INSTITUT DE RECHERCHE EN INFORMATIQUE FONDAMENTALE

Typing Records, Maps, and Structs

Giuseppe CASTAGNA

Outline

Records are finite maps from “labels” (or “keys”) to values

Focus on two usages for records:

- **structs:** a predefined finite set of labels mapped into values of different types
  e.g., a person: `{name="Alice", age=23, born=(1999,9,30)}`
- **dictionaries/maps:** dynamically generated labels mapped into values of the same type
  e.g., a symbol table `{"x"=(int,block),"y"=(double,global),"foo"=(fun,extern)}`

How to type a language where:

- records mix both usages
- types include union, intersection, and negation types

Problems:

- How to define and decide subtyping
- How to type operations on records

Not in this talk: how to type mutable records (see paper’s appendix)

Records

Short history

- Proposed in 1965 by C.A.R. Hoare for ALGOL
- rapidly adopted by ALGOL, but also by Simula 67 yielding the concepts of object

Records

Short history

- Proposed in 1965 by C.A.R. Hoare for ALGOL
- rapidly adopted by ALGOL, but also by Simula 67 yielding the concepts of object
- Since then they are used for a wide palette of purposes such as:
  - structured data (as originally proposed by Hoare),
  - relations (as in relational databases),
  - maps (a.k.a., associative arrays, dictionaries, hashes, lookup-tables),
  - modules,
  - objects,
  - configuration files,
  - data serialization

Records

Short history

- Proposed in 1965 by C.A.R. Hoare for ALGOL
- rapidly adopted by ALGOL, but also by Simula 67 yielding the concepts of object
- Since then they are used for a wide palette of purposes such as:
  - structured data (as originally proposed by Hoare),
  - relations (as in relational databases),
  - maps (a.k.a., associative arrays, dictionaries, hashes, lookup-tables),
  - modules,
  - objects,
  - configuration files,
  - data serialization

Maps:

- All keys of a single map have the same type and so do the values they are mapped to.
- It is not necessary to know all the keys at compile time: they can be dynamically discovered.
- It is sensible to give a default value.
- Keys may be indexed: it is possible to iterate over them.
- Keys are values: keys used for map selection are results of expressions.
- Accessing a key that is not defined yields a specific value (e.g., `nil`, `null`, ..)
- Maps are (often) mutable data structures with mutable components.

Structs:

- Different keys in the same structure can be mapped to values of different types.
- Keys may be restricted to a specific set (e.g., strings, atoms, identifiers, ...).
- It is necessary to know all the different keys at compile time.
- Keys do not support indexing.
- Keys are not necessarily values. Struct access is by nominal keys
- Accessing a key that is not defined yields an error.
- Structs are (often) immutable data-structures but may have mutable components.

Maps:

- All keys of a single map have the same type and so do the values they are mapped to.
- It is not necessary to know all the keys at compile time: they can be dynamically discovered.
- It is sensible to give a default value.
- Keys may be indexed: it is possible to iterate over them.
- Keys are values: keys used for map selection are results of expressions.
- Accessing a key that is not defined yields a specific value (e.g., `nil`, `null`, ..)
- Maps are (often) mutable data structures with mutable components.

Structs:

- Different keys in the same structure can be mapped to values of different types.
- Keys may be restricted to a specific set (e.g., strings, atoms, identifiers, ...).
- It is necessary to know all the different keys at compile time.
- Keys do not support indexing.
- Keys are not necessarily values. Struct access is by nominal keys
- Accessing a key that is not defined yields an error.
- Structs are (often) immutable data-structures but may have mutable components.

Expressions

$$e \quad ::= \quad c \mid x \mid \lambda x.e \mid ee \mid \ldots \qquad \text{lambda-calculus}$$

Expressions

$$
\begin{array}{rcll}
e & ::= & c \mid x \mid \lambda x.e \mid ee \mid \ldots & \text{lambda-calculus} \\
 & \mid & \{\ell = e, \ldots, \ell = e\} & \text{record} \\
 & \mid & e.\ell & \text{selection} \\
 & \mid & e \backslash \ell & \text{deletion} \\
 & \mid & e + e & \text{concatenation}
\end{array}
$$

Expressions

$$
\begin{array}{llll}
e & ::= & c \mid x \mid \lambda x.e \mid ee \mid \ldots & \text{lambda-calculus} \\
 & \mid & \{\ell = e, \ldots, \ell = e\} & \text{record} \\
 & \mid & e.\ell & \text{selection} \\
 & \mid & e \backslash \ell & \text{deletion} \\
 & \mid & e + e & \text{concatenation}
\end{array}
\left.\vphantom{\begin{array}{l} a \\ b \\ c \end{array}}\right\} \text{struct operations}
$$

Expressions

$$
\begin{array}{rcll}
e & ::= & c \mid x \mid \lambda x.e \mid ee \mid \ldots & \text{lambda-calculus} \\
& \mid & \{\ell = e, \ldots, \ell = e\} & \text{record} \\
& \mid & e.\ell & \text{selection} \\
& \mid & e\backslash\ell & \text{deletion} \\
& \mid & e + e & \text{concatenation} \\
& \mid & e.[e] & \text{map access} \\
& \mid & e\backslash[e] & \text{map deletion} \\
& \mid & e{\leftarrow}\langle[e]{=}e\rangle & \text{map update}
\end{array}
$$

struct operations

map operations

Expressions

$$
\begin{array}{rcll}
e & ::= & c \mid x \mid \lambda x.e \mid ee \mid \ldots & \text{lambda-calculus} \\
 & \mid & \{\ell = e, \ldots, \ell = e\} & \text{record} \\
 & \mid & e.\ell & \text{selection} \\
 & \mid & e \backslash \ell & \text{deletion} \\
 & \mid & e + e & \text{concatenation} \\
 & \mid & e.[e] & \text{map access} \\
 & \mid & e \backslash [e] & \text{map deletion} \\
 & \mid & e \leftarrow \langle [e] = e \rangle & \text{map update}
\end{array}
$$

$\left.\begin{array}{l}\text{selection}\\ \text{deletion}\\ \text{concatenation}\end{array}\right\}$ struct operations

$\left.\begin{array}{l}\text{map access}\\ \text{map deletion}\\ \text{map update}\end{array}\right\}$ map operations

Expressions

$$
\begin{array}{rcll}
e & ::= & c \mid x \mid \lambda x.e \mid ee \mid \ldots & \text{lambda-calculus} \\
& \mid & \{\ell = e, \ldots, \ell = e\} & \text{record} \\
& \mid & e.\ell & \text{selection} \\
& \mid & e \backslash \ell & \text{deletion} \\
& \mid & e + e & \text{concatenation} \\
& \mid & e.[e] & \text{map access} \\
& \mid & e \backslash [e] & \text{map deletion} \\
& \mid & e \leftarrow \langle [e] = e \rangle & \text{map update}
\end{array}
\quad
\begin{array}{l}
\left.\begin{array}{l} \\ \\ \\ \end{array}\right\} \text{struct operations} \\
\left.\begin{array}{l} \\ \\ \\ \end{array}\right\} \text{map operations}
\end{array}
$$

$$\{\ell_1 = v_1, \ldots, \ell_n = v_n\}.\ell \quad \rightsquigarrow \quad v_i \qquad \text{if } \ell \equiv \ell_i \text{ for some } i \in [1..n]$$

$$\{\ell_1 = v_1, \ldots, \ell_n = v_n\}.[\ell] \quad \rightsquigarrow \quad \begin{cases} v_i & \text{if } \ell \equiv \ell_i \text{ for some } i \in [1..n] \\ \texttt{nil} & \text{otherwise} \end{cases}$$

Expressions

$$
\begin{array}{rcll}
e & ::= & c \mid x \mid \lambda x.e \mid ee \mid \ldots & \text{lambda-calculus} \\
 & \mid & \{\ell = e, \ldots, \ell = e\} & \text{record} \\
 & \mid & e.\ell & \text{selection} \\
 & \mid & e\backslash\ell & \text{deletion} \\
 & \mid & e + e & \text{concatenation} \\
 & \mid & e.[e] & \text{map access} \\
 & \mid & e\backslash[e] & \text{map deletion} \\
 & \mid & e \leftarrow \langle [e] = e \rangle & \text{map update}
\end{array}
$$

selection, deletion, concatenation: struct operations

map access, map deletion, map update: map operations

$$\{\ell_1 = v_1, \ldots, \ell_n = v_n\}.\ell \quad \rightsquigarrow \quad v_i \quad \text{if } \ell \equiv \ell_i \text{ for some } i \in [1..n]$$

$$\{\ell_1 = v_1, \ldots, \ell_n = v_n\}.[\ell] \quad \rightsquigarrow \quad \begin{cases} v_i & \text{if } \ell \equiv \ell_i \text{ for some } i \in [1..n] \\ \texttt{nil} & \text{otherwise} \end{cases}$$

Types

Requirements

- Record types cover both map and struct usages
- Union, intersection, negation types (application to Elixir and Ballerina)

Types

Requirements

- Record types cover both map and struct usages
- Union, intersection, negation types (application to Elixir and Ballerina)

Examples in Elixir

Types

Requirements

- Record types cover both map and struct usages
- Union, intersection, negation types (application to Elixir and Ballerina)

Examples in Elixir

```
type t= %{:foo => atom,
          optional(:bar) => atom,
          optional(atom) => integer}

# atoms = colon-prefixed user-defined constants such as :foo and :bar
```

Types

Requirements

- Record types cover both map and struct usages
- Union, intersection, negation types (application to Elixir and Ballerina)

Examples in Elixir

```
type t= %{:foo => atom,
          optional(:bar) => atom,
          optional(atom) => integer}

# atoms = colon-prefixed user-defined constants such as :foo and :bar

type r= %{:output => :ok,  :socket => socket} or
        %{:output => :err, :message => (:timeout or {:delay,integer})}
```

Types

Requirements

- Record types cover both map and struct usages
- Union, intersection, negation types (application to Elixir and Ballerina)

Examples in Elixir

```
type t= %{:foo => atom,
          optional(:bar) => atom,
          optional(atom) => integer}

# atoms = colon-prefixed user-defined constants such as :foo and :bar

type r= %{:output => :ok,  :socket => socket} or
        %{:output => :err, :message => (:timeout or {:delay,integer})}
```

union types

Examples (continued)

```
def m :: %{:foo => atom,
           optional(:bar) => atom,
           optional(atom) => integer}
```

Examples (continued)

```
def m :: %{:foo => atom,
           optional(:bar) => atom,
           optional(atom) => integer}

m.foo                        /* colon can be omitted in dot selection */
# type atom
```

Examples (continued)

```
def m :: %{:foo => atom,
           optional(:bar) => atom,
           optional(atom) => integer}

m.foo                        /* colon can be omitted in dot selection */
# type atom
m.bar
#=> Type error: key :bar may be undefined
```

Examples (continued)

```
def m :: %{:foo => atom,
           optional(:bar) => atom,
           optional(atom) => integer}

m.foo                        /* colon can be omitted in dot selection */
# type atom
m.bar
#=> Type error: key :bar may be undefined
m[:bar]
# type atom or nil
```

Examples (continued)

```
def m :: %{:foo => atom,
           optional(:bar) => atom,
           optional(atom) => integer}

m.foo                         /* colon can be omitted in dot selection */
# type atom
m.bar
#=> Type error: key :bar may be undefined
m[:bar]
# type atom or nil
m[:other]
# type integer or nil
```

Examples (continued)

```
def m :: %{:foo => atom,
           optional(:bar) => atom,
           optional(atom) => integer}

m.foo                              /* colon can be omitted in dot selection */
# type atom
m.bar
#=> Type error: key :bar may be undefined
m[:bar ]
# type atom or nil
m[ m.foo ]
# type atom or integer or nil
```

Examples (continued)

```
def m :: %{:foo => atom,
           optional(:bar) => atom,
           optional(atom) => integer}

m.foo                        /* colon can be omitted in dot selection */
# type atom
m.bar
#=> Type error: key :bar may be undefined
m[:bar ]
# type atom or nil
m[ m.foo ]
# type atom or integer or nil
m[41+1]
# type nil   (... and a warning)
```

Type Syntax

Let $\ell \in \mathscr{L}$ range over keys:

$$
\begin{array}{llcll}
\textbf{Expr types} & t & ::= & \mathsf{Int} \mid \mathsf{Bool} \mid t \to t \mid \ldots & \text{type constructors} \\
 & & \mid & t \vee t \mid t \wedge t \mid \neg t \mid \mathbb{1} \mid \mathbb{0} & \text{set-theoretic types} \\
 & & \mid & \{f, \ldots, f\} & \text{record types}
\end{array}
$$

Type Syntax

Let $\ell \in \mathscr{L}$ range over keys:

$$
\begin{array}{llcll}
\textbf{Expr types} & t & ::= & \mathsf{Int} \mid \mathsf{Bool} \mid t \to t \mid \ldots & \text{type constructors} \\
 & & \mid & t \vee t \mid t \wedge t \mid \neg t \mid \mathbb{1} \mid \mathbb{0} & \text{set-theoretic types} \\
 & & \mid & \{f, \ldots, f\} & \text{record types} \\
\\
\textbf{Field types} & f & ::= & \ell = t & \text{mandatory field type} \\
 & & \mid & \ell \Rightarrow t & \text{optional field type}
\end{array}
$$

Type Syntax

Let $\ell \in \mathscr{L}$ range over keys:

$$
\begin{array}{llcll}
\textbf{Expr types} & t & ::= & \mathsf{Int} \mid \mathsf{Bool} \mid t \to t \mid \ldots & \text{type constructors} \\
 & & \mid & t \vee t \mid t \wedge t \mid \neg t \mid \mathbb{1} \mid \mathbb{0} & \text{set-theoretic types} \\
 & & \mid & \{f, \ldots, f\} & \text{record types} \\
 & & & & \\
\textbf{Field types} & f & ::= & \ell = t & \text{mandatory field type} \\
 & & \mid & \ell \Rightarrow t & \text{optional field type}
\end{array}
\left.\vphantom{\begin{array}{l} a \\ b \end{array}}\right\} \text{struct types}
$$

Type Syntax

Let $\ell \in \mathscr{L}$ range over keys:

$$\begin{array}{llcll}
\textbf{Expr types} & t & ::= & \mathsf{Int} \mid \mathsf{Bool} \mid t \to t \mid \dots & \text{type constructors} \\
& & \mid & t \vee t \mid t \wedge t \mid \neg t \mid \mathbb{1} \mid \mathbb{0} & \text{set-theoretic types} \\
& & \mid & \{f, \dots, f\} & \text{record types} \\
& & & & \\
\textbf{Field types} & f & ::= & \ell = t & \left.\begin{array}{l}\text{mandatory field type} \\ \text{optional field type}\end{array}\right\} \text{struct types} \\
& & \mid & \ell \Rightarrow t & \\
& & \mid & k \Rightarrow t & \left.\begin{array}{l}\text{map type} \\ \\ \text{any key} \\ \text{different flavors}\end{array}\right\} \text{map types} \\
& & & & \\
\textbf{Key types} & k & ::= & _ & \\
& & \mid & \dots &
\end{array}$$

Type Syntax

Let $\ell \in \mathscr{L}$ range over keys:

$$
\begin{array}{llcll}
\textbf{Expr types} & t & ::= & \mathrm{Int} \mid \mathrm{Bool} \mid t \to t \mid \dots & \text{type constructors} \\
 & & \mid & t \vee t \mid t \wedge t \mid \neg t \mid \mathbb{1} \mid \mathbb{0} & \text{set-theoretic types} \\
 & & \mid & \{f, \dots, f\} & \text{record types} \\
\textbf{Field types} & f & ::= & \ell = t & \text{mandatory field type} \\
 & & \mid & \ell \Rightarrow t & \text{optional field type} \\
 & & \mid & k \Rightarrow t & \text{map type} \\
\textbf{Key types} & k & ::= & \_ & \text{any key} \\
 & & \mid & \dots & \text{different flavors}
\end{array}
$$

struct types

map types

Ballerina: $k ::= \_$ with $\mathscr{L} = \{v \mid v \text{ is a string}\}$

TypeScript: $k ::= \_ \mid \texttt{string} \mid \texttt{number} \mid \texttt{symbol}$ with $\mathscr{L} = \{v \mid v \text{ is a value}\}$

Erlang TypeSpec: $k ::= \_ \mid t$ with $\mathscr{L} = \{v \mid v \text{ is a value}\}$

Encodings

Open record type

$\{f_1, \ldots, f_n, \_ \Rightarrow \mathbb{1}\}$ (records containing at least the fields $f_1, \ldots, f_n$)

Closed record type

$\{f_1, \ldots, f_n, \_ \Rightarrow \mathbb{0}\}$ (records containing exactly the fields $f_1, \ldots, f_n$)

Encodings

Open record type

$\{f_1, \ldots, f_n, \_ \Rightarrow \mathbb{1}\}$ (records containing <u>at least</u> the fields $f_1, \ldots, f_n$)

Closed record type

$\{f_1, \ldots, f_n, \_ \Rightarrow \mathbb{0}\}$ (records containing <u>exactly</u> the fields $f_1, \ldots, f_n$)

Top record type

$\{\_ \Rightarrow \mathbb{1}\}$ (all records)

Encodings

Open record type

$\{f_1, \ldots, f_n, \_ \Rightarrow \mathbb{1}\}$ (records containing <u>at least</u> the fields $f_1, \ldots, f_n$)

Closed record type

$\{f_1, \ldots, f_n, \_ \Rightarrow \mathbb{0}\}$ (records containing <u>exactly</u> the fields $f_1, \ldots, f_n$)

Top record type

$\{\_ \Rightarrow \mathbb{1}\}$ (all records)

Elixir example

```
type t= %{:foo => atom,
          optional(:bar) => atom,
          optional(atom) => integer }
```

$$t = \{\texttt{:foo} = \text{atom}, \texttt{:bar} \Rightarrow \text{atom}, \text{atom} \Rightarrow \text{int}, \_ \Rightarrow \mathbb{0}\}$$

Encodings

Open record type

$\{f_1, \ldots, f_n, \_ \Rightarrow \mathbb{1}\}$ (records containing at least the fields $f_1, \ldots, f_n$)

Closed record type

$\{f_1, \ldots, f_n, \_ \Rightarrow \mathbb{0}\}$ (records containing exactly the fields $f_1, \ldots, f_n$)

Top record type

$\{\_ \Rightarrow \mathbb{1}\}$ (all records)

Elixir example

```
type t= %{:foo => atom,
          optional(:bar) => atom,
          optional(atom) => integer }
```

$$t = \{\texttt{:foo} = \text{atom}, \texttt{:bar} \Rightarrow \text{atom}, \text{atom} \Rightarrow \text{int}, \_ \Rightarrow \mathbb{0}\}$$

Encodings

Open record type

$\{f_1, \ldots, f_n, \_ \Rightarrow \mathbb{1}\}$ (records containing <u>at least</u> the fields $f_1, \ldots, f_n$)

Closed record type

$\{f_1, \ldots, f_n, \_ \Rightarrow \mathbb{0}\}$ (records containing <u>exactly</u> the fields $f_1, \ldots, f_n$)

Top record type

$\{\_ \Rightarrow \mathbb{1}\}$ (all records)

Elixir example

```
type t= %{:foo => atom,
          optional(:bar) => atom,
          optional(atom) => integer, ... }
```

$$t = \{\texttt{:foo} = \text{atom}, \texttt{:bar} \Rightarrow \text{atom}, \text{atom} \Rightarrow \text{int}, \_ \Rightarrow \mathbb{1}\}$$

Type System

$$[\text{Recd}]\ \frac{\Gamma \vdash e_1 : t_1 \quad \cdots \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash \{\ell_1 = e_1, \ldots, \ell_n = e_n\} : \boldsymbol{\{}\ell_1 = t_1, \ldots, \ell_n = t_n, \_ \Rightarrow \mathbb{0}\boldsymbol{\}}}$$

Type System

$$[\text{Recd}]\ \frac{\Gamma \vdash e_1 : t_1 \quad \cdots \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash \{\ell_1 = e_1, \ldots, \ell_n = e_n\} : \mathbf{\{}\ell_1 = t_1, \ldots, \ell_n = t_n, \_ \Rightarrow \mathbb{0}\mathbf{\}}}$$

$$[\text{Sel}]\ \frac{\Gamma \vdash e : t \leq \mathbf{\{}\ell = \mathbb{1}, \_ \Rightarrow \mathbb{1}\mathbf{\}}}{\Gamma \vdash e.\ell : t.\ell} \qquad [\text{Del}]\ \frac{\Gamma \vdash e : t \leq \mathbf{\{}\_ \Rightarrow \mathbb{1}\mathbf{\}}}{\Gamma \vdash e\backslash\ell : t\backslash\ell}$$

$$[\text{Conc}]\ \frac{\Gamma \vdash e_1 : t_1 \leq \mathbf{\{}\_ \Rightarrow \mathbb{1}\mathbf{\}} \qquad \Gamma \vdash e_2 : t_2 \leq \mathbf{\{}\_ \Rightarrow \mathbb{1}\mathbf{\}}}{\Gamma \vdash e_1 \mathbin{\boldsymbol{+}} e_2 : t_1 \mathbin{\boldsymbol{+}} t_2}$$

Type System

$$[\text{Recd}]\ \frac{\Gamma \vdash e_1 : t_1 \quad \cdots \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash \{\ell_1 = e_1, \ldots, \ell_n = e_n\} : \boldsymbol{\{}\ell_1 = t_1, \ldots, \ell_n = t_n,\ \_ \Rightarrow \mathbb{0}\boldsymbol{\}}}$$

$$[\text{Sel}]\ \frac{\Gamma \vdash e : t \leq \boldsymbol{\{}\ell = \mathbb{1},\ \_ \Rightarrow \mathbb{1}\boldsymbol{\}}}{\Gamma \vdash e.\ell : t.\ell} \qquad [\text{Del}]\ \frac{\Gamma \vdash e : t \leq \boldsymbol{\{}\_ \Rightarrow \mathbb{1}\boldsymbol{\}}}{\Gamma \vdash e\backslash\ell : t\backslash\ell}$$

$$[\text{Conc}]\ \frac{\Gamma \vdash e_1 : t_1 \leq \boldsymbol{\{}\_ \Rightarrow \mathbb{1}\boldsymbol{\}} \qquad \Gamma \vdash e_2 : t_2 \leq \boldsymbol{\{}\_ \Rightarrow \mathbb{1}\boldsymbol{\}}}{\Gamma \vdash e_1 + e_2 : t_1 + t_2}$$

$$[\text{M-Sel}]\ \frac{\Gamma \vdash e_1 : t_1 \leq \boldsymbol{\{}\_ \Rightarrow 1\boldsymbol{\}} \qquad \Gamma \vdash e_2 : t_2 \leq \mathscr{L}}{\Gamma \vdash e_1.[e_2] : t_1.[t_2]} \qquad [\text{M-Del}]\ \frac{\Gamma \vdash e_1 : t_1 \leq \boldsymbol{\{}\_ \Rightarrow 1\boldsymbol{\}} \qquad \Gamma \vdash e_2 : t_2 \leq \mathscr{L}}{\Gamma \vdash e_1\backslash[e_2] : t_1\backslash[t_2]}$$

$$[\text{M-Upd}]\ \frac{\Gamma \vdash e_1 : t_1 \leq \boldsymbol{\{}\_ \Rightarrow 1\boldsymbol{\}} \qquad \Gamma \vdash e_2 : t_2 \leq \mathscr{L} \qquad \Gamma \vdash e_3 : t_3}{\Gamma \vdash e_1 {\leftarrow} \langle [e_2] {=} e_3 \rangle : t_1 {\leftarrow} \langle [t_2] {=} t_3 \rangle}$$

Type System

$$[\text{Recd}]\ \frac{\Gamma \vdash e_1 : t_1 \quad \cdots \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash \{\ell_1 = e_1, \ldots, \ell_n = e_n\} : \{\ell_1 = t_1, \ldots, \ell_n = t_n, \_ \Rightarrow \mathbb{0}\}}$$

$$[\text{Sel}]\ \frac{\Gamma \vdash e : t \leq \{\ell = \mathbb{1}, \_ \Rightarrow \mathbb{1}\}}{\Gamma \vdash e.\ell : t.\ell} \qquad [\text{Del}]\ \frac{\Gamma \vdash e : t \leq \{\_ \Rightarrow \mathbb{1}\}}{\Gamma \vdash e \backslash \ell : t \backslash \ell}$$

$$[\text{Conc}]\ \frac{\Gamma \vdash e_1 : t_1 \leq \{\_ \Rightarrow \mathbb{1}\} \qquad \Gamma \vdash e_2 : t_2 \leq \{\_ \Rightarrow \mathbb{1}\}}{\Gamma \vdash e_1 + e_2 : t_1 + t_2}$$

$$[\text{M-Sel}]\ \frac{\Gamma \vdash e_1 : t_1 \leq \{\_ \Rightarrow \mathbb{1}\} \qquad \Gamma \vdash e_2 : t_2 \leq \mathscr{L}}{\Gamma \vdash e_1.[e_2] : t_1.[t_2]} \qquad [\text{M-Del}]\ \frac{\Gamma \vdash e_1 : t_1 \leq \{\_ \Rightarrow \mathbb{1}\} \qquad \Gamma \vdash e_2 : t_2 \leq \mathscr{L}}{\Gamma \vdash e_1 \backslash [e_2] : t_1 \backslash [t_2]}$$

$$[\text{M-Upd}]\ \frac{\Gamma \vdash e_1 : t_1 \leq \{\_ \Rightarrow \mathbb{1}\} \qquad \Gamma \vdash e_2 : t_2 \leq \mathscr{L} \qquad \Gamma \vdash e_3 : t_3}{\Gamma \vdash e_1 \leftarrow \langle [e_2] = e_3 \rangle : t_1 \leftarrow \langle [t_2] = t_3 \rangle}$$

Type System

1. define subtyping

$$[\text{Recd}]\ \frac{\Gamma \vdash e_1 : t_1 \quad \cdots \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash \{\ell_1 = e_1, \dots, \ell_n = e_n\} : \{\ell_1 = t_1, \dots, \ell_n = t_n, \_ \Rightarrow \mathbb{0}\}}$$

$$[\text{Sel}]\ \frac{\Gamma \vdash e : t \leq \{\ell = \mathbb{1}, \_ \Rightarrow \mathbb{1}\}}{\Gamma \vdash e.\ell : t.\ell} \qquad [\text{Del}]\ \frac{\Gamma \vdash e : t \leq \{\_ \Rightarrow \mathbb{1}\}}{\Gamma \vdash e\backslash\ell : t\backslash\ell}$$

$$[\text{Conc}]\ \frac{\Gamma \vdash e_1 : t_1 \leq \{\_ \Rightarrow \mathbb{1}\} \qquad \Gamma \vdash e_2 : t_2 \leq \{\_ \Rightarrow \mathbb{1}\}}{\Gamma \vdash e_1 + e_2 : t_1 + t_2}$$

$$[\text{M-Sel}]\ \frac{\Gamma \vdash e_1 : t_1 \leq \{\_ \Rightarrow 1\} \qquad \Gamma \vdash e_2 : t_2 \leq \mathscr{L}}{\Gamma \vdash e_1.[e_2] : t_1.[t_2]} \qquad [\text{M-Del}]\ \frac{\Gamma \vdash e_1 : t_1 \leq \{\_ \Rightarrow 1\} \qquad \Gamma \vdash e_2 : t_2 \leq \mathscr{L}}{\Gamma \vdash e_1\backslash[e_2] : t_1\backslash[t_2]}$$

$$[\text{M-Upd}]\ \frac{\Gamma \vdash e_1 : t_1 \leq \{\_ \Rightarrow 1\} \qquad \Gamma \vdash e_2 : t_2 \leq \mathscr{L} \qquad \Gamma \vdash e_3 : t_3}{\Gamma \vdash e_1 \leftarrow \langle [e_2] = e_3 \rangle : t_1 \leftarrow \langle [t_2] = t_3 \rangle}$$

Type System

1. define subtyping
2. define typeoperators

$$\text{[Recd]}\ \frac{\Gamma \vdash e_1 : t_1 \quad \cdots \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash \{\ell_1 = e_1, \dots, \ell_n = e_n\} : \{\ell_1 = t_1, \dots, \ell_n = t_n, \_ \Rightarrow \mathbb{0}\}}$$

$$\text{[Sel]}\ \frac{\Gamma \vdash e : t \leq \{\ell = \mathbb{1}, \_ \Rightarrow \mathbb{1}\}}{\Gamma \vdash e.\ell : t.\ell} \qquad \text{[Del]}\ \frac{\Gamma \vdash e : t \leq \{\_ \Rightarrow \mathbb{1}\}}{\Gamma \vdash e \backslash \ell : t \backslash \ell}$$

$$\text{[Conc]}\ \frac{\Gamma \vdash e_1 : t_1 \leq \{\_ \Rightarrow \mathbb{1}\} \qquad \Gamma \vdash e_2 : t_2 \leq \{\_ \Rightarrow \mathbb{1}\}}{\Gamma \vdash e_1 + e_2 : t_1 + t_2}$$

$$\text{[M-Sel]}\ \frac{\Gamma \vdash e_1 : t_1 \leq \{\_ \Rightarrow 1\} \qquad \Gamma \vdash e_2 : t_2 \leq \mathscr{L}}{\Gamma \vdash e_1.[e_2] : t_1.[t_2]} \qquad \text{[M-Del]}\ \frac{\Gamma \vdash e_1 : t_1 \leq \{\_ \Rightarrow 1\} \qquad \Gamma \vdash e_2 : t_2 \leq \mathscr{L}}{\Gamma \vdash e_1 \backslash [e_2] : t_1 \backslash [t_2]}$$

$$\text{[M-Upd]}\ \frac{\Gamma \vdash e_1 : t_1 \leq \{\_ \Rightarrow 1\} \qquad \Gamma \vdash e_2 : t_2 \leq \mathscr{L} \qquad \Gamma \vdash e_3 : t_3}{\Gamma \vdash e_1 \leftarrow \langle [e_2] = e_3 \rangle : t_1 \leftarrow \langle [t_2] = t_3 \rangle}$$

How to proceed

Subtyping

1. See record values as particular functions (QC-functions)
2. Interpret types as sets of values $\Rightarrow$ record types as sets of QC-functions
3. Define subtyping as set containment
4. Deduce from the set-theoretic interpretation how to decide record subtyping
5. Derive a backtracking-free algorithm

Type operators

1. Problem: handle negated record types
2. Solution: Embed negation in a new record type atom so that every subtype of $\{\_ \Rightarrow \mathbb{1}\}$ is equivalent to a union of these atoms
3. Define type operators on unions of these atoms

Simpler case: Ballerina's maps

$$
\begin{array}{rlll}
\textbf{Record types} & \mathrm{R} & ::= & \{f, \ldots, f\} \\
\textbf{Field types} & f & ::= & \ell = t \;\mid\; \ell \Rightarrow t \;\mid\; \_ \Rightarrow t
\end{array}
$$

Simpler case: Ballerina's maps

Record types	R	$::=$	$\{f, \ldots, f\}$
Field types	f	$::=$	$\ell = t \mid \ell \Rightarrow t \mid \_ \Rightarrow t$

Definition (Quasi-Constant function)

A QC-function f in $X \to Y$ is a total function that is constant but on a finite subset of X

- $f = \{\![x_1 = y_1, \ldots, x_n = y_n, \_ = y_o]\!\}$ (representation of f)
- $\mathrm{dom}(f) = \{x_1, \ldots, x_n\}$ (domain of f)
- $\mathrm{deflt}(f) = y_o$ (default value of f)

Simpler case: Ballerina's maps

$$\begin{array}{rlll} \textbf{Record types} & \mathrm{R} & ::= & \{f, \ldots, f\} \\ \textbf{Field types} & f & ::= & \ell = t \mid \ell \Rightarrow t \mid \_ \Rightarrow t \end{array}$$

Definition (Quasi-Constant function)

A QC-function $\not\ell$ in $X \to Y$ is a total function that is constant but on a finite subset of X

- $\not\ell = \{\!\![x_1 = y_1\,, \ldots, x_n = y_n\,, \_ = y_o]\!\!\}$ (representation of $\not\ell$)
- $\mathrm{dom}(\not\ell) = \{x_1, \ldots, x_n\}$ (domain of $\not\ell$)
- $\mathrm{deflt}(\not\ell) = y_o$ (default value of $\not\ell$)

- A Record value r is a QC-function in $\mathscr{L} \to \mathrm{Values} \cup \{\bot\}$ such that $\mathrm{deflt}(r) = \bot$
- A Record type R is a QC-function in $\mathscr{L} \to \mathrm{Types} \cup \{\bot\}$

Simpler case: Ballerina's maps

$$\begin{array}{rrcl} \textbf{Record types} & \mathrm{R} & ::= & \{f, \ldots, f\} \\ \textbf{Field types} & f & ::= & \ell = t \mid \ell \Rightarrow t \mid \_ \Rightarrow t \end{array}$$

Definition (Quasi-Constant function)

A QC-function $\mathscr{f}$ in $X \to Y$ is a total function that is constant but on a finite subset of X

- $\mathscr{f} = \{\![x_1 = y_1, \ldots, x_n = y_n, \_ = y_o]\!\}$ (representation of $\mathscr{f}$)
- $\mathrm{dom}(\mathscr{f}) = \{x_1, \ldots, x_n\}$ (domain of $\mathscr{f}$)
- $\mathrm{deflt}(\mathscr{f}) = y_o$ (default value of $\mathscr{f}$)

- A Record value r is a QC-function in $\mathscr{L} \to \mathrm{Values} \cup \{\bot\}$ such that $\mathrm{deflt}(r) = \bot$
- A Record type R is a QC-function in $\mathscr{L} \to \mathrm{Types} \cup \{\bot\}$

Set-theoretic interpretation of record types

A record type R is the set of all QC-functions $r : \mathscr{L} \to \mathrm{Values} \cup \{\bot\}$ such that
- $\mathrm{deflt}(r) = \bot$ and
- $r(\ell) \in \mathrm{R}(\ell)$ for all $\ell \in \mathscr{L}$.

Deciding subtyping (simpler case)

1. Deciding subtyping = deciding emptiness: $t_1 \leq t_2 \iff t_1 \wedge \neg t_2 \leq \mathbb{0}$

Deciding subtyping (simpler case)

1. Deciding subtyping = deciding emptiness: $t_1 \leq t_2 \iff t_1 \wedge \neg t_2 \leq \mathbb{0}$
2. Disjunctive normal form: Every type $t \leq \{\_ \Rightarrow \mathbb{1}\}$ is equivalent to a type of the form:

$$\bigvee_{i \in I} \left(\bigwedge_{p \in P_i} \mathrm{R}_p \;\wedge\; \bigwedge_{n \in N_i} \neg \mathrm{R}_n \right) \tag{1}$$

Deciding subtyping (simpler case)

1. Deciding subtyping = deciding emptiness: $t_1 \leq t_2 \iff t_1 \wedge \neg t_2 \leq \mathbb{0}$

2. Disjunctive normal form: Every type $t \leq \{\_ \Rightarrow \mathbb{1}\}$ is equivalent to a type of the form:

$$\bigvee_{i \in I} \left(\bigwedge_{p \in P_i} \mathrm{R}_p \wedge \bigwedge_{n \in N_i} \neg \mathrm{R}_n \right) \tag{1}$$

3. Deciding emptyness of (1):

Deciding subtyping (simpler case)

1. Deciding subtyping = deciding emptiness: $t_1 \leq t_2 \iff t_1 \wedge \neg t_2 \leq \mathbb{0}$

2. Disjunctive normal form: Every type $t \leq \{\_ \Rightarrow \mathbb{1}\}$ is equivalent to a type of the form:

$$\bigvee_{i \in I} \left(\bigwedge_{p \in P_i} \mathrm{R}_p \;\wedge\; \bigwedge_{n \in N_i} \neg \mathrm{R}_n \right) \tag{1}$$

3. Deciding emptyness of (1): $\bigwedge_{p \in P} \mathrm{R}_p \wedge \bigwedge_{n \in N} \neg \mathrm{R}_n \leq \mathbb{0}$

Deciding subtyping (simpler case)

1. Deciding subtyping = deciding emptiness: $t_1 \leq t_2 \iff t_1 \wedge \neg t_2 \leq \mathbb{0}$

2. Disjunctive normal form: Every type $t \leq \{\_ \Rightarrow \mathbb{1}\}$ is equivalent to a type of the form:

$$\bigvee_{i \in I} \left(\bigwedge_{p \in P_i} \mathrm{R}_p \wedge \bigwedge_{n \in N_i} \neg \mathrm{R}_n \right) \tag{1}$$

3. Deciding emptyness of (1): $\bigwedge_{p \in P} \mathrm{R}_p \wedge \bigwedge_{n \in N} \neg \mathrm{R}_n \leq \mathbb{0}$ iff

either $\bigwedge_{p \in P} \mathrm{deflt}(\mathrm{R}_p) = \varnothing$, or $\forall \iota : N \to \bigcup_{i \in P \cup N} \mathrm{dom}(\mathrm{R}_i) \cup \{\_\}$:

$$\left(\exists \ell \in \mathscr{L}. \left(\bigwedge_{p \in P} \mathrm{R}_p(\ell) \subseteq \bigvee_{n \in N | \iota(n) = \ell} \mathrm{R}_n(\ell) \right) \right) \text{ or } \left(\exists n_\circ \in N. \left(\iota(n_\circ) = \_ \right) \text{ and } \left(\bigwedge_{p \in P} \mathrm{deflt}(\mathrm{R}_p) \leq \mathrm{deflt}(\mathrm{R}_{n_\circ}) \right) \right)$$

Deciding subtyping (simpler case)

1. Deciding subtyping = deciding emptiness: $t_1 \leq t_2 \iff t_1 \wedge \neg t_2 \leq \mathbb{0}$

2. Disjunctive normal form: Every type $t \leq \{\_ \Rightarrow \mathbb{1}\}$ is equivalent to a type of the form:

$$\bigvee_{i\in I}\left(\bigwedge_{p\in P_i} \mathrm{R}_p \;\wedge\; \bigwedge_{n\in N_i} \neg \mathrm{R}_n\right) \tag{1}$$

3. Deciding emptyness of (1): $\bigwedge_{p\in P} \mathrm{R}_p \wedge \bigwedge_{n\in N} \neg \mathrm{R}_n \leq \mathbb{0}$ iff

either $\bigwedge_{p\in P} \mathrm{deflt}(\mathrm{R}_p) = \varnothing$, or $\forall\iota : N \to \bigcup_{i\in P\cup N} \mathrm{dom}(\mathrm{R}_i)\cup\{\_\}$:

$$\left(\exists \ell\in\mathscr{L}.\left(\bigwedge_{p\in P} \mathrm{R}_p(\ell) \subseteq \bigvee_{n\in N|\iota(n)=\ell} \mathrm{R}_n(\ell)\right)\right) \text{ or } \left(\exists n_\circ\in N.\left(\iota(n_\circ) = \_\right) \text{ and } \left(\bigwedge_{p\in P} \mathrm{deflt}(\mathrm{R}_p) \leq \mathrm{deflt}(\mathrm{R}_{n_\circ})\right)\right)$$

Deciding subtyping (simpler case)

1. Deciding subtyping = deciding emptiness: $t_1 \leq t_2 \iff t_1 \wedge \neg t_2 \leq \mathbb{0}$

2. Disjunctive normal form: Every type $t \leq \{\_ \Rightarrow \mathbb{1}\}$ is equivalent to a type of the form:

$$\bigvee_{i\in I}\left(\bigwedge_{p\in P_i} \mathrm{R}_p \wedge \bigwedge_{n\in N_i} \neg \mathrm{R}_n\right) \tag{1}$$

3. Deciding emptyness of (1): $\bigwedge_{p\in P} \mathrm{R}_p \wedge \bigwedge_{n\in N} \neg \mathrm{R}_n \leq \mathbb{0}$ iff

either $\bigwedge_{p\in P} \mathrm{deflt}(\mathrm{R}_p) = \varnothing$, or $\forall \iota : N \to \bigcup_{i\in P\cup N} \mathrm{dom}(\mathrm{R}_i) \cup \{\_\}$:

$$\left(\exists \ell \in \mathscr{L}. \left(\bigwedge_{p\in P} \mathrm{R}_p(\ell) \subseteq \bigvee_{n\in N \mid \iota(n)=\ell} \mathrm{R}_n(\ell)\right)\right) \text{ or } \left(\exists n_\circ \in N. \left(\iota(n_\circ) = \_\right) \text{ and } \left(\bigwedge_{p\in P} \mathrm{deflt}(\mathrm{R}_p) \leq \mathrm{deflt}(\mathrm{R}_{n_\circ})\right)\right)$$

Deciding subtyping (simpler case)

1. Deciding subtyping = deciding emptiness: $t_1 \leq t_2 \iff t_1 \wedge \neg t_2 \leq \mathbb{0}$

2. Disjunctive normal form: Every type $t \leq \{\_ \Rightarrow \mathbb{1}\}$ is equivalent to a type of the form:

$$\bigvee_{i \in I} \Big(\bigwedge_{p \in P_i} \mathrm{R}_p \;\wedge\; \bigwedge_{n \in N_i} \neg \mathrm{R}_n \Big) \tag{1}$$

possible backtracking

3. Deciding emptyness of (1): $\bigwedge_{p \in P} \mathrm{R}_p \wedge \bigwedge_{n \in N} \neg \mathrm{R}_n \leq \mathbb{0}$ iff

either $\bigwedge_{p \in P} \mathrm{deflt}(\mathrm{R}_p) = \varnothing$, or $\forall \iota : N \to \bigcup_{i \in P \cup N} \mathrm{dom}(\mathrm{R}_i) \cup \{\_\}$:

$$\Big(\exists \ell \in \mathscr{L} . \big(\bigwedge_{p \in P} \mathrm{R}_p(\ell) \subseteq \bigvee_{n \in N | \iota(n) = \ell} \mathrm{R}_n(\ell) \big) \Big) \text{ or } \Big(\exists n_\circ \in N . \big(\iota(n_\circ) = \_ \big) \text{ and } \big(\bigwedge_{p \in P} \mathrm{deflt}(\mathrm{R}_p) \leq \mathrm{deflt}(\mathrm{R}_{n_\circ}) \big) \Big)$$

Backtracking-free subtyping algorithm (simpler case)

Problem:

Decide the emptiness of $\bigwedge_{R \in P} R \wedge \bigwedge_{R \in N} \neg R$

Backtracking-free subtyping algorithm (simpler case)

Problem:

Decide the emptiness of $\bigwedge_{\mathrm{R} \in P} \mathrm{R} \wedge \bigwedge_{\mathrm{R} \in N} \neg \mathrm{R}$

Algorithm:

1. Move intersection inside for $\bigwedge_{\mathrm{R} \in P} \mathrm{R}$ yielding R_o

Backtracking-free subtyping algorithm (simpler case)

Problem:

Decide the emptiness of $\bigwedge_{R \in P} R \wedge \bigwedge_{R \in N} \neg R$

Algorithm:

1. Move intersection inside for $\bigwedge_{R \in P} R$ yielding R_o
2. Compute emptiness:

$$R_o \wedge \bigwedge_{R \in N} \neg R \leq \mathbb{0} \qquad \iff$$

Backtracking-free subtyping algorithm (simpler case)

Problem:

Decide the emptiness of $\bigwedge_{\mathrm{R}\in P} \mathrm{R} \wedge \bigwedge_{\mathrm{R}\in N} \neg\mathrm{R}$

Algorithm:

1. Move intersection inside for $\bigwedge_{\mathrm{R}\in P} \mathrm{R}$ yielding R_o
2. Compute emptiness:

$$\mathrm{R}_o \wedge \bigwedge_{\mathrm{R}\in N} \neg\mathrm{R} \leq \mathbb{0} \quad\Longleftrightarrow\quad (\mathrm{R}_\circ \leq \mathbb{0}) \text{ or } \Phi(\mathrm{R}_\circ, N)$$

where

$$\begin{aligned}
\Phi(\mathrm{R}_\circ, \varnothing) &= \texttt{false} \\
\Phi(\mathrm{R}_\circ, N \cup \{\mathrm{R}\}) &= \texttt{if } \ \mathrm{deflt}(\mathrm{R}_\circ) \leq \mathrm{deflt}(\mathrm{R}) \\
&\quad\ \texttt{then } \forall \ell \in L\,.\,(\,\mathrm{R}_\circ(\ell) \leq \mathrm{R}(\ell) \text{ or } \Phi(\mathrm{R}_\circ \wedge \{\ell : \texttt{not}(\mathrm{R}(\ell))\}, N)\,) \\
&\quad\ \texttt{else } \ \Phi(\mathrm{R}_\circ, N)
\end{aligned}$$

with $\mathrm{R}_o \not\leq \mathbb{0}$ and $L = \bigcup_{\mathrm{R}\in N\cup\{\mathrm{R}_\circ\}} \mathrm{dom}(\mathrm{R})$.

Type operators

Easy for single record types:

$$
\begin{aligned}
\mathsf{R}.\ell &= \mathsf{R}(\ell) && \text{if } \mathsf{R}(\ell) \wedge \bot \leq \mathbb{0} \\
\mathsf{R}.[t] &= \begin{cases} \bigvee_{\ell \in t} \mathsf{R}(\ell) & \text{if } \bigvee_{\ell \in t} \mathsf{R}(\ell) \wedge \bot \leq \mathbb{0} \\ \bigvee_{\ell \in t} \mathsf{R}(\ell) \setminus \bot \vee \mathtt{nil} & \text{otherwise} \end{cases} \\
(\mathsf{R}_1 + \mathsf{R}_2)(\ell) &= \begin{cases} \mathsf{R}_2(\ell) & \text{if } \mathsf{R}_2(\ell) \wedge \bot \leq \mathbb{0} \\ (\mathsf{R}_2(\ell) \setminus \bot) \vee \mathsf{R}_1(\ell) & \text{otherwise} \end{cases}
\end{aligned}
$$

Type operators

Easy for single record types:

$$
\begin{array}{rcll}
\mathsf{R}.\ell & = & \mathsf{R}(\ell) & \text{if } \mathsf{R}(\ell) \wedge \bot \leq \mathbb{0} \\
\mathsf{R}.[t] & = & \begin{cases} \bigvee_{\ell \in t} \mathsf{R}(\ell) & \\ \bigvee_{\ell \in t} \mathsf{R}(\ell) \setminus \bot \vee \mathtt{nil} & \end{cases} & \begin{array}{l} \text{if } \bigvee_{\ell \in t} \mathsf{R}(\ell) \wedge \bot \leq \mathbb{0} \\ \text{otherwise} \end{array} \\
(\mathsf{R}_1 + \mathsf{R}_2)(\ell) & = & \begin{cases} \mathsf{R}_2(\ell) & \\ (\mathsf{R}_2(\ell) \setminus \bot) \vee \mathsf{R}_1(\ell) & \end{cases} & \begin{array}{l} \text{if } \mathsf{R}_2(\ell) \wedge \bot \leq \mathbb{0} \\ \text{otherwise} \end{array}
\end{array}
$$

A different representation for record types:

Let $\tau ::= t \mid \bot$

$$\mathcal{R} ::= \langle\!\langle (\tau_\ell)_{\ell \in L} \,;\, t_0 \,;\, S \rangle\!\rangle$$

It denotes $\{\ell_1 = \tau_{\ell_1}, \ldots, \ell_n = \tau_{\ell_n}, \_ \Rightarrow t_0\} \setminus \bigvee_{s \in S} \{\ell_1 \Rightarrow \mathbb{1}, \ldots, \ell_n \Rightarrow \mathbb{1}, \_ \Rightarrow \neg s\}$
meaning $\forall s \in S$ there exists a label not in L whose value is of type $s \in S$

Type operators

Easy for single record types:

$$
\begin{aligned}
\mathrm{R}.\ell \quad &= \quad \mathrm{R}(\ell) && \text{if } \mathrm{R}(\ell) \wedge \bot \leq \mathbb{0} \\
\mathrm{R}.[\,t\,] \quad &= \quad \begin{cases} \bigvee_{\ell \in t} \mathrm{R}(\ell) & \text{if } \bigvee_{\ell \in t} \mathrm{R}(\ell) \wedge \bot \leq \mathbb{0} \\ \bigvee_{\ell \in t} \mathrm{R}(\ell) \setminus \bot \vee \texttt{nil} & \text{otherwise} \end{cases} \\
(\mathrm{R}_1 + \mathrm{R}_2)(\ell) \quad &= \quad \begin{cases} \mathrm{R}_2(\ell) & \text{if } \mathrm{R}_2(\ell) \wedge \bot \leq \mathbb{0} \\ (\mathrm{R}_2(\ell) \setminus \bot) \vee \mathrm{R}_1(\ell) & \text{otherwise} \end{cases}
\end{aligned}
$$

A different representation for record types:

Let $\tau ::= t \mid \bot$

usual record type

$$\mathcal{R} ::= \langle\!\langle (\tau_\ell)_{\ell \in L} \,;\, t_0 \,;\, S \rangle\!\rangle$$

It denotes $\{\ell_1 = \tau_{\ell_1}, \ldots, \ell_n = \tau_{\ell_n}, \_ \Rightarrow t_0\} \setminus \bigvee_{s \in S} \{\ell_1 \Rightarrow \mathbb{1}, \ldots, \ell_n \Rightarrow \mathbb{1}, \_ \Rightarrow \neg s\}$

meaning $\forall s \in S$ there exists a label not in L whose value is of type $s \in S$

Type operators

Easy for single record types:

$$
\begin{aligned}
\mathrm{R}.\ell &= \mathrm{R}(\ell) && \text{if } \mathrm{R}(\ell) \wedge \bot \leq \mathbb{0} \\
\mathrm{R}.[t] &= \begin{cases} \bigvee_{\ell \in t} \mathrm{R}(\ell) & \text{if } \bigvee_{\ell \in t} \mathrm{R}(\ell) \wedge \bot \leq \mathbb{0} \\ \bigvee_{\ell \in t} \mathrm{R}(\ell) \setminus \bot \vee \mathtt{nil} & \text{otherwise} \end{cases} \\
(\mathrm{R}_1 + \mathrm{R}_2)(\ell) &= \begin{cases} \mathrm{R}_2(\ell) & \text{if } \mathrm{R}_2(\ell) \wedge \bot \leq \mathbb{0} \\ (\mathrm{R}_2(\ell) \setminus \bot) \vee \mathrm{R}_1(\ell) & \text{otherwise} \end{cases}
\end{aligned}
$$

embedded negation

A different representation for record types:

Let $\tau ::= t \mid \bot$

usual record type

$$\mathcal{R} ::= \langle\!\langle (\tau_\ell)_{\ell \in L} \,;\, t_0 \,;\, S \rangle\!\rangle$$

It denotes $\{\ell_1 = \tau_{\ell_1}, \ldots, \ell_n = \tau_{\ell_n}, \_ \Rightarrow t_0\} \setminus \bigvee_{s \in S} \{\ell_1 \Rightarrow \mathbb{1}, \ldots, \ell_n \Rightarrow \mathbb{1}, \_ \Rightarrow \neg s\}$

meaning $\forall s \in S$ there exists a label not in L whose value is of type $s \in S$

Type operators

Properties of the new representation:

1. Union normal form: Every $t \leq \{\_ \Rightarrow \mathbb{1}\}$ is equivalent to a type of the form $\bigvee_{i \in I} \mathcal{R}_i$

Type operators

Properties of the new representation:

1. Union normal form: Every $t \leq \{\_ \Rightarrow \mathbb{1}\}$ is equivalent to a type of the form $\bigvee_{i\in I} \mathcal{R}_i$
2. Effective transformation: It is easy to transform $\bigvee_{i\in I} \left(\bigwedge_{p\in P_i} \mathrm{R}_p \ \wedge\ \bigwedge_{n\in N_i} \neg \mathrm{R}_n\right)$ into an equivalent $\bigvee_{j\in J} \mathcal{R}_j$

Type operators

Properties of the new representation:

1. Union normal form: Every $t \leq \{\_ \Rightarrow \mathbb{1}\}$ is equivalent to a type of the form $\bigvee_{i \in I} \mathcal{R}_i$
2. Effective transformation: It is easy to transform $\bigvee_{i \in I} \left(\bigwedge_{p \in P_i} \mathrm{R}_p \;\wedge\; \bigwedge_{n \in N_i} \neg \mathrm{R}_n \right)$ into an equivalent $\bigvee_{j \in J} \mathcal{R}_j$
3. Type operators: Operators on single $\mathcal{R}$'s as defined as those on single R's.

Type operators

Properties of the new representation:

1. Union normal form: Every $t \leq \{\_ \Rightarrow \mathbb{1}\}$ is equivalent to a type of the form $\bigvee_{i\in I} \mathcal{R}_i$
2. Effective transformation: It is easy to transform $\bigvee_{i\in I} \left(\bigwedge_{p\in P_i} \mathrm{R}_p \ \wedge \ \bigwedge_{n\in N_i} \neg \mathrm{R}_n\right)$ into an equivalent $\bigvee_{j\in J} \mathcal{R}_j$
3. Type operators: Operators on single $\mathcal{R}$'s as defined as those on single R's.

Type operators for $t \leq \{\_ \Rightarrow \mathbb{1}\}$

$$
\begin{array}{rcl}
(\bigvee_{i\in I} \mathcal{R}_i).\ell & = & \bigvee_{i\in I}(\mathcal{R}_i.\ell) \\
(\bigvee_{i\in I} \mathcal{R}_i).[t] & = & \bigvee_{i\in I}(\mathcal{R}_i.[t]) \\
(\bigvee_{i\in I} \mathcal{R}_i) + (\bigvee_{j\in J} \mathcal{R}_j) & = & \bigvee_{i\in I, j\in J}(\mathcal{R}_i + \mathcal{R}_j)
\end{array}
$$

General Case: Erlang's maps

Record types	R	$::=$	$\{f, \ldots, f\}$
Field types	f	$::=$	$\ell = t \mid \ell \Rightarrow t \mid \_ \Rightarrow t \mid t \Rightarrow t$

General Case: Erlang's maps

Record types	R	$::=$	$\{f, \ldots, f\}$
Field types	f	$::=$	$\ell = t \mid \ell \Rightarrow t \mid \_ \Rightarrow t \mid \boldsymbol{t \Rightarrow t}$

General Case: Erlang's maps

Record types	R	$::=$	$\{f, \ldots, f\}$
Field types	f	$::=$	$\ell = t \mid \ell \Rightarrow t \mid \_ \Rightarrow t \mid t \Rightarrow t$

The problem of overlapping domains

Consider (Elixir syntax)

```
type t= %{ {integer,term} => atom, ... } and %{ {term,integer} => integer, ... }
```

General Case: Erlang's maps

$$\begin{array}{rrcl} \textbf{Record types} & R & ::= & \{f, \ldots, f\} \\ \textbf{Field types} & f & ::= & \ell = t \mid \ell \Rightarrow t \mid \_ \Rightarrow t \mid t \Rightarrow t \end{array}$$

The problem of overlapping domains

Consider (Elixir syntax)

```
type t= %{ {integer,term} => atom, ... } and %{ {term,integer} => integer, ... }
```

Set-theoretically equivalent to

```
type t= %{ {integer, not integer} => atom,
           {not integer, integer} => integer,
           { integer , integer }  => none, ...}          // none = empty-type
```

General Case: Erlang's maps

Record types $\mathrm{R} \quad ::= \quad \{f, \ldots, f\}$

Field types $f \quad ::= \quad \ell = t \mid \ell \Rightarrow t \mid \_ \Rightarrow t \mid t \Rightarrow t$

The problem of overlapping domains

Consider (Elixir syntax)

```
type t= %{ {integer,term} => atom, ... } and %{ {term,integer} => integer, ... }
```

Set-theoretically equivalent to

```
type t= %{ {integer, not integer} => atom,
           {not integer, integer} => integer,
           { integer , integer }  => none, ...}          // none = empty-type
```

Existing solutions

Luau: records of type `t` must have fields of type `{integer,integer}` undefined

TypeScript: restrict key-types to avoid overlapping

Chosen solution (TypeScript-like)

Record types	R	$::=$	$\{f, \ldots, f\}$
Field types	f	$::=$	$\ell = t \mid \ell \Rightarrow t \mid k \Rightarrow t$
Keys types	k	$::=$	$\_ \mid \mathrm{Atom} \mid \mathrm{Int} \mid \mathrm{String} \mid \mathbb{1} \times \mathbb{1} \mid \{\_ \Rightarrow \mathbb{1}\}$

Rationale

- less expressive but easier to understand
- simple extension of the previous theory
- can be extended with row polymorphism (work in progress)
- other solutions are possible as long as key-types do not overlap

Chosen solution (TypeScript-like)

Record types	R	$::=$	$\{f, \ldots, f\}$
Field types	f	$::=$	$\ell = t \mid \ell \Rightarrow t \mid k \Rightarrow t$
Keys types	k	$::=$	$\_ \mid \mathrm{Atom} \mid \mathrm{Int} \mid \mathrm{String} \mid \mathbb{1}\times\mathbb{1} \mid \{\_ \Rightarrow \mathbb{1}\}$

Rationale

- less expressive but easier to understand
- simple extension of the previous theory
- can be extended with row polymorphism (work in progress)
- other solutions are possible as long as key-types do not overlap

Example:

```
type t= %{:foo => atom(),
          optional(:bar) => atom,
          optional(atom) => integer,
          optional(map) => list([{atom,term}])
          optional({term,term,term}) => function }
```

Extension of the previous theory

Notation:

$\mathrm{deflt}(\mathsf{R})$ becames a product indexed on $\mathbb{K} = \{\text{Atom}, \text{Int}, \text{String}, \mathbb{1} \times \mathbb{1}, \mathbf{\{\_ \Rightarrow} \mathbb{1}\mathbf{\}}\}$

Extension of the previous theory

Notation:

$\mathrm{deflt}(\mathrm{R})$ becames a product indexed on $\mathbb{K} = \{\mathrm{Atom}, \mathrm{Int}, \mathrm{String}, \mathbb{1}\times\mathbb{1}, \boldsymbol{\{}\_ \Rightarrow \mathbb{1}\boldsymbol{\}}\}$

Subtyping:

The definition of Φ changes:

$$
\begin{array}{rcl}
\Phi(\mathrm{R}_\circ, \varnothing) & = & \texttt{false} \\
\Phi(\mathrm{R}_\circ, N \cup \{\mathrm{R}\}) & = & \texttt{if } \mathrm{deflt}(\mathrm{R}_\circ) \leq \mathrm{deflt}(\mathrm{R}) \\
& & \ \texttt{then } \forall \ell \in L\,.\,(\,\mathrm{R}_\circ(\ell) \leq \mathrm{R}(\ell) \texttt{ or } \Phi(\mathrm{R}_\circ \wedge \boldsymbol{\{}\ell : \texttt{not}(\mathrm{R}(\ell))\boldsymbol{\}}, N)\,) \\
& & \ \texttt{else } \Phi(\mathrm{R}_\circ, N)
\end{array}
$$

Extension of the previous theory

Notation:

$\mathrm{deflt}(\mathrm{R})$ becames a product indexed on $\mathbb{K} = \{\mathrm{Atom}, \mathrm{Int}, \mathrm{String}, \mathbb{1}\times\mathbb{1}, \{\_ \Rightarrow \mathbb{1}\}\}$

Subtyping:

The definition of Φ changes:

$$
\begin{array}{rcl}
\Phi(\mathrm{R}_\circ, \varnothing) & = & \mathtt{false} \\
\Phi(\mathrm{R}_\circ, N \cup \{\mathrm{R}\}) & = & \mathtt{if}\ \ \forall k \in \mathbb{K}\,.\,(\mathrm{deflt}(\mathrm{R}_\circ))_k \leq (\mathrm{deflt}(\mathrm{R}))_k \\
 & & \ \mathtt{then}\ \forall \ell \in L\,.\,(\,\mathrm{R}_\circ(\ell) \leq \mathrm{R}(\ell)\ \mathtt{or}\ \Phi(\mathrm{R}_\circ \wedge \{\ell : \mathtt{not}(\mathrm{R}(\ell))\}, N)\,) \\
 & & \ \mathtt{else}\ \Phi(\mathrm{R}_\circ, N)
\end{array}
$$

Extension of the previous theory

Notation:

$\mathrm{deflt}(\mathrm{R})$ becames a product indexed on $\mathbb{K} = \{\mathrm{Atom}, \mathrm{Int}, \mathrm{String}, \mathbb{1}\times\mathbb{1}, \{\_ \Rightarrow \mathbb{1}\}\}$

Subtyping:

The definition of Φ changes:

$$
\begin{array}{rcl}
\Phi(\mathrm{R}_\circ, \varnothing) & = & \texttt{false} \\
\Phi(\mathrm{R}_\circ, N \cup \{\mathrm{R}\}) & = & \texttt{if} \quad \forall k \in \mathbb{K}\,.\, (\mathrm{deflt}(\mathrm{R}_\circ))_k \leq (\mathrm{deflt}(\mathrm{R}))_k \\
 & & \;\texttt{then} \;\; \forall \ell \in L\,.\,(\, \mathrm{R}_\circ(\ell) \leq \mathrm{R}(\ell) \;\text{ or }\; \Phi(\mathrm{R}_\circ \wedge \{\ell : \texttt{not}(\mathrm{R}(\ell))\}, N)\,) \\
 & & \;\texttt{else} \;\; \Phi(\mathrm{R}_\circ, N)
\end{array}
$$

Type operators:

Only the record representation needs to change:

$$\mathcal{R} ::= \langle\!\langle (\tau_\ell)_{\ell \in L} \,;\, t_0 \,;\, S \rangle\!\rangle$$

Extension of the previous theory

Notation:

deflt(R) becames a product indexed on $\mathbb{K} = \{\text{Atom}, \text{Int}, \text{String}, \mathbb{1}\times\mathbb{1}, \{\_ \Rightarrow \mathbb{1}\}\}$

Subtyping:

The definition of Φ changes:

$$\begin{array}{rcl}
\Phi(R_\circ, \varnothing) & = & \texttt{false} \\
\Phi(R_\circ, N \cup \{R\}) & = & \texttt{if}\ \ \forall k \in \mathbb{K}.\,(\mathrm{deflt}(R_\circ))_k \leq (\mathrm{deflt}(R))_k \\
& & \ \texttt{then}\ \forall \ell \in L.\,(\,R_\circ(\ell) \leq R(\ell)\ \text{ or }\ \Phi(R_\circ \wedge \{\ell : \texttt{not}(R(\ell))\}, N)\,) \\
& & \ \texttt{else}\ \Phi(R_\circ, N)
\end{array}$$

Type operators:

Only the record representation needs to change:

$$\mathcal{R} ::= \langle\!\langle (\tau_\ell)_{\ell\in L}\,;\,(t_1\times\ldots\times t_n)\,;\,\{(s^i_1\times\ldots\times s^i_n) \mid i \in I\}\,\rangle\!\rangle$$

Extension of the previous theory

Notation:

$\mathsf{deflt}(\mathrm{R})$ becames a product indexed on $\mathbb{K} = \{\mathsf{Atom}, \mathsf{Int}, \mathsf{String}, \mathbb{1}\times\mathbb{1}, \{\_ \Rightarrow \mathbb{1}\}\}$

Subtyping:

The definition of Φ changes:

$$\begin{array}{rcl}
\Phi(\mathrm{R}_\circ, \varnothing) & = & \mathtt{false} \\
\Phi(\mathrm{R}_\circ, N \cup \{\mathrm{R}\}) & = & \mathtt{if}\ \ \forall k \in \mathbb{K}\,.\,(\mathsf{deflt}(\mathrm{R}_\circ))_k \leq (\mathsf{deflt}(\mathrm{R}))_k \\
& & \ \mathtt{then}\ \ \forall \ell \in L\,.\,(\,\mathrm{R}_\circ(\ell) \leq \mathrm{R}(\ell)\ \text{ or }\ \Phi(\mathrm{R}_\circ \wedge \{\ell : \mathtt{not}(\mathrm{R}(\ell))\}, N)\,) \\
& & \ \mathtt{else}\ \ \Phi(\mathrm{R}_\circ, N)
\end{array}$$

Type operators:

Only the record representation needs to change:

$$\mathcal{R} ::= \langle\!\langle (\tau_\ell)_{\ell\in L}\,;\,(t_1\times\ldots\times t_n)\,;\,\{(s^i_1\times\ldots\times s^i_n) \mid i\in I\}\,\rangle\!\rangle$$

where $\langle\!\langle (\tau_\ell)_{\ell\in L}\,;\,(t_1\times\ldots\times t_n)\,;\,\varnothing\,\rangle\!\rangle$ is $\{\ell_1 = \tau_{\ell_1}, \ldots, \ell_n = \tau_{\ell_n}, k_1 \Rightarrow t_1, \ldots, k_n \Rightarrow t_n\}$ and each $(s_1\times\ldots\times s_n)$ implies there exists k_i and a label $\ell \in k_i \setminus L$ mapped into a value in $\neg s_i$

Conclusion and future work

Summary

- A type system for mixed usage of structs and maps
- Difficult in the presence of set-theoretic types
- Boils down to the definition of subtyping and of appropriate type operators
- I gave algorithms to decide/compute them and proved their soundness
- Being implemented for Elixir and under consideration for Ballerina

Future work

- Row polymorphism
- Allow the programmer to define how to partition tuples of key-types