type *ListDyn*. We will be using the fold and unfold terms (to enable recursive types). Since fold and unfold, like values, require type annotations, for the sake of readability, we will use the names *ListDyn* and *ListDyn'* as an abbreviation of the *ListDyn* type and it's one-step unfolding, respectively:

$$\begin{aligned} ListDyn &\triangleq \mu L.\langle Nil : Unit, Cons : (Dyn, L)\rangle \\ ListDyn' &\triangleq \langle Nil : Unit, Cons : (Dyn, ListDyn)\rangle \end{aligned}$$

The standard terms for building generic lists as well as the lists themselves can be defined in the following manner:

```
let nil =
  fold [ListDyn] <Nil=unit> as ListDyn' in
let cons = \v . \l . fold [ListDyn]
  <Cons=(v, l) as (Dyn, ListDyn)> as ListDyn' in
```

```
let isnil = \l . case (unfold [ListDyn] l) of
    <Nil=n> => True
  | <Cons=c> => False in
let hd = \l . case (unfold [ListDyn] l) of
    <Nil=n> => Error "***Exception: empty list"
  | <Cons=c> => proj 1 2 c in
let tl = \l . case (unfold [ListDyn] l) of
    <Nil=n> => Error "***Exception: empty list"
  | <Cons=c> => proj 2 2 c in
let list123 = cons 1 (cons 2 (cons 3 nil)) in
let listtrue1 = cons true (cons 1 nil) in
hd listtrue1 + 1
```

However, some definitions seem odd. For example, cons takes two arguments (an element to be inserted in the head of a list, and said list) and returns the representation of the list: the *Cons* tag containing a pair (2-tuple) with the element in the first position and the list in the second position. Both the tag and the tuple must have type annotations. The tag is annotated with the one step unfolding of *ListDyn* because it has the correct type structure that a tag accepts in it's annotation (that of a variant constructor). The same can be seen in the type annotation in the tuple, which can only have a tuple type in it's annotation. Although these annotations seem too excessive, they are necessary to guide the type system and ensure the expressions are typed with the correct (and intended) types. Therefore, when we interpret the expression a *blame error* is produced. The *blame error* is produced when the code evaluates to

```
(((True : Bool => Dyn) : Dyn => Int)
: Int => Int) + (1 : Int => Int)
```

In this expressions we can see the same cast that produced a blame in the previous example (*MaybeDyn*). This cast essentially means we are trying to force a term of type Bool to be of type Int. This happens because we are trying to add the head of the list $listtrue1$, the value true which is of type Bool, with the integer 1.

10 Conclusion

Type constructors, particularly products and sums and their generalizations, tuples and variants, have been used to build algebraic data types by providing a mechanism to simulate combination and alternation. These type constructors form the building blocks of algebraic data types, who inherits the typing discipline of type constructors. In this paper we define an extended type system and cast insertion rules for the standard terms of these type constructors that enable elements of compound types to be declared as static or dynamic.

Acknowledgments

This work is partially funded by FCT within project Elven POCI-01-0145-FEDER-016844, Project 9471 - Reforçar a Investigação, o Desenvolvimento Tecnológico e a Inovação (Project 9471-RIDTI) and by Fundo Comunitário Europeu FEDER.

References

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. 1991. Dynamic typing in a statically typed language. *ACM transactions on programming languages and systems (TOPLAS)* 13, 2 (1991), 237–268.
[2] Amal Ahmed, Robert Bruce Findler, Jeremy G Siek, and Philip Wadler. 2011. Blame for all. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 201–214.
[3] Arthur I Baars and S Doaitse Swierstra. 2002. Typing dynamic typing. In *ACM SIGPLAN Notices*, Vol. 37. ACM, 157–166.
[4] Matteo Cimini and Jeremy G Siek. 2016. The gradualizer: a methodology and algorithm for generating gradual type systems. *ACM SIGPLAN Notices* 51, 1 (2016), 443–455.
[5] Matteo Cimini and Jeremy G Siek. 2017. Automatically generating the dynamic semantics of gradually typed languages. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 789–803.
[6] Ronald Garcia and Matteo Cimini. 2015. Principal type schemes for gradual programs. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 303–315.
[7] Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N Freund, and Cormac Flanagan. 2006. Sage: Practical hybrid checking for expressive types and specifications. In *Proceedings of the Workshop on Scheme and Functional Programming*. 93–104.
[8] Fritz Henglein. 1994. Dynamic typing: Syntax and proof theory. *Science of Computer Programming* 22, 3 (1994), 197–230.
[9] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. 2004. Dynamic typing with dependent types. In *Exploring new frontiers of theoretical informatics*. Springer, 437–450.
[10] Benjamin C Pierce. 2002. *Types and programming languages*. MIT press.
[11] Jeremy G Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Vol. 6. 81–92.
[12] Jeremy G Siek and Manish Vachharajani. 2008. Gradual typing with unification-based inference. In *Proceedings of the 2008 symposium on Dynamic languages*. ACM, 7.
[13] Satish Thatte. 1988. Type inference with partial types. In *International Colloquium on Automata, Languages, and Programming*. Springer, 615–629.
[14] Satish Thatte. 1989. Quasi-static typing. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 367–381.
[15] Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage migration: from scripts to programs. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 964–974.