Dot-types and Their Implementation

Tao Xue and Zhaohui Luo

Department of Computer Science
Royal Holloway, University of London

July 2, 2012

Outline

Simple Type Theory and Modern Type Theory

Church's simple type theory – Montague Semantics:

Modern type theories (eg, Martin-Löf's type theory) – Type-Theoretical Semantics:

Simple Type Theory and Modern Type Theory

Church's simple type theory – Montague Semantics:

- Base types ("single-sorted"): e and t
- Composite types: $e \to t$, $(e \to t) \to t$,
- Formulas in HOL

Modern type theories (eg, Martin-Löf's type theory) – Type-Theoretical Semantics:

Simple Type Theory and Modern Type Theory

Church's simple type theory – Montague Semantics:

- Base types ("single-sorted"): e and t
- Composite types: $e \to t$, $(e \to t) \to t$,
- Formulas in HOL

Modern type theories (eg, Martin-Löf's type theory) –
Type-Theoretical Semantics:

- Many types of entities ("many-sorted")
  Table, *Man*, *Human*, *Phy*, are all types (of certain entities).
- Different MTTs have different embedded logics:
  - Martin-Löf's type theory: first-order logic.
  - Impredicative UTT: higher-order logic.

Montague Semantics

Montague Semantics (Montague 1974) is base on Church's simple type theory.

Montague Semantics

Montague Semantics (Montague 1974) is base on Church's simple type theory.

Common nouns (as functional subsets of entities)

- $[[man]], [[human]], [[book]] : e \rightarrow t$

Montague Semantics

Montague Semantics (Montague 1974) is base on Church's simple type theory.

Common nouns (as functional subsets of entities)

- $[[man]], [[human]], [[book]] : e \to t$

Verbs (as subset of entities)

- $[[walk]] : e \to t$
- $[[John\ walks]] = [[walk]](j),\ if\ j = [[John]] : e$

Montague Semantics

Montague Semantics (Montague 1974) is base on Church's simple type theory.

Common nouns (as functional subsets of entities)

- $[[man]], [[human]], [[book]] : e \to t$

Verbs (as subset of entities)

- $[[walk]] : e \to t$
- $[[John\ walks]] = [[walk]](j),\ if\ j = [[John]] : e$

Adjectives (as function from subsets to subsets)

- $[[nice]] : (e \to t) \to (e \to t)$
- $[[nice\ book]] = [[nice]]([[book]]) : e \to t$

Montague Semantics

Montague Semantics (Montague 1974) is base on Church's simple type theory.

Common nouns (as functional subsets of entities)

- $[[man]], [[human]], [[book]] : e \to t$

Verbs (as subset of entities)

- $[[walk]] : e \to t$
- $[[John\ walks]] = [[walk]](j),\ if\ j = [[John]] : e$

Adjectives (as function from subsets to subsets)

- $[[nice]] : (e \to t) \to (e \to t)$
- $[[nice\ book]] = [[nice]]([[book]]) : e \to t$

(or subset of entities)

- $[[nice]] : e \to t$

Type-Theoretical Semantics

Ranta (1994) studied various semantics issues of natural languages in Martin-Löf's type theory, introduced the basic ideas of type-theoretical semantics.

Type-Theoretical Semantics

Ranta (1994) studied various semantics issues of natural languages in Martin-Löf's type theory, introduced the basic ideas of type-theoretical semantics.

In the type-theoretical semantics, the common nouns are interpreted as *Type*, e.g. *Man*, *Human* and *Book* are types:

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

Type-Theoretical Semantics

Ranta (1994) studied various semantics issues of natural languages in Martin-Löf's type theory, introduced the basic ideas of type-theoretical semantics.

In the type-theoretical semantics, the common nouns are interpreted as *Type*, e.g. *Man*, *Human* and *Book* are types:

$$[[man]], [[human]], [[book]] : \mathit{Type}$$

verbs and adjectives are interpreted as predicates:

$$\begin{array}{lcl} [[nice]] & : & [[book]] \to \mathit{Prop} \\ [[read]] & : & [[human]] \to [[book]] \to \mathit{Prop} \end{array}$$

where *Prop* is the type of propositions.

Type-Theoretical Semantics

Let's consider a kind of dependent type called Σ-types.

$\Sigma x : A.B(x)$ is a type, if:

- A is a type.
- B is an A-indexed family of types.

$(a, b) : \Sigma x : A.B(x)$ such that $a : A$ and $b : B(a)$.

Type-Theoretical Semantics

Let's consider a kind of dependent type called Σ-types.

$\Sigma x : A.B(x)$ is a type, if:

- A is a type.
- B is an A-indexed family of types.

$(a, b) : \Sigma x : A.B(x)$ such that $a : A$ and $b : B(a)$.

$$[[\textit{nice book}]] = \Sigma x : [[\textit{book}]].[[\textit{nice}]](x)$$

Type-Theoretical Semantics

If we consider the following example:

John walks.

$$\begin{array}{lcl} [[walk]] & : & [[human]] \rightarrow Prop \\ [[John]] & : & [[man]] \end{array}$$

Type-Theoretical Semantics

If we consider the following example:

John walks.

$$\begin{array}{lcl} [[walk]] & : & [[human]] \rightarrow Prop \\ [[John]] & : & [[man]] \end{array}$$

How can we interpret "John walks":

$$??[[walk]]([[John]])??$$

Coercive Subtyping

Coercive subtyping is an abbreviation mechanism for subtype: A is a subtype of B ($A <_c B$), if there is a unique implicit coercion c from type A to type B.

Coercive Subtyping

Coercive subtyping is an abbreviation mechanism for subtype:
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 $c(a)$.

Coercive Subtyping

Coercive subtyping is an abbreviation mechanism for subtype:
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 $c(a)$.

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

Coercive Subtyping

Coercive subtyping is an abbreviation mechanism for subtype:
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 $c(a)$.

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

$$[[man]] <_c [[human]]$$

$$[[walk]]([[John]]) = [[walk]](c([[John]]))$$

Outline

Copredication

- The lunch yesterday was delicious but took forever.
- John picked up and mastered a book.

Copredication

- The lunch yesterday was delicious but took forever.
- John picked up and mastered a book.

lunch	*food*	*delicious*
	event	*forever*

Copredication

- The lunch yesterday was delicious but took forever.
- John picked up and mastered a book.

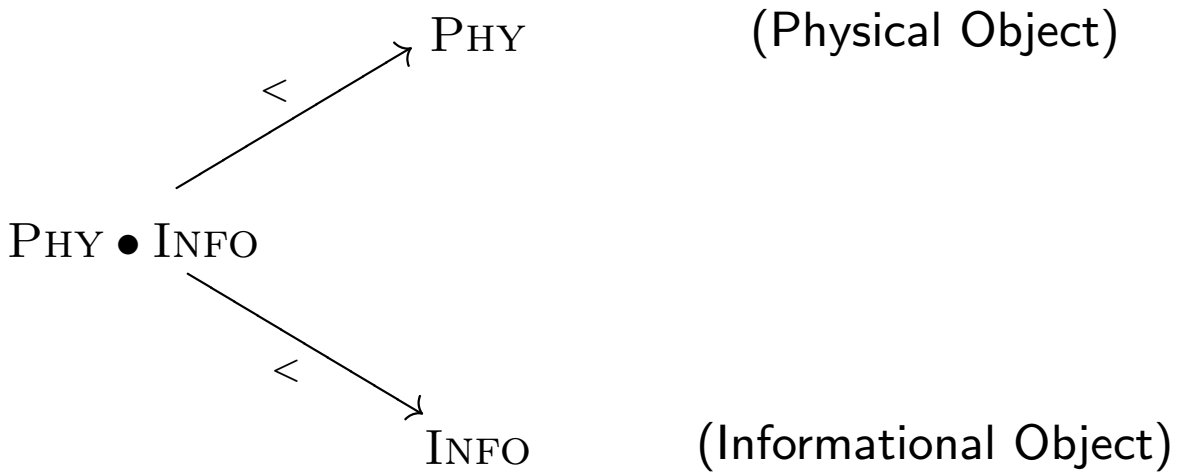
lunch	*food*	*delicious*
	event	*forever*
book	*physical object*	*pickup*
	informational object	*master*

Dot-types

Dot-types, or sometimes called dot objects or complex types, were introduced by Pustejovsky in the Generative Lexicon Theory (1995), and studied by many others, like Asher (2008, 2011), Luo (2010, 2011).

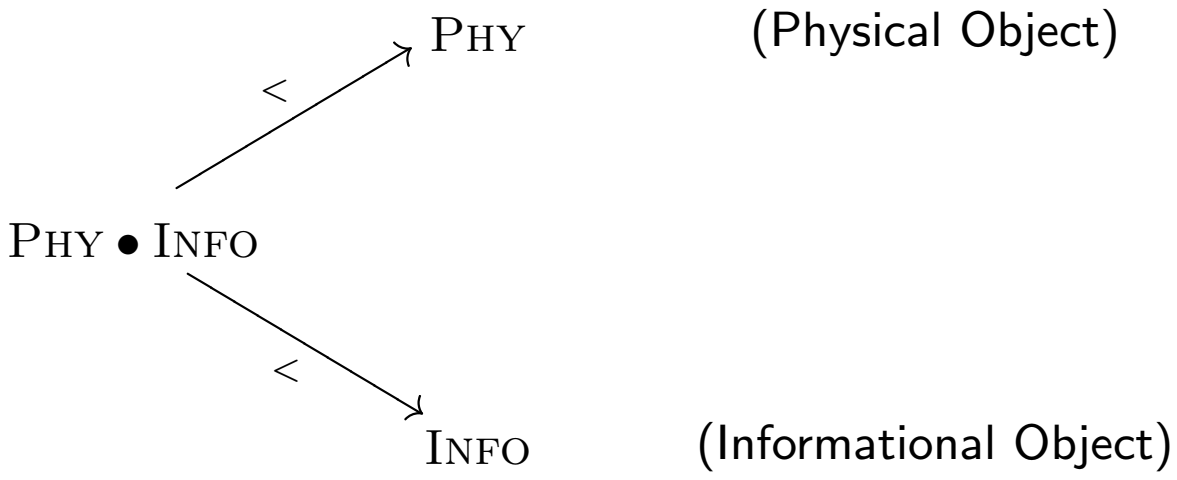
Dot-types

Dot-types, or sometimes called dot objects or complex types, were introduced by Pustejovsky in the Generative Lexicon Theory (1995), and studied by many others, like Asher (2008, 2011), Luo (2010, 2011).



Dot-types

Dot-types, or sometimes called dot objects or complex types, were introduced by Pustejovsky in the Generative Lexicon Theory (1995), and studied by many others, like Asher (2008, 2011), Luo (2010, 2011).



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

John picked up and mastered a book.

$$
\begin{array}{rcl}
[[\textit{book}]] & < & \textsc{Phy} \bullet \textsc{Info} \\
[[\textit{pick up}]] & : & [[\textit{human}]] \rightarrow \textsc{Phy} \rightarrow \textit{Prop} \\
[[\textit{master}]] & : & [[\textit{human}]] \rightarrow \textsc{Info} \rightarrow \textit{Prop}
\end{array}
$$

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

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

Therefore, [[*pick up*]] and [[*master*]] can both be used in a context where terms of type $[[\textit{human}]] \to [[\textit{book}]] \to \textit{Prop}$

$b : [[book]]$

$[[pick\ up]]\ [[John]]\ b$

$[[master]]\ [[John]]\ b$

Dot-types

$A \bullet B$ is a dot-type if A and B are types for different and incompatible aspects of the objects.
$\langle a, b \rangle : A \bullet B$ is a dot-term, if $a : A$, $b : B$ and $A \bullet B$ is a valid dot-type
$\langle a, b \rangle_{p1} = a : A \qquad \langle a, b \rangle_{p2} = b : B$

Dot-types

$A \bullet B$ is a dot-type if A and B are types for different and incompatible aspects of the objects.
$\langle a, b \rangle : A \bullet B$ is a dot-term, if $a : A$, $b : B$ and $A \bullet B$ is a valid dot-type
$\langle a, b \rangle_{p_1} = a : A \qquad \langle a, b \rangle_{p_2} = b : B$

Two important ingredients in type-theoretic definition of dot-types:

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

Definition of 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 form of T as :

$$\mathscr{C}(T) =_{def} \begin{cases} SUP(T) & \text{if } T \neq T_1 \bullet T_2 \\ \mathscr{C}(T_1) \cup \mathscr{C}(T_2) & T = T_1 \bullet T_2 \end{cases}$$

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

Rules of Dot-types 1

Formation Rule

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

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

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

Figure: The rules of Dot-type 1

Rules of Dot-types 2

Coercion Propagation

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

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

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

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

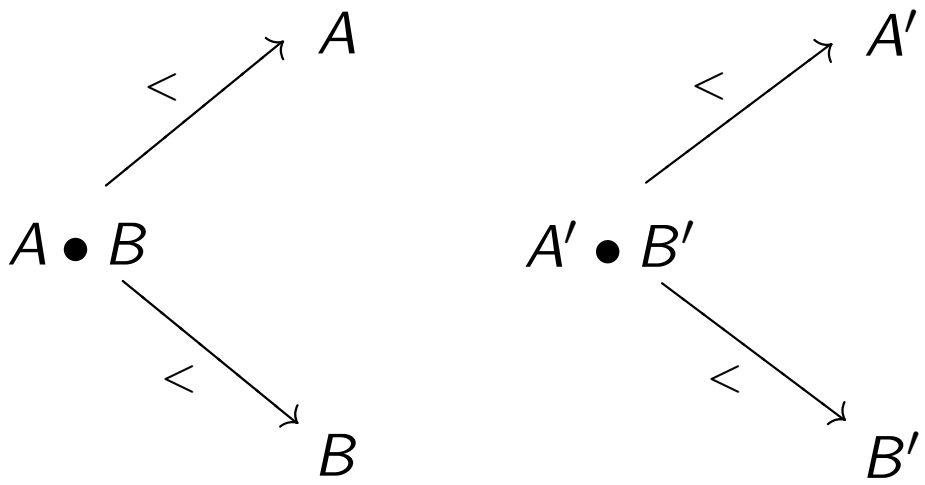
$$\frac{\Gamma \vdash A \bullet B : \mathbf{Type} \quad \Gamma \vdash A' \bullet B' : \mathbf{Type} \quad \Gamma \vdash A <_{c_1} A' : \mathbf{Type} \quad \Gamma \vdash B <_{c_2} B' : \mathbf{Type}}{\Gamma \vdash A \bullet B <_{d[c_1,c_2]} A' \bullet B' : \mathbf{Type}}$$

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

Figure: The rules of Dot-type 2

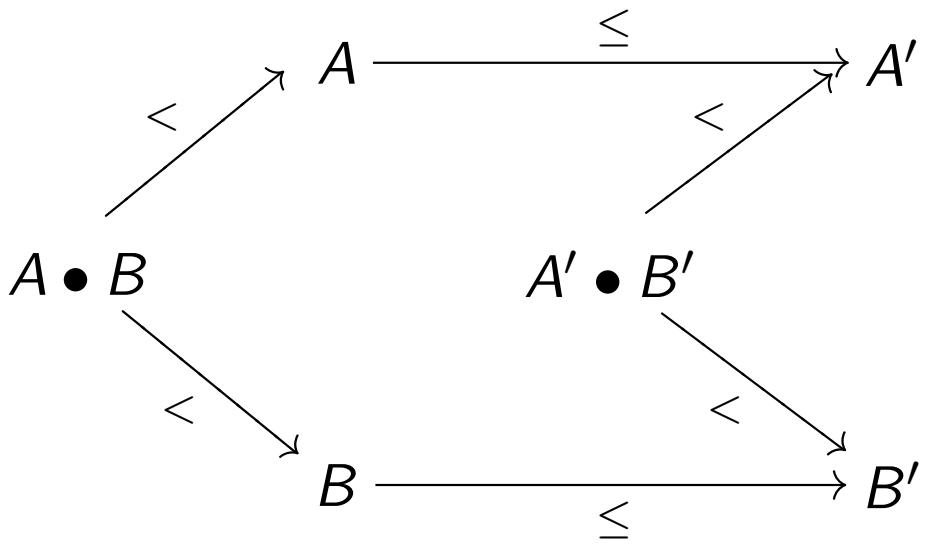
Propagations

Consider two dot-types $A \bullet B$ and $A' \bullet B'$



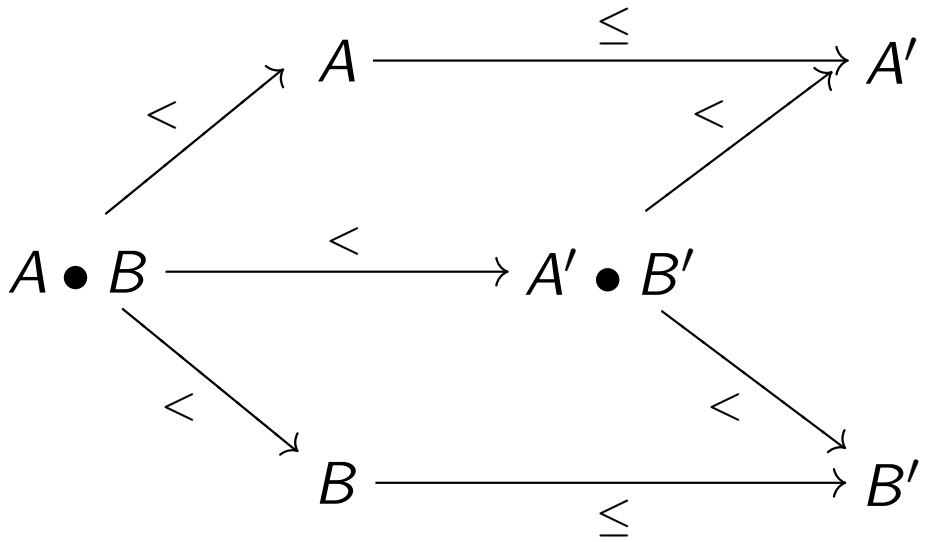
Propagations

If $A \leq A'$ and $B \leq B'$



Propagations

If $A \leq A'$ and $B \leq B'$



Propagations

pick up and read a book

Propagations

pick up and read a book

book contains *readable* information, compared to *radio program* which does not have a *readable* information aspect

Propagations

pick up and read a book

book contains *readable* information, compared to *radio program* which does not have a *readable* information aspect

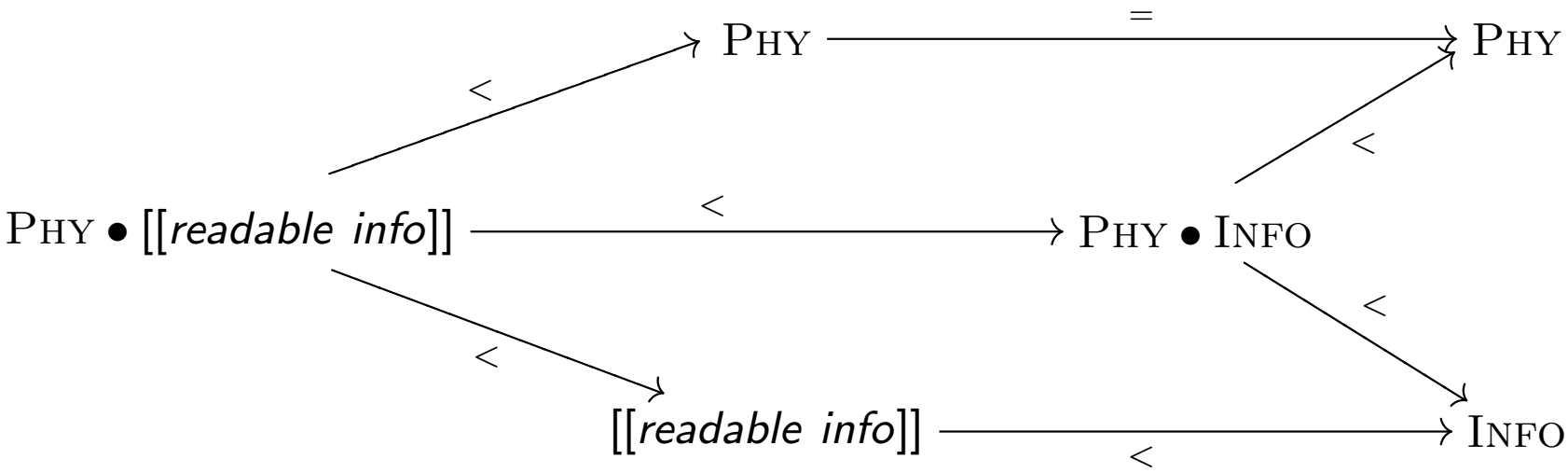
$$
\begin{array}{rcl}
[[\textit{readable}]] & : & \textsc{Info} \to \textit{Prop} \\
[[\textit{readable info}]] & = & \Sigma x : \textsc{Info}.[[\textit{readable}]](x) < \textsc{Info} \\
[[\textit{book}]] & < & \textsc{Phy} \bullet [[\textit{readable info}]]
\end{array}
$$

Propagations

pick up and read a book

book contains *readable* information, compared to *radio program* which does not have a *readable* information aspect

$$
\begin{array}{rcl}
[[\textit{readable}]] & : & \textsc{Info} \to \textit{Prop} \\
[[\textit{readable info}]] & = & \Sigma x : \textsc{Info}.[[\textit{readable}]](x) < \textsc{Info} \\
[[\textit{book}]] & < & \textsc{Phy} \bullet [[\textit{readable info}]]
\end{array}
$$

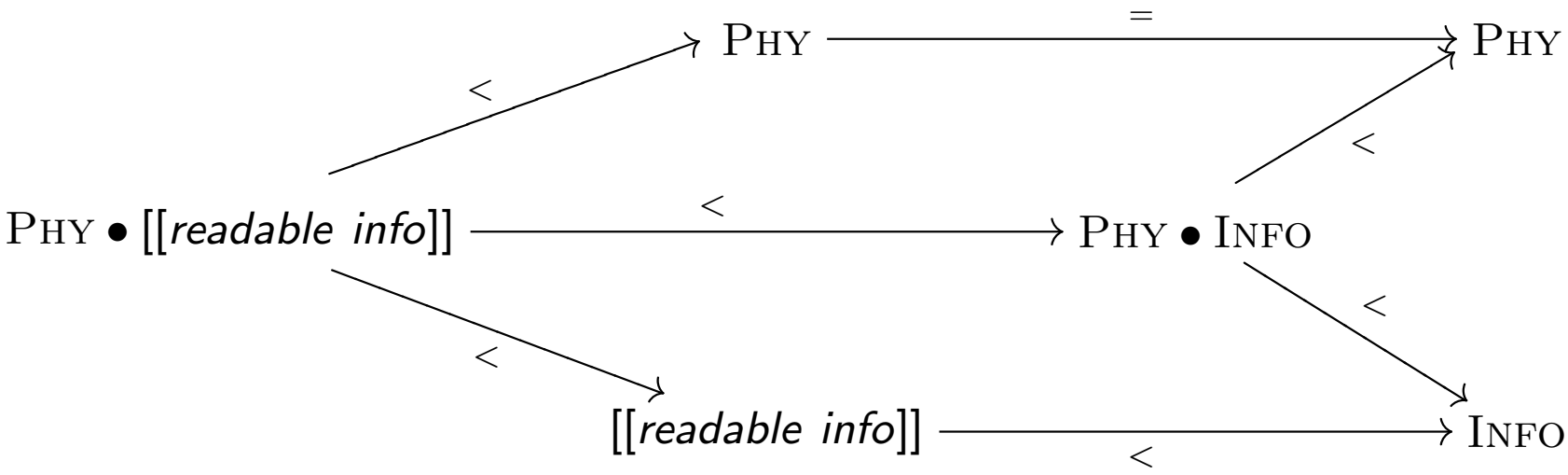


Propagations

pick up and read a book

book contains *readable* information, compared to *radio program* which does not have a *readable* information aspect

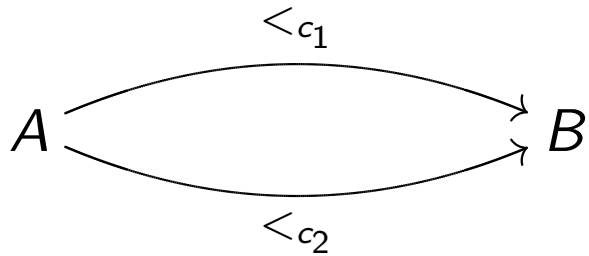
$$
\begin{array}{rcl}
[[\textit{readable}]] & : & \textsc{Info} \to \textit{Prop} \\
[[\textit{readable info}]] & = & \Sigma x : \textsc{Info}.[[\textit{readable}]](x) < \textsc{Info} \\
[[\textit{book}]] & < & \textsc{Phy} \bullet [[\textit{readable info}]]
\end{array}
$$



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

Coherence

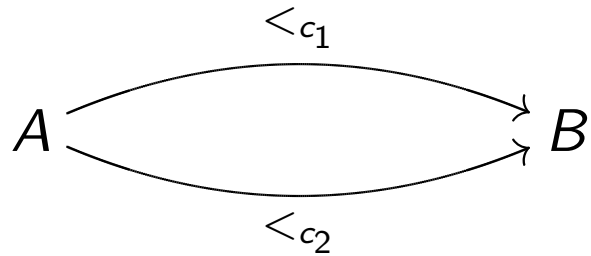
There's unique coercion between two types.



$$c_1 = c_2 : A \to B$$

Coherence

There's unique coercion between two types.



$$c_1 = c_2 \colon A \to B$$

Proposition

The coercions p_1, p_2, d_1, d_2 and d are coherent together.

$A \bullet B$ vs. $A \times B$

$A \times B$ is called a (non-dependent) product type .
If $a\colon A$ and $b\colon B$, then $(a, b)\colon A \times B$

$$(a, b)_{\pi_1} = a : A \qquad (a, b)_{\pi_2} = b : B$$

$A \bullet B$ vs. $A \times B$

$A \times B$ is called a (non-dependent) product type .
If $a : A$ and $b : B$, then $(a, b) : A \times B$

$$(a, b)_{\pi_1} = a : A \qquad (a, b)_{\pi_2} = b : B$$

A and B do not share components in $A \bullet B$

$A \bullet B$ vs. $A \times B$

$A \times B$ is called a (non-dependent) product type .
If $a\colon A$ and $b\colon B$, then $(a, b)\colon A \times B$

$$(a, b)_{\pi_1} = a : A \qquad (a, b)_{\pi_2} = b : B$$

A and B do not share components in $A \bullet B$

1. $A \bullet A$ is not a well formed dot-type
   $A \times A$ is a well formed product type.
2. In dot-types, p_1 and p_2 can both be coercions.
   In product types, only one of π_1 and π_2 can be coercion otherwise coherence may fail.

$A \bullet B$ vs. $A \times B$

$A \times B$ is called a (non-dependent) product type .
If $a\colon A$ and $b\colon B$, then $(a, b)\colon A \times B$

$$(a, b)_{\pi_1} = a : A \qquad (a, b)_{\pi_2} = b : B$$

A and B do not share components in $A \bullet B$

1. $A \bullet A$ is not a well formed dot-type
   $A \times A$ is a well formed product type.
2. In dot-types, p_1 and p_2 can both be coercions.
   In product types, only one of π_1 and π_2 can be coercion otherwise coherence may fail.

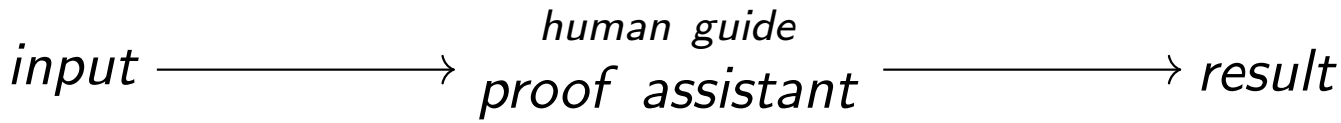
$$A \times A <_{\pi_1} A$$

$$A \times A <_{\pi_2} A$$

Outline

Plastic

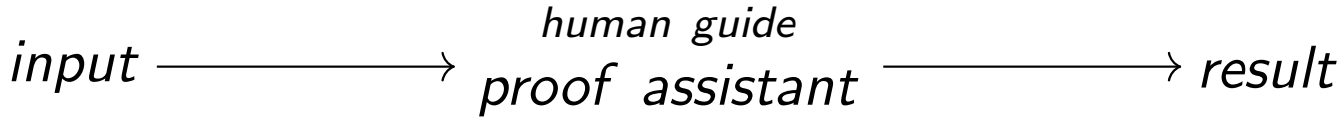
A proof assistant is an interactive theorem prover which assist with formal proofs by man-machine collaboration.



Plastic is a proof assistant which implements Luo's Logical Framework(Luo 1994). Coercive subtyping is implemented in Plastic.

Plastic

A proof assistant is an interactive theorem prover which assist with formal proofs by man-machine collaboration.



Plastic is a proof assistant which implements Luo's Logical Framework(Luo 1994). Coercive subtyping is implemented in Plastic.

Use proof assistant to formalize type-theoretical semantics.

Plastic: a simple example

"John walks"

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

Plastic

Like many other proof assistants (Coq, Matita...) various data types are defined in the library of Plastic, such as Σ types or product types.

Plastic

Like many other proof assistants (Coq, Matita...) various data types are defined in the library of Plastic, such as Σ types or product types.

However, dot-types are different with these data types. To check the sharing of the component, we need to check all the coercion relations of the term and its constituents in the context.

Plastic

Like many other proof assistants (Coq, Matita...) various data types are defined in the library of Plastic, such as Σ types or product types.

However, dot-types are different with these data types. To check the sharing of the component, we need to check all the coercion relations of the term and its constituents in the context.

We have to define the dot-types directly in the proof assistant.

Dot-types in Plastic

- If we have two types A, B which do not share components, we could simply define a type M of type $A \bullet B$:
  $> [M = A * B];$
- If we have two terms a, b, $a : A$ and $b : B$, we can define a dot term $\langle a, b \rangle$:
  $> [m = dot < a, b >];$

The coercion from $A \bullet B$ to A and B will be added automatically.

Dot-types in Plastic

In the following examples, the types share components in different ways and, therefore, none of them could be defined as a dot-type or dot term, they fail and warnings will be shown in all the following cases.

- The two constituents are the same:
  $> [M = A * A];$
- $A \bullet C$ and $A \bullet B$ have the same component A:
  $> [M = (A * C) * (A * B)];$
- A is a subtype of B, by definition of component $A \in \mathscr{C}(A) \cap \mathscr{C}(B)$
  $> [c : A- > B];$
  $> Coercion = c;$
  $> [M = A * B];$
- a and b are both terms of type A
  $> [a, b : A];$
  $> [ab = dot < a, b >];$

Dot-types in Plastic

In the following examples, the types share components in different ways and, therefore, none of them could be defined as a dot-type or dot term, they fail and warnings will be shown in all the following cases.

- The two constituents are the same:
  $> [M = A * A];$
  illegal dot-type or dot-term
- $A \bullet C$ and $A \bullet B$ have the same component A:
  $> [M = (A * C) * (A * B)];$
  illegal dot-type or dot-term
- A is a subtype of B, by definition of component $A \in \mathscr{C}(A) \cap \mathscr{C}(B)$
  $> [c : A -> B];$
  $> Coercion = c;$
  $> [M = A * B];$
  illegal dot-type or dot-term
- a and b are both terms of type A
  $> [a, b : A];$
  $> [ab = dot < a, b >];$
  illegal dot-type or dot-term

Dot-types in Plastic

John picked up and mastered a book

We should contain the following data:

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

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

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

We can interpret "and" as:

```
> [AND: (A:Type) (A->A->A)]
> [And = AND (Human ==> (Book ==> Prop))];
```

Examples

In Plastic notion, for $f : A ==> B$ and $a : A$, $ap\_\ A\ B\ f\ a$ is application $f(a)$

```
("John picked up and mastered b")

> [sentence = ap_ Book Prop (ap_ Human (Book==>Prop)
                          (And pickup master) John) b];

("John picked up and mastered a book")

> [sentence2 = Ex Book [b:Book](ap_ Book Prop (ap_ Human (Book==>Prop)
                          (And pickup master) John) b)];
```

Thank you