Types by automata

Klara Zielińska

Department of Computer Science
Uniwersytet Wrocławski

Forum Informatyki Teoretycznej, 2015

Outline

1. Motivation
   - Types and their semantics should be isomorphic
   - Some type constructions
   - Terms seem insufficient as types
2. Solution
   - Types as sets with possibly many representations
   - Automata representation

Types Isomorphic To The Semantics of Types

- I wanted to types be isomorphic with their semantics.
  - Precisely: a value v is of a type t iff $v \in [\![t]\!]$; and $[\![]\!]$ is a bijection.
- Why?
- Because it corresponds to intuitions and it makes reasoning about types easier.
- If someone does not believe, let he compare reasoning about the set of primitive natural numbers and a set of all algebraic expressions describing natural numbers.

Types Isomorphic To The Semantics of Types

- I wanted to types be isomorphic with their semantics.
  - Precisely: a value v is of a type t iff $v \in [\![t]\!]$; and $[\![]\!]$ is a bijection.
- Why?
- Because it corresponds to intuitions and it makes reasoning about types easier.
- If someone does not believe, let he compare reasoning about the set of primitive natural numbers and a set of all algebraic expressions describing natural numbers.

Types Isomorphic To The Semantics of Types

- I wanted to types be isomorphic with their semantics.
  - Precisely: a value v is of a type t iff $v \in [\![t]\!]$; and $[\![]\!]$ is a bijection.
- Why?
- Because it corresponds to intuitions and it makes reasoning about types easier.
- If someone does not believe, let he compare reasoning about the set of primitive natural numbers and a set of all algebraic expressions describing natural numbers.

Types Isomorphic To The Semantics of Types

- I wanted to types be isomorphic with their semantics.
  - Precisely: a value v is of a type t iff $v \in [\![t]\!]$; and $[\![]\!]$ is a bijection.
- Why?
- Because it corresponds to intuitions and it makes reasoning about types easier.
- If someone does not believe, let he compare reasoning about the set of primitive natural numbers and a set of all algebraic expressions describing natural numbers.

Outline

1. Motivation
   - Types and their semantics should be isomorphic
   - Some type constructions
   - Terms seem insufficient as types
2. Solution
   - Types as sets with possibly many representations
   - Automata representation

Non-deterministic Type Alternatives

- I wanted to have natural type alternatives.
  - Precisely: a value v is of a type $t_1|t_2$ iff v is of the type t_1 or v is of the type t_2.
- A consequence of such alternatives is that we can write strongly statically typed programs in a more mathematical natural way — we do not need constructors.
- Specifically, excessive redefinitions of values — which have no semantical meaning and are required only to satisfy a type system — can be avoided.

Non-deterministic Type Alternatives

- I wanted to have natural type alternatives.
  - Precisely: a value v is of a type $t_1|t_2$ iff v is of the type t_1 or v is of the type t_2.
- A consequence of such alternatives is that we can write strongly statically typed programs in a more mathematical natural way — we do not need constructors.
- Specifically, excessive redefinitions of values — which have no semantical meaning and are required only to satisfy a type system — can be avoided.

Non-deterministic Type Alternatives

- I wanted to have natural type alternatives.
  - Precisely: a value v is of a type $t_1|t_2$ iff v is of the type t_1 or v is of the type t_2.
- A consequence of such alternatives is that we can write strongly statically typed programs in a more mathematical natural way — we do not need constructors.
- Specifically, excessive redefinitions of values — which have no semantical meaning and are required only to satisfy a type system — can be avoided.

Non-deterministic Type Alternatives

- I wanted to have natural type alternatives.
  - Precisely: a value v is of a type $t_1|t_2$ iff v is of the type t_1 or v is of the type t_2.
- A consequence of such alternatives is that we can write strongly statically typed programs in a more mathematical natural way — we do not need constructors.
- Specifically, excessive redefinitions of values — which have no semantical meaning and are required only to satisfy a type system — can be avoided.

Two Kinds of Equi-Recursive Types

- In type systems we may want to have induction-like recursive types — e.g., lists (the least X such that $X = \mathtt{NIL} \mid \mathtt{Int} * X$)
- and co-induction-like recursive types — e.g., streams (the greatest X such that $X = \mathtt{NIL} \mid \mathtt{Int} * X$).
- I wanted to have a type system with both of these recursive constructions (precisely, equi-recursive).
  - The reason: some operations are valid only on values of one kind of these recursions.
  - E.g., `length` is a proper operation on (finite) lists, but not on streams (that can be infinite).

Two Kinds of Equi-Recursive Types

- In type systems we may want to have induction-like recursive types — e.g., lists (the least X such that $X = \texttt{NIL} \mid \texttt{Int} * X$)
- and co-induction-like recursive types — e.g., streams (the greatest X such that $X = \texttt{NIL} \mid \texttt{Int} * X$).
- I wanted to have a type system with both of these recursive constructions (precisely, equi-recursive).
    - The reason: some operations are valid only on values of one kind of these recursions.
    - E.g., `length` is a proper operation on (finite) lists, but not on streams (that can be infinite).

Two Kinds of Equi-Recursive Types

- In type systems we may want to have induction-like recursive types — e.g., lists (the least X such that $X = \mathtt{NIL} \mid \mathtt{Int} * X$)
- and co-induction-like recursive types — e.g., streams (the greatest X such that $X = \mathtt{NIL} \mid \mathtt{Int} * X$).
- I wanted to have a type system with both of these recursive constructions (precisely, equi-recursive).
  - The reason: some operations are valid only on values of one kind of these recursions.
  - E.g., `length` is a proper operation on (finite) lists, but not on streams (that can be infinite).

Two Kinds of Equi-Recursive Types

- In type systems we may want to have induction-like recursive types — e.g., lists (the least X such that $X = \mathtt{NIL} \mid \mathtt{Int} * X$)
- and co-induction-like recursive types — e.g., streams (the greatest X such that $X = \mathtt{NIL} \mid \mathtt{Int} * X$).
- I wanted to have a type system with both of these recursive constructions (precisely, equi-recursive).
  - The reason: some operations are valid only on values of one kind of these recursions.
  - E.g., `length` is a proper operation on (finite) lists, but not on streams (that can be infinite).

Two Kinds of Equi-Recursive Types

- In type systems we may want to have induction-like recursive types — e.g., lists (the least X such that $X = \mathtt{NIL} \mid \mathtt{Int} * X$)
- and co-induction-like recursive types — e.g., streams (the greatest X such that $X = \mathtt{NIL} \mid \mathtt{Int} * X$).
- I wanted to have a type system with both of these recursive constructions (precisely, equi-recursive).
  - The reason: some operations are valid only on values of one kind of these recursions.
  - E.g., `length` is a proper operation on (finite) lists, but not on streams (that can be infinite).

Outline

Terms Are Impaired Types

- Commonly types are identified with terms, but ...
- The natural way to check types given by terms fails for non-deterministic alternatives.
  - E.g., checking if types $\mathtt{Bool} * (\mathtt{Int}|\mathtt{Float})$, $\mathtt{Bool} * \mathtt{Int} \mid \mathtt{Bool} * \mathtt{Float}$ are semantically equal cannot be done in the standard way — that is by a structural induction.
- The above example also shows that when we have natural type alternatives, then term-types are not isomorphic with their semantics (because the terms are different, but they are semantically equivalent).
- Moreover, terms fail to properly model type systems w.r.t. recursive equations.
  - E.g., the equation $X = \mathtt{NIL} \mid \mathtt{Int} * X$ may be solved by 2 types (a list and a stream), but there is 0 or 1 term — depending on the terms definition — that satisfies it.

Terms Are Impaired Types

- Commonly types are identified with terms, but ...
- The natural way to check types given by terms fails for non-deterministic alternatives.
  - E.g., checking if types $\mathtt{Bool} * (\mathtt{Int}|\mathtt{Float})$, $\mathtt{Bool} * \mathtt{Int} \,|\, \mathtt{Bool} * \mathtt{Float}$ are semantically equal cannot be done in the standard way — that is by a structural induction.
- The above example also shows that when we have natural type alternatives, then term-types are not isomorphic with their semantics (because the terms are different, but they are semantically equivalent).
- Moreover, terms fail to properly model type systems w.r.t. recursive equations.
  - E.g., the equation $X = \mathtt{NIL} \mid \mathtt{Int} * X$ may be solved by 2 types (a list and a stream), but there is 0 or 1 term — depending on the terms definition — that satisfies it.

Terms Are Impaired Types

- Commonly types are identified with terms, but ...
- The natural way to check types given by terms fails for non-deterministic alternatives.
  - E.g., checking if types $\mathtt{Bool} * (\mathtt{Int}|\mathtt{Float})$, $\mathtt{Bool} * \mathtt{Int} \,|\, \mathtt{Bool} * \mathtt{Float}$ are semantically equal cannot be done in the standard way — that is by a structural induction.
- The above example also shows that when we have natural type alternatives, then term-types are not isomorphic with their semantics (because the terms are different, but they are semantically equivalent).
- Moreover, terms fail to properly model type systems w.r.t. recursive equations.
  - E.g., the equation $X = \mathtt{NIL} \mid \mathtt{Int} * X$ may be solved by 2 types (a list and a stream), but there is 0 or 1 term — depending on the terms definition — that satisfies it.

Terms Are Impaired Types

- Commonly types are identified with terms, but ...
- The natural way to check types given by terms fails for non-deterministic alternatives.
  - E.g., checking if types $\mathtt{Bool} * (\mathtt{Int}|\mathtt{Float})$, $\mathtt{Bool} * \mathtt{Int} \mid \mathtt{Bool} * \mathtt{Float}$ are semantically equal cannot be done in the standard way — that is by a structural induction.
- The above example also shows that when we have natural type alternatives, then term-types are not isomorphic with their semantics (because the terms are different, but they are semantically equivalent).
- Moreover, terms fail to properly model type systems w.r.t. recursive equations.
  - E.g., the equation $X = \mathtt{NIL} \mid \mathtt{Int} * X$ may be solved by 2 types (a list and a stream), but there is 0 or 1 term — depending on the terms definition — that satisfies it.

Terms Are Impaired Types

- Commonly types are identified with terms, but ...
- The natural way to check types given by terms fails for non-deterministic alternatives.
  - E.g., checking if types $\mathtt{Bool} * (\mathtt{Int}|\mathtt{Float})$, $\mathtt{Bool} * \mathtt{Int} \,|\, \mathtt{Bool} * \mathtt{Float}$ are semantically equal cannot be done in the standard way — that is by a structural induction.
- The above example also shows that when we have natural type alternatives, then term-types are not isomorphic with their semantics (because the terms are different, but they are semantically equivalent).
- Moreover, terms fail to properly model type systems w.r.t. recursive equations.
  - E.g., the equation $X = \mathtt{NIL} \mid \mathtt{Int} * X$ may be solved by 2 types (a list and a stream), but there is 0 or 1 term — depending on the terms definition — that satisfies it.

Outline

1. Motivation
   - Types and their semantics should be isomorphic
   - Some type constructions
   - Terms seem insufficient as types
2. Solution
   - Types as sets with possibly many representations
   - Automata representation

Type Theory

- Alternative type theory
  - we define types as sets of values (of some universe),
  - we define representations of these sets
    (chosen accordingly to our needs).
- One of such representations may be terms.

Type Theory

- Alternative type theory
  - we define types as sets of values (of some universe),
  - we define representations of these sets (chosen accordingly to our needs).
- One of such representations may be terms.

Type Theory

- Alternative type theory
  - we define types as sets of values (of some universe),
  - we define representations of these sets (chosen accordingly to our needs).
- One of such representations may be terms.

Type Theory

- Alternative type theory
  - we define types as sets of values (of some universe),
  - we define representations of these sets
    (chosen accordingly to our needs).
- One of such representations may be terms.

Outline

1. Motivation
   - Types and their semantics should be isomorphic
   - Some type constructions
   - Terms seem insufficient as types
2. Solution
   - Types as sets with possibly many representations
   - Automata representation

Automata Representation

- Generally, values correspond to trees (possibly infinite) or accessible pointed graphs (APG).
- So we can represent sets of values (types) by automata on trees/APGs.
- Mentioned type alternatives may be represented by non-determinism in automata.
- Generally, mentioned recursive types (with two kinds of recursion) may be represented by automata with the Rabin acceptance condition. (Niwiński, 97)
- After some non-essential restriction, recursive types can be represented by weak Büchi automata.
- We have algorithms on these automata that allow to perform type-checking.

Automata Representation

- Generally, values correspond to trees (possibly infinite) or accessible pointed graphs (APG).
- So we can represent sets of values (types) by automata on trees/APGs.
- Mentioned type alternatives may be represented by non-determinism in automata.
- Generally, mentioned recursive types (with two kinds of recursion) may be represented by automata with the Rabin acceptance condition. (Niwiński, 97)
- After some non-essential restriction, recursive types can be represented by weak Büchi automata.
- We have algorithms on these automata that allow to perform type-checking.

Automata Representation

- Generally, values correspond to trees (possibly infinite) or accessible pointed graphs (APG).
- So we can represent sets of values (types) by automata on trees/APGs.
- Mentioned type alternatives may be represented by non-determinism in automata.
- Generally, mentioned recursive types (with two kinds of recursion) may be represented by automata with the Rabin acceptance condition. (Niwiński, 97)
- After some non-essential restriction, recursive types can be represented by weak Büchi automata.
- We have algorithms on these automata that allow to perform type-checking.

Automata Representation

- Generally, values correspond to trees (possibly infinite) or accessible pointed graphs (APG).
- So we can represent sets of values (types) by automata on trees/APGs.
- Mentioned type alternatives may be represented by non-determinism in automata.
- Generally, mentioned recursive types (with two kinds of recursion) may be represented by automata with the Rabin acceptance condition. (Niwiński, 97)
- After some non-essential restriction, recursive types can be represented by weak Büchi automata.
- We have algorithms on these automata that allow to perform type-checking.

Automata Representation

- Generally, values correspond to trees (possibly infinite) or accessible pointed graphs (APG).
- So we can represent sets of values (types) by automata on trees/APGs.
- Mentioned type alternatives may be represented by non-determinism in automata.
- Generally, mentioned recursive types (with two kinds of recursion) may be represented by automata with the Rabin acceptance condition. (Niwiński, 97)
- After some non-essential restriction, recursive types can be represented by weak Büchi automata.
- We have algorithms on these automata that allow to perform type-checking.

Automata Representation

- Generally, values correspond to trees (possibly infinite) or accessible pointed graphs (APG).
- So we can represent sets of values (types) by automata on trees/APGs.
- Mentioned type alternatives may be represented by non-determinism in automata.
- Generally, mentioned recursive types (with two kinds of recursion) may be represented by automata with the Rabin acceptance condition. (Niwiński, 97)
- After some non-essential restriction, recursive types can be represented by weak Büchi automata.
- We have algorithms on these automata that allow to perform type-checking.

Example of Checking Types

- Let *fst* be a projection of products to their first axis, and
- x be of the type $int * int$ or $float * float$, then
- $fst\ x$ is of the type int or $float$.

Example of Checking Types

Assume

- Ω is a universal type,
- $\Gamma = fst : (int|float) * \Omega \to int|float,$ <- this must be inferred
- $\Gamma' = fst : (int|float) * \Omega \to int|float,\ \ x : int * int \,|\, float * float\ .$

Then we can do the following type checking

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}(int|float) * \Omega \to int|float \ \subseteq \\ (int|float) * \Omega \to int|float\end{array}}{\Gamma' \vdash fst : (int|float) * \Omega \to int|float} \qquad \dfrac{int * int|float * float \subseteq (int|float) * \Omega}{\Gamma' \vdash x : (int|float) * \Omega}}{\Gamma' \vdash fst\ x : int|float}}{\Gamma \vdash (\lambda x.\ fst\ x) : int * int \,|\, float * float \to int|float}$$

where $\leq$ may be computed by translating a term representation to an automata representation and checking inclusion between languages accepted by automata.

Problems

- Type-checking by Rabin automata is expensive,
  but weak Büchi automata may be in the practical range.
- Non-standard classes of automata are needed.
  - Automata on infinitely branched trees/APGs are needed to support functional values.
    - Such automata expose problems to complement them.
  - Automata over infinite alphabets may be needed to properly handle tuples and records.
- Some kind of automata unification algorithm is needed to perform type-checking for polymorphic types.

Problems

- Type-checking by Rabin automata is expensive, but weak Büchi automata may be in the practical range.
- Non-standard classes of automata are needed.
  - Automata on infinitely branched trees/APGs are needed to support functional values.
    - Such automata expose problems to complement them.
  - Automata over infinite alphabets may be needed to properly handle tuples and records.
- Some kind of automata unification algorithm is needed to perform type-checking for polymorphic types.

Problems

- Type-checking by Rabin automata is expensive, but weak Büchi automata may be in the practical range.
- Non-standard classes of automata are needed.
  - Automata on infinitely branched trees/APGs are needed to support functional values.
    - Such automata expose problems to complement them.
  - Automata over infinite alphabets may be needed to properly handle tuples and records.
- Some kind of automata unification algorithm is needed to perform type-checking for polymorphic types.

Problems

- Type-checking by Rabin automata is expensive, but weak Büchi automata may be in the practical range.
- Non-standard classes of automata are needed.
  - Automata on infinitely branched trees/APGs are needed to support functional values.
    - Such automata expose problems to complement them.
  - Automata over infinite alphabets may be needed to properly handle tuples and records.
- Some kind of automata unification algorithm is needed to perform type-checking for polymorphic types.

Problems

- Type-checking by Rabin automata is expensive,
  but weak Büchi automata may be in the practical range.
- Non-standard classes of automata are needed.
  - Automata on infinitely branched trees/APGs are needed to support functional values.
    - Such automata expose problems to complement them.
  - Automata over infinite alphabets may be needed to properly handle tuples and records.
- Some kind of automata unification algorithm is needed to perform type-checking for polymorphic types.

Problems

- Type-checking by Rabin automata is expensive, but weak Büchi automata may be in the practical range.
- Non-standard classes of automata are needed.
  - Automata on infinitely branched trees/APGs are needed to support functional values.
    - Such automata expose problems to complement them.
  - Automata over infinite alphabets may be needed to properly handle tuples and records.
- Some kind of automata unification algorithm is needed to perform type-checking for polymorphic types.

Problems

- Type-checking by Rabin automata is expensive,
  but weak Büchi automata may be in the practical range.
- Non-standard classes of automata are needed.
  - Automata on infinitely branched trees/APGs are needed to support functional values.
    - Such automata expose problems to complement them.
  - Automata over infinite alphabets may be needed to properly handle tuples and records.
- Some kind of automata unification algorithm is needed to perform type-checking for polymorphic types.