Type Checking by Domain Analysis in Ampersand

Stef Joosten[1,2] and Sebastiaan J.C. Joosten[3,4]

[1] Open Universiteit Nederland, P.O. Box 2960, HEERLEN, the Netherlands
[2] Ordina NV, Nieuwegein
[3] Eindhoven University of Technology, P.O. Box 513, Eindhoven, the Netherlands
[4] Radboud University, Nijmegen, the Netherlands
corr: stef.joosten@ou.nl

Abstract. In the process of incorporating subtyping in relation algebra, an algorithm was found to derive the subtyping relation from the program to be checked. By using domain analysis rather than type inference, this algorithm offers an attractive visualization of the type derivation process. This visualization can be used as a graphical proof that the type system has assigned types correctly. An implementation is linked to in this paper, written in Haskell. The algorithm has been tried and tested in Ampersand, a language that uses relation algebra for the purpose of designing information systems.

This is a preprint to https://doi.org/10.1007/978-3-319-24704-5_14.

1 Introduction

In building information systems, the challenge is to translate business policy into a running system that can support that policy. According to the Business Rules Method [1], a business policy is best described by a set of agreements called business rules. In our view, creating an information system should *only* involve writing down these agreements in a formal language, a compiler should do the rest. Ampersand is a project in which such a compiler was developed [2]. It uses a variant of heterogeneous relation algebra [3] as formal language for three reasons: relation algebra is suited for symbolic manipulation, relation algebra is close to natural language, and relation algebra is easily implemented through SQL.

Specialization and overloading are used frequently in business rules, and Ampersand supports this. For this purpose, a slight modification of the heterogeneous relation algebra was implemented, and a new type system was developed. Van der Woude et al. describes this slightly modified algebra in [4]. In line with the typing rules of that paper, this paper describes a type system for Ampersand.

The type system has two purposes that are common for type systems. First, the type system should *assign one type* to every term in the script. This means that tools which use the script as input, have access to the type information,

and can use it for its analysis. This way, a type system helps improve the output of such tools. Second, the type system should give *feedback to the user*. This includes alerting the user to type errors or warnings, and can even be feedback in the form of type derivations. This way, a type system helps the user to improve her code.

While type systems have been around for quite a while, we tried out a novel approach to typing. Our type system differs from conventional systems in two important ways:

- Even though type inference is quite common since the Hindley–Milner type system [5], rules for subtyping (aka generalization or specialization) are typically made explicit. This paper will show that type rules for subtyping can be derived too. This enabled a very uniform and appealing syntax in the Ampersand language, in which relations, domains and subdomains can be declared implicitly.
- Typically, type systems construct tree-shaped proofs in a type deduction system. Our system, however, uses a directed graph rather than a tree to derive types. This allows us to use simple and intuitive algorithms for doing type derivation. It also means that the type checker can explain its calculations by drawing the relevant portion of the graph.

The result appeals to our desire for simplicity and elegance. In order to validate this result beyond the usual toy examples, we have implemented the type system in the Ampersand compiler and tried it in practice. Soon, we found that the lack of distinction between rules and declarations led to unexpected results in scripts with errors. For practical reasons, it is imperative that erroneous scripts yield error messages that make sense to their authors. So we were forced to introduce the distinction between declarations (of relations), statements to specify generalization, and rules in Ampersand, which now has a more conventional type checker.

Still, the elegant properties remain. We present the type system in the hope that the reader finds joy in its elegance, finds another use for it, or finds the lessons learned from our attempt instructive.

2 Related work

There is a large body of work on type systems. The rationale behind a good type system is to verify programs mechanically in order to prevent erroneous runs. Within the category of syntactically correct programs, a type system should distinguish those programs that can be interpreted semantically from those without such an interpretation. On top of that, a type system may forbid or warn against input that is unexpected.

The type system proposed in this article was designed to cater for subtyping in Ampersand. Already in the 90's it was recognized that subtyping increases the expressive power of a type system as well as the power of its mechanical verification [6–9]. However, this comes at a price. Subtyping can have performance

penalties or it may incur problems with the decidability of type inference, for instance [10, 11]. All type systems we have studied assume there is, or define, an explicit subtype definition. The type system proposed in this paper derives the subtyping from the same program it analyses.

Undecidability of a type system does not mean it cannot be used in practice. Dynamically typed languages work around this issue by checking type correctness on run time, keeping track of type information at runtime. When some of that type information is calculated statically, the result is called hybrid type checking [12]. Our work takes the orthogonal approach: we derive type information, and refuse input for which this derivation failed. The user always has the option to add type information himself, in case the derivation was not powerful enough.

Ampersand is based on a variant of heterogeneous relation algebra described in [4]. Outside of the Ampersand project, we do not know of any type checker for this algebra. However, in the area of machine learning, the work of [13] seems to be working in the same setting. This work does not describe a type checker, but the notion of subclass coincides with ours, and applies to relations in the same way. In addition, the work - like ours - tries to derive subclass information without requiring additional input. Except for this work, and the scope of Ampersand, we could not find a setting in which the same variant of heterogeneous relation algebra would apply. Besides a shared notion of subsets in relation algebra, there are only differences: assigning a type to an entity for 'question answering', as solved in [13], at best gives a heuristic to perform dynamic type checking. Ampersand is designed for static type checking only.

3 Problem definition

This paper focuses on the problem to assign to every term in Ampersand precisely one type, which is formulated in Equation 2. Ampersand interprets each concept[5] as a set and each relation as a set of pairs. Instances of a concept are called *atoms* and are elements in those sets. Ampersand lets its user model relations in combination with concepts. If, for example, the user defines a relation $account_{\langle Person, IBAN \rangle}$, she wants to be sure that the relation is populated with instances of those concepts, i.e. the relation contains pairs of persons and IBAN-numbers only. So, if $\langle$`Peter`, `NL99BANK0123456789`$\rangle$ is a pair within this relation, the type system must ensure that:

- `Peter` is an instance of $Person$ and any concept that is more general (e.g. $LegalEntity$).
- `NL99BANK0123456789` is an instance of $IBAN$ and any concept that is more general (e.g. $Accountnumber$).

The type system aims to:

[5] The notions *concept*, *atom*, *relation*, *term*, and *rule* are defined formally in section 4.

– maintain the algebraic properties of homogeneous relation algebra (i.e. the axioms of Tarski).
– ensure that calculations with relational terms maintain type correctness.

These first two requirements have been elaborated and published [4]. This study by van der Woude and Joosten analyses how heterogeneous relation algebra fails to maintain all of Tarski's axioms. It blames the complement operator and solves that problem by using the binary minus operator instead. That result is used in Equation 18 of the current paper. In everyday use, this means that one of Tarski's axioms is restricted in a way that does not inhibit practical applications.

Ampersand poses requirements to the type system. Some of these were trivial to implement, and have been omitted from the details in the paper. Others will be treated in the sections hereafter:

– By typing a term, we restrict the values it may have at run time. This can improve run time performance in many cases. In Ampersand, every term gets exactly one type. This requirement is called soundness, and is formalized at the end of this section in Equation 2.
– In order to make it easier to reuse names, the type system must allow overloading.
  For example, a user is free to declare both

$$account_{\langle Person, IBAN\rangle} \quad \text{and}$$
$$account_{\langle Purchase, IBAN\rangle}$$

  in the same script. This introduces two different relations, just because $Person$ and $Purchase$ are distinct concepts. Upon each use of a declaration, the type system must allow omission of the type in cases where no confusion can arise. In this paper, overloading of relations will not be discussed.
– In order to facilitate collaborative development and code reuse, the type system must cope with specialization.
  For example, if an apartment is a home, Ampersand allows the user to state: $Apartment \preceq Home$. This means that every instance of *Apartment* is an instance of $Home$ as well. The consequence is that all relations that work with $Home$ are applicable to *Apartment* as well. In this interpretation, the word generalization has the same meaning and may be used as a synonym. Specialization can be helpful for reusing code that was written for $Home$. Allowing specialization will be an emergent property of our type system
– The type system must allow intuitive explanation of results to users. We aimed to achieve this by using a graph as visualization of our type system.

Relation algebra uses binary relations. In Ampersand, each relation r has a type $\mathfrak{T}(r) = \langle A, B\rangle$, which is a tuple of concepts. As a shorthand for 'r with type $\langle A, B\rangle$', we write $r_{\langle A,B\rangle}$. A relation contains a set of pairs, the elements of which are called *atoms*. Each pair $\langle a, b\rangle$ has a source atom a and a target atom b. To distinguish between a relation and the set of pairs it contains, we use a function $\mathfrak{I}(\cdot)$. We will also use $\mathfrak{I}(\cdot)$ to indicate the set of atoms in concepts and types. This is defined formally in Definition 2.

Every atom is an element of a concept, which is called the type of that atom. Formally, the typing of two atoms a and b in a relation $r_{\langle A,B\rangle}$ is described by:

$$\langle a, b\rangle \in \mathfrak{I}(r_{\langle A,B\rangle}) \Rightarrow a \in \mathfrak{I}(A) \ \wedge \ b \in \mathfrak{I}(B) \tag{1}$$

The function $\mathfrak{I}(\cdot)$ is not available to the type system, so to ensure Property 1, the type system will reason with type terms rather than atoms. For each term t there are the type-terms $dom(t)$ and $cod(t)$[6]. For each concept C, there is the type-term $pop(C)$. The set of all source atoms in a relation r is indicated by $dom(r)$ (pronounced: domain of r) and the set of all target atoms in that relation is indicated by $cod(r)$ (codomain of r). The set of all atoms that are an instance of concept C can be indicated by $pop(C)$ (population of C). Note that $\mathfrak{I}(C)$ (the interpretation of the concept C) is equal to $\mathfrak{I}(pop(C))$ (the interpretation of the concept C interpreted as a type-term). Similar equalities will be used to established the soundness of our algorithm.

The problem for the type system to solve is to assign to every term t precisely one type $\mathfrak{T}(t)$ such that:

$$\mathfrak{T}(t) = \langle A, B\rangle \ \Rightarrow \ \mathfrak{I}(dom(t)) \subseteq \mathfrak{I}(pop(A)) \ \wedge \ \mathfrak{I}(cod(t)) \subseteq \mathfrak{I}(pop(B)) \tag{2}$$

This equation specifies soundness. The task of the type algorithm is to ensure that every term t has precisely one type $\mathfrak{T}(t)$ and to decide whether a script can satisfy Property 2 at runtime.

4 Definitions

In order to describe the type system, we need definitions of the notions *atom*, *concept*, *relation*, *term*, and *rule*.

Since we describe a type system, it is not necessary for the reader to know what an Ampersand script does. Nevertheless, it might help with the intuitions, so we give those here: A script contains a set of rules. A rule is the equality between two terms, built from relations. Rules can be thought of as invariants throughout the execution of a program. Based on the relations used in them, Ampersand will generate a database and some interfaces. The phase in which Ampersand takes a script, and turns it into a database, is what we will refer to as compile time. The phase in which a (possibly different) user interacts with the database, is what we will refer to as runtime. The database can then be used to calculate which atom-pairs are in the relations, from which Ampersand can decide whether all rules are satisfied. If not, the last change is reverted, returning the database to a 'safe' state. The changes to the database are not specified at compile time, but given by the database user at runtime. This means that Ampersand scripts do not have a notion of execution, and that atoms are only a runtime concept.

[6] Later, we will also introduce $inter(s, t)$ as a type-term

Atoms are values that have no internal structure, meant to represent data elements in a database. From a business perspective, atoms are used to represent concrete items of the world, such as `Peter`, `1`, or `the king of France`. By convention throughout the remainder of this paper, variables a, b, and c are used to represent *atoms*. The set of all atoms is called $\mathbb{A}$. Each atom is an instance of a *concept*.

Concepts are names we use to classify atoms in a meaningful way. For example, you might choose to classify `Peter` as a person, and `074238991` as a telephone number. We will use variables A, B, C, D to represent concepts. Let us call the set of all concepts in an Ampersand script $\mathbb{C}$. The expression $a \in A$ means that atom a is an instance of concept A. In the syntax of Ampersand, concepts form a separate syntactic category, allowing a parser to recognize them as concepts. The declaration of $A \preceq B$ (pronounce: A is a B) in an Ampersand script states that any instance of A is an instance of B as well. We call this *specialization*, but it is also known as *generalization* or *subtyping*. Specialization is needed to allow statements such as: "An orange is a fruit that ....".

Relations are used to represent sets of facts (i.e. statements that are true in a business context), to be stored and maintained as data in a computer. As data changes over time, so do the contents of these relations. In this paper relations are represented by variables r, s, and d. We represent the declaration of a relation r in an Ampersand script by $r_{\langle A,B\rangle}$, in which A is the source concept and B the target concept. The relation $\mathbb{I}_A$ represents the *identity relation* of concept A. The relation $\mathbb{V}_{A\times B}$ represents the *universal relation* over concepts A and B. The set of all identifiers that represent relations in an Ampersand script, is called $\mathbb{D}$. It is defined by:

$$r \in \mathbb{D} \text{ iff } r_{\langle A,B\rangle} \text{ occurs in the Ampersand script.} \tag{3}$$

The meaning of relations in Ampersand is defined by an interpretation function $\mathfrak{I}$. It maps each relation to a set of facts. The declaration of $r_{\langle A,B\rangle}$ implies $r \in \mathbb{D}$, and $A, B \in \mathbb{C}$. Furthermore, it is a runtime requirement that the pairs in r are contained in its type:

$$\langle a, b\rangle \in \mathfrak{I}(r) \Rightarrow \ a \in \mathfrak{I}(pop(A)) \wedge b \in \mathfrak{I}(pop(B)) \tag{4}$$

Terms are used to combine relations using operators. The set of terms is called $\mathbb{T}$. It is defined by:

Definition 1 (terms).

The set of terms, $\mathbb{T}$, is the smallest set that satisfies, for $r, s \in \mathbb{T}$, $d \in \mathbb{D}$ and $A, B \in \mathbb{C}$.

$$d \in \mathbb{T} \qquad \textit{(every relation is a term)} \tag{5}$$
$$(r \cap s) \in \mathbb{T} \qquad \textit{(intersection)} \tag{6}$$
$$(r - s) \in \mathbb{T} \qquad \textit{(difference)} \tag{7}$$
$$(r; s) \in \mathbb{T} \qquad \textit{(composition)} \tag{8}$$
$$r^{\smile} \in \mathbb{T} \qquad \textit{(converse)} \tag{9}$$
$$\mathbb{I}_A \in \mathbb{T} \qquad \textit{(identity)} \tag{10}$$
$$\mathbb{V}_{A \times B} \in \mathbb{T} \qquad \textit{(full set)} \tag{11}$$

Throughout the remainder of this paper, terms are represented by variables r, s, d, and t.

The meaning of terms in Ampersand is an extension of interpretation function $\mathfrak{I}$. Let A and B be finite sets of atoms, then $\mathfrak{I}$ maps all terms to the set of facts for which that term stands.

Definition 2 (interpretation of terms).
For every $A, B \in \mathbb{C}$ and $r, s \in \mathbb{T}$

$$\mathfrak{I}(r \cap s) = \{\langle a, b\rangle |\ \langle a, b\rangle \in \mathfrak{I}(r) \textit{ and } \langle a, b\rangle \in \mathfrak{I}(s)\} \tag{12}$$
$$\mathfrak{I}(r - s) = \{\langle a, b\rangle |\ \langle a, b\rangle \in \mathfrak{I}(r) \textit{ and } \langle a, b\rangle \notin \mathfrak{I}(s)\} \tag{13}$$
$$\mathfrak{I}(r; s) = \{\langle a, c\rangle |\ \textit{for some } b,\ \langle a, b\rangle \in \mathfrak{I}(r) \textit{ and } \langle b, c\rangle \in \mathfrak{I}(s)\} \tag{14}$$
$$\mathfrak{I}(r^{\smile}) = \{\langle b, a\rangle |\ \langle a, b\rangle \in \mathfrak{I}(r)\} \tag{15}$$
$$\mathfrak{I}(\mathbb{I}_A) = \{\langle a, a\rangle |\ a \in A\} \tag{16}$$
$$\mathfrak{I}(\mathbb{V}_{A \times B}) = \{\langle a, b\rangle |\ a \in A, b \in B\} \tag{17}$$

In fact, Ampersand has even more operators: the complement (prefix unary $-$), kleene closure operators (postfix $^+$ and *), left- and right residuals (infix $\backslash$ and $/$), relational addition (infix $\dagger$), and product (infix $\times$). These do not introduce new concepts, just more terms. We have constrained this exposition to the operators mentioned above, which is sufficient for explaining the type system.

To solve the problems with the complement operator in heterogeneous relation algebra [4], Ampersand uses a binary difference operator as in Equation 7. It is used to define a complement as a unary prefix operator $-$ for relations of which the type is known.

$$\mathfrak{T}(r) = \langle A, B\rangle \ \Rightarrow\ -r \ =\ \mathbb{V}_{A \times B} - r \tag{18}$$

After approval of the script by the type system, every term has a unique type. Since scripts with type errors cannot be executed, ordinary users of Ampersand never get to see any behavior of the complement other than they can predict with Tarski's axioms.

After defining concepts, relations and terms, let us now define rules. Rules are used to impose constraints on the data in relations.

A rule is a pair of terms $r, s \in \mathbb{T}$. We indicate the set of all rules in an Ampersand script by $\mathbb{R}$. To indicate that a pair of terms (r, s) is in $\mathbb{R}$, we will write:

$$\text{RULE } r = s$$

The rule RULE $r = s$ imposes the following restriction on the data in Ampersand:

$$\mathfrak{I}(r) = \mathfrak{I}(s)$$

For a user, this means that a rule restricts the possible populations in the database to those that satisfy the rule.

Note that a declaration $r_{\langle A,B \rangle}$ in Ampersand can be represented by the following rule:

$$\text{RULE } r = r \cap \mathbb{V}_{A \times B}$$

5 Domain analysis

The core idea of the proposed type algorithm is an analysis of domains. A domain is a set of atoms. We introduce type-terms to represent such sets; our algorithm will act on type-terms. There are four functions that yield type-terms: $dom(\cdot)$, $cod(\cdot)$, $inter(\cdot,\cdot)$ and $pop(\cdot)$, with the following interpretation.

Definition 3 (Interpretation of type-terms).
For every $a, b \in \mathbb{A}$ *and* $r, s \in \mathbb{T}$

$$\mathfrak{I}(dom(r)) = \{a \mid \langle a, b \rangle \in \mathfrak{I}(r)\} \tag{19}$$
$$\mathfrak{I}(cod(r)) = \{b \mid \langle a, b \rangle \in \mathfrak{I}(r)\} \tag{20}$$
$$\mathfrak{I}(inter(r, s)) = \mathfrak{I}(cod(r)) \cap \mathfrak{I}(dom(s)) \tag{21}$$
$$\mathfrak{I}(pop(A)) = \mathfrak{I}(A) \tag{22}$$

Note that we use the word *type-term* to indicate the intermediate structures used by the type system. The word *type* is used for a pair of concepts.

Since $\mathfrak{I}(cod(r)) = \mathfrak{I}(dom(r^\smile))$, we can treat $cod(r)$ as a shorthand notation for $dom(r^\smile)$. Similarly, $\mathfrak{I}(pop(A)) = \mathfrak{I}(A)$, so we can treat $pop(\cdot)$ as a shorthand notation, too. In fact, we could have avoided introducing $cod(\cdot)$ and $pop(\cdot)$, and rely solely on $dom(\cdot)$. However, the use of $cod(\cdot)$ and $pop(\cdot)$ makes it easier for the reader to keep track of the terms involved in these type terms. We will treat $inter(s, t)$ as is, without creating a shorthand: the obvious shorthand $dom(s \cap t^\smile)$ contains a term that is not necessarily well typed.

Listing 1.1 introduces three relations, `r[A*C]`, `s[A*B]`, and `t[B*C]` and one rule: "`r = s;t`". This rule has four terms: "`r`", "`s;t`", "`s`", and "`t`". Domain analysis can be used to derive the type of expressions. For example, in the

Listing 1.1. A type correct Ampersand script

```
3 RELATION r[A*C]
4 RELATION s[A*B]
5 RELATION t[B*C]
6 RULE r = s;t
```

context of listing 1.1, we can derive:

$$\begin{array}{l} \mathfrak{I}(dom(s;t)) \\ \subseteq \\ \mathfrak{I}(dom(s)) \\ = \\ \mathfrak{I}(dom(s_{A*B})) \\ \subseteq \\ \mathfrak{I}(pop(A)) \end{array} \qquad \begin{array}{l} \mathfrak{I}(cod(s;t)) \\ \subseteq \\ \mathfrak{I}(cod(t)) \\ = \\ \mathfrak{I}(cod(t_{B*C})) \\ \subseteq \\ \mathfrak{I}(pop(C)) \end{array}$$

Together these two calculations prove that term $s;t$ has type $\langle A, C\rangle$.

The domain analysis introduces a relation *sub* between type terms. The intention is that *sub* is the counterpart of the subset relation $\subseteq$. It translates the observation that $\mathfrak{I}(dom(s;t)) \subseteq \mathfrak{I}(dom(t))$ to type terms by stating that $dom(s;t)$ *sub* $dom(t)$. By constructing the *sub* relation consistently for all operators in every term in listing 1.1, the algorithm from section 6 constructs a type-graph. The result is shown in figure 1. Every vertex (ellipse) in this graph represents a set of type-terms: in this particular example, the type system has distinguished 10 distinct sets. Type-terms that are proven to represent the same set of atoms are printed inside the same ellipse. A directed path from a vertex containing type-term t_1 to a vertex containing type-term t_2, indicates that atoms in $\mathfrak{I}(t_1)$ are also in $\mathfrak{I}(t_2)$. Each derivation such as the two above can be seen as a path in a type graph. In fact, the type graph even shows shorter derivations, when possible.

The type graph can be interpreted as a collection of derivations, that can be used to prove that an expression has a certain type. Because the graph itself contains all proofs that all terms have precisely one type, there is no need to write down all calculations. The graph itself can serve as a compact representation of all the necessary calculations.

Let a second example illustrate how mistakes are analyzed. Consider a script with a type error in Listing 1.2. The type graph of this script is represented in Figure 2. One of the errors is a mismatch between $cod(s)$ and $dom(t)$ (on line 6 position 11). Because Ampersand treats concepts `A` and `B` as sets with an empty intersection, this is treated as an error. We have marked that error in red in Figure 2. Another error is for example that the domain of `r` (on line 6 position 6) is *sub* of `A` and also of `B`. In general, every term that leads to more than one pop-vertex is erroneous, as well as the terms that lead to no pop-vertex. In the Ampersand type system, a single erroneous term is sufficient reason to reject the

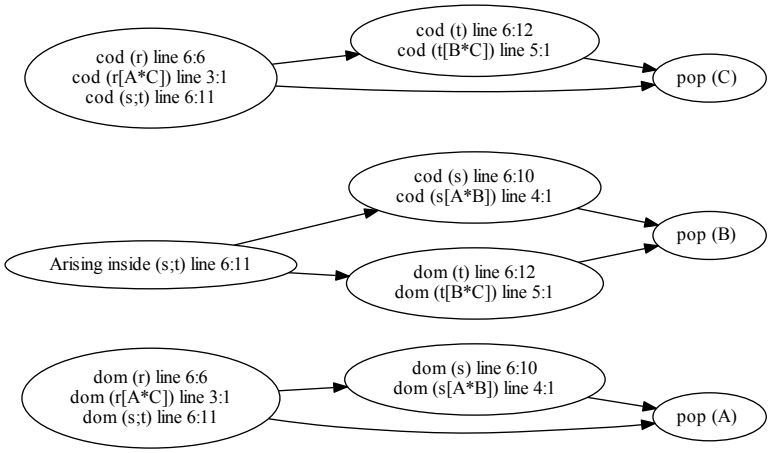

Fig. 1. Type graph for Listing 1.1

Listing 1.2. A type incorrect Ampersand script

```
3 RELATION r[A*C]
4 RELATION s[B*A]
5 RELATION t[B*C]
6 RULE r = s;t
```

entire script. As a consequence, every term in a type-correct script has precisely one type.

6 Algorithm

This section explains the algorithm of the proposed type system. A toy version of the domain analysis can be found online[7]. It is meant to communicate the idea and play with it in GHCi.

The algorithm for type checking is an algorithm on type graphs, for which we introduce a set of vertices $\mathbb{N}$ and a set of edges $sub_{\langle \mathbb{N},\mathbb{N} \rangle}$. Each vertex v represents a set of atoms. The relation sub represents edges in the type graph. So v sub v' represents an edge from vertex v to vertex v', which means that the set of atoms v is a subset of v'.

[7] `http://cs.ru.nl/~B.Joosten/ampTypes/`

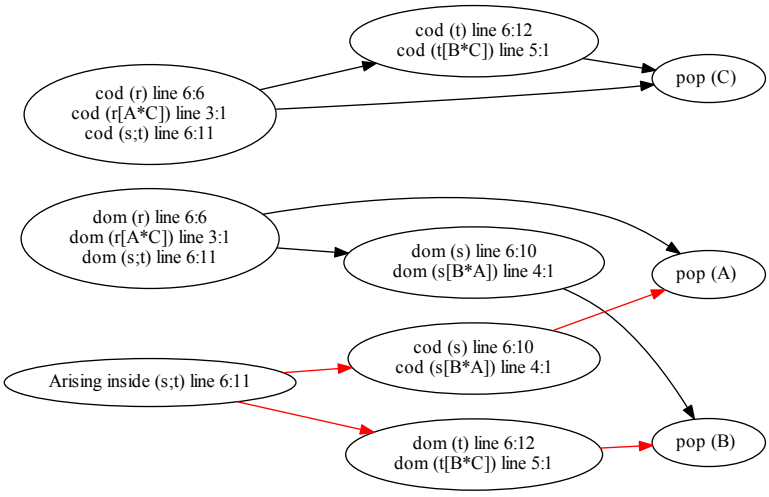

Fig. 2. Type graph for Listing 1.2

For each term in the script, edges are computed using the following rules.

$$dom(\mathbb{V}_{A\times B})\ sub\ pop(A) \wedge cod(\mathbb{V}_{A\times B})\ sub\ pop(B) \tag{23}$$
$$dom(r_{\langle A,B\rangle})\ sub\ pop(A) \wedge cod(r_{\langle A,B\rangle})\ sub\ pop(B) \tag{24}$$
$$dom(\mathbb{I}_A)\ sub\ pop(A) \wedge cod(\mathbb{I}_A)\ sub\ pop(A) \tag{25}$$
$$dom(r \cap s)\ sub\ dom(r) \wedge dom(r \cap s)\ sub\ dom(s) \tag{26}$$
$$cod(r \cap s)\ sub\ cod(r) \wedge cod(r \cap s)\ sub\ cod(s) \tag{27}$$
$$dom(r - s)\ sub\ dom(r) \wedge cod(r - s)\ sub\ cod(r) \tag{28}$$
$$dom(r; s)\ sub\ dom(r) \wedge cod(r; s)\ sub\ cod(s) \tag{29}$$
$$dom(r^\smile)\ sub\ cod(r) \wedge cod(r^\smile)\ sub\ dom(r) \tag{30}$$
$$dom(r)\ sub\ cod(r^\smile) \wedge cod(r)\ sub\ dom(r^\smile) \tag{31}$$

Note that we cannot state that $pop(A)\ sub\ dom(\mathbb{V}_{A\times B})$, because B can be empty. In that case, $\mathfrak{I}(dom(\mathbb{V}_{A\times B}))$ is a proper subset of $\mathfrak{I}(pop(A))$. The composition term $r; s$ needs a set of atoms beside the domain and codomain. The reason is that the interpretation $\mathfrak{I}(r; s)$ contains an existential quantifier, for which we require a type. For that purpose we introduce $inter(r, s)$ and the following rule:

$$inter(r, s)\ sub\ cod(r)\ \wedge\ inter(r, s)\ sub\ dom(s) \tag{32}$$

In addition, we add edges for every rule RULE $r = s$:

$$dom(r)\ sub\ dom(s) \wedge cod(r)\ sub\ cod(s) \wedge dom(s)\ sub\ dom(r) \wedge cod(s)\ sub\ cod(r) \tag{33}$$

Recall that RULE $r = s$ implies $\mathfrak{I}(r) = \mathfrak{I}(s)$. Using Definition 2, the reader should now be able to verify that Equations 23 to 33 all satisfy:

$$n_1 \ sub \ n_2 \Rightarrow \mathfrak{I}(n_1) \subseteq \mathfrak{I}(n_2) \tag{34}$$

This means that an arrow connecting domains n_1 and n_2 in Figures 1 and 2 may be read as a subset relation that has been recognized (by the type algorithm) between these domains. In fact, Equations 23 through 33 describe the edges between vertices for all possible terms. As the compiler traverses the parse tree recursively, it visits all terms in the script and collects relevant edges on the way.

The type system must establish that each term gets precisely one type. It does so by taking all $pop(\cdot)$ vertices it encounters when traversing the graph from the term that is to be typed. For this, we introduce the relation of pre-types $P : \mathbb{T} \times \mathbb{C}$ using sub^* as the reflexive transitive closure of sub:

$$P = \{\langle x, C\rangle \mid x \ sub^* \ pop(C)\} \tag{35}$$

The relation P is total for terms: for every domain or subdomain x, there is a term $pop(C)$ such that $\langle x, C\rangle \in P$. This holds because we require every declaration to be declared with a type. So according to Equations 23 to 25, P is total for those. The only terms we can construct according to Definition 1 are terms that are smaller than the declarations they are made of. So for every term x:

$$(\exists_{C\in\mathbb{C}} \ \langle dom(x), C\rangle \in P) \wedge (\exists_{C\in\mathbb{C}} \ \langle cod(x), C\rangle \in P) \tag{36}$$

Using this property, we could easily make a type system that is complete in the sense that it assigns a type to every term, by picking any such concept arbitrarily. However, the fact that there is a choice often indicates a mistake of an Ampersand user. So, instead of choosing an arbitrary concept, the compiler emits an error message forcing the user to make that choice.

If one of these vertices is smallest with respect to sub^*, it is used as the type for that term. If not, a type error is shown to the user. In other words, the typing function is governed by these rules:

$$\mathfrak{T}(t) = \langle A, B\rangle \Rightarrow \langle dom(t), A\rangle \in P \wedge \langle cod(t), B\rangle \in P \tag{37}$$

$$\mathfrak{T}(t) = \langle A, B\rangle \wedge \langle dom(t), A'\rangle \in P \Rightarrow \langle pop(A), A'\rangle \in P \tag{38}$$

$$\mathfrak{T}(t) = \langle A, B\rangle \wedge \langle cod(t), B'\rangle \in P \Rightarrow \langle pop(B), B'\rangle \in P \tag{39}$$

In this manner, type checking involves computing a Kleene closure over the relation sub. The compiler uses the Warshall algorithm for computing the closure, giving it polynomial ($O(n^3)$ with n the number of type-terms) complexity.

7 Fulfillment of requirements

In the previous sections, we have explained and illustrated how the type system works. The soundness of the type system is specified by Equation 2. Equations 35 and 37 yield:

$$\mathfrak{T}(t) = \langle A, B\rangle \Rightarrow dom(t) \ sub^* \ pop(A) \wedge cod(t) \ sub^* \ pop(B)$$

Together with Equation 34, and transitivity of $\subseteq$, this yields:

$$\mathfrak{T}(t) = \langle A, B \rangle \Rightarrow \mathfrak{I}(dom(t)) \subseteq \mathfrak{I}(pop(A)) \wedge \mathfrak{I}(cod(t)) \subseteq \mathfrak{I}(pop(B))$$

So we have established soundness, i.e. our type system satisfies Equation 2.

For sound scripts, P (Equation 35) is total. Since P associates one or more concepts to the domain or codomain of every term, P is a total relation. We have chosen to implement the algorithm by choosing the smallest concept instead of an arbitrary one. In the Example from Listing 1.2, the concepts P associates to $inter(s, t)$ are A and B: choosing an arbitrary one would be unsound. If the smallest concept is not unique, as in our example, the type algorithm emits an error message that forces the user to make a choice. By resolving such errors detected by Ampersand, the user can more easily be alerted to unintended scripts. In an Ampersand script without type errors, each term gets a unique type in a way that may be more predictable for the user.

The type graph can be used for a visual check: If for every vertex v there is a path to precisely one smallest $pop(\cdot)$ vertex, the script has no errors. This is precisely what was illustrated in figures 1 and 2. A reviewer of this paper suggested highlighting the relevant paths in the graph for each type and type error, which we think would be a great tool to assist Ampersand users.

The user has syntax to specialize concepts, e.g. $A \preceq B$ as defined in Section 4. The type system can already handle such statements, if we assume them to be syntax sugar for rules. The statement, $A \preceq B$ can be expressed as:

$$\text{RULE } \mathbb{I}_A = \mathbb{I}_B \cap \mathbb{I}_A \tag{40}$$

By Equation 33 the type algorithm calculates the right edges in the type graph, from which a pre-order of concepts is constructed.

The user is given control over intersection of concepts by insisting that every set must have a type. If the type algorithm computes an intersection set between two concepts, but that vertex is not associated a $pop(\cdot)$ of some concept, it is forbidden. This gives the user full control, because he can always add a statement of the same form as Equation 40 to make the type system accept what he wanted.

The type graph can be interpreted in term of good old sets, and it is easy to see that it represents type calculations as was done with figure 1. In our teaching practice, this has shown to appeal to the intuition of students, making it easy to explain.

The static typing is enabled by this type system because it reasons solely with concepts and not with atoms. For this reason, the implementation in the Ampersand compiler aborts after emitting type errors, or it proceeds if there are no mistakes.

8 Discussion

The authors appreciate domain analysis for visualizing the type system, as illustrated in figures 1 and 2. For large scripts, the type graph loses value because the

user loses overview. But in smaller, yet complicated scripts, it gives an insightful outlook both on the correctness of the script as of its incorrectness.

In some cases, the type system might reject a script in which a user has correctly represented a desirable situation. For example, this script will be rejected:

$$\begin{aligned} &\text{DECLARE} && r_{\langle A,B\rangle} \\ &\text{DECLARE} && s_{\langle C,D\rangle} \\ &\text{RULE} && \mathbb{I}_E = \mathbb{I}_B \cap \mathbb{I}_C \\ &\text{RULE} && r;s = \mathbb{V}_{A\times D} \end{aligned}$$

In this script, the first two lines declare the relations r and s. The third line introduces a concept E, for which the type system will know that it is smaller than B and C. The $inter(r,s)$ vertex arising in the last rule is known to be a subset of both B and C. Even though this means that it must be a subset of $\mathbb{I}_E$, the type system does not discover this automatically. At this point, the Ampersand user is required to be more specific, and change the last line to:

$$\text{RULE } r;\mathbb{I}_E;s = \mathbb{V}_{A\times D}$$

This introduces $inter(\mathbb{I}_E,s)$ and $inter(r,(\mathbb{I}_E;s))$ as terms, which are both typed as $pop(E)$. For the Ampersand user, adding such terms should feel like type casting. We point out to users that type errors can often be resolved by adding type information to the script. The user should add type information to "help" the type system. This appears to be easy to explain and quite intuitive for users.

As indicated earlier, Ampersand has more operators, including the complement (unary $-$), product[8] ($\times$), and union ($\cup$). These can be implemented by using the previously introduced difference, intersection, and full relation, given that the $\cdot$ placeholders can be replaced by the correct concepts:

$$\begin{aligned} -r &= \mathbb{V}_{\cdot\times\cdot} - r \\ (r \cup s) &= -(-r \cap -s) \\ r \times s &= r;\mathbb{V}_{\cdot\times\cdot};s \end{aligned}$$

These definitions require a type for the full relation $\mathbb{V}_{\cdot\times\cdot}$ to be given. The user may specify this type, but we do not require this. Separate heuristics try to infer the type, but require the user to specify the type if the type system cannot.

The type system allows the use of both $\mathbb{V}$ and $\mathbb{I}$ without type. As an example, consider a script with the relation $r_{\langle A,A\rangle}$. The union of r and $\mathbb{I}$ would be expressible in terms of the ambiguous $\mathbb{V}$:

$$(r \cup \mathbb{I}) = \mathbb{V} - ((\mathbb{V} - r) \cap (\mathbb{V} - \mathbb{I}))$$

[8] The definition below is the one currently implemented in ampersand, even though the name 'product' and the symbol $\times$ might suggest another operator. We are open to suggestions for a better name.

In this case, the different occurrences of $\mathbb{V}$ may still refer to different relations. They are separated by their position in the script. After applying a heuristic to guess the type of each $\mathbb{V}$, the type system obtains an expression to which it can apply the methods discussed in this paper:

$$(r \cup \mathbb{I}_A) = \mathbb{V}_{A\times A} - ((\mathbb{V}_{A\times A} - r) \cap (\mathbb{V}_{A\times A} - \mathbb{I}_A))$$

For this heuristic, the type of any surrounding declared relation(s) is used.

Note that $(r \cup \mathbb{I}_A)$ and $(r \cup \mathbb{I}_B)$ may be terms with a very different interpretation. So, it is necessary to disambiguate the type of every expression in a script. In the above, the type system picks $(r \cup \mathbb{I}_A)$ as the intended meaning of $(r \cup \mathbb{I})$. In each case with multiple choices, the type system produces an error, alerting the user about the ambiguity.

So far, experience with the type algorithm shows that the amount of type information needed in an Ampersand script is reasonable in the eyes of users. In most cases where the type of a term can obviously be deduced by a user, the type system infers that type. There are limitations, of course. The constraint that every term should get a type is restrictive in the sense that some scripts will not be admissible. This often is desired behavior, as is the case in Listing 1.2. In such cases, the Ampersand compiler produces an error message (type error) to help the user identify and fix the mistake.

A unique property of this type system is that the order of subtypes emerges from the script itself. In practice, however, this makes debugging an Ampersand script difficult, even when the user is presented with the partial order on concepts. Suppose for example that the user specifies:

$$\begin{aligned}&\text{RULE } r \cap \mathbb{V}_{B\times A} = r\\ &\text{RULE } r; r \cap \mathbb{I}_A = \mathbb{I}_A\end{aligned}$$

The first line in this script says that r is of type $\langle B, A\rangle$ (or, to be precise, of a type $\langle B', A'\rangle$ with $B' \preceq B$ and $A' \preceq A$). The last line can be thought of as type incorrect: the user may have intended $\mathbb{I}_A \subseteq r^{\smile}; r$. Given the current script, however, we can derive (and our algorithm derives):

$$\begin{aligned}\mathfrak{I}(A) &= \mathfrak{I}(dom(A))\\ &= \mathfrak{I}(dom(r; r \cap A))\\ &\subseteq \mathfrak{I}(dom(r))\\ &= \mathfrak{I}(dom(r \cap \mathbb{V}_{B\times A}))\\ &\subseteq \mathfrak{I}(dom(\mathbb{V}_{B\times A}))\\ &\subseteq \mathfrak{I}(B)\end{aligned}$$

This implies that $A \preceq B$, so no error message is produced. To make matters worse, even when $\mathbb{I}_B \cap \mathbb{I}_A = \mathbb{I}_B$ is added to intentionally imply $B \preceq A$, a type error does occur, saying that $A = B$ could be derived. At no point is the user alerted to the fact that $r; r$ contains an error at the ;. This shows that when errors are produced, they may not point to the problem. In a practical setting in

industry, this leads to unacceptably confusing errors for users. Suppose, for example, that the user makes the mistake of omitting a converse operator ($(\cdot)^{\smile}$). The consequence is that the compiler checks whether the type *Person* corresponds to *Account* (which is in the eyes of the user obviously not the case). The type checker finds an interpretation in which $Person \preceq Account$. The resulting error messages, if any, will be unintelligble in the eyes of the user. As a result, the type algorithm proposed here was *not* adopted in the Ampersand compiler.

9 Conclusion

This paper shows that domain analysis can be used as a mechanism for type checking. The type algorithm presented is unconventional: it does not work with proof trees. This results in an attractive type graph, which is useful for explaining the type-correctness or type-incorrectness of a script. The type graph is a comprehensive representation of all proofs that are needed to show that every term has precisely one type. Because of this, the diagram has demonstrated to be quite convincing in debates among Ampersand professionals in practice. The approach is not only simple, but it allows compilers with advanced features such as specialization and overloading with hardly any extra effort. The simplicity of the approach is illustrated by the haskell code corresponding to this article, which contains the complete algorithm in under 200 lines of Haskell code, and can be found at: `http://cs.ru.nl/~B.Joosten/ampTypes/` The simplicity is also valued in the Ampersand project, because this yields code that is maintainable because of its simplicity.

This type algorithm adds a new feature to type checking: it uses information from the very rules it is checking to enhance the concept pre-order.

10 Acknowledgements

We thank the reviewers for their comments. A special thanks to the reviewer who tried our code on the examples in this paper.

References

1. The Business Rules Group: Business rules manifesto – the principles of rule independence. available from http://www.BusinessRulesGroup.org (January 2003)
2. Michels, G., Joosten, S., van der Woude, J., Joosten, S.: Ampersand: Applying relation algebra in practice. In: Proceedings of the 12th conference on Relational and Algebraic Methods in Computer Science RAMICS'11. Lecture Notes in Computer Science 6663, Berlin, Springer-Verlag (2011) 280–293
3. Maddux, R.: Relation Algebras. Volume 150 of Studies in Logic and the Foundations of Mathematics. Elsevier Science (2006)
4. van der Woude, J., Joosten, S.: Relational heterogeneity relaxed by subtyping. In: Proceedings of the 12th conference on Relational and Algebraic Methods in Computer Science. Lecture Notes in Computer Science 6663, Berlin, Springer-Verlag (2011) 347–361

5. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences **17** (1978) 348–375
6. Marlow, S., Wadler, P.: A practical subtyping system for erlang. SIGPLAN Not. **32**(8) (August 1997) 136–149
7. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems (TOPLAS) **16**(6) (1994) 1811–1841
8. Mitchell, J.C.: Type inference with simple subtypes. Journal of functional programming **1**(03) (1991) 245–285
9. Amadio, R.M., Cardelli, L.: Subtyping recursive types. ACM Transactions on Programming Languages and Systems (TOPLAS) **15**(4) (1993) 575–631
10. Tiuryn, J., Urzyczyn, P.: The subtyping problem for second-order types is undecidable. In: Logic in Computer Science, Symposium on, IEEE Computer Society (1996) 74–74
11. Wells, J.B.: The undecidability of mitchell's subtyping relationship. Technical report, Boston University Computer Science Department (1995)
12. Lindahl, T., Sagonas, K.: Practical type inference based on success typings. In: Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming, ACM (2006) 167–178
13. Tomás, D., Vicedo, J.L.: Minimally supervised question classification on fine-grained taxonomies. Knowledge and information systems **36**(2) (2013) 303–334