It's the type theory baby!

A gentle introduction to type theory (well, not so gentle . . .)

Michele Finelli
m@biodec.com
BioDec

Index

Index

This is a talk about types

So . . . what is a type in programming languages ?

1. A type is a **property**.
2. A type is **assigned to a term**.
3. So a type determines that a term (*i.e.* a “program fragment”) has a well defined **meaning**.

This is a talk about types

So . . . what is a type in programming languages ?

1. A type is a **property**.
2. A type is **assigned to a term**.
3. So a type determines that a term (*i.e.* a "program fragment") has a well defined **meaning**.

This is a talk about types

So . . . what is a type in programming languages ?

1. A type is a **property**.
2. A type is **assigned to a term**.
3. So a type determines that a term (*i.e.* a "program fragment") has a well defined **meaning**.

This is a talk about types

So . . . what is a type in programming languages ?

1. A type is a **property**.
2. A type is **assigned to a term**.
3. So a type determines that a term (*i.e.* a “program fragment”) has a well defined **meaning**.

What do we mean by “meaning” ?

So . . . which are the **acceptable meanings** of a program fragment ? Which of the following is an **acceptable property** ?

1. Is it being an IEEE 754-2008 float ?
2. Is the assurance that a given memory location will not be changed ?
3. Is the statement that a given programming structure has a certain size ?
4. What about this ? — N are the natural numbers, Perm is a permutation.

$$\forall n, \forall f : N \to N, \exists \pi \in \mathrm{Perm}, \forall 1 \leq i, j \leq n$$
$$i \leq j \implies f(\pi(i)) \leq f(\pi(j))$$

What do we mean by “meaning” ?

So . . . which are the **acceptable meanings** of a program fragment ? Which of the following is an **acceptable property** ?

1. Is it being an IEEE 754-2008 float ?
2. Is the assurance that a given memory location will not be changed ?
3. Is the statement that a given programming structure has a certain size ?
4. What about this ? — N are the natural numbers, Perm is a permutation.

$$\forall n, \forall f : N \to N, \exists \pi \in \text{Perm}, \forall 1 \leq i, j \leq n$$
$$i \leq j \implies f(\pi(i)) \leq f(\pi(j))$$

BIODEC Evolving ICT Infrastructures

What do we mean by “meaning” ?

So . . . which are the **acceptable meanings** of a program fragment ? Which of the following is an **acceptable property** ?

1. Is it being an IEEE 754-2008 float ?
2. Is the assurance that a given memory location will not be changed ?
3. Is the statement that a given programming structure has a certain size ?
4. What about this ? — N are the natural numbers, Perm is a permutation.

$$\forall n, \forall f : N \to N, \exists \pi \in \text{Perm}, \forall 1 \leq i, j \leq n$$
$$i \leq j \implies f(\pi(i)) \leq f(\pi(j))$$

What do we mean by "meaning" ?

So . . . which are the **acceptable meanings** of a program fragment ? Which of the following is an **acceptable property** ?

1. Is it being an IEEE 754-2008 float ?
2. Is the assurance that a given memory location will not be changed ?
3. Is the statement that a given programming structure has a certain size ?
4. What about this ? — N are the natural numbers, Perm is a permutation.

$$\forall n, \forall f : N \to N, \exists \pi \in \mathrm{Perm}, \forall 1 \leq i, j \leq n$$
$$i \leq j \implies f(\pi(i)) \leq f(\pi(j))$$

BIODEC
Evolving ICT Infrastructures

What do we mean by “meaning” ?

So . . . which are the **acceptable meanings** of a program fragment ? Which of the following is an **acceptable property** ?

1. Is it being an IEEE 754-2008 float ?
2. Is the assurance that a given memory location will not be changed ?
3. Is the statement that a given programming structure has a certain size ?
4. What about this ? — N are the natural numbers, Perm is a permutation.

$$\forall n, \forall f : N \to N, \exists \pi \in \text{Perm}, \forall 1 \leq i, j \leq n$$
$$i \leq j \implies f(\pi(i)) \leq f(\pi(j))$$

BIODEC
Evolving ICT Infrastructures

Meaningful answers

1. Is being an IEEE 754-2008 float a **property** ? **Yes.** It is a typical variable typing judgement.
2. Is the assurance that a given memory location will not be changed, a **property** ? **Yes.** It is a safety property (not so easy to express).

Meaningful answers

1. Is being an IEEE 754-2008 float a **property** ? **Yes**. It is a typical variable typing judgement.
2. Is the assurance that a given memory location will not be changed, a **property** ? **Yes**. It is a safety property (not so easy to express).

Meaningful answers

1. Is being an IEEE 754-2008 float a **property** ? **Yes**. It is a typical variable typing judgement.
2. Is the assurance that a given memory location will not be changed, a **property** ? **Yes**. It is a safety property (not so easy to express).

Meaningful answers

1. Is being an IEEE 754-2008 float a **property** ? **Yes**. It is a typical variable typing judgement.
2. Is the assurance that a given memory location will not be changed, a **property** ? **Yes**. It is a safety property (not so easy to express).

Meaningful answers

3. Is the statement that a given programming structure has a certain size, a **property** ? **Yes.** It could be a dependent type judgement, for example.
4. What about this ? — N are the natural numbers, Perm is a permutation.

$$\forall n, \forall f : N \to N, \exists \pi \in \text{Perm}, \forall 1 \leq i, j \leq n$$
$$i \leq j \implies f(\pi(i)) \leq f(\pi(j))$$

Yes again. It states that there is an algorithm that sorts the natural numbers.

Meaningful answers

3. Is the statement that a given programming structure has a certain size, a **property** ? **Yes**. It could be a dependent type judgement, for example.

4. What about this ? — N are the natural numbers, Perm is a permutation.

$$\forall n, \forall f : N \to N, \exists \pi \in \text{Perm}, \forall 1 \leq i, j \leq n$$
$$i \leq j \implies f(\pi(i)) \leq f(\pi(j))$$

Yes again. It states that there is an algorithm that sorts the natural numbers.

Meaningful answers

3. Is the statement that a given programming structure has a certain size, a **property** ? **Yes**. It could be a dependent type judgement, for example.
4. What about this ? — N are the natural numbers, Perm is a permutation.

$$\forall n, \forall f : N \to N, \exists \pi \in \text{Perm}, \forall 1 \leq i, j \leq n$$
$$i \leq j \implies f(\pi(i)) \leq f(\pi(j))$$

Yes again. It states that there is an algorithm that sorts the natural numbers.

Meaningful answers

3. Is the statement that a given programming structure has a certain size, a **property** ? **Yes**. It could be a dependent type judgement, for example.
4. What about this ? — N are the natural numbers, Perm is a permutation.

$$\forall n, \forall f : N \to N, \exists \pi \in \text{Perm}, \forall 1 \leq i, j \leq n$$
$$i \leq j \implies f(\pi(i)) \leq f(\pi(j))$$

Yes again. It states that there is an algorithm that sorts the natural numbers.

Meaningful answers

3. Is the statement that a given programming structure has a certain size, a **property** ? **Yes**. It could be a dependent type judgement, for example.
4. What about this ? — N are the natural numbers, Perm is a permutation.

$$\forall n, \forall f : N \to N, \exists \pi \in \text{Perm}, \forall 1 \leq i, j \leq n$$
$$i \leq j \implies f(\pi(i)) \leq f(\pi(j))$$

Yes again. It states that there is an algorithm that sorts the natural numbers.

A property is a logical formula

- A type judgement about a program fragment is *simply* a **logical formula** that is satisfied by that program.
- But a program does not “stand still” . . . it *runs*, it *gets executed*: a formula *au contraire* is **always** true or false, how do we reconcile this **static versus dynamic** conundrum ?

A property is a logical formula

- A type judgement about a program fragment is *simply* a **logical formula** that is satisfied by that program.
- But a program does not “stand still” . . . it *runs*, it *gets executed*: a formula *au contraire* is **always** true or false, how do we reconcile this **static versus dynamic** conundrum ?

A property is a logical formula

- A type judgement about a program fragment is *simply* a **logical formula** that is satisfied by that program.
- But a program does not "stand still" . . . it *runs*, it *gets executed*: a formula *au contraire* is **always** true or false, how do we reconcile this **static versus dynamic** conundrum ?

Formulas are eternal !

- Since the **provability** of a logical formula does not change with time, we must assume that also the meaning of the fragment of code is **absolutely determined** and that it does not depend on being executed.
- **Execution** is just the reduction of the code to something simpler but equivalent: the **result of the computation**.

Formulas are eternal !

- Since the **provability** of a logical formula does not change with time, we must assume that also the meaning of the fragment of code is **absolutely determined** and that it does not depend on being executed.
- **Execution** is just the reduction of the code to something simpler but equivalent: the **result of the computation**.

Formulas are eternal !

- Since the **provability** of a logical formula does not change with time, we must assume that also the meaning of the fragment of code is **absolutely determined** and that it does not depend on being executed.
- **Execution** is just the reduction of the code to something simpler but equivalent: the **result of the computation**.

Propositions as types is a powerful notion

Providing a type to a term subsumes some properties of the code fragment to which the type is attached:

- the code must be *pure*, *i.e.* without **side effects** since the meaning of the code can not depend on something external to the program;
- since the formula is completely determined by its variables, we have a notion of **functionality** of the related code;
- we have a notion of **immutability**.

Propositions as types is a powerful notion

Providing a type to a term subsumes some properties of the code fragment to which the type is attached:

- the code must be *pure*, *i.e.* without **side effects** since the meaning of the code can not depend on something external to the program;
- since the formula is completely determined by its variables, we have a notion of **functionality** of the related code;
- we have a notion of **immutability**.

Propositions as types is a powerful notion

Providing a type to a term subsumes some properties of the code fragment to which the type is attached:

- the code must be *pure*, *i.e.* without **side effects** since the meaning of the code can not depend on something external to the program;
- since the formula is completely determined by its variables, we have a notion of **functionality** of the related code;
- we have a notion of **immutability**.

Propositions as types is a powerful notion

Providing a type to a term subsumes some properties of the code fragment to which the type is attached:

- the code must be *pure*, *i.e.* without **side effects** since the meaning of the code can not depend on something external to the program;
- since the formula is completely determined by its variables, we have a notion of **functionality** of the related code;
- we have a notion of **immutability**.

Sounds familiar ?

- Purity, or “no side effects” ?
- Functions as the building blocks.
- Immutable values.

Yes ! I am talking about **functional programming** !

Sounds familiar ?

- Purity, or “no side effects” ?
- Functions as the building blocks.
- Immutable values.

Yes ! I am talking about **functional programming** !

Sounds familiar ?

- Purity, or “no side effects” ?
- Functions as the building blocks.
- Immutable values.

Yes ! I am talking about **functional programming** !

Sounds familiar ?

- Purity, or "no side effects" ?
- Functions as the building blocks.
- Immutable values.

Yes ! I am talking about **functional programming** !

Sounds familiar ?

- Purity, or “no side effects” ?
- Functions as the building blocks.
- Immutable values.

Yes ! I am talking about **functional programming** !

Index

BIODEC
Evolving ICT Infrastructures

Or, how we got the *propositions as types* interpretation

This section is a brief outline of type theory, mainly focused on the the branch that studied the interpretation of:

- propositions as types,
- proofs as programs,

$$\text{Program} : \text{Type} \Leftrightarrow \text{Proof} : \text{Proposition}$$

- execution as normalization.

Or, how we got the *propositions as types* interpretation

This section is a brief outline of type theory, mainly focused on the the branch that studied the interpretation of:

- propositions as types,
- proofs as programs,

  $$\text{Program} : \text{Type} \Leftrightarrow \text{Proof} : \text{Proposition}$$

- execution as normalization.

Or, how we got the *propositions as types* interpretation

This section is a brief outline of type theory, mainly focused on the the branch that studied the interpretation of:

- propositions as types,
- proofs as programs,

$$\text{Program} : \text{Type} \Leftrightarrow \text{Proof} : \text{Proposition}$$

- execution as normalization.

Or, how we got the *propositions as types* interpretation

This section is a brief outline of type theory, mainly focused on the the branch that studied the interpretation of:

- propositions as types,
- proofs as programs,

$$\text{Program} : \text{Type} \Leftrightarrow \text{Proof} : \text{Proposition}$$

- execution as normalization.

Or, how we got the *propositions as types* interpretation

This section is a brief outline of type theory, mainly focused on the the branch that studied the interpretation of:

- propositions as types,
- proofs as programs,

$$\text{Program} : \text{Type} \Leftrightarrow \text{Proof} : \text{Proposition}$$

- execution as normalization.

Propositions as types

. . . or “Formulae as Types”, Curry-Howard-de Bruijn Correspondence, Brouwer’s Dictum, and others.

- The analogy between a certain kind of logical formulas, namely the **propositions**, and the meaning given to **types** is indeed a **mathematical theorem**.
- It is usually referred as the **Curry - Howard Isomorphism**.

Propositions as types

…or "Formulae as Types", Curry-Howard-de Bruijn Correspondence, Brouwer's Dictum, and others.

- The analogy between a certain kind of logical formulas, namely the **propositions**, and the meaning given to **types** is indeed a **mathematical theorem**.
- It is usually referred as the **Curry - Howard Isomorphism**.

Propositions as types

…or "Formulae as Types", Curry-Howard-de Bruijn Correspondence, Brouwer's Dictum, and others.

- The analogy between a certain kind of logical formulas, namely the **propositions**, and the meaning given to **types** is indeed a **mathematical theorem**.
- It is usually referred as the **Curry - Howard Isomorphism**.

Propositions as types

Propositions as Types is a notion with breadth. It applies to a range of logics including propositional, predicate, second-order, intuitionistic, classical, modal, and linear. It underpins the foundations of functional programming, explaining features including functions, records, variants, parametric polymorphism, data abstraction, continuations, linear types, and session types. (Wadler, Propositions as types)

Propositions as types

(Wadler, continued)

- *Why should it be the case that intuitionistic natural deduction, as developed by Gentzen in the 1930s, and simply-typed λ-calculus, as developed by Church around the same time for an unrelated purpose, should be discovered thirty years later to be essentially identical?*
- *The logician Hindley and the computer scientist Milner independently developed the same type system, now dubbed Hindley-Milner.*
- *The logician Girard and the computer scientist Reynolds independently developed the same calculus, now dubbed Girard-Reynolds.*

Propositions as types

(Wadler, continued)

- *Why should it be the case that intuitionistic natural deduction, as developed by Gentzen in the 1930s, and simply-typed λ-calculus, as developed by Church around the same time for an unrelated purpose, should be discovered thirty years later to be essentially identical?*
- *The logician Hindley and the computer scientist Milner independently developed the same type system, now dubbed Hindley-Milner.*
- *The logician Girard and the computer scientist Reynolds independently developed the same calculus, now dubbed Girard-Reynolds.*

Propositions as types

(Wadler, continued)

- *Why should it be the case that intuitionistic natural deduction, as developed by Gentzen in the 1930s, and simply-typed λ-calculus, as developed by Church around the same time for an unrelated purpose, should be discovered thirty years later to be essentially identical?*
- *The logician Hindley and the computer scientist Milner independently developed the same type system, now dubbed Hindley-Milner.*
- *The logician Girard and the computer scientist Reynolds independently developed the same calculus, now dubbed Girard-Reynolds.*

BIODEC Evolving ICT Infrastructures

Propositions as types

(Wadler, continued)

- *Why should it be the case that intuitionistic natural deduction, as developed by Gentzen in the 1930s, and simply-typed λ-calculus, as developed by Church around the same time for an unrelated purpose, should be discovered thirty years later to be essentially identical?*
- *The logician Hindley and the computer scientist Milner independently developed the same type system, now dubbed Hindley-Milner.*
- *The logician Girard and the computer scientist Reynolds independently developed the same calculus, now dubbed Girard-Reynolds.*

Propositions as types

(Wadler, continued)

- In 1934, Curry observed a curious fact, relating a theory of functions to a theory of implication. Every type of a function ($A \to B$) could be read as a proposition ($A \supset B$), and under this reading the type of any given function would always correspond to a provable proposition.
- Conversely, for every provable proposition there was a function with the corresponding type.

Propositions as types

(Wadler, continued)

- *In 1934, Curry observed a curious fact, relating a theory of functions to a theory of implication. Every type of a function ($A \to B$) could be read as a proposition ($A \supset B$), and under this reading the type of any given function would always correspond to a provable proposition.*
- *Conversely, for every provable proposition there was a function with the corresponding type.*

BIODEC
Evolving ICT Infrastructures

Propositions as types

(Wadler, continued)

- *In 1934, Curry observed a curious fact, relating a theory of functions to a theory of implication. Every type of a function ($A \to B$) could be read as a proposition ($A \supset B$), and under this reading the type of any given function would always correspond to a provable proposition.*
- *Conversely, for every provable proposition there was a function with the corresponding type.*

Propositions as types

(Wadler, continued)

- *In 1969, Howard circulated a xeroxed manuscript. It was not published until 1980, where it appeared in a Festschrift dedicated to Curry. Motivated by Curry's observation, Howard pointed out that there is a similar correspondence between natural deduction, on the one hand, and simply-typed λ-calculus, on the other, and he made explicit the third and deepest level of the correspondence (...), that simplification of proofs corresponds to evaluation of programs.*

Propositions as types

(Wadler, continued)

- *In 1969, Howard circulated a xeroxed manuscript. It was not published until 1980, where it appeared in a Festschrift dedicated to Curry. Motivated by Curry's observation, Howard pointed out that there is a similar correspondence between natural deduction, on the one hand, and simply-typed λ-calculus, on the other, and he made explicit the third and deepest level of the correspondence (. . .), that simplification of proofs corresponds to evaluation of programs.*

Propositions as types

Isomorphism

BIODEC
Evolving ICT Infrastructures

Index

BIODEC
Evolving ICT Infrastructures

Type systems helps

I will give a few examples taken from a recent research that shows the relevance of static typing (as opposed to dynamic typing) to build correct code in less time (ICSE 2014, OOPSLA 2012).

ICSE 2014 How Do API Documentation . . .

> *The presence of a static type system had a significant positive effect on development time: Subjects using the statically typed language required between 15 and 89 minutes less time for solving the task.*

OOPSLA 2012 Static Type Systems (Sometimes) . . .

We gave 27 subjects five programming tasks and found that the type systems had a significant impact on the development time: for three of five tasks me measured apositive impact of the static type system, for two tasks we measured a positive impact of the dynamic type system.

ICFP 2010 Experience Report: Haskell as a Reagent

At the end, the question was: is the extra effort needed for maintaining code written in two languages justified? Do we get any advantage out of combining two high-level, but quite different, languages? As we try to show in this paper, in our experience the answer is affirmative, sometimes in non obvious ways.

Index

Why do not we program in the only one (*true*) typed language ?

- Because it does not exist !
- Historically, languages have been built before the theory formalized algorithms and techniques — the concept of monad as a way of encapsulating *state* in functional languages is in (Moggi, Information and Computation 93, 1991).
- If types are logical formulas we still have the problem of choosing the right **logical theory**.
- Moreover there is not an agreement on what is the right amount of typing: we are full of endless *this feature versus that feature* like …

Why do not we program in the only one (*true*) typed language ?

- Because it does not exist !
- Historically, languages have been built before the theory formalized algorithms and techniques — the concept of monad as a way of encapsulating *state* in functional languages is in (Moggi, Information and Computation 93, 1991).
- If types are logical formulas we still have the problem of choosing the right **logical theory**.
- Moreover there is not an agreement on what is the right amount of typing: we are full of endless *this feature versus that feature* like . . .

Why do not we program in the only one (*true*) typed language ?

- Because it does not exist !
- Historically, languages have been built before the theory formalized algorithms and techniques — the concept of monad as a way of encapsulating *state* in functional languages is in (Moggi, Information and Computation 93, 1991).
- If types are logical formulas we still have the problem of choosing the right **logical theory**.
- Moreover there is not an agreement on what is the right amount of typing: we are full of endless *this feature versus that feature* like . . .

Why do not we program in the only one (*true*) typed language ?

- Because it does not exist !
- Historically, languages have been built before the theory formalized algorithms and techniques — the concept of monad as a way of encapsulating *state* in functional languages is in (Moggi, Information and Computation 93, 1991).
- If types are logical formulas we still have the problem of choosing the right **logical theory**.
- Moreover there is not an agreement on what is the right amount of typing: we are full of endless *this feature versus that feature* like …

BIODEC
Evolving ICT Infrastructures

Why do not we program in the only one (*true*) typed language ?

- Because it does not exist !
- Historically, languages have been built before the theory formalized algorithms and techniques — the concept of monad as a way of encapsulating *state* in functional languages is in (Moggi, Information and Computation 93, 1991).
- If types are logical formulas we still have the problem of choosing the right **logical theory**.
- Moreover there is not an agreement on what is the right amount of typing: we are full of endless *this feature versus that feature* like . . .

Typed vs untyped

> *Languages that do not restrict the range of variables are called **untyped languages**: they do not have types or, equivalently, have a single universal type that contains all values. In these languages, operations may be applied to inappropriate arguments: the result may be a fixed arbitrary value, a fault, an exception, or an unspecified effect. (Cardelli, Type Systems)*

Untyped or *uni*typed languages ?

- Lisp is the typical untyped language . . . is Ruby a typed language ?
- Or Javascript (Node) ?
- People deal with the issue of having a type system **which allows bad things to happen** saying that . . .

Untyped or *uni*typed languages ?

- Lisp is the typical untyped language . . . is Ruby a typed language ?
- Or Javascript (Node) ?
- People deal with the issue of having a type system **which allows bad things to happen** saying that . . .

Untyped or *uni*typed languages ?

- Lisp is the typical untyped language . . . is Ruby a typed language ?
- Or Javascript (Node) ?
- People deal with the issue of having a type system **which allows bad things to happen** saying that . . .

Untyped or *uni*typed languages ?

- Lisp is the typical untyped language . . . is Ruby a typed language ?
- Or Javascript (Node) ?
- People deal with the issue of having a type system **which allows bad things to happen** saying that . . .

Static *vs* dynamic

. . . the type system is *dynamic* or that their terms are *duck typed*.

- Come on ! It does not make any sense: either a language is typed, or it is not.
- A language can be very useful even if it is not typed: this is not the point.
- Because the real issue is that even (*statically*) typed languages **can fail**, because of . . .

Static *vs* dynamic

...the type system is *dynamic* or that their terms are *duck typed*.

- Come on ! It does not make any sense: either a language is typed, or it is not.
- A language can be very useful even if it is not typed: this is not the point.
- Because the real issue is that even (*statically*) typed languages **can fail**, because of ...

Static *vs* dynamic

…the type system is *dynamic* or that their terms are *duck typed*.

- Come on ! It does not make any sense: either a language is typed, or it is not.
- A language can be very useful even if it is not typed: this is not the point.
- Because the real issue is that even (*statically*) typed languages **can fail**, because of …

Static *vs* dynamic

. . . the type system is *dynamic* or that their terms are *duck typed*.

- Come on ! It does not make any sense: either a language is typed, or it is not.
- A language can be very useful even if it is not typed: this is not the point.
- Because the real issue is that even (*statically*) typed languages **can fail**, because of . . .

Safe *vs* unsafe

...**safety**. Quoting again Cardelli:

> *In reality, certain statically checked languages do not ensure **safety**. That is, their set of forbidden errors does not include all untrapped errors. (...) For example (...) C has many unsafe and widely used features, such as pointer arithmetic and casting. It is interesting to notice that **the first five of the ten commandments for C programmers are directed at compensating for the weak-checking aspects of C**. Some of the problems caused by weak checking in C have been alleviated in C++, and even more have been addressed in Java, confirming a trend away from weak checking.*

Explicit *vs* implicit

- The real burden in using types is the **extra work needed to specify types**: see what happens in C++ (even with *auto*) or Java.
- **The real advantage** is in using languages where types can be *inferred* by the compiler, with none to little help from the programmer.
- Even better if the *typing relation* is decidable (Scala type system, for example, is known to be Turing complete).

Explicit *vs* implicit

- The real burden in using types is the **extra work needed to specify types**: see what happens in C++ (even with *auto*) or Java.
- **The real advantage** is in using languages where types can be *inferred* by the compiler, with none to little help from the programmer.
- Even better if the *typing relation* is decidable (Scala type system, for example, is known to be Turing complete).

Explicit *vs* implicit

- The real burden in using types is the **extra work needed to specify types**: see what happens in C++ (even with *auto*) or Java.
- **The real advantage** is in using languages where types can be *inferred* by the compiler, with none to little help from the programmer.
- Even better if the *typing relation* is decidable (Scala type system, for example, is known to be Turing complete).

Explicit *vs* implicit

- The real burden in using types is the **extra work needed to specify types**: see what happens in C++ (even with *auto*) or Java.
- **The real advantage** is in using languages where types can be *inferred* by the compiler, with none to little help from the programmer.
- Even better if the *typing relation* is decidable (Scala type system, for example, is known to be Turing complete).

Index

Do not miss an opportunity

- A functional programming language without a type system **misses an opportunity**.
- A type system whose term language is not isomorphic to the programming langage **misses an opportunity**.
- Functional programming and type systems are two faces of the same thing: **do not use one without the other**.

Do not miss an opportunity

- A functional programming language without a type system **misses an opportunity.**
- A type system whose term language is not isomorphic to the programming langage **misses an opportunity.**
- Functional programming and type systems are two faces of the same thing: **do not use one without the other.**

Do not miss an opportunity

- A functional programming language without a type system **misses an opportunity.**
- A type system whose term language is not isomorphic to the programming langage **misses an opportunity.**
- Functional programming and type systems are two faces of the same thing: **do not use one without the other.**

Do not miss an opportunity

- A functional programming language without a type system **misses an opportunity.**
- A type system whose term language is not isomorphic to the programming langage **misses an opportunity.**
- Functional programming and type systems are two faces of the same thing: **do not use one without the other.**

Questions

?

Thanks & see you soon . . .

Thanks for paying attention.

The biggest Italian DevOps meeting is in Bologna, April 10^{th}, 2015.

IDI2015 Incontro DevOps Italia 2015: http://incontrodevops.it/idi2015/

Thanks & see you soon ...

Thanks	for paying attention.
The biggest Italian DevOps meeting	is in Bologna, April 10th, 2015.
IDI2015	Incontro DevOps Italia 2015: `http://incontrodevops.it/idi2015/`

BIODEC
Evolving ICT Infrastructures

Thanks & see you soon . . .

Thanks for paying attention.

The biggest Italian DevOps meeting is in Bologna, April 10th, 2015.

IDI2015 Incontro DevOps Italia 2015: http://incontrodevops.it/idi2015/

BIODEC Evolving ICT Infrastructures

Thanks & see you soon . . .

Thanks for paying attention.

The biggest Italian DevOps meeting is in Bologna, April 10^{th}, 2015.

IDI2015 Incontro DevOps Italia 2015: http://incontrodevops.it/idi2015/