Dot-types and Their Implementation

Tao Xue and Zhaohui Luo*

taoxue@cs.rhul.ac.uk
zhaohui.luo@hotmail.co.uk

Abstract. Dot-types, as proposed by Pustejovsky and studied by many others, are special data types useful in formal semantics to describe interesting linguistic phenomena such as copredication. In this paper, we present an implementation of dot-types in the proof assistant Plastic base on their formalization in modern type theories.

1 Introduction

Dot-types, or sometimes called dot objects or complex types, were introduced by Pustejovsky in the Generative Lexicon Theory [19] and studied by many others, including [3]. Intuitively, a dot-type is formed from two constituent types that present distinct aspects of those objects in the dot-type. For example, a book may be considered to have two aspects: one informational (eg, when it is read) and the other physical (eg, when it is picked up). One may therefore consider a dot-type PHY • INFO whose objects, including books, have both physical and informational aspects. In particular, such objects can be involved in the linguistic phenomenon of copredication and dot-types play a promising role in its analysis and formalisation.

Although the meaning of dot-types is intuitively clear, its proper formal account seems surprisingly difficult and tricky (see [2] for a discussion). Researchers have made several proposals to model dot-types formally including, for example, [3, 4] and [6, 7]. Besides discussions on whether the proposed solutions do capture and therefore give successful formal accounts of dot-types, most of these proposals are considered in the Montagovian setting. In [13], the second author has proposed a formal treatment of dot-types in modern type theories[1] (MTTs) with the help of coercive subtyping and it is argued that, because in the formal semantics based on MTTs common nouns are interpreted as types (rather than predicates as in the Montague semantics), the linguistic phenomena such as copredication can be given satisfactory treatments by means of dot-types.

In this paper, we present an implementation of dot-types in the proof assistant Plastic [5], based on the formalisation of dot-types in MTTs. As far as we

* Partially supported by the research grant F/07-537/AJ of the Leverhulme Trust in U.K.

[1] Modern type theories may be classified into the predicative type theories such as Martin-Löf's type theory [18, 16] and the impredicative type theories such as the Calculus of Constructions (CC) [9] and the Unifying Theory of dependent Types (UTT) [11].

know, this is the first attempt to implement the dot-types.[2] It allows us to use dot-types in the development of formal semantics in proof assistants and, at the same time, gives us a better understanding of dot-types and their relationship with other data types in type theory.

Dot-types are not ordinary inductive types, as found in the MTT-based proof assistants such as Agda [1], Coq and Plastic. In particular, for $A \bullet B$ to be a dot-type, the constituent types A and B should not share components (see the main text for the formal definition). In an implementation of dot-types, this special condition of type formation must be checked and adhered to. In order to make sure of this, we have to implement the dot-types as special data types, different from ordinary inductive types. We shall show how this is done in our implementation in Plastic.

Dot-types are introduced informally in section 2, where we focus on the idea that the constituent types of a dot-type do not share common components. In section 3, we discuss the formal formulation of dot-types in MTTs. The implementation of dot-types are presented in section 4, where we will first briefly introduce the proof assistant Plastic and then explain how to implement dot-types in Plastic with several examples to illustrate the use.

2 Dot-types in Formal Semantics: an Introduction

In the Generative Lexicon Theory [19], Pustejovsky has introduced the idea of employing dot-types to model various linguistic data that involve objects that have distinct aspects. Typical examples are concerned about copredication, where different aspects of a word are selected when predication comes into force. For example, in the following sentences [3], the words 'lunch' and 'book' both have two distinct aspects to be selected: in (1) a 'lunch' was delicious as food and took forever as an event, and in (2) a 'book' was picked up as a physical object and mastered as an informational object.

(1) The lunch yesterday was delicious but took forever.

(2) John picked up and mastered the mathematical book.

There have been studies of dot-types in various formal systems or semi-formal systems including, for example, [19, 4, 3, 21]. Most of these proposals are given in the Montagovian setting where, in particular, common nouns are interpreted as predicates. It has been argued in [13] that the way that CNs are interpreted in the Montagovian setting is incompatible with the subtyping postulates that the type of entities has subtypes *Event* (of events), PHY (of physical objects), INFO (of informational objects), etc. This leads to unnecessary difficulties and formal

[2] In [14] the second author has presented some examples in Coq [8] and, since Coq does not support dot-types, he had to use Σ-types to mimic dot-types, although fully aware of the fact that this is in general impossible and leads to incoherence, since there is no guarantee that the constituent types of a dot-type do not share components.

complications when formalising dot-types. On the other hand, if we interpret CNs as types, as in the formal semantics based on MTTs, the treatment becomes straightforward and satisfactory [13]. (See section 3.1 for further details.)

Usually the two aspects involved in a dot-type are incompatible: in the above examples, Food and Event are incompatible and so are the physical and informational objects. This incompatibility of the two aspects that form a dot-type was expressed by Pustejovsky as follows:

> *Dot objects have a property that I will refer to as* inherent polysemy. *This is the ability to appear in selectional contexts that are contradictory in type specification.* [20]

In other words, an important feature is that, to form a dot-type $A \bullet B$, its constituent types A and B should not share common parts. For instance,

- PHY $\bullet$ PHY should not be a dot-type because its constituent types are the same type PHY.
- PHY $\bullet$ (PHY $\bullet$ INFO) should not be a dot-type because its constituent types PHY and PHY $\bullet$ INFO share the component PHY.

Put in another way, a dot-type $A \bullet B$ can only be formed if the types A and B do not share any components: it is a dot-type only when the constituent types A and B present different and incompatible aspects of the objects.

This incompatibility is one of the two key features based on which dot-types are introduced in MTTs [13]: it is stipulated that the constituent types of a dot-type do not share components. The other feature is that the relationships between the dot-type and its constituent types are captured by means of coercive subtyping so that the dot-type is the subtype of both of its constituent types. We now turn to the type-theoretic formulation of dot-types.

3 Dot-types in Modern Type Theories

In this section, we show how dot-types can be introduced in modern type theories with the help of coercive subtyping [13]. We will first explain informally, in the formal semantics based on MTTs, called *type-theoretical semantics* henceforth. how to use dot-types to interpret copredication in natural language. Then we will lay down the formal rules of the dot-types in modern type theories.

3.1 Dot-types and Coercive Subtyping

Type-Theoretical Semantics. In [22] Ranta has studied various semantic issues of natural languages in Martin-Löf's type theory, introducing the basic ideas of type-theoretical semantics based on MTTs. Unlike Montague grammar in which common nouns like Man and $Human$ are interpreted as functional subsets (or predicates) of entities, in the type-theoretical semantics based on modern type theories, common nouns are interpreted as types. For instance, in

Montague grammar, Man and $Human$ are interpreted as objects of type $e \to t$, where e is the type of entities and t the type of propositions. In type-theoretical semantics instance, the interpretations of Man, $Human$ and $Book$ are types:

$$[[man]], [[human]], [[book]] : Type$$

This is natural in a modern type theory, which is many-sorted in the sense that there are many types like $[[man]]$ and $[[book]]$ consisting of objects standing for different sorts of entities, while the simple type theory may be thought of as single sorted in the sense that there is the type e of all entities. In a type-theoretical semantics, verbs and adjectives are interpreted as predicates. For example, we can have

$$[[nice]] : [[book]] \to Prop$$
$$[[read]] : [[human]] \to [[book]] \to Prop$$

where $Prop$ is the type of propositions.

Let's we consider a kind of dependent type called Σ-types. It basically means that if A is a type and B is an A-indexed family of types, then $\Sigma(A, B)$, or sometimes written as $\Sigma x{:}A.B(x)$, is a type, consisting of pairs (a, b) such that a is of type A and b is of type $B(a)$.

Modified common noun phrases could be interpreted by means of Σ-types: for instance,

$$[[nice\ book]] = \Sigma([[book]], [[nice]])$$

Coercive subtyping. Coercive subtyping was introduced in [12]. The basic idea of coercive subtyping is to consider subtyping as an abbreviation mechanism: A is a subtype of B ($A <_c B$), if there is a unique implicit coercion c from type A to type B. If so, an object a of type A can be used in any context that expects an object of type B, and it is equal to $c(a)$. The formal coercive definition rule is defined as:

$$\frac{\Gamma \vdash f : B \to C \quad \Gamma \vdash a : A \quad \Gamma \vdash A <_c B : Type}{\Gamma \vdash f(a) = f(c(a)) : C}$$

For instance, one may consider the type of men to be a subtype of the type of human beings by declaring a coercion between them: $[[man]] <_m [[human]]$. If we assume that $walk$ be interpreted as $[[walk]] : [[human]] \to Prop$ and $Jack$ as $[[Jack]] : [[man]]$, we could interpret the sentence (3) as (4).

(3) Jack walks.

(4) $[[walk]]([[Jack]])$

The reason that (4) is well-typed is that $[[man]]$ is now a subtype of $[[human]]$, an appropriate coercion can be inserted to fill up the gap in the term in (4).

Notation We shall adopt the following notational abbreviations, writing

- $A < B$ for $A <_c B$: **Type** for some c,
- $A \leq B$ for $A = B$: **Type** or $A < B$.

Dot-type and coercive subtyping. Intuitively, a dot-type should be a subtype of its constituent types. For instance, it is natural to think that the type consisting of the objects with both aspects of food and event be a subtype of Food as well as a subtype of Event. Similarly, the type consisting of objects with both physical and informational aspects should be a subtype of the type PHY of physical objects and a subtype of the type of informational aspect:

$$\textsc{Phy} \bullet \textsc{Info} <_{c_1} \textsc{Phy}$$

$$\textsc{Phy} \bullet \textsc{Info} <_{c_2} \textsc{Info}$$

Consider sentence (2) again. In a type-theoretical semantics, we may assume that

$$\begin{aligned} [[book]] &< \textsc{Phy} \bullet \textsc{Info} \\ [[pick\ up]] &: [[human]] \to \textsc{Phy} \to Prop \\ [[master]] &: [[human]] \to \textsc{Info} \to Prop \end{aligned}$$

Because of the above subtyping relationship (and contravariance of subtyping for the function types), we have

$$\begin{aligned} [[pick\ up]] &: [[human]] \to \textsc{Phy} \to Prop \\ &< [[human]] \to \textsc{Phy} \bullet \textsc{Info} \to Prop \\ &< [[human]] \to [[book]] \to Prop \end{aligned}$$

$$\begin{aligned} [[master]] &: [[human]] \to \textsc{Info} \to Prop \\ &< [[human]] \to \textsc{Phy} \bullet \textsc{Info} \to Prop \\ &< [[human]] \to [[book]] \to Prop \end{aligned}$$

Therefore, $[[pick\ up]]$ and $[[master]]$ can both be used in a context where terms of type $[[human]] \to [[book]] \to Prop$ are required and the interpretation of the sentence (2) can proceed as intended.

However, as we mentioned above, there are some difficulties if we do the same thing in Montagovian settings, we can show it with a simple case. Take the example of "heavy book", in Montague semantics, we should have

$$\begin{aligned} [[heavy]] &: (\textsc{Phy} \to t) \to (\textsc{Phy} \to t) \\ [[book]] &: \textsc{Phy} \bullet \textsc{Info} \to t \end{aligned}$$

In order to interpret "heavy book" as [[heavy]]([[book]]), we need

$$\textsc{Phy} \bullet \textsc{Info} \to t < \textsc{Phy} \to t$$

By contravariance, we need

$$\textsc{Phy} < \textsc{Phy} \bullet \textsc{Info}$$

But this is not the case, the subtype relation is actually in another way around.

3.2 Dot-types in Type Theory: a Formal Formulation

In the following, we present a type-theoretic treatment of dot-types with the help of coercive subtyping. There are two important ingredients in this type-theoretic definition:

1. The constituent types of a dot-type should not share common components.
2. A dot-type, if well-formed, should be a subtype of both of its constituent types.

Because of (1), the first and the most important thing is to define the notion of component and, when doing this, because of (2), the set of components of a type should contain those of all of its constituents and super-types. This is formally given by means of the following definition.

Definition 1 (component) *Let* T:**Type** *be a type in the empty context. Then,* $\mathscr{C}(T)$*, the set of components of* T*, is defined according to the normal form* [3] *of* T *as :*

$$\mathscr{C}(T) =_{def} \begin{cases} SUP(T) & \text{if the normal form of } T \text{ is not of the form } X \bullet Y \\ \mathscr{C}(T_1) \cup \mathscr{C}(T_2) & \text{if the normal form of } T \text{ is } T_1 \bullet T_2 \end{cases}$$

where $SUP(T) = \{T' | T \leq T'\}$.

Now, we give the formal rules for the dot-types in Figure 1 [4]. Note that, in the formation rule, we require that the constituent types do not share common components:

$$\mathscr{C}(A) \cap \mathscr{C}(B) = \emptyset$$

According to the rules in Figure 1, $A \bullet B$ is a subtype of A and a subtype of B. In other words, an object of the dot-type $A \bullet B$ can be regarded as an object of type A, in a context requiring an object of A, and can also be regarded as an object of type B in a context requiring an object of B.

Finally, the subtyping relations are propagated through the dot-types, by means of the coercions d_1, d_2 and d as specified in the last three rules in Figure 1.

Remark 1. It is worth to point out that, under the definition of component and rules of dot-types, there is no "universal supertype" of all types, and that dot-types are always between incompatible "maximal types".

[3] Intuitively, in a modern type theory with strong normalization and Church-Rosser properties, every process of computation starting from a well-typed term terminates and it computes to a (unique) normal form.

[4] In the formation rule of Figure 1, $\Gamma \vdash$ **valid** means that Γ is a valid context.

Formation Rule

$$\frac{\Gamma \vdash \textbf{valid} \quad \langle\rangle \vdash A : \textbf{Type} \quad \langle\rangle \vdash B : \textbf{Type} \quad \mathscr{C}(A) \cap \mathscr{C}(B) = \emptyset}{\Gamma \vdash A \bullet B : \textbf{Type}}$$

Introduction Rule

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B \quad \Gamma \vdash A \bullet B : \textbf{Type}}{\Gamma \vdash \langle a, b\rangle : A \bullet B}$$

Elimination Rules

$$\frac{\Gamma \vdash c : A \bullet B}{\Gamma \vdash p_1(c) : A} \qquad \frac{\Gamma \vdash c : A \bullet B}{\Gamma \vdash p_2(c) : B}$$

Computation Rules

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B \quad \Gamma \vdash A \bullet B : \textbf{Type}}{\Gamma \vdash p_1(\langle a, b\rangle) = a : A} \qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B \quad \Gamma \vdash A \bullet B : \textbf{Type}}{\Gamma \vdash p_2(\langle a, b\rangle) = b : B}$$

Projections as Coercions

$$\frac{\Gamma \vdash A \bullet B : \textbf{Type}}{\Gamma \vdash A \bullet B <_{p_1} A : \textbf{Type}} \qquad \frac{\Gamma \vdash A \bullet B : \textbf{Type}}{\Gamma \vdash A \bullet B <_{p_2} B : \textbf{Type}}$$

Coercion Propagation

$$\frac{\Gamma \vdash A \bullet B : \textbf{Type} \quad \Gamma \vdash A' \bullet B' : \textbf{Type} \quad \Gamma \vdash A <_{c_1} A' : \textbf{Type} \quad \Gamma \vdash B = B' : \textbf{Type}}{\Gamma \vdash A \bullet B <_{d_1[c_1]} A' \bullet B' : \textbf{Type}}$$

where $d_1[c_1](\langle a, b\rangle) = \langle c_1(a), b\rangle$.

$$\frac{\Gamma \vdash A \bullet B : \textbf{Type} \quad \Gamma \vdash A' \bullet B' : \textbf{Type} \quad \Gamma \vdash A = A' : \textbf{Type} \quad \Gamma \vdash B <_{c_2} B' : \textbf{Type}}{\Gamma \vdash A \bullet B <_{d_2[c_2]} A' \bullet B' : \textbf{Type}}$$

where $d_2[c_2](\langle a, b\rangle) = \langle a, c_2(b)\rangle$.

$$\frac{\Gamma \vdash A \bullet B : \textbf{Type} \quad \Gamma \vdash A' \bullet B' : \textbf{Type} \quad \Gamma \vdash A <_{c_1} A' : \textbf{Type} \quad \Gamma \vdash B <_{c_2} B' : \textbf{Type}}{\Gamma \vdash A \bullet B <_{d[c_1,c_2]} A' \bullet B' : \textbf{Type}}$$

where $d[c_1, c_2](\langle a, b\rangle) = \langle c_1(a), c_2(b)\rangle$.

Fig. 1. The rules of Dot-type

Propagations. To explain the propagation rule, we can think of interpreting the phrase

pick up and read the book

Instead of simply considering book having physical and informational aspect, we might think book contains *readable* information, compared to *radio program* which does not have a *readable* informational aspect. So we could interpret

$$\begin{aligned} [[readable]] &: \textsc{Info} \to Prop \\ [[readable\ info]] &= \Sigma(\textsc{Info}, [[readable]]) \\ [[book]] &< \textsc{Phy} \bullet [[readable\ info]] \end{aligned}$$

With the coercion relation we have for Σ-types [15],

$$\Sigma(\textsc{Info}, [[readable]]) < \textsc{Info}$$

we have $[[readable\ info]] < \textsc{Info}$ and trivially we have $\textsc{Phy} = \textsc{Phy}$. So we could get the following through propagation rule

$$[[book]] < \textsc{Phy} \bullet [[readable\ info]] < \textsc{Phy} \bullet \textsc{Info}$$

It is compatible with the example we've explained above.

Coherence. When we consider type theory subtyping relation, coherence [12] is the most important property that is required to hold. Informally, coherence means that coercion between any two types is unique. Put in another way, if there are two coercions c and c' from type A to type B, c and c' are required to be equal. Actually, this property makes the subtyping relation like a forest. One may intuitively understand the importance of this property like this: it guarantees that there's no ambiguity when we use a coercion.

Since the constituent types of a well-formed dot-type do not share components, it is straightforward to prove the following coherence property.

Proposition 2 *(coherence) The coercions p_1, p_2, d_1, d_2 and d are coherent together.*

Note that coherence is important as it guarantees the correctness of employing the projections p_1 and p_2 and the propagation operator d as coercions, and hence the subtyping relationships $A \bullet B <_{p_1} A$ and $A \bullet B <_{p_2} B$.

If the constituent types of a dot-type shared a common component, coherence would fail, like in product type. For instance, A and $A \bullet B$ share the component A. If $A \bullet (A \bullet B)$ were a dot-type, with the transitivity rule [5] there would be the following two coercions p_1 and $p_2 \circ p_1$:

$$A \bullet (A \bullet B) <_{p_1} A$$

[5] Transitivity for the coercion means that, if we have two coercions $A <_{c_1} B$ and $B <_{c_2} C$, then there's coercion from A to $C(A <_{c_2 \circ c_1} C)$ where $c_2 \circ c_1 \equiv [x{:}A](c_2(c_1(x)))$ is the composition of c_1 and c_2 .

$$A \bullet (A \bullet B) <_{p_2 \circ p_1} A$$

which are between the same types but not equal, coherence would then fail.

One may find that, when a dot-type $A \bullet B$ is well-formed, its behavior is similar to that of a product type $A \times B$: intuitively, its objects are pairs and the projections p_1 and p_2 correspond to the projection operations π_1 and π_2 in the product type, respectively. However, there are two important differences between dot-types and product types:

1. The constituent types of a dot-type do not share components, while in a product type the constituent parts can possibly share component. For instance, $A \times A$ is a well-formed product type, but $A \bullet A$ is not a well-formed dot-type.
2. It is fine for both of the projections p_1 and p_2 for dot-types to be coercions (Proposition 2), but for product types, only one of them can be coercions for, otherwise, coherence would fail [10].

4 Implementation

We have shown how to formalize the dot-types in type theory in the last section. As we have proof assistants which have implemented various data types, we would also like to put dot-types into proof assistant. However unlike inductive types such as the product types or Σ-types which could be defined with inductive schemata, dot-type cannot be simply be defined in a library of the proof assistant. The main reason is that we need to check whether the constituents of the dot-type share components. Especially in our definition of component, we need to check all the coercion relations of the term and its constituents in the context. This is not covered by existing approaches to define inductive types in the libraries of proof assistants. So we have to proceed in a hard way: defining dot-types directly in a proof assistant.

In this section, we present how we define dot-types in the proof assistant Plastic, and show how to use it.

4.1 Proof Assistants and Plastic

A proof assistant is a piece of software to assist with the formal proofs in a man-machine interactive way, based on constructive mathematical proofs. One can define mathematical problems in the provided formal language, choose the right strategy or algorithms in the library to achieve the proof. Modern type theories have been implemented in the proof assistants such as Agda [1], Coq [8], Matita [17] and Plastic [5] used in applications to formalization of mathematics and verification of programs.

Plastic is an implementation of LF presented in chapter 9 of [11] with extension for inductive families, universes. In the library of the proof assistant, we have various predefined inductive types and an second order logic. We also have implemented coercive subtyping in Plastic.

With the help of proof assistants like Plastic, we can also implement the formal semantics, which would help us to study the type-theoretical semantics for linguistic issue.

Here, we do not explain the technical details of using Plastic. Instead, we present a very simple example to show how we use Plastic to develop formal semantics.

Example 3 *Consider the example (3) "Jack walks" in section 2. Its semantics (4) can be presented by the last line of the following code in Plastic:*

```
> [Man, Human :Type];
> [c : Man -> Human];
> Coercion = c;
> [Jack : Man];
> [walk : Human -> Prop];
> [Jack_walks = walk(Jack)];
```

In the code, "Coercion = c" makes the function c as a coercion from Man to $Human$. With the help of coercion, the term $Jack\_walks$ is well typed: $Jack\_walks = walk(Jack) = walk(c(Jack)) : Prop$.

4.2 Dot-types in Plastic

As explained above, the dot-types have to be directly implemented in Plastic and, at the same time, the associated subtyping relations have to be specified.

- In the syntax of Plastic, we use $A * B$ to present the dot-type $A \bullet B$ and $dot < a, b >$ to present dot term $< a, b >$.
- When we declare a new dot-type $A \bullet B$, or a dot term $< a, b >$ where a:A and b:B, we will first check whether it is a proper dot-type. If $\mathcal{C}(A) \cap \mathcal{C}(B) = \emptyset$, $A \bullet B$ will be a legal dot-type or $< a, b >$ will be a legal dot term; otherwise, they will be rejected and an error message '*dot-type should not share component*' will be shown.
- Once a dot-type $A \bullet B$ or dot term $< a, b >$ is considered to be well-formed (legal), we will consider the coercions generated from the dot-type $A \bullet B$. We will add $[x{:}A \bullet B]p_1(x)$ and $[x{:}A \bullet B]p_2(x)$ as coercions from $A \bullet B$ to A and B [6].
- Furthermore, we will check the existing coercions of other dot-types to see whether there are cases for coercion propagation, if so, Plastic will add the new coercion generated by the coercion propagation into context.

In the implementation, we define some reductions for the computation rules for the projections p_1 and p_2, and the propagation operators d, d_1, and d_2.

[6] The system will automatically assign new metavariable names $cx1$, $cx2$, ... to the new introduced coercions by the dot-type rules

Assume A, B, C, D:$Type$, a:A, b:B, $A <_{c_1} C$, $B <_{c_2} D$, we have:

$$p_1(< a, b >) \rhd a$$
$$p_2(< a, b >) \rhd b$$
$$d[c_1, c_2](< a, b >) \rhd < c_1(a), c_2(b) >$$
$$d_1[c_1](< a, b >) \rhd < c_1(a), b >$$
$$d_2[c_2](< a, b >) \rhd < a, c_2(b) >$$

In the following, we present three main algorithms in our implementation. First we need to give an algorithm to calculate the components of a type.

Algorithm 4 *(checking components) Given a type A, we will calculate the component of A, $\mathscr{C}(A)$, in the following way.*

1. *Check the form of A to see whether it is a dot-type or not.*
2. *If A is not a dot-type, check all the coercion relations in the context to find out every type T which satisfies $A <_c T$ with some coercion c. $\mathscr{C}(A)$ is the set of all these super type T.*
3. *If A is a dot-type of the form $T_1 \bullet T_2$, $\mathscr{C}(A) = \mathscr{C}(T_1) \cup \mathscr{C}(T_2)$. (The algorithm is called recursively.)*

The second algorithm deals with the introduction of dot-types.

Algorithm 5 *When defining a type to be a dot-type $A \bullet B$:*

1. *Check the context to see whether A and B are already defined types. If so, go to next step; otherwise alert the type is not defined and end the algorithm.*
2. *Calculate $\mathscr{C}(A)$ and $\mathscr{C}(B)$ to see whether the intersection of these two is empty. If so go to the next step; if not, alert that dot-type do not share component and end the algorithm.*
3. *Check the existing coercions, to see whether $A \bullet B$ has already been considered. If so, simply finish the algorithm; otherwise go to the next step.*
4. *Add coercion from $A \bullet B$ to A and from $A \bullet B$ to B into the context, add the coercions generated by transitivity as well.*
5. *Check the existing coercion of dot-type in the context to add coercions introduced by propagation rules. For every existing coercion dot-type $C \bullet D$,*
   - *if there's a coercion c_1 from A to C, and a coercion c_2 from B to D, add a new coercion $[x{:}A \bullet B]d[c_1, c_2](x)$ from $A \bullet B$ to $C \bullet D$.*
   - *if there's a coercion c_1 from A to C, and B equals to D, add a new coercion $[x{:}A \bullet B]d_1[c_1](x)$ from $A \bullet B$ to $C \bullet D$.*
   - *if there's a coercion c_2 from B to D, and A equals to C, add a new coercion $[x{:}A \bullet B]d_2[c_2](x)$*
   - *otherwise, do nothing.*
6. *Check the transitivity possibilities of the new generated coercion.*

The third algorithm deals with the introduction of dot-terms (similar to that for dot-type introduction).

Algorithm 6 *When defining a term to be a dot term* $< a, b >$*:*

1. *Check the context to see whether* a *and* b *are defined terms. If so take the types of* a *and* b*, let say* A *and* B*, and go to the next step; otherwise alert the term is not defined and end the algorithm.*
2. *Calculate* $\mathscr{C}(A)$ *and* $\mathscr{C}(B)$ *to see whether the intersection of these two is empty. If so go to the next step; if not, alert that dot-type do not share component and end the algorithm.*
3. *Check the existing coercions, to see whether* $A \bullet B$ *has already been considered. If so, simply finish the algorithm; otherwise go to the next step.*
4. *Add coercion from* $A \bullet B$ *to* A *and from* $A \bullet B$ *to* B *into the context, add the coercions generated by transitivity as well.*
5. *Check the existing coercion of dot-type in the context. For every existing coercion dot-type* $C \bullet D$*,*
   - *if there's a coercion* c_1 *from* A *to* C*, and a coercion* c_2 *from* B *to* D*, add a new coercion* $[x{:}A \bullet B]d[c_1, c_2](x)$ *from* $A \bullet B$ *to* $C \bullet D$*.*
   - *if there's a coercion* c_1 *from* A *to* C*, and* B *equals to* D*, add a new coercion* $[x{:}A \bullet B]d_1[c_1](x)$ *from* $A \bullet B$ *to* $C \bullet D$*.*
   - *if there's a coercion* c_2 *from* B *to* D*, and* A *equals to* C*, add a new coercion* $[x{:}A \bullet B]d_2[c_2](x)$
   - *otherwise, do nothing.*
6. *Check the transitivity possibilities of the new generated coercion.*

Another part we should take care of, is that, since we need to consider the propagation rules of dot-types, when we introduce a new coercion, it links two existing dot-types and generates a new coercion through the propagation rules. So when we introduce a new coercion, we should also check all the dot-types in the context to see whether there're types satisfy the conditions of propagation rule, and add corresponding coercions for the propagation rules.

4.3 Examples of Dot-types in Plastic

In this subsection, we will first give some abstract examples to show how to declare a dot-type in Plastic, what we will get from the declaration, and some examples of illegal declaration of dot-types. Then we will give a concrete case to interpret sentences in natural language into code in Plastic.

Example 7 *We can define a dot-type or a dot-term simply in the following way:*

1. *If we have two types* A*,* B *which do not share components, we could simply define a type* M *of type* $A * B$ *like this:*

```
> [M = A*B];
```

The system will generate two coercions $cx1$ from $A * B$ to A and $cx2$ from $A * B$ to B.

2. *We can also define a dot term . If we have two terms a, b, a:A and b:B, we can define a dot term $m =< a, b >$ like this:*

   ```
   > [m = dot<a,b>];
   ```

   Now m is defined to be a dot term $dot < a, b >$ and it is of type $A * B$. The system will generate two coercions $cx1$ from $A * B$ to A and $cx2$ from $A * B$ to B.

Example 8 *In the following examples, the types share components in different ways, none of them could be defined as a dot-type or dot term, alert will be shown in all the following cases.*

1. *The two parts are the same*

   ```
   > [M = A*A];
   ```

2. *$A * C$ and $A * B$ have the same component A*

   ```
   > [M = (A*C)*(A*B)];
   ```

3. *A is a subtype of B, as definition of component $\mathscr{C}(A) \cap \mathscr{C}(B) = \{A\}$*

   ```
   > [c:A->B];
   > Coercion = c;
   > [M = A*B];
   ```

4. *a and b are both of type A, but $A * A$ is not a legal dot-type, so $dot < a, b >$ is not a legal dot term.*

   ```
   > [a,b:A];
   > [ab = dot<a,b>];
   ```

5. *a is of type A and b is of type B, while A is a subtype of B. As shown above, $A * B$ is not a legal dot-type, hence $dot < a, b >$ is not a legal dot term.*

   ```
   > [a:A];
   > [b:B];
   > [c:A->B];
   > Coercion = c;
   > [ab = dot<a,b>];
   ```

Example 9 *When we have dot-type $A * B$, $A <_{c1} C$ and $B <_{c2} D$, if we claim $C * D$ to be another dot-type, coercions from the propagation rule will also be added.*

```
> [c1:A->C];
> [c2:B->D];
> Coercion = c1;
> Coercion = c2;
> [M1 = A*B];
> [M2 = C*D];
```

In this example several coercions will be added according to the dot-type rule and transitivity. $cx1$ *from* $A*B$ *to* A, $cx2$ *from* $A*B$ *to* B, $< c1, cx1 >= [x{:}(A*B)]cx3(cx1\ x)$ *by transitivity from* $A*B$ *to* C, $< c2, cx1 >= [x{:}(A*B)]cx4(cx1\ x)$ *by transitivity from* $A*B$ *to* D, $cx3$ *from* $C*D$ *to* C *and* $cx4$ *from* $C*D$ *to* D. *However, we will get one more coercion from the propagation rule, there will be a coercion* $cx5$ *from* $A*B$ *to* $C*D$ *and* $cx5 = [x{:}A*B]d[c1, c2](x)$, *where for any dot term* $dot < a, b >$ *of dot-type* $A*B$, $d[c1, c2]dot < a, b >= dot < c1(a), c2(b) >$.

Now, let's use a concrete example to show how we could interpret natural language in Plastic:

Example 10 *Let's consider the sentence*

John picked up and mastered a book

We should contain the following data:

```
> [PhyInfo = Phy*Info];
> [cb : Book -> PhyInfo];
> Coercion = cb;
> [John:Human];
> [book:Book];
```

The verbs 'picked up' and 'mastered' are of the following types, where "==>" is Plastic notation for function type :

```
> [pickup : Human ==> (Phy ==> Prop)];
> [master : Human ==> (Info ==> Prop)];
```

So we could interpret "John picked up 'book' " and "John mastered 'book' "s separately, and use the predefined connect operator and:$Prop- > Prop- > Prop$ *to connect them. However this hasn't presented our meaning, it is actually for the sentence:*

John picked up 'book' and John mastered 'book'.

We want 'and' to connect 'picked up' with 'mastered' , so we consider 'and' as a generic semantic kind: for any type A, [[AND]](A) is of kind $A \to A \to A$. *For A being* $Human ==> Book ==> Prop$,

$$And = [[AND]](Human ==> Book ==> Prop)$$

In particular, the term "And pickup master" is well-typed, thanks to the coercive subtyping relations, including the contravariance for subtyping function

types, as explained in section 2. However, this seems still not enough, book : Book refers to a special book, so this interpretation some how is for the sentence "John picked up and mastered 'book'. ", where 'book' is a special book. More naturally, "John picked up and mastered a book" should stands for "There's a book, John picked up and mastered it". So we need to use the Ex notation in Plastic for exists as well. The full code is in the appendix.

References

1. The Agda proof assistant. `http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php` (2008)
2. Asher, N.: A type driven theory of predication with complex types. Fundamenta Infor. 84(2) (2008)
3. Asher, N.: Lexical Meaning in Context: A Web of Words. Cambridge University Press (2011)
4. Asher, N., Pustejovsky, J.: Word meaning and commonsense metaphysics (2005)
5. Callaghan, P., Luo, Z.: An implementation of LF with coercive subtyping and universes. Journal of Automated Reasoning 27(1), 3–27 (2001)
6. Cooper, R.: Copredication, quantification and frames. Logical Aspects of Computational Linguistics (LACL'2011). LNAI 6736 (2011)
7. Cooper, R.: Copredication, dynamic generalized quantification and lexical innovation by coercion. Proceedings of GL2007, the Fourth International Workshop on Generative Approaches to the Lexicon (2007)
8. The Coq Development Team: The Coq Proof Assistant Reference Manual (Version 8.1), INRIA (2007)
9. Coquand, T., Huet, G.: The calculus of constructions. Infor. and Computation 76(2/3) (1988)
10. Luo, Y.: Coherence and Transitivity in Coercive Subtyping. Ph.D. thesis, University of Durham (2005)
11. Luo, Z.: Computation and Reasoning: A Type Theory for Computer Science. Oxford Univ Press (1994)
12. Luo, Z.: Coercive subtyping. J of Logic and Computation 9(1), 105–130 (1999)
13. Luo, Z.: Type-theoretical semantics with coercive subtyping. Semantics and Linguistic Theory 20 (SALT20), Vancouver (2010)
14. Luo, Z.: Contextual analysis of word meanings in type-theoretical semantics. Logical Aspects of Computational Linguistics (LACL'2011). LNAI 6736 (2011)
15. Luo, Z., Luo, Y.: Transitivity in coercive subtyping. Infor. and Computation 197(1-2) (2005)
16. Martin-Löf, P.: Intuitionistic Type Theory. Bibliopolis (1984)
17. The Matita proof assistant. `http://matita.cs.unibo.it/` (2008)
18. Nordström, B., Petersson, K., Smith, J.: Programming in Martin-Löf's Type Theory: An Introduction. Oxford University Press (1990)
19. Pustejovsky, J.: The Generative Lexicon. MIT (1995)
20. Pustejovsky, J.: A survey of dot objects. Manuscript (2005)
21. Pustejovsky, J.: Mechanisms of coercion in a general theory of selection (2011)
22. Ranta, A.: Type-Theoretical Grammar. Oxford University Press (1994)

A Example code in Plastic

```
> import Sol_All;
> import FnCoercion;
> import SigmaCoercion
> [Human, Phy, Info, Book :Type];
> [PhyInfo = Phy*Info];
> [cb:Book -> PhyInfo];
> Coercion = cb;
> [pickup:Human ==> (Phy ==> Prop)];
> [master:Human ==> (Info ==> Prop)];
> [John:Human];
> [book:Book];

("John picked up  'book' and John mastered  'book' ")

> [ pickup_book = (ap_ Book Prop (ap_ Human (Book==>Prop)  pickup John) book];
> [ master_book = (ap_ Book Prop (ap_ Human (Book==>Prop)  master John) book];
> [ sentence1 = and pickup_book master_book];


("John picked up and mastered 'book' ")

> [AND: (A:Type) (A->A->A)]
> [And = AND (Human ==> Book ==> Prop)];
> [sentence2 = ap_ Book Prop (ap_ Human (Book==>Prop)
                              (And pickup master) John) book];


("John picked up and mastered a book")

> [sentence3 = Ex Book [b:Book](ap_ Book Prop (ap_ Human (Book==>Prop)
                              (And pickup master) John) b)];
```