A Branding Strategy for Business Types

Avraham Shinnar and Jérôme Siméon(✉)

IBM Research, 1101 Kitchawan Rd, Yorktown Heights, NY 10598, USA
{shinnar,simeon}@us.ibm.com

Abstract. In the course of building a compiler from business rules to a database run-time, we encounter the need for a type system that includes a class hierarchy and subtyping in the presence of complex record operations. Since our starting point is based on structural typing and targets a data-centric language, we develop an approach inspired by Wadler's work on XML types (Siméon and Wadler 2003). Our proposed type system has strong similarities with branded or tagged objects, combining nominal and structural typing, and is designed to support a rich set of operations on records commonly found in database languages. We show soundness of the type system and illustrate its use on two of the intermediate languages involved in our compiler for business rules: a calculus for pattern matching with aggregation (CAMP) that captures rules semantics, and the nested relational algebra (NRA) used for optimization and code generation. We show type soundness for both languages. The approach and correctness proofs are fully mechanized using the Coq proof-assistant.

1 Introduction

In order to support the growing amount of data routinely processed by applications, many programming languages are being developed (Cooper et al. 2007; Green et al. 2012) or extended (Cheney et al. 2013; Meijer et al. 2006) with query-like capabilities. Existing database techniques can then be leveraged for scalability through compilation to a database query language such as SQL (Grust et al. 2009) or a cloud library such as Spark (Armbrust et al. 2015). When extending object-oriented languages with such query capabilities, one of the key challenges is type system integration: database languages (e.g., SQL or the relational algebra) most often rely on structural types and feature rich operations on records (e.g., record concatenation), while object-oriented languages most often rely on nominal types and feature rich forms of subtyping. Inspired by Wadler's work on XML types (Siméon and Wadler 2003) as well as recent work on brands (Jones et al. 2015) or tagged objects (Lee et al. 2015), we propose a new type system that provides an elegant and natural way to combine nominal and structural typing in the presence of complex record operations, and illustrate its use in the context of a compiler for business rules.

Business Rules. Our work takes place as part of META (Arnold et al. 2016), a project in which rules are used to program applications that combine event processing and analytics. The use of rules is common in business intelligence

S. Lindley et al. (Eds.): Wadler Festschrift, LNCS 9600, pp. 367–387, 2016.
DOI: 10.1007/978-3-319-30936-1_20

```
public class Entity
{
  Int id;
}

public class Client
  extends Entity
{
  String name;
}

public class Marketer
  extends Entity
{
  String name;
  Int client;
}

public final class DirectMarketer
  extends Marketer
{
  List<String> targets;
}

public class ProductRep
  extends Marketer
{
  List<String> products;
}

public final class C2Ps
{
  Int client;
  List<String> reps;
}
```

Fig. 1. Business object model.

applications as they can encode complex data-centric policies in a flexible manner. Languages for business rules go back to production rules (Forgy 1981), and modern specimens include Drools (Bali 2009) and JRules (Boyer and Mili 2011), the latter being our focus. Rules are written against collection(s) of objects described in a Business Object Model (BOM), which is essentially an object-oriented class hierarchy. Figure 1 shows an example BOM for a marketing application. A general `Entity` class is used to describe objects with an `id` attribute which serves as a primary key. Other classes derive from `Entity` and include `Client`s with an attribute `name` of type string, and `Marketer`s with attributes `name` of type string and `client` of type `Client`. Two additional classes corresponding to categories of marketers are also defined: `DirectMarketer` with an attribute `targets` and `ProductRep` with an attribute `products`, both of which of type list of strings. The `extends` clause indicates the derivation hierarchy for each class, and when omitted means the class derives from the built-in `Object` class. Finally, class `C2Ps` is used to materialize a mapping from clients to their product reps, computed through the rule shown in Fig. 2.

That rule consists of a condition (`when`, Lines 2–7) and an action (`then`, Line 8). In effect, expressions in such rules perform pattern matching over an implicit value, in the context of an environment (variable bindings). The initial implicit value is not set and the initial environment is a single variable containing the collection of objects against which the rule is being matched. The `rule` construct sets the implicit value to be that input collection and the rule condition is matched against any of those objects and "fires" when a match is found. In our example, the condition binds variable `C` to a `Client`, and then aggregates all `ProductRep` objects `P` for whom `C`'s `id` is the same as `P`'s `client`. The aggregation expression is essentially an embedded query, with a syntax similar to that

```
1  rule FindProductReps {
2    when {
3      C: Client();
4      Ps: aggregate {
5            P: ProductRep(client == C.id);
6          } do { List<String> {P.name}; }
7    }
8    then { insert new C2Ps(C, Ps); }
9  }
```

Fig. 2. Rule computing a reverse mapping from clients to product reps.

of comprehensions (Trinder and Wadler 1988). The `List<String> { P.name }` expression builds a new collection containing the names of all matching elements resulting from the aggregate, and that collection is bound to variable `Ps`. The action part of the rule in Line 8 creates a new object of class `C2Ps` which materializes the mapping from `C` to `Ps`. Environment manipulation (e.g., binding variable `P`) is reflected in the compiler as record operations (e.g., concatenating a record with field `P` to the record for the current environment).

A Branding Strategy. A calculus and compilation scheme for business rules targeting the nested relational algebra (NRA) for optimization and scalability was proposed in (Shinnar et al. 2015). Proper typing is essential in order to exploit optimization techniques for bulk data processing developed by the database community (Beeri and Kornatzky 1993; Cluet and Moerkotte 1993; Fegaras and Maier 2000; May et al. 2006). However, most existing type systems for database languages, including the one we used in (Shinnar et al. 2015), rely on a purely structural approach which does not account for the nominal aspects of object-oriented data models and types. In addition, those typically assume homogeneous collections of objects and do not account for the subtyping relation implicit in the class hierarchy. In the context of Object-Oriented Databases, OQL relies on a purely nominal type system (Alagic 1999), while underlying algebras or calculi for optimization rely on a purely structural type system (Fegaras and Maier 2000; Cluet and Moerkotte 1993). A notable exception is Wadler's work on a type system for XML and XQuery presented in (Siméon and Wadler 2003), which integrates nominal and structural typing by annotating values and types with type names related through a derivation hierarchy. However, the focus on XML types and lack of support for records means it cannot be directly applied to classic object models or to operators from the nested relational algebra.

Inspired by this work, we propose a novel type system that can handle collections of objects in a class hierarchy that we believe is well suited for integration with database languages or language-integrated query systems (Cooper et al. 2007; Grust et al. 2009; Cheney et al. 2013). The approach relies on a notion of branded types and objects, where brands can be used to annotate values and types. Relationship between brands is captured by a derivation relation which encodes the nominal part of the semantics. From a structural typing stand point,

we support open and closed records along with the corresponding subtyping relation, in the presence of a rich set of record operations that include projection and concatenation. From a nominal typing stand point, we support inheritance and (up/down) casting. Most of the treatment is language-agnostic, in that we define a set of core operators over brands independently from the host language. We then illustrate how those core operations can be integrated within two languages used in our business rules compiler: a calculus which captures rules' pattern matching semantics, and the nested relational algebra.

Paper Outline. The rest of the paper is organized as follows. In Sect. 2, we provide an overview for the proposed branding approach through examples. In Sect. 3, we describe the data model and type system. In Sect. 4, we introduce the subtyping relation and a notion of well-formed models which captures object-oriented schemas. In Sect. 5, we define key operators on records, bags and branded values along with their typing rules. Finally, in Sect. 6, we apply the proposed type system to a compiler from rules to NRA. Section 7 reviews related work and Sect. 8 concludes the paper with a brief report on our implementation and a discussion of future work.

2 Overview

Even though our work is motivated by business rules, most of the approach can be explained in terms of core operations over an object model. We first explain how classes and objects are encoded with brands, present key operations on those, then illustrate their use as part of our rules compiler.

Branded Model. In essence, branded types are types annotated with a name (a brand). Figure 3 shows the branded model corresponding to the Business Object Model from Fig. 1. Each class is described by a brand (the class name) and a type (the type of values of that class). For instance, the `Client` class corresponds to *Brand* Client which contains values of type record with attributes *id* of type `INT` and *name* of type `STRING`. We use [ ] to denote records, and { } to denote collection types[1]. The '[..]' (resp. '[|]') notation indicates open (resp. closed) records. An open record can be extended with new attributes while a closed one cannot. In our example, all branded types use open record, except for `C2Ps` and `DirectMarketer` which are declared as `final` in the source object model.

The brand derivation hierarchy is listed at the bottom of Fig. 3 and captures the nominal component of the type hierarchy. For instance, brand ProductRep derives from brand Marketer which itself derives from brand Entity. We assume the presence of an Object brand whose type is that of all possible records and which encodes the built-in Object class. We use Any to denote the top of the brand hierarchy. Compared to Fig. 1, we eliminate the `extends` relation, i.e., we fully expand the type for each class, and expose the inheritance relation in the

[1] We rely on a bag semantics but support for other collection types can be added easily, e.g., along the lines of (Fegaras and Maier 2000).

Brand Object [..]

Brand Entity [*id* : INT ..]

Brand Marketer
[*id* : INT;
name : STRING;
client : INT ..]

Brand ProductRep
[*id* : INT;
name : STRING;
client : INT;
products : {STRING} ..]

Brand Client
[*id* : INT;
name : STRING ..]

Brand DirectMarketer
[*id* : INT;
name : STRING;
client : INT;
targets : {STRING} |]

Brand C2Ps
[*client* : INT;
reps : {STRING} |]

Object *derives from* Any
Entity *derives from* Object
Client *derives from* Entity
Marketer *derives from* Entity
DirectMarketer *derives from* Marketer
ProductRep *derives from* Marketer
C2Ps *derives from* Object

Fig. 3. Branded object model.

derivation hierarchy[2]. This separation follows directly from (Siméon and Wadler 2003) and is central to our approach in that it allows to clearly separate the nominal and structural aspects of the type system. The branded model along with its derivation hierarchy is well-formed on the condition that the types of two brands related through (nominal) derivation be properly related through (structural) subtyping. For instance, adding the relation C2Ps *derives from* Entity would be invalid since the type for brand C2Ps does not have and *id* attribute.

Note that the model allows any type to be branded. For instance, INT is the type of integer values but we could give integers an object-like treatment by defining a brand for them. We could then define a special kind of integer for ids through derivation, as illustrated by the following declarations.

Brand Int INT
Brand Id INT
Id *derives from* Int

Through subtyping, every operation taking a branded value of type *Brand* Int also takes a value of type *Brand* Id, but not the converse.

Branded Values. In essence, branded values are values annotated with a name (a brand). Figure 4 shows examples of values in our data model. In each case we provide a name for the value (variable), a type and the value itself. All the

[2] The formal model actually relies on the reflexive and transitive closure of the derivation hierarchy which must be a partial order (i.e., without cycles).

```
M1Id : INT = 1

C1 : [ id : INT; name : STRING |] = [ id : 1 ; name : "Disney" ]
C2 : [ id : INT ..] = [ id : 1 ; name : "Disney" ]

M1 : Brand DirectMarketer =
  brand DirectMarketer ([ id : 101 ; name : "John"; client : 1;
                          targets : { "iOS", "Gawker" } ])

M2 : Brand Marketer =
  brand DirectMarketer ([ id : 101 ; name : "John"; client : 1;
                          targets : { "iOS", "Gawker" } ])

Clients : {Brand Client} =
  { brand Client ([ id : 1 ; name : "Disney" ]),
    brand Client ([ id : 2 ; name : "Netflix" ]),
    brand Client ([ id : 3 ; name : "HBO" ]) }

Marketers : {Brand Marketer} =
  { brand DirectMarketer ([ id : 101 ; name : "John"; client : 1;
                            targets : { "iOS"; "Gawker" } ]),
    brand ProductRep ([ id : 102 ; name : "Julie"; client : 1;
                        products : { "Star Wars" } ]),
    brand DirectMarketer ([ id : 103 ; name : "Jack"; client : 3;
                            targets : { "Instagram" } ])}

WM : {Brand Entity} = Clients ∪ Marketers
```

Fig. 4. Branded values.

examples are well-typed, i.e., the given value has the given type. The first three are simple values without brands: `M1Id` is an integer value, `C1` and `C2` are records. Note that `C1` (resp. `C2`) is declared as having a closed (resp. open) record type. `M1` is a value with brand DirectMarketer whose content is a record with id 101 and name *"John"*, working for client with id 1 and targeting *"iOS"* and *"Gawker"*. Value `M2` is the same value but declared with type *Brand* Marketer, which is also valid since DirectMarketer derives from Marketer. The remaining of Fig. 4 shows examples of collections. `Clients` is an homogeneous collection of values with brand Client, and `Marketers` a collection of object in the sub-brand hierarchy for brand Marketer. Finally the last collection `WM` is the union of `Clients` and `Marketers` and can be typed as a collection of values with brand Entity.

Brand Operations. We provide three core operations on brands: branding, unbranding, and casting. The first two are constructors/destructors for branded values. For instance, the following operations create a value with brands Id, Client and Entity. (We use the notation: `Var` : $Type = Expr \Downarrow Value$ to indicate that variable Var has type $Type$ and value $Value$ which is the result of $Expr$).

$$\begin{aligned}
\texttt{M1Id'} &: \mathit{Brand}\ \mathsf{Id} = \mathsf{brand\ Id}\ (\texttt{M1Id}) \Downarrow \mathsf{brand\ Id}\ (1)\\
\texttt{C1'} &: \mathit{Brand}\ \mathsf{Client} = \mathsf{brand\ Client}\ (\texttt{C1}) \Downarrow\\
&\qquad \mathsf{brand\ Client}\ ([\mathit{id} : 1; \mathit{name} : \text{“}\mathit{Disney}\text{”}])\\
\texttt{C1''} &: \mathit{Brand}\ \mathsf{Entity} = \mathsf{brand\ Client}\ (\texttt{C1}) \Downarrow\\
&\qquad \mathsf{brand\ Client}\ ([\mathit{id} : 1; \mathit{name} : \text{“}\mathit{Disney}\text{”}])\\
\texttt{C1'''} &: \mathit{Brand}\ \mathsf{Entity} = \mathsf{brand\ Entity}\ (\texttt{C1}) \Downarrow\\
&\qquad \mathsf{brand\ Entity}\ ([\mathit{id} : 1; \mathit{name} : \text{“}\mathit{Disney}\text{”}])
\end{aligned}$$

Note that branding checks that the value being branded has the type declared for the corresponding brand. In the previous examples, `C1` is declared as a closed record with attributes *id* and *name* and is indeed a subtype of the corresponding open record for both brands Client and Entity. However, the following two operations are not well typed, since the type for `C1` (resp. `M1Id`) isn't a subtype of the type for brand Marketer (resp. Entity).

$$\begin{aligned}
&\mathsf{brand\ Marketer}\ (\texttt{C1})\\
&\mathsf{brand\ Entity}\ (\texttt{M1Id})
\end{aligned}$$

Unbranding, denoted !, always succeeds on a branded value and returns its content. Unbranding uses the static type of the brand (which can be changed with a cast) to determine the type of the returned contents. Here are a few examples, also illustrating record operations such as field access (denoted $.A$ where A is an attribute) and record projection (denoted $\pi_{\overline{A_i}}$ where $\overline{A_i}$ is a set of attributes). Note that the typing rule for projection yields a closed record type.

$$\begin{aligned}
\texttt{Id1} &: \texttt{INT} = !\texttt{M1Id'} \Downarrow 1\\
\texttt{C1id} &: \texttt{INT} = !\texttt{C1'}.\mathit{id} \Downarrow 1\\
\texttt{C1proj} &: [\mathit{id} : \texttt{INT}\ |] = \pi_{\mathit{id}}(!\texttt{C1'}) \Downarrow [\mathit{id} : 1]
\end{aligned}$$

Unbranding is typed using the statically known brand type, which means that $!\texttt{C1''}.\mathit{name}$ is not well typed. In other words, the fields that are present in `C1''` but not declared for the brand Entity are hidden, unless casting is first applied to the value. This once again corresponds to the standard behavior in an OO context. At run-time, casting checks if a given brand is a supertype (or the same) as a brand's dynamic type and returns the original branded value or fails[3]. The typing rule for casting asserts that the returned brand has the provided brand type. This allows safe downcasting on brands, allowing code to enrich the static type of a brand at the cost of a runtime check. Here are some examples of successful casts applied in the context of the nested relational algebra, in which success (resp. failure) is represented as a singleton (resp. empty) bag.

$$\begin{aligned}
\texttt{UpM1Id} &: \{\mathit{Brand}\ \mathsf{Int}\} = (\mathsf{Int})\ \texttt{M1Id'} \Downarrow \{\mathsf{brand\ Id}\ (1)\}\\
\texttt{UpC1} &: \{\mathit{Brand}\ \mathsf{Entity}\} = (\mathsf{Entity})\ \texttt{C1'} \Downarrow\\
&\qquad \{\mathsf{brand\ Client}\ ([\mathit{id} : 1; \mathit{name} : \text{“}\mathit{Disney}\text{”}])\}\\
\texttt{DownC1} &: \{\mathit{Brand}\ \mathsf{Client}\} = (\mathsf{Client})\ \texttt{C1''} \Downarrow\\
&\qquad \{\mathsf{brand\ Client}\ ([\mathit{id} : 1; \mathit{name} : \text{“}\mathit{Disney}\text{”}])\}
\end{aligned}$$

[3] What failure means may vary as we will show when applying casting in the context the two intermediate languages (CAMP and NRA) involved in our rules compiler.

The first two examples are upcasts: from *Brand* Id to *Brand* Int, and from *Brand* Client to *Brand* Entity. The third example is a downcast from *Brand* Entity to *Brand* Client. We now give some examples of cast failures, all of which return the empty bag in the context of the nested relational algebra.

$$\texttt{Fail1} : \{\mathit{Brand}\ \mathsf{Entity}\} = (\mathsf{Entity})\ \texttt{M1Id'} \Downarrow \{\}$$
$$\texttt{Fail2} : \{\mathit{Brand}\ \mathsf{Client}\} = (\mathsf{Client})\ \texttt{C1'''} \Downarrow \{\}$$
$$\texttt{Fail3} : \{\mathit{Brand}\ \mathsf{ProductRep}\} = (\mathsf{ProductRep})\ \texttt{M1} \Downarrow \{\}$$

Note that the specific failure semantics for NRA is unusual from a classic OO perspective, but it enables easy integration with other relational operators. For instance, the following combination of flatten, map (χ) and casting can be used to select all objects in variable `WM` that are product reps. In that example, **In** stands for the current element in the input collection processed by the map. The output of the map is a bag of either singleton or empty bags which can then be flattened. The final type is a bag of *Brand* ProductRep as expected.

$$\begin{aligned}\texttt{Reps} : \{\mathit{Brand}\ \mathsf{ProductRep}\} = \mathit{flatten}(\chi_{\langle(\mathsf{ProductRep})\ \mathbf{In}\rangle}(\texttt{WM})) \Downarrow \\ \{\mathsf{ProductRep}([\mathit{id} : 102; \mathit{name} : \text{“}\mathit{Julie}\text{”}; \mathit{client} : 1; \\ \mathit{products} : \{\text{“}\mathit{Star\ Wars}\text{”}\}])\}\end{aligned}$$

Rules Compiler. With that notion of branded values and types in hand, we extend the rules calculus and compiler proposed in (Shinnar et al. 2015) to provide proper support for the class hierarchy. Going back to the example rule from Fig. 2, the binding within the aggregate expression on Line 5 can be translated to the nested-relational algebra as follows:

$$\bowtie^{d}_{\langle[P:\mathbf{In}.it]\rangle}\Big(\sigma_{\langle!(\mathbf{In}.C).id=!(\mathbf{In}.it).client\rangle}\Big(\rho_{it/\{br\}}\Big(\bowtie^{d}_{\langle[br:(ProductRep)\ (\mathbf{In}.it)]\rangle}(\mathbf{In})\Big)\Big)\Big) \tag{1}$$

From an input bag of records (rightmost **In**) each containing a working memory elements (in attribute *it*) and a binding for variable `C` (in attribute C), select those whose brand is ProductRep. Then, select those whose *client* attribute is the same as the client id ($!(\mathbf{In}.C).id$). Then, add the attribute P containing the current value in attribute *it* to each remaining record. Note that the notion of current value (**In**) depends on context and its scope changes within every sub-operator between angle brackets $\langle..\rangle$. In the first operand of the select (σ) operator, it is bound to each record from the bag returned by its second operand. The $\bowtie^{d}$ operator (traditionally called dependent join in the database literature (Cluet and Moerkotte 1993)) performs record concatenation, adding attribute P to every record in its input. The expression makes use of *casting* and *unbranding*. The first expression within the right-most $\bowtie^{d}$ uses a cast to filter the input objects with the right brand ($(ProductRep)$ ($\mathbf{In}.it$)) and adds a temporary field *br* in the input record that contains values of type {*Brand* ProductRep}. The next NRA operator (ρ) is used to unnest the content of attribute *br*, then re-bind it to attribute *it*, which now has type *Brand* ProductRep. The subsequent

selection operator can now access the *client* attribute safely by unbranding the input value $(!(\textbf{IN}.it).client)$.

Remarks. Before we move to a formal treatment of the proposed model, a few aspects are worth pointing out. Firstly, support for both open and closed records is a key requirement: open records allow to model classes and inheritance, while closed records are essential when using relational operations such as Cartesian product or (dependent) join both of which perform record concatenation. We handle both by combining: (i) subtyping between open and closed records, (ii) record operations, notably projection, that allow to coerce an open record into a closed record. Secondly, the combination of operations on brands and records enables a rich feature set that includes classic features from OO language, many of which can be expressed as combinations of our core operators. The presentation here focuses on data-centric languages and notably, we do not attempt to model features such as methods or object identity.

3 Data and Types

In this section, we first define the data model and core type system with the notion of brands. Note that both rely only on a hierarchy of brand names.

3.1 Data Model

Values in our data model, the set $\mathcal{D}$, are atoms, records, bags or branded values. We assume a sufficiently large set of atoms $a, b, ...$ including integers in $\mathcal{Z}$, strings in $\mathcal{S}$, the Boolean values true and false, and a null value written nil. A bag is a multiset of values in $\mathcal{D}$; we write $\emptyset$ for the empty bag and $\{d_1, ..., d_n\}$ for the bag containing the values $d_1, ..., d_n$.

A record is a mapping from a finite set of attributes to values in $\mathcal{D}$, where attribute names are drawn from a sufficiently large set $A, B, ...$ We write $[\,]$ for the empty record and $[\overline{A_i : d_i}]$ for the record mapping A_i to d_i. We define the concatenation $x * y$ of two records as a mapping from $\mathrm{dom}(x) \cup \mathrm{dom}(y)$ to values, such that $[x * y](A) = x(A)$ if $A \in \mathrm{dom}(x)$ and $[x * y](A) = y(A)$ if $A \in \mathrm{dom}(y) \setminus \mathrm{dom}(x)$. Records x and y are $\mathsf{compatible}$ if $\forall A \in \mathrm{dom}(x) \cap \mathrm{dom}(y), x(A) = y(A)$. We define the sum $[x + y]$ of compatible records as the union $x \cup y$, and leave it undefined for non-compatible records.

A branded value is a pair of a brand name and a value, where brand names are drawn from a sufficiently large set $\mathsf{A}, \mathsf{B}, ...$ We write $\mathsf{brand\ A}\ (d)$ for the value d branded with A. We assume a derivation hierarchy, which is a partial order relation δ between brands, with a most general brand denoted Any. We write $\delta(\mathsf{A}, \mathsf{A}')$ to denote that A *derives from* $\mathsf{A'}$ (through reflexive and transitive closure). For every brand A, we assume that $\delta(\mathsf{A}, \mathsf{Any})$. The fact that δ is a partial order is essential for the soundness of the type system. We allow a brand name to derive from several brand names which accounts for multiple inheritance.

Finally, we assume structural equality between values, denoted $d_1 \doteq d_2$, and defined inductively as follows. Two atomic values are equal if their underlying

built-in equality (between two booleans, two strings, two integers) holds. Two bags are equal if they have the same values (compared through equality) modulo permutation. Two record are equal if they have the same set of attributes and the values associated to corresponding attribute are equal. Branded values are equal if they have the same brand and their contents are equal.

3.2 Types

Our types include primitive types `NIL`, `INT`, `BOOL`, and `STRING`. The type of a (homogeneous) bag with elements of type τ is written $\{\tau\}$. We also assume the existence of a top (resp. bottom) type denoted $\top$ (resp. $\bot$).

We define two kinds of record types: closed and open. $[\overline{A_i : \tau_i}\ |]$ is the type of a *closed* record with attributes A_i of type τ_i. $[\overline{A_i : \tau_i}\ ..]$ is the type of an *open* record with attributes A_i of type τ_i. When there is no ambiguity, we sometimes write $[\overline{A_i : \tau_i}]$ for a record type that can be either open or closed.

As with data records, we define a notion of compatibility for record types. Two open (resp. closed) record types are considered compatible if all their overlapping attributes (resp. known overlapping attributes) have the same type. Note that two records can have compatible types but incompatible data. Record concatenation, $[\overline{A_i : \tau_{A_i}} * \overline{B_j : \tau_{B_j}}|]$ concatenates two record types, favoring the types of A's attributes in case of conflict and is only defined for closed records. Record merge, $[\overline{A_i : \tau_{A_i}} + \overline{B_j : \tau_{B_j}}]$, also concatenates the record types and is defined for both open and closed records, but only if they are compatible.

Finally, *Brand* A is the type of values with brand A.

4 Subtyping and Models

In this section, we define structural subtyping and the notion of well-formed model used to capture class hierarchies. We then describe what it means for a value to be well-typed and show this notion is consistent with subtyping.

4.1 Subtyping

We can now define the subtyping relation, which is still purely structural, depending only on the given derivation hierarchy δ over brand names. Figure 5 defines $\tau \preccurlyeq_\delta \tau'$, the subtyping relation between two types given δ. The first two rules simply place $\top$ (resp. $\bot$) at the top (resp. bottom) of the type hierarchy. A type is a subtype of itself (SRefl). A bag of type τ is a subtype of a bag of type τ' iff, τ is a subtype of τ' (SBag). A type with brand A is a subtype of a type with brand A' iff A *derives from* A' in δ (SBrand).

A record is a subtype of a closed record iff, it is also closed, has the same domain (same attributes), and each of its attributes' type is a subtype of the corresponding attribute in the super type (SClosed). The subtype of an open record can be open or closed, and it must have at least the same attributes, with the proper subtyping relationship between the attribute in common (SOpen). In

$$\frac{}{\tau \preccurlyeq_\delta \top}\ (\text{STop}) \qquad \frac{}{\bot \preccurlyeq_\delta \tau}\ (\text{SBottom}) \qquad \frac{}{\tau \preccurlyeq_\delta \tau}\ (\text{SRefl})$$

$$\frac{\tau \preccurlyeq_\delta \tau'}{\{\tau\} \preccurlyeq_\delta \{\tau'\}}\ (\text{SBag}) \qquad \frac{\delta(\mathsf{A}, \mathsf{A'})}{\mathit{Brand}\ \mathsf{A} \preccurlyeq_\delta \mathit{Brand}\ \mathsf{A'}}\ (\text{SBrand})$$

$$\frac{\tau_1 \preccurlyeq_\delta \tau_1' \quad \ldots \quad \tau_n \preccurlyeq_\delta \tau_n'}{[A_1 : \tau_1, \ldots, A_n : \tau_n\ |] \preccurlyeq_\delta [A_1 : \tau_1', \ldots, A_n : \tau_n'\ |]}\ (\text{SClosed})$$

$$\frac{\tau_1 \preccurlyeq_\delta \tau_1' \quad \ldots \quad \tau_n \preccurlyeq_\delta \tau_n'}{[A_1 : \tau_1, \ldots, A_n : \tau_n, \overline{B_i : \tau_i''}] \preccurlyeq_\delta [A_1 : \tau_1', \ldots, A_n : \tau_n'\ ..]}\ (\text{SOpen})$$

Fig. 5. Subtyping. $\boxed{\tau \preccurlyeq_\delta \tau'}$

other words, open records support both width and depth subtyping, while closed records support only depth subtyping.

Figure 5 does not include rules for transitivity and antisymmetry which are both consequences of the definition. Subtyping is indeed a partial order over types, as stated by the following theorem[4].

Theorem 1 (Subtype is a Partial Order). *Given a derivation hierarchy δ, for all types τ_1, τ_2, τ_3:*

$$(\tau_1 \preccurlyeq_\delta \tau_1)$$
$$\textit{if}\ (\tau_1 \preccurlyeq_\delta \tau_2) \wedge (\tau_2 \preccurlyeq_\delta \tau_3)\ \textit{then}\ (\tau_1 \preccurlyeq_\delta \tau_3)$$
$$\textit{if}\ (\tau_1 \preccurlyeq_\delta \tau_2) \wedge (\tau_2 \preccurlyeq_\delta \tau_1)\ \textit{then}\ (\tau_1 = \tau_2)$$

4.2 Models

A model μ_δ is a relation from brands to types wrt a given derivation hierarchy δ. We write $\langle\rangle_\delta$ for the empty model and $\langle \overline{\mathsf{A}_i \mapsto \tau_i} \rangle_\delta$ for the model mapping A_i to type $\mu_\delta(\mathsf{A}_i) = \tau_i$.

Definition 1 (Well-Formed Model). A model μ_δ is well-formed with respect to a brand derivation hierarchy δ, denoted $\models \mu_\delta$, iff:

$$\forall \mathsf{A}, \mathsf{A}',\ \text{if}\ \delta(\mathsf{A}, \mathsf{A}')\ \text{then}\ \mu_\delta(\mathsf{A}) \preccurlyeq_\delta \mu_\delta(\mathsf{A}')$$

4.3 Typed Data

We use the notation $\mu_\delta \vdash d : \tau$ to mean that data d has type τ in the context of model μ_δ. Figure 6 defines the typing rules for data. The first few rules are trivial: every data has type $\top$ (TTop), and atoms have the type corresponding to the kind of data they belong to (TNil, TInt, TString, TTrue, TFalse). As expected, no data can have the type $\bot$.[5] Bags are values of uniform types (TEmpty, TBag).

[4] All theorems in the paper have been verified using Coq.

[5] Note that $\bot$ is still useful to have in the type system, for instance the type $\{\bot\}$ is the most precise type for the empty bag.

$$\frac{}{\mu_\delta \vdash d : \top}\ (\text{TTop}) \qquad \frac{}{\mu_\delta \vdash \mathsf{nil} : \texttt{NIL}}\ (\text{TNil}) \qquad \frac{d \in \mathcal{Z}}{\mu_\delta \vdash d : \texttt{INT}}\ (\text{TInt})$$

$$\frac{d \in \mathcal{S}}{\mu_\delta \vdash d : \texttt{STRING}}\ (\text{TString}) \qquad \frac{}{\mu_\delta \vdash \mathsf{true} : \texttt{BOOL}}\ (\text{TTrue}) \qquad \frac{}{\mu_\delta \vdash \mathsf{false} : \texttt{BOOL}}\ (\text{TFalse})$$

$$\frac{}{\mu_\delta \vdash \emptyset : \{\tau\}}\ (\text{TEmpty}) \qquad \frac{\mu_\delta \vdash d_1 : \tau \quad ... \quad \mu_\delta \vdash d_n : \tau}{\mu_\delta \vdash \{d_1, ..., d_n\} : \{\tau\}}\ (\text{TBag})$$

$$\frac{\mu_\delta \vdash d_1 : \tau_1 \quad ... \quad \mu_\delta \vdash d_n : \tau_n}{\mu_\delta \vdash [A_1 : d_1, ..., A_n : d_n] : [A_1 : \tau_1, ..., A_n : \tau_n\ |]}\ (\text{TClosed})$$

$$\frac{\mu_\delta \vdash d_1 : \tau_1 \quad ... \quad \mu_\delta \vdash d_n : \tau_n \quad \mu_\delta \vdash d'_1 : \tau'_1 \quad ... \quad \mu_\delta \vdash d'_k : \tau'_k}{\mu_\delta \vdash [A_1 : d_1, ..., A_n : d_n, \overline{B_i : d'_i}] : [A_1 : \tau_1, ..., A_n : \tau_n\ ..]}\ (\text{TOpen})$$

$$\frac{\mu_\delta(A) = \tau \quad \delta(A, A') \quad \mu_\delta \vdash d : \tau}{\mu_\delta \vdash \mathsf{brand\ A\ } (d) : \mathit{Brand\ A'}}\ (\text{TBrand})$$

Fig. 6. Typed data. $\boxed{\mu_\delta \vdash d : \tau}$

For a closed record type, a record value belongs to that type if it has the same attributes as its type and if each attribute value has the corresponding attribute type (TClosed). For an open record type, a record value belongs to that type if it has at least the attributes defined in its type and for those, each attribute value has the corresponding attribute type (TOpen). Those attributes not declared in the open record type are only assumed to be well typed. The rule for brands simply state that the content of a branded value must be of the type declared for that brand in the model μ_δ (TBrand).

The key property of the typing rules for data is that they are sound with respect to subtyping, as expressed by the following theorem.

Theorem 2 (Soundness of Typing Rules for Data). *Given a well-formed model, i.e., μ_δ such that $\models \mu_\delta$:*

$$\textit{if } (\mu_\delta \vdash d : \tau) \wedge (\tau \preccurlyeq_\delta \tau') \textit{ then } (\mu_\delta \vdash d : \tau')$$

I.e., if data d has type τ in the context of μ_δ, and τ is a subtype of τ', then data d has type τ' in the context of μ_δ.

5 Operators

We now define operations over the proposed data model. Besides operations on records and bags typically found in data languages (notably record concatenation and projection), we include operations for branding and unbranding. We leave casting for the next section, as the semantics of cast depends on the specific ways failure is handled in the host language.

5.1 Syntax and Semantics

Unary or binary *operators* are basic operations over the data model defined as functions. Most of those are only defined for data with a specific type. We provide an informal dynamic semantics, followed by typing rules which gives the precise conditions under which those operators are well-typed. A small denotational semantics is given for reference in Appendix A.

Definition 2 (Operators).

$$\begin{aligned}(\text{uops})\ \oplus d \quad &::= \mathit{ident}\ d \mid \{d\} \mid \mathit{flatten}\ d \mid \neg d \mid [A\!:\!d] \mid d.A \\ &\mid\ d\!-\!A \mid \pi_{\overline{A_i}} \mid \mathsf{brand}\ \mathsf{A}\ (d) \mid\ !d \\ (\text{bops})\ d_1 \otimes d_2 &::= d_1 = d_2 \mid d_1 \in d_2 \mid d_1 \cup d_2 \mid d_1 * d_2 \mid d_1 + d_2\end{aligned}$$

In order of presentation, the unary operators do the following:

$\mathit{ident}\ d$	returns d.
$\{d\}$	constructs a singleton bag containing the value d.
$\mathit{flatten}\ d$	flattens a bag of bags.
$\neg d$	negates a Boolean.
$[A\!:\!d]$	constructs a record with a single attribute A and value d.
$d.A$	accesses the value associated with attribute A in record d.
$d\!-\!A$	returns a record with all attributes of d except A.
$\pi_{\overline{A_i}}(d)$	returns the projection of record d over attributes $\overline{A_i}$.
$\mathsf{brand}\ \mathsf{A}\ (d)$	returns a value with brand A containing d.
$!d$	returns the content of branded value d.

In order of presentation, the binary operators do the following:

$d_1 = d_2$	compares two values for equality.
$d_1 \in d_2$	returns true if and only if d_1 is an element of bag d_2.
$d_1 \cup d_2$	returns the union of two bags.
$d_1 * d_2$	concatenates two records, favoring d_1 for overlapping attributes.
$d_1 + d_2$	returns a singleton bag with the concatenation of the two records if they are compatible, and returns $\emptyset$ otherwise.

Operators can be easily extended (e.g., for arithmetics or aggregation). *flatten* corresponds to a single-level flattening of a nested bag. Record operations are sufficient to support all standard relational and nested relational operators. The last two unary operators correspond to branding and unbranding. Branding, $\mathsf{brand}\ \mathsf{A}\ (d)$, creates a new value with brand A and content d. Unbranding, $!d$, returns the content of a branded value d. The definition for those does not make any assumption about the model of the corresponding brands, but consistency with a given model results from their typing rules which are given next.

5.2 Typing

Given a well-formed model μ_δ, the type signatures of unary operators, written $\mu_\delta \vdash \oplus d : \tau \to \tau'$ (i.e., operator $\oplus$ applied to a value d of type τ has return type τ'), are as follows:

$$\begin{array}{rcl@{\qquad}rcl}
\mu_\delta \vdash \mathit{ident}\, d & : & \tau \to \tau & \mu_\delta \vdash [A\!:\!d] & : & \tau \to [A:\tau\ |] \\
\mu_\delta \vdash \neg d & : & \texttt{BOOL} \to \texttt{BOOL} & \mu_\delta \vdash d.A & : & [A:\tau] \to \tau \\
\mu_\delta \vdash \{d\} & : & \tau \to \{\tau\} & \mu_\delta \vdash d{-}A & : & [A:\tau, \overline{B_i:\tau_i}] \to [\overline{B_i:\tau_i}] \\
\mu_\delta \vdash \mathit{flatten}\, d & : & \{\{\tau\}\} \to \{\tau\} & \mu_\delta \vdash \pi_{\overline{A_i}} & : & [\overline{A_i:\tau_i}] \to [\overline{A_i:\tau_i}\ |]
\end{array}$$

$$\begin{array}{rcl}
\mu_\delta \vdash \mathsf{brand\ A}\ (d) & : & \mu_\delta(\mathsf{A}) \to \mathit{Brand}\ \mathsf{A} \\
\mu_\delta \vdash\ !d & : & \mathit{Brand}\ \mathsf{A} \to \mu_\delta(\mathsf{A})
\end{array}$$

Note that record construction ($[A:d]$) and projection ($\pi_{\overline{A_i}}$) always return a closed record, and that field access ($d.A$) and field removal ($d{-}A$) work on both closed and open records. The typing rules for branding (resp. unbranding) takes as input (resp. returns) values of the type associated with its brand in μ_δ.

Given a well-formed model μ_δ, the type signatures of binary operators, written $\mu_\delta \vdash d_1 \otimes d_2 : \tau_1 \to \tau_2 \to \tau_3$ (i.e., operator $\otimes$ applied to values d_1 of type τ_1 and d_2 of type τ_2 has return type τ_3), are as follows:

$$\begin{array}{l}
\mu_\delta \vdash d_1 = d_2 : \tau \to \tau \to \texttt{BOOL} \\
\mu_\delta \vdash d_1 \in d_2 : \tau \to \{\tau\} \to \texttt{BOOL} \\
\mu_\delta \vdash d_1 \cup d_2 : \{\tau\} \to \{\tau\} \to \{\tau\} \\
\mu_\delta \vdash d_1 * d_2 : [\overline{A_i:\tau_{A_i}}\ |] \to [\overline{B_j:\tau_{B_j}}\ |] \to [\overline{A_i:\tau_{A_i}} * \overline{B_j:\tau_{B_j}}\ |] \\
\mu_\delta \vdash d_1 + d_2 : [\overline{A_i:\tau_{A_i}}\ |] \to [\overline{B_j:\tau_{B_j}}\ |] \to \{[\overline{A_i:\tau_{A_i}} + \overline{B_j:\tau_{B_j}}\ |]\} \\
\mu_\delta \vdash d_1 + d_2 : [\overline{A_i:\tau_{A_i}}] \to [\overline{B_j:\tau_{B_j}}] \to \{[\overline{A_i:\tau_{A_i}} + \overline{B_j:\tau_{B_j}}\ ..]\}\ \mathit{otherwise}
\end{array}$$

Note that record concatenation is well typed only for closed records, while record merge is well typed for both closed and open records. If both records in a merge are closed the type of its record output is closed, otherwise it is open.

We end this section with a theorem showing operators are total under typing conditions i.e., given value(s) of the correct input type(s), an operator always returns a value of the expected output type. The formulation relies on the evaluation judgments ($\oplus d \Downarrow d$ for unary operators and $d \otimes d \Downarrow d$ for binary operators) defined in Appendix A.

Theorem 3 (Typed Operators Consistency). *Given a well-formed model, i.e., μ_δ such that $\models \mu_\delta$, typing for unary and binary operators is consistent:*

$$\begin{array}{c}
\mathit{if}\ (\mu_\delta \vdash d : \tau) \wedge (\mu_\delta \vdash \oplus : \tau \to \tau') \\
\mathit{then}\ \exists d', (\oplus d \Downarrow d') \wedge (\mu_\delta \vdash d' : \tau')
\end{array}$$

$$\begin{array}{c}
\mathit{if}\ (\mu_\delta \vdash d_1 : \tau_1) \wedge (\mu_\delta \vdash d_2 : \tau_2) \wedge (\mu_\delta \vdash \otimes : \tau_1 \to \tau_2 \to \tau_3) \\
\mathit{then}\ \exists d', (d_1 \otimes d_2 \Downarrow d') \wedge (\mu_\delta \vdash d' : \tau_3)
\end{array}$$

6 Business Rules Compiler

We now show how the proposed brand model and type system can be applied in the context of a business rules compiler. Due to space limitations, we focus on the changes from the version without brands from (Shinnar et al. 2015).

6.1 CAMP

The Calculus for Aggregating Matching Patterns was proposed in (Shinnar et al. 2015) to model the query fragment of production rules (Forgy 1981) with aggregation (Boyer and Mili 2011). That query fragment is a (nested) pattern matched against working memory. CAMP patterns scrutinize an implicit datum (**it**), in the context of an environment (**env**) mapping variables to data. Patterns may fail if they do not match the given data. Match failure, denoted **err**, is not fatal and can trigger alternative pattern matching attempts.

Definition 3 (CAMP Syntax).

$$\begin{aligned}(\text{patterns})\ p ::= {} & d \mid \oplus p \mid p_1 \otimes p_2 \mid \textbf{map}\ p \mid \textbf{assert}\ p \mid p_1 \| p_2 \mid \textbf{it} \\ & \mid \textbf{let it} = p_1\ \textbf{in}\ p_2 \mid \textbf{env} \mid \textbf{let env} \mathrel{+}= p_1\ \textbf{in}\ p_2 \mid \textbf{cast}\ \mathsf{A}\end{aligned}$$

Definition 3 shows the syntax for CAMP with one expression added for casting. d returns a constant data. Unary ($\oplus$) and binary ($\otimes$) operators can be applied to the result of a pattern or patterns. **map** p maps a pattern p over the implicit data **it**. Assuming that **it** is a bag, the result is the bag of results obtained from matching p against each datum in **it**. Failing matches are skipped. The **assert** p construct allows a pattern p to conditionally cause match failure. If p evaluates to false, matching fails, otherwise, it returns the empty record $[]$. The $p_1 \| p_2$ construct allows for recovery from match failure: if p_1 matches successfully, p_2 is ignored; if p_1 fails to match, p_2 is evaluated. **it** returns the datum being matched. **let it** $= p_1$ **in** p_2 binds the implicit datum to the result of a pattern. **env** reifies the current environment as a record, which can then be manipulated via standard record operators. **let env** $+= p_1$ **in** p_2, adds new bindings to the environment. The result of matching p_1 must be a record, which is interpreted as a reified environment. If the current environment is compatible with the new one (all common attributes have equal values) they are merged and the pattern p_2 is evaluated with the merged environment. If they are incompatible, the pattern fails. Merge captures the standard semantics in rules languages where multiple bindings of the same variable must bind to the same value.

We add a new expression, **cast** A, which casts **it** to a given brand A and accounts for the specific failure semantics in CAMP. It returns the same value if it succeeds, and **err** if it fails. The semantics (resp. typing) for that extension of CAMP proceeds identically to the one described in (Shinnar et al. 2015), except that it takes a derivation hierarchy (resp. a model) as additional context. Instead of the evaluation judgment $\sigma \vdash p \,@\, d \Downarrow d?$, we use $\delta; \sigma \vdash p \,@\, d \Downarrow d?$, where δ is a derivation hierarchy, σ an environment binding variables to values, p a pattern, d an (implicit) input value, and $d?$ the successful match or a match failure. All the evaluation rules pass that additional context along, and the rules for casting are as follows.

$$\frac{\delta(\mathsf{A}', \mathsf{A})}{\delta; \sigma \vdash \textbf{cast}\ \mathsf{A} \mathbin{@} \mathsf{brand}\ \mathsf{A}'\ (d) \Downarrow \mathsf{brand}\ \mathsf{A}'\ (d)}\ (\text{PCast})$$

$$\frac{\neg\delta(\mathsf{A}', \mathsf{A})}{\delta; \sigma \vdash \textbf{cast}\ \mathsf{A} \mathbin{@} \mathsf{brand}\ \mathsf{A}'\ (d) \Downarrow \textbf{err}}\ (\text{PCastFail})$$

Instead of the typing judgment $\Gamma \vdash p : \tau_0 \to \tau_1$, we use $\mu_\delta; \Gamma \vdash p : \tau_0 \to \tau_1$, where μ_δ is a well-formed model, Γ a type environment, p a pattern, τ_0 the type of the (implicit) input, and τ_1 the type of successful matches. All the typing rules pass that additional context along, and the rule for casting is as follows.

$$\frac{}{\mu_\delta; \Gamma \vdash \textbf{cast}\ \mathsf{A} : \mathit{Brand}\ \mathsf{A}' \to \mathit{Brand}\ \mathsf{A}}\ (\text{TPCast})$$

Note that the dynamic semantics only requires δ which means evaluation can proceed entirely based on the brand name derivation with no need for structural checks. This suggests techniques for efficient representation and manipulation of objects should also apply in our context. The model μ_δ still needs to be passed at compile time as it is used in the typing rules for branding and unbranding operators (See Sect. 5). Type soundness holds for CAMP with brands.

Theorem 4 (Soundness of Type System for CAMP). *Given a well-formed model, i.e., μ_δ such that $\models \mu_\delta$:*

$$\begin{array}{c} \textit{if}\ (\mu_\delta; \Gamma \vdash p : \tau_0 \to \tau_1) \wedge (\mu_\delta \vdash \sigma : \Gamma) \wedge (\mu_\delta \vdash d_0 : \tau_0) \\ \textit{then}\ \exists d_1?, (\delta; \sigma \vdash p \mathbin{@} d_0 \Downarrow d_1?) \wedge (\mu_\delta \vdash d_1? : \tau_1) \end{array}$$

6.2 NRA

We use the NRA from (Shinnar et al. 2015) with one additional expression for casting. Other operators, e.g., ρ for unnesting, can be defined in terms of this core algebra (Cluet and Moerkotte 1993).

Definition 4 (NRA Syntax).

$$\begin{array}{rl} \text{(queries)}\ q ::= & d \mid \textbf{In} \mid \oplus q \mid q_1 \otimes q_2 \mid \chi_{\langle q_2 \rangle}(q_1) \mid \sigma_{\langle q_2 \rangle}(q_1) \\ & \mid\ q_1 \times q_2 \mid \bowtie^d_{\langle q_2 \rangle}(q_1) \mid q_1 \parallel q_2 \mid (\mathsf{A})\ q \end{array}$$

Here, d returns constant data, **In** returns the context value (usually a bag or a record), and $\oplus$ and $\otimes$ are the unary and binary operators from Sect. 5. χ is the map operation on bags, σ is selection, $\times$ is Cartesian product. The *dependent join*, $\bowtie^d$, evaluates q_2 with its context set to each value in the bag resulting from evaluating q_1, then concatenates records from q_1 and q_2 as in a Cartesian product. The $\parallel$ expression, which we call *default*, evaluates its first operand and returns its value, unless that value is $\emptyset$, in which case it returns the value of its second operand (as default).

$(\mathsf{A})\ q$, casts the result of q to a brand A. It returns a singleton bag containing that same value if the cast succeeds, and $\emptyset$ otherwise. The semantics (resp. typing) for the NRA proceeds identically to the one described in (Shinnar et al. 2015), except that it takes a derivation hierarchy (resp. a model) as additional context. Instead of the evaluation judgment $q\ @\ d \Downarrow d'$, we use $\delta \vdash q\ @\ d \Downarrow d'$, where δ is a derivation hierarchy, q an expression, d the (implicit) input value, and d' the output value. All the evaluation rules pass that additional context along, and the rules for casting are as follows.

$$\frac{\delta \vdash q\ @\ d_0 \Downarrow \mathsf{brand}\ \mathsf{A}'\ (d) \quad \delta(\mathsf{A}', \mathsf{A})}{\delta \vdash (\mathsf{A})\ q\ @\ d_0 \Downarrow \{\mathsf{brand}\ \mathsf{A}'\ (d)\}}\ (\text{Cast})$$

$$\frac{\delta \vdash q\ @\ d_0 \Downarrow \mathsf{brand}\ \mathsf{A}'\ (d) \quad \neg\delta(\mathsf{A}', \mathsf{A})}{\delta \vdash (\mathsf{A})\ q\ @\ d_0 \Downarrow \{\}}\ (\text{CastFail})$$

Instead of the typing judgment $q : \tau_0 \to \tau_1$, we use $\mu_\delta \vdash q : \tau_0 \to \tau_1$, where μ_δ is a well-formed model, q an expression, τ_0 the type of the (implicit) input, and τ_1 the type of the output. All the typing rules pass that additional context along, and the rule for casting is as follows.

$$\frac{\mu_\delta \vdash q : \tau \to \mathit{Brand}\ \mathsf{A}'}{\mu_\delta \vdash (\mathsf{A})\ q : \tau \to \{\mathit{Brand}\ \mathsf{A}\}}\ (\text{TCast})$$

Type soundness holds for NRA with brands.

Theorem 5 (Soundness of Type System for NRA). *Given a well-formed model, i.e., μ_δ such that $\models \mu_\delta$:*

$$\textit{if}\ (\mu_\delta \vdash q : \tau_0 \to \tau_1) \wedge (\mu_\delta \vdash d_0 : \tau_0)\ \textit{then}\ \exists d_1, (\delta \vdash q\ @\ d_0 \Downarrow d_1) \wedge (\mu_\delta \vdash d_1 : \tau_1)$$

6.3 From CAMP to NRA

Translation from CAMP to NRA proceeds identically as the one in (Shinnar et al. 2015), with one additional rule for the cast pattern:

$$[\![\mathbf{cast}\ \mathsf{A}]\!] = (\mathsf{A})\ (\mathbf{In}.D)$$

We now restate the key correctness theorems of semantics and type preservation for that translation. A CAMP pattern that evaluates to d becomes an NRA query that evaluates to $\{d\}$ and a CAMP pattern that evaluates to **err** becomes an NRA query that evaluates to $\emptyset$.

Theorem 6 (Correctness of Translation from CAMP to NRA).

$$\delta; \sigma \vdash p\ @\ d_1 \Downarrow d_2 \iff \delta \vdash [\![p]\!]\ @\ ([E : \sigma] * [D : d_1]) \Downarrow \{d_2\}$$
$$\delta; \sigma \vdash p\ @\ d_1 \Downarrow \mathbf{err} \iff \delta \vdash [\![p]\!]\ @\ ([E : \sigma] * [D : d_1]) \Downarrow \emptyset$$

Theorem 7 (Type Preservation). *Given a well-formed model, i.e., μ_δ such that $\models \mu_\delta$:*

$$\texttt{CAMP}\leftrightarrow\texttt{NRA}: \quad \mu_\delta; \Gamma \vdash p : \tau_0 \to \tau_1 \Leftrightarrow \mu_\delta \vdash [\![p]\!] : [E : \Gamma, D : \tau_0] \to \{\tau_1\}\,.$$

7 Related Work

Our work is related to several areas of databases and programming languages research which have all been influenced by Philip Wadler's work. Compiling rules to a database backend shares similarities with database-supported execution of programming languages with embedded queries, such as Links (Cooper et al. 2007) and others (Cheney et al. 2013). The Nested Relational Algebra is closely related to languages based on comprehensions such as (Trinder and Wadler 1988), and others (Tannen et al. 1992; Beeri and Kornatzky 1993; Cluet and Moerkotte 1993; Fegaras and Maier 2000). The idea to use of type annotations with a structural subtyping relation to marry nominal and structural typing is inspired by (Siméon and Wadler 2003) and others (Lee et al. 2015; Jones et al. 2015). Compared to (Siméon and Wadler 2003) we must account for the different data model which includes bags and records. This allows us to shed some of the complexity inherent to XML and XML Schema, such as derivation by restriction or substitution groups. However, the subtyping relation requires specific care to handle complex record operations. Compared to (Jones et al. 2015) and (Lee et al. 2015), we do not address typing for a full OO programming language but we integrate brands into a language suitable for bulk data processing.

Typing issues similar to ours are found in object-oriented databases, which support queries over nested objects (Berler et al. 2000). Implementations of OQL (Cluet and Moerkotte 1993) also compile queries into a nested algebra or calculus (Fegaras and Maier 2000; Trigoni and Bierman 2001; Cluet and Moerkotte 1993; Beeri and Kornatzky 1993; Claußen et al. 1997). At the surface language level, proposed type systems (Alagic 1999; Trigoni and Bierman 2001) rely on a purely nominal approach. In contrast, the type system for underlying algebras used for optimization (Cluet and Moerkotte 1993) involves structural typing and record operations but does not address inheritance. The approach presented here unifies both those type systems in a simple and natural way, allowing us to prove type-preservation for the resulting compiler.

8 Conclusion

In this paper, we have described a novel type system for languages that involve complex record operations in the context of a class hierarchy. Our compiler includes a translator from JRules to CAMP, a backend that generates Javascript code from NRA, and a query optimizer. Brands are integrated in the full pipeline and, with the exception of the initial front-end translation and Javascript generation, it has been fully mechanized and verified using the Coq proof assistant.

We are currently pursuing further development on both theoretical and practical aspects of our compiler. From a theoretical standpoint, we are exploring type inference and extensions with intersection types. From a practical standpoint, we are looking into the use types for optimization, and code-generation for various database runtimes. Beyond his work on XML types, we are indebted to Phil Wadler's long held interest in cross pollination between programming languages and databases without which this work would simply not exist.

Acknowledgments. We want to thank Joshua Auerbach, Martin Hirzel, and Louis Mandel, for their feedback on earlier versions of this draft.

A Operators Semantics

Figure 7 provides a denotational semantics for core unary operators using the judgment $\oplus d_1 \Downarrow d_2$ where $\oplus$ is the operator, d_1 the value it is applied to and d_2 the resulting value. Note that when writing e.g., $\{d\} \Downarrow \{d\}$, the left hand-side is really the application of the operator on d. I.e., it is really meant as $\lambda x.\{x\}(d) \Downarrow \{d\}$.

$$\frac{}{\mathit{ident}\ d \Downarrow d}\ (\text{Ident}) \qquad \frac{}{\{d\} \Downarrow \{d\}}\ (\text{Coll})$$

$$\frac{}{\mathit{flatten}(\{\}) \Downarrow \{\}}\ (\text{Flatten}_\emptyset) \qquad \frac{\mathit{flatten}\ (d_0) \Downarrow d_0'}{\mathit{flatten}\ (\{\{\overline{d_i}\}\} \cup d_0) \Downarrow \{\overline{d_i}\} \cup d_0'}\ (\text{Flatten}_\cup)$$

$$\frac{}{\neg\mathsf{true} \Downarrow \mathsf{false}}\ (\text{Not}_T) \qquad \frac{}{\neg\mathsf{false} \Downarrow \mathsf{true}}\ (\text{Not}_F)$$

$$\frac{}{[A:d] \Downarrow [A:d]}\ (\text{Rec}) \qquad \frac{d = [A : d_0; B_1 : d_1 ... B_n : d_n]}{d.A \Downarrow d_0}\ (\text{Field})$$

$$\frac{d = [A : d_0; B_1 : d_1; ... B_n : d_n]}{d - A \Downarrow [B_1 : d_1 ... B_n : d_n]}\ (\text{Remove})$$

$$\frac{d = [A_1 : d_1; ... A_n : d_n; B_1 : d_1'; ... B_n : d_n']}{\pi_{\overline{A_i}}(d) \Downarrow [A_1 : d_1 ... A_n : d_n]}\ (\text{Project})$$

$$\frac{}{\mathsf{brand\ A}\ (d) \Downarrow \mathsf{brand\ A}\ (d)}\ (\text{Brand}) \qquad \frac{d = \mathsf{brand\ A}\ (d')}{!d \Downarrow d'}\ (\text{Unbrand})$$

Fig. 7. Unary operators semantics. $\boxed{\oplus d \Downarrow d}$

Figure 8 provides a denotational semantics for core binary operators using the judgment $d_1 \otimes d_2 \Downarrow d_3$ where $\otimes$ is the operator, d_1 and d_2 the values it is applied to and d_3 the resulting value. Note that for concat and merge operators we rely on underlying merge and concat defined at the data model level (see Sect. A).

$$\frac{d_1 \doteq d_2}{d_1 = d_2 \Downarrow \mathsf{true}}\ (\mathrm{Eq}) \qquad \frac{\neg d_1 \doteq d_2}{d_1 = d_2 \Downarrow \mathsf{false}}\ (\neg\ \mathrm{Eq})$$

$$\frac{}{d \in \{\} \Downarrow \mathsf{false}}\ (\mathrm{In}_\emptyset) \qquad \frac{d \doteq d_1}{d \in \{d_1\} \cup d_2 \Downarrow \mathsf{true}}\ (\mathrm{In}_\cup)$$

$$\frac{\neg d \doteq d_1 \quad d \in d_2 \Downarrow d'}{d \in \{d_1\} \cup d_2 \Downarrow d'}\ (\neg\ \mathrm{In}_\cup) \qquad \frac{}{\{\overline{d_i}\} \cup \{\overline{d'_i}\} \Downarrow \{\overline{d_i}; \overline{d'_i}\}}\ (\cup)$$

$$\frac{}{[\overline{A_i : d_i}] * [\overline{A'_i : d'_i}] \Downarrow [\overline{A_i : d_i} * \overline{A'_i : d'_i}]}\ (\mathrm{Concat})$$

$$\frac{}{[\overline{A_i : d_i}] + [\overline{A'_i : d'_i}] \Downarrow [\overline{A_i : d_i} + \overline{A'_i : d'_i}]}\ (\mathrm{Merge})$$

Fig. 8. Binary operators semantics. $\boxed{d \otimes d \Downarrow d}$

References

Alagic, S.: Type-checking OQL queries in the ODMG type systems. ACM Trans. Database Syst. **24**(3), 319–360 (1999)

Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., et al.: Spark SQL: relational data processing in Spark. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 1383–1394 (2015)

Arnold, M., Grove, D., Herta, B., Hind, M., Hirzel, M., Iyengar, A., Mandel, L., Saraswat, V.A., Shinnar, A., Siméon, J., Takeuchi, M., Tardieu, O., Zhang, W.: META: middleware for events, transactions, and analytics. IBM J. Res. Devel. (IBMRD). (2016, to appear)

Bali, M.: Drools JBoss Rules 5.0 Developer's Guide. Packt Publishing, Birmingham (2009)

Beeri, C., Kornatzky, Y.: Algebraic optimization of object-oriented query languages. Theor. Comput. Sci. **116**(1&2), 59–94 (1993)

Berler, M., Cattell, R.G.G., Barry, D.K.: The Object Data Standard: ODMG 3.0. Morgan Kaufmann, San Francisco (2000)

Boyer, J., Mili, H.: IBM websphere ILOG JRules. In: Boyer, J., Mili, H. (eds.) Agile Business Rule Development, pp. 215–242. Springer, Heidelberg (2011)

Cheney, J., Lindley, S., Wadler, P.: A practical theory of language-integrated query. In: International Conference on Functional Programming (ICFP), pp. 403–416 (2013)

Claußen, J., Kemper, A., Moerkotte, G., Peithner, K.: Optimizing queries with universal quantification in object-oriented and object-relational databases. In: Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB), pp. 286–295 (1997)

Cluet, S., Moerkotte, G.: Nested queries in object bases. In: Workshop on Database Programming Languages (DBPL), pp. 226–242 (1993)

Cooper, E., Lindley, S., Yallop, J.: Links: web programming without tiers. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2006. LNCS, vol. 4709, pp. 266–296. Springer, Heidelberg (2007)

Fegaras, L., Maier, D.: Optimizing object queries using an effective calculus. ACM Trans. Database Syst. (TODS) **25**(4), 457–516 (2000)

Forgy, C.L.: OPS5 user's manual. Technical report. 2397, CMU (1981)

Green, T.J., Aref, M., Karvounarakis, G.: LogicBlox, platform and language: a tutorial. In: Barceló, P., Pichler, R. (eds.) Datalog 2.0 2012. LNCS, vol. 7494, pp. 1–8. Springer, Heidelberg (2012)

Grust, T., Mayr, M., Rittinger, J., Schreiber, T.: Ferry: Database-supported program execution. In: International Conference on Management of Data (SIGMOD), pp. 1063–1066 (2009)

Jones, T., Homer, M., Noble, J.: Brand objects for nominal typing. In: 29th European Conference on Object-Oriented Programming, ECOOP 2015, 5–10 July 2015, Prague, Czech Republic, pp. 198–221 (2015)

Lee, J., Aldrich, J., Shaw, T., Potanin, A.: A theory of tagged objects. In: 29th European Conference on Object-Oriented Programming, ECOOP 2015, 5–10 July 2015, Prague, Czech Republic (2015)

May, N., Helmer, S., Moerkotte, G.: Strategies for query unnesting in XML databases. Trans. Database Syst. (TODS) **31**(3), 968–1013 (2006)

Meijer, E., Beckman, B., Bierman, G.: Linq: reconciling object, relations and XML in the.NET framework. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp. 706–706 (2006)

Shinnar, A., Siméon, J., Hirzel, M.: A pattern calculus for rule languages: expressiveness, compilation, and mechanization. In: 29th European Conference on Object-Oriented Programming, ECOOP 2015, 5–10 July 2015, Prague, Czech Republic, pp. 542–567 (2015)

Siméon, J., Wadler, P.: The essence of XML. In: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, 15–17 January 2003, pp. 1–13 (2003)

Tannen, V., Buneman, P., Wong, L.: Naturally embedded query languages. In: Hull, R., Biskup, J. (eds.) ICDT 1992. LNCS, vol. 646, pp. 140–154. Springer, Heidelberg (1992)

Trigoni, A., Bierman, G.: Inferring the principal type and the schema requirements of an OQL query. In: Read, B. (ed.) BNCOD 2001. LNCS, vol. 2097, pp. 185–201. Springer, Heidelberg (2001)

Trinder, P., Wadler, P.: List comprehensions and the relational calculus. In: Proceedings of 1988 Glasgow Workshop on Functional Programming, Rothesay, Scotland, pp. 115–123, August 1988