Refinement Kinds

Type-safe Programming with Practical Type-level Computation

Bernardo Toninho
(joint work with Luís Caires)
FCT-UNL & NOVA LINCS

(Presented at OOPSLA'19)

December 16, 2019

Motivation

Why ~~Refinement Kinds~~ / Another-Fancy-Kind-System?

- Modern software computes with both values and types:
  - Types as data (e.g. reflection, ad-hoc polymorphism)
  - Meta-programming (e.g. toolchains, web apps, etc)

Motivation

Why ~~Refinement Kinds~~ / Another-Fancy-Kind-System?

- Modern software computes with both values and types:
  - Types as data (e.g. reflection, ad-hoc polymorphism)
  - Meta-programming (e.g. toolchains, web apps, etc)
- Such idioms move outside the boundaries of type safety.

Motivation

Why ~~Refinement Kinds~~ / Another-Fancy-Kind-System?

- Modern software computes with both values and types:
  - Types as data (e.g. reflection, ad-hoc polymorphism)
  - Meta-programming (e.g. toolchains, web apps, etc)
- Such idioms move outside the boundaries of type safety.
- . . . but we would really like to have type safety!

Motivation

An Example

- Consider a generic function genConstr that for **any** non-empty record type R returns a function to initialise a mutable record with the field types of R.

Motivation

An Example

- Consider a generic function genConstr that for **any** non-empty record type R returns a function to initialise a mutable record with the field types of R.
- Let Person $= \langle \mathbf{name}{:}\mathsf{String}; \mathbf{age} : \mathsf{Int}\rangle :: \mathsf{Type}$

Motivation
An Example

- Consider a generic function genConstr that for **any** non-empty record type R returns a function to initialise a mutable record with the field types of R.
- Let Person $= \langle$**name**:String; **age** : Int$\rangle$ :: Type
- We want genConstr Person to work like:

$$\text{genConstr Person}$$
$$\longrightarrow^{*}$$
$$\lambda x{:}\text{String}.\lambda y{:}\text{Int}.\langle \mathit{age} = \textbf{ref}\ y; \mathit{name} = \textbf{ref}\ x\rangle$$
$$: \text{String} \rightarrow \text{Int} \rightarrow \langle \mathit{age} : \textbf{ref}\ \text{Int}; \mathit{name} : \textbf{ref}\ \text{String}\rangle$$

Motivation

An Example

- Consider a generic function genConstr that for **any** non-empty record type R returns a function to initialise a mutable record with the field types of R.
- Let Person $= \langle$**name**:String; **age** : Int$\rangle$ :: Type
- We want genConstr Person to work like:

$$\text{genConstr Person}$$
$$\longrightarrow^*$$
$$\lambda x{:}\text{String}.\lambda y{:}\text{Int}.\langle \textit{age} = \textbf{ref}\, y; \textit{name} = \textbf{ref}\, x\rangle$$
$$: \text{String} \to \text{Int} \to \langle \textit{age} : \textbf{ref}\,\text{Int}; \textit{name} : \textbf{ref}\,\text{String}\rangle$$

Motivation
An Example

- Consider a generic function genConstr that for **any** non-empty record type R returns a function to initialise a mutable record with the field types of R.
- Let Person $= \langle \textbf{name}{:}\mathsf{String}; \textbf{age} : \mathsf{Int}\rangle :: \mathsf{Type}$
- We want genConstr Person to work like:

$$\begin{array}{c} \mathsf{genConstr}\ \mathsf{Person} \\ \longrightarrow^{*} \\ \lambda x{:}\mathsf{String}.\lambda y{:}\mathsf{Int}.\langle \mathit{age} = \textbf{ref}\ y; \mathit{name} = \textbf{ref}\ x\rangle \\ : \mathsf{String} \rightarrow \mathsf{Int} \rightarrow \langle \mathit{age} : \textbf{ref}\ \mathsf{Int}; \mathit{name} : \textbf{ref}\ \mathsf{String}\rangle \end{array}$$

Motivation

Goals

Desiderata:

- Static type-checking of such “highly plastic” code idioms.
- Clean, expressive and practical.
- Reasonable type-theory

Motivation

Goals

Desiderata:

- Static type-checking of such “highly plastic” code idioms.
- Clean, expressive and practical.
- Reasonable type-theory

Refinement Kinds:

- Natural transposition of refinement types to the kind level.
- Structural properties of types as kind refinements.

Outline

Types and Kinds

Types classify expressions:

- 0 : Int
- succ : Int $\rightarrow$ Int
- joe : $\langle$**name** : String; **age** : Int$\rangle$

Types and Kinds

Types classify expressions:

- $0 : \mathsf{Int}$
- $\mathsf{succ} : \mathsf{Int} \to \mathsf{Int}$
- $\mathsf{joe} : \langle \mathbf{name} : \mathsf{String}; \mathbf{age} : \mathsf{Int} \rangle$

Kinds classify types:

- $\mathsf{Int} \to \mathsf{Int} :: \mathsf{Type}$
- $\langle \mathbf{name} : \mathsf{String}; \mathbf{age} : \mathsf{Int} \rangle :: \mathsf{Type}$
- $\mathsf{List} :: \mathbf{Type} \to \mathbf{Type}$

Refinement Types

Refinement types [FP91,BBFGM11]:

- $M : \{x{:}T \mid \varphi(x)\}$
- M is of type T

Refinement Types

Refinement types [FP91,BBFGM11]:

- $M : \{x{:}T \mid \varphi(x)\}$
- M is of type T
- φ logical predicate that M must satisfy (e.g. of some SMT-decidable theory).

$$\frac{\Gamma \models \varphi\{M/t\} \quad \Gamma \vdash M : T}{\Gamma \vdash M : \{x{:}T \mid \varphi\}}$$

Refinement Types

Refinement types [FP91,BBFGM11]:

- $M : \{x{:}T \mid \varphi(x)\}$
- M is of type T
- φ logical predicate that M must satisfy (e.g. of some SMT-decidable theory).

$$\frac{\Gamma \models \varphi\{M/t\} \quad \Gamma \vdash M : T}{\Gamma \vdash M : \{x{:}T \mid \varphi\}}$$

- $\{x{:}\text{Int} \mid \text{even}(x)\}$
- $\{x{:}\text{Int} \mid x > 0\}$

Refinement Types

Refinement types [FP91,BBFGM11]:

- $M : \{x{:}T \mid \varphi(x)\}$
- M is of type T
- φ logical predicate that M must satisfy (e.g. of some SMT-decidable theory).

$$\frac{\Gamma \models \varphi\{M/t\} \quad \Gamma \vdash M : T}{\Gamma \vdash M : \{x{:}T \mid \varphi\}}$$

- $\{x{:}\mathsf{Int} \mid \mathsf{even}(x)\}$
- $\{x{:}\mathsf{Int} \mid x > 0\}$
- Types **declaratively** specify properties of their inhabitants.

Refinement Kinds

Specialized kinds classify types:

- $\mathsf{Int} \rightarrow \mathsf{Int} :: \mathsf{Fun}$

Refinement Kinds

Specialized kinds classify types:

- $\mathsf{Int} \rightarrow \mathsf{Int} :: \mathsf{Fun}$
- $\langle \mathbf{name} : \mathsf{String}; \mathbf{age} : \mathsf{Int} \rangle :: \mathsf{Rec}$

Refinement Kinds

Specialized kinds classify types:

- $\mathsf{Int} \rightarrow \mathsf{Int} :: \mathsf{Fun}$
- $\langle \mathbf{name} : \mathsf{String}; \mathbf{age} : \mathsf{Int} \rangle :: \mathsf{Rec}$

Refinement Kinds:

- $T :: \{t{::}\mathcal{K} \mid \varphi(t)\}$
- T is of kind $\mathcal{K}$

Refinement Kinds

Specialized kinds classify types:

- $\mathsf{Int} \rightarrow \mathsf{Int} :: \mathsf{Fun}$
- $\langle \textbf{name} : \mathsf{String}; \textbf{age} : \mathsf{Int} \rangle :: \mathsf{Rec}$

Refinement Kinds:

- $T :: \{t{::}\mathcal{K} \mid \varphi(t)\}$
- T is of kind $\mathcal{K}$
- φ logical predicate that T must satisfy (e.g. of some SMT-decidable theory).

$$\frac{\Gamma \models \varphi\{T/t\} \qquad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}$$

Refinement Kinds

Specialized kinds classify types:

- $\mathsf{Int} \to \mathsf{Int} :: \{f :: \mathsf{Fun} \mid \mathsf{auto}(f)\}$
- $\langle \textbf{name} : \mathsf{String}; \textbf{age} : \mathsf{Int} \rangle :: \mathsf{Rec}$

Refinement Kinds:

- $T :: \{t{::}\mathcal{K} \mid \varphi(t)\}$
- T is of kind $\mathcal{K}$
- φ logical predicate that T must satisfy (e.g. of some SMT-decidable theory).

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}$$

- Kinds **declaratively** specify **structural** properties of their inhabitants.

Refinement Kinds

Specialized kinds classify types:

- $\mathsf{Int} \to \mathsf{Int} :: \{f :: \mathsf{Fun} \mid \mathsf{auto}(f)\}$
- $\langle \textbf{name} : \mathsf{String}; \textbf{age} : \mathsf{Int}\rangle :: \{r :: \mathsf{Rec} \mid \textbf{name} \in \mathsf{lab}(r)\}$

Refinement Kinds:

- $T :: \{t{::}\mathcal{K} \mid \varphi(t)\}$
- T is of kind $\mathcal{K}$
- φ logical predicate that T must satisfy (e.g. of some SMT-decidable theory).

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}$$

- Kinds **declaratively** specify **structural** properties of their inhabitants.

Outline

Extending a Record Type

Generically extending a record **type** with a new field:

- Nm – the kind of record labels
- lab(s) – the set of labels in record type s
- $\langle \ell : T \rangle @ S$ – record type constructor (i.e. type-level lists)

Extending a Record Type

Generically extending a record **type** with a new field:

- Nm – the kind of record labels
- lab(s) – the set of labels in record type s
- $\langle \ell : T \rangle @ S$ – record type constructor (i.e. type-level lists)

addRecFieldT ::

Extending a Record Type

Generically extending a record **type** with a new field:

- Nm – the kind of record labels
- lab(s) – the set of labels in record type s
- $\langle \ell : T \rangle @ S$ – record type constructor (i.e. type-level lists)

addRecFieldT :: Πl::Nm.

Extending a Record Type

Generically extending a record **type** with a new field:

- Nm – the kind of record labels
- $\text{lab}(s)$ – the set of labels in record type s
- $\langle \ell : T \rangle @ S$ – record type constructor (i.e. type-level lists)

$$\text{addRecFieldT} :: \Pi l{::}\text{Nm}.\, \Pi t{::}\text{Type}.$$

Extending a Record Type

Generically extending a record **type** with a new field:

- Nm – the kind of record labels
- $\mathsf{lab}(s)$ – the set of labels in record type s
- $\langle \ell : T \rangle @ S$ – record type constructor (i.e. type-level lists)

$$\text{addRecFieldT} :: \Pi l{::}\mathsf{Nm}.\, \Pi t{::}\mathsf{Type}.\, \Pi r{::}\{s{::}\mathsf{Rec} \mid l \notin \mathbf{lab}(s)\}.$$

Extending a Record Type

Generically extending a record **type** with a new field:

- Nm – the kind of record labels
- lab(s) – the set of labels in record type s
- $\langle \ell : T\rangle @S$ – record type constructor (i.e. type-level lists)

$$\text{addRecFieldT} :: \Pi l{::}\text{Nm}.\, \Pi t{::}\text{Type}.\, \Pi r{::}\{s{::}\text{Rec} \mid l \notin \textbf{lab}(s)\}.\, \text{Rec}$$

Extending a Record Type

Generically extending a record **type** with a new field:

- Nm – the kind of record labels
- $\text{lab}(s)$ – the set of labels in record type s
- $\langle \ell : T \rangle @ S$ – record type constructor (i.e. type-level lists)

$$\text{addRecFieldT} :: \Pi l{::}\text{Nm}.\, \Pi t{::}\text{Type}.\, \Pi r{::}\{s{::}\text{Rec} \mid l \notin \textbf{lab}(s)\}.\, \text{Rec}$$
$$\text{addRecFieldT} \triangleq \lambda l{::}\text{Nm}.$$

Extending a Record Type

Generically extending a record **type** with a new field:

- Nm – the kind of record labels
- lab(s) – the set of labels in record type s
- $\langle \ell : T \rangle @ S$ – record type constructor (i.e. type-level lists)

$$\text{addRecFieldT} :: \Pi l{::}\text{Nm}.\, \Pi t{::}\text{Type}.\, \Pi r{::}\{s{::}\text{Rec} \mid l \notin \textbf{lab}(s)\}.\, \text{Rec}$$
$$\text{addRecFieldT} \triangleq \lambda l{::}\text{Nm}.\lambda t{::}\text{Type}.$$

Extending a Record Type

Generically extending a record **type** with a new field:

- Nm – the kind of record labels
- lab(s) – the set of labels in record type s
- $\langle \ell : T \rangle @ S$ – record type constructor (i.e. type-level lists)

$$\text{addRecFieldT} :: \Pi l{::}\text{Nm}.\, \Pi t{::}\text{Type}.\, \Pi r{::}\{s{::}\text{Rec} \mid l \notin \mathbf{lab}(s)\}.\, \text{Rec}$$
$$\text{addRecFieldT} \triangleq \lambda l{::}\text{Nm}.\lambda t{::}\text{Type}.\lambda r{::}\{s{::}\text{Rec} \mid l \notin \mathbf{lab}(s)\}.$$

Extending a Record Type

Generically extending a record **type** with a new field:

- Nm – the kind of record labels
- lab(s) – the set of labels in record type s
- $\langle \ell : T \rangle @ S$ – record type constructor (i.e. type-level lists)

$$\text{addRecFieldT} :: \Pi l{::}\text{Nm}.\, \Pi t{::}\text{Type}.\, \Pi r{::}\{s{::}\text{Rec} \mid l \notin \textbf{lab}(s)\}.\, \text{Rec}$$
$$\text{addRecFieldT} \triangleq \lambda l{::}\text{Nm}.\lambda t{::}\text{Type}.\lambda r{::}\{s{::}\text{Rec} \mid l \notin \textbf{lab}(s)\}.\langle l : t\rangle @ r$$

Extending a Record Type

Generically extending a record **expression** with a new field:

$$\text{addRecFieldT} :: \Pi l{::}\text{Nm}.\, \Pi t{::}\text{Type}.\, \Pi r{::}\{s{::}\text{Rec} \mid l \notin \textbf{lab}(s)\}.\, \text{Rec}$$

addField :

addField ≜

Extending a Record Type

Generically extending a record **expression** with a new field:

$$\text{addRecFieldT} :: \Pi l{::}\text{Nm}.\, \Pi t{::}\text{Type}.\, \Pi r{::}\{s{::}\text{Rec} \mid l \notin \textbf{lab}(s)\}.\, \text{Rec}$$

$$\text{addField} : \quad \forall l{::}\text{Nm}.$$

$$\text{addField} \triangleq \quad \Lambda l{::}\text{Nm}.$$

Extending a Record Type

Generically extending a record **expression** with a new field:

$$\mathsf{addRecFieldT} :: \Pi l{::}\mathsf{Nm}.\,\Pi t{::}\mathsf{Type}.\,\Pi r{::}\{s{::}\mathsf{Rec} \mid l \notin \mathbf{lab}(s)\}.\,\mathsf{Rec}$$

$$\mathsf{addField} : \quad \forall l{::}\mathsf{Nm}.\forall t{::}\mathsf{Type}.$$

$$\mathsf{addField} \triangleq \quad \Lambda l{::}\mathsf{Nm}.\Lambda t{::}\mathsf{Type}.$$

Extending a Record Type

Generically extending a record **expression** with a new field:

$$\text{addRecFieldT} :: \Pi l{::}\text{Nm}.\, \Pi t{::}\text{Type}.\, \Pi r{::}\{s{::}\text{Rec} \mid l \notin \textbf{lab}(s)\}.\, \text{Rec}$$

$$\text{addField} : \quad \forall l{::}\text{Nm}.\forall t{::}\text{Type}.\forall r{::}\{s{::}\text{Rec} \mid l \notin \textbf{lab}(s)\}.$$

$$\text{addField} \triangleq \quad \Lambda l{::}\text{Nm}.\Lambda t{::}\text{Type}.\Lambda r{::}\{s{::}\text{Rec} \mid l \notin \textbf{lab}(s)\}.$$

Extending a Record Type

Generically extending a record **expression** with a new field:

$$\text{addRecFieldT} :: \Pi l{::}\text{Nm}.\, \Pi t{::}\text{Type}.\, \Pi r{::}\{s{::}\text{Rec} \mid l \notin \textbf{lab}(s)\}.\, \text{Rec}$$

$$\begin{aligned} \text{addField} :\; & \forall l{::}\text{Nm}.\forall t{::}\text{Type}.\forall r{::}\{s{::}\text{Rec} \mid l \notin \textbf{lab}(s)\}. \\ & t \to r \to \text{addRecFieldT}\ l\ t\ r \\ \text{addField} \triangleq\; & \Lambda l{::}\text{Nm}.\Lambda t{::}\text{Type}.\Lambda r{::}\{s{::}\text{Rec} \mid l \notin \textbf{lab}(s)\}. \\ & \lambda x{:}t.\lambda y{:}r.\langle l = x\rangle @ y \end{aligned}$$

Extending a Record Type

Generically extending a record **expression** with a new field:

$$\text{addRecFieldT} :: \Pi l{::}\text{Nm}.\, \Pi t{::}\text{Type}.\, \Pi r{::}\{s{::}\text{Rec} \mid l \notin \mathbf{lab}(s)\}.\, \text{Rec}$$

$$\begin{aligned}
\text{addField} :\quad & \forall l{::}\text{Nm}.\forall t{::}\text{Type}.\forall r{::}\{s{::}\text{Rec} \mid l \notin \mathbf{lab}(s)\}.\\
& t \to r \to \text{addRecFieldT}\ l\ t\ r\\
\text{addField} \triangleq\quad & \Lambda l{::}\text{Nm}.\Lambda t{::}\text{Type}.\Lambda r{::}\{s{::}\text{Rec} \mid l \notin \mathbf{lab}(s)\}.\\
& \lambda x{:}t.\lambda y{:}r.\langle l = x\rangle @ y
\end{aligned}$$

Extending a Record Type

Generically extending a record **expression** with a new field:

$$\text{addRecFieldT} :: \Pi l{::}\text{Nm}.\, \Pi t{::}\text{Type}.\, \Pi r{::}\{s{::}\text{Rec} \mid l \notin \textbf{lab}(s)\}.\, \text{Rec}$$

$$\begin{aligned}
\text{addField} :&\quad \forall l{::}\text{Nm}.\forall t{::}\text{Type}.\forall r{::}\{s{::}\text{Rec} \mid l \notin \textbf{lab}(s)\}. \\
&\quad t \to r \to \text{addRecFieldT}\ l\ t\ r \\
\text{addField} \triangleq&\quad \Lambda l{::}\text{Nm}.\Lambda t{::}\text{Type}.\Lambda r{::}\{s{::}\text{Rec} \mid l \notin \textbf{lab}(s)\}. \\
&\quad \lambda x{:}t.\lambda y{:}r.\langle l = x\rangle @ y
\end{aligned}$$

Extending a Record Type

Generically extending a record **expression** with a new field:

$$\text{addRecFieldT} :: \Pi l{::}\text{Nm}.\,\Pi t{::}\text{Type}.\,\Pi r{::}\{s{::}\text{Rec} \mid l \notin \textbf{lab}(s)\}.\,\text{Rec}$$

$$\begin{aligned}
\text{addField} :\quad & \forall l{::}\text{Nm}.\forall t{::}\text{Type}.\forall r{::}\{s{::}\text{Rec} \mid l \notin \textbf{lab}(s)\}. \\
& t \to r \to \text{addRecFieldT}\ l\ t\ r \\
\text{addField} \triangleq\quad & \Lambda l{::}\text{Nm}.\Lambda t{::}\text{Type}.\Lambda r{::}\{s{::}\text{Rec} \mid l \notin \textbf{lab}(s)\}. \\
& \lambda x{:}t.\lambda y{:}r.\langle l = x\rangle @ y
\end{aligned}$$

$$\begin{aligned}
&\text{addField}\ \textit{name}\ \text{String}\ \langle age : \text{Int}\rangle\ \text{“jack”}\ \langle age = 20\rangle \\
&: \text{addRecFieldT}\ \textit{name}\ \text{String}\ \langle age : \text{Int}\rangle
\end{aligned}$$

Extending a Record Type

Generically extending a record **expression** with a new field:

$$\text{addRecFieldT} :: \Pi l{::}\text{Nm}.\, \Pi t{::}\text{Type}.\, \Pi r{::}\{s{::}\text{Rec} \mid l \notin \textbf{lab}(s)\}.\, \text{Rec}$$

$$\begin{aligned}
\text{addField} :\ & \forall l{::}\text{Nm}.\forall t{::}\text{Type}.\forall r{::}\{s{::}\text{Rec} \mid l \notin \textbf{lab}(s)\}.\\
& t \to r \to \text{addRecFieldT}\ l\ t\ r\\
\text{addField} \triangleq\ & \Lambda l{::}\text{Nm}.\Lambda t{::}\text{Type}.\Lambda r{::}\{s{::}\text{Rec} \mid l \notin \textbf{lab}(s)\}.\\
& \lambda x{:}t.\lambda y{:}r.\langle l = x\rangle @ y
\end{aligned}$$

$$\begin{aligned}
& \text{addField}\ \textit{name}\ \text{String}\ \langle age : \text{Int}\rangle\ \text{“jack”}\ \langle age = 20\rangle\\
& : \text{addRecFieldT}\ \textit{name}\ \text{String}\ \langle age : \text{Int}\rangle\\
& \longrightarrow^*\\
& \langle \textit{name} = \text{“jack”}; \textit{age} = 20\rangle : \langle \textit{name} : \text{String}; \textit{age} : \text{Int}\rangle
\end{aligned}$$

Generic Constructor Generator

Easier to use an accumulator parameter:

$$\text{genConstr Person } \langle\rangle \ \langle\rangle \rightarrow^* \lambda x{:}\text{String}.\lambda y{:}\text{Int}.\langle \textit{age} = \textbf{ref}\, y; \textit{name} = \textbf{ref}\, x \rangle$$

Generic Constructor Generator

Easier to use an accumulator parameter:

$$\text{genConstr Person } \langle\rangle\ \langle\rangle \to^* \lambda x{:}\text{String}.\lambda y{:}\text{Int}.\langle \mathit{age} = \textbf{ref}\, y; \mathit{name} = \textbf{ref}\, x\rangle$$

A definition for genConstr:

$$\text{genConstr} \triangleq$$

Generic Constructor Generator

Easier to use an accumulator parameter:

$$\text{genConstr Person}\ \langle\rangle\ \langle\rangle \rightarrow^* \lambda x{:}\text{String}.\lambda y{:}\text{Int}.\langle \textit{age} = \textbf{ref}\, y; \textit{name} = \textbf{ref}\, x\rangle$$

A definition for genConstr:

$$\text{genConstr} \triangleq \quad \Lambda R{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.$$

Generic Constructor Generator

Easier to use an accumulator parameter:

$$\text{genConstr Person } \langle\rangle\ \langle\rangle \to^{*} \lambda x{:}\text{String}.\lambda y{:}\text{Int}.\langle \textit{age} = \textbf{ref}\ y; \textit{name} = \textbf{ref}\ x\rangle$$

A definition for genConstr:

$$\begin{aligned}\text{genConstr} \triangleq\ & \Lambda R{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.\\ & \Lambda C{::}\{c{::}\text{Rec} \mid \textbf{lab}(c)\#\textbf{lab}(R)\}.\lambda v{:}C.\end{aligned}$$

Generic Constructor Generator

Easier to use an accumulator parameter:

$$\text{genConstr Person } \langle\rangle\ \langle\rangle \to^* \lambda x{:}\text{String}.\lambda y{:}\text{Int}.\langle \mathit{age} = \textbf{ref}\ y; \mathit{name} = \textbf{ref}\ x\rangle$$

A definition for genConstr:

$$\begin{aligned}\text{genConstr} \triangleq\ & \Lambda R{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.\\ & \Lambda C{::}\{c{::}\text{Rec} \mid \textbf{lab}(c)\#\textbf{lab}(R)\}.\lambda v{:}C.\\ & \lambda x{:}\textbf{headType}(R).\end{aligned}$$

Generic Constructor Generator

Easier to use an accumulator parameter:

$$\text{genConstr Person } \langle\rangle\ \langle\rangle \rightarrow^* \lambda x{:}\text{String}.\lambda y{:}\text{Int}.\langle \textit{age} = \textbf{ref}\, y; \textit{name} = \textbf{ref}\, x\rangle$$

A definition for genConstr:

$$\begin{aligned}\text{genConstr} \triangleq\ & \Lambda R{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.\\ & \Lambda C{::}\{c{::}\text{Rec} \mid \textbf{lab}(c)\#\textbf{lab}(R)\}.\lambda v{:}C.\\ & \lambda x{:}\textbf{headType}(R).\textbf{case}\ R\ \textbf{of}\\ & \qquad\qquad\quad \Rightarrow\\ & \quad \mid \qquad\qquad\quad \Rightarrow\end{aligned}$$

Generic Constructor Generator

Easier to use an accumulator parameter:

$$\text{genConstr Person } \langle\rangle\ \langle\rangle \to^* \lambda x{:}\text{String}.\lambda y{:}\text{Int}.\langle \textit{age} = \textbf{ref}\, y; \textit{name} = \textbf{ref}\, x\rangle$$

A definition for genConstr:

$$\begin{aligned} \text{genConstr} \triangleq\ & \Lambda R{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \\ & \Lambda C{::}\{c{::}\text{Rec} \mid \textbf{lab}(c)\#\textbf{lab}(R)\}.\lambda v{:}C. \\ & \lambda x{:}\textbf{headType}(R).\textbf{case}\ R\ \textbf{of} \\ & \quad \langle \ell : T\rangle @ \langle\rangle \Rightarrow \\ & \mid \qquad\qquad\quad \Rightarrow \end{aligned}$$

Generic Constructor Generator

Easier to use an accumulator parameter:

$$\text{genConstr Person } \langle\rangle\ \langle\rangle \rightarrow^* \lambda x{:}\text{String}.\lambda y{:}\text{Int}.\langle \mathit{age} = \textbf{ref}\, y; \mathit{name} = \textbf{ref}\, x\rangle$$

A definition for genConstr:

$$\begin{aligned}\text{genConstr} \triangleq\ & \Lambda R{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.\\ & \Lambda C{::}\{c{::}\text{Rec} \mid \textbf{lab}(c)\#\textbf{lab}(R)\}.\lambda v{:}C.\\ & \lambda x{:}\textbf{headType}(R).\textbf{case}\ R\ \textbf{of}\\ & \quad \langle \ell : T\rangle @ \langle\rangle \Rightarrow \langle \ell = \textbf{ref}\, x\rangle @ v\\ & \quad \mid \qquad\qquad \Rightarrow\end{aligned}$$

Generic Constructor Generator

Easier to use an accumulator parameter:

$$\text{genConstr Person } \langle\rangle \ \langle\rangle \to^* \lambda x{:}\text{String}.\lambda y{:}\text{Int}.\langle \textit{age} = \textbf{ref}\, y; \textit{name} = \textbf{ref}\, x\rangle$$

A definition for genConstr:

$$\begin{aligned}\text{genConstr} \triangleq\ & \Lambda R{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.\\ & \Lambda C{::}\{c{::}\text{Rec} \mid \textbf{lab}(c)\#\textbf{lab}(R)\}.\lambda v{:}C.\\ & \lambda x{:}\textbf{headType}(R).\textbf{case}\ R\ \textbf{of}\\ & \quad \langle \ell : T\rangle@\langle\rangle \Rightarrow \langle \ell = \textbf{ref}\, x\rangle@v\\ & \ \mid \langle \ell : T\rangle@R' \Rightarrow\end{aligned}$$

Generic Constructor Generator

Easier to use an accumulator parameter:

$$\text{genConstr Person } \langle\rangle\ \langle\rangle \to^{*} \lambda x{:}\text{String}.\lambda y{:}\text{Int}.\langle \textit{age} = \textbf{ref}\, y; \textit{name} = \textbf{ref}\, x\rangle$$

A definition for genConstr:

$$\begin{aligned}\text{genConstr} \triangleq\ & \Lambda R{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.\\ & \Lambda C{::}\{c{::}\text{Rec} \mid \textbf{lab}(c)\#\textbf{lab}(R)\}.\lambda v{:}C.\\ & \lambda x{:}\textbf{headType}(R).\textbf{case}\ R\ \textbf{of}\\ & \quad \langle \ell : T\rangle@\langle\rangle \Rightarrow \langle \ell = \textbf{ref}\ x\rangle@v\\ & \ \mid \langle \ell : T\rangle@R' \Rightarrow \text{genConstr}\ R'\end{aligned}$$

Generic Constructor Generator

Easier to use an accumulator parameter:

$$\text{genConstr Person } \langle\rangle\ \langle\rangle \to^* \lambda x{:}\text{String}.\lambda y{:}\text{Int}.\langle \textit{age} = \textbf{ref}\ y; \textit{name} = \textbf{ref}\ x\rangle$$

A definition for genConstr:

$$\begin{aligned}\text{genConstr} \triangleq\ & \Lambda R{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.\\ & \Lambda C{::}\{c{::}\text{Rec} \mid \textbf{lab}(c)\#\textbf{lab}(R)\}.\lambda v{:}C.\\ & \lambda x{:}\textbf{headType}(R).\textbf{case}\ R\ \textbf{of}\\ & \quad \langle \ell : T\rangle @ \langle\rangle \Rightarrow \langle \ell = \textbf{ref}\ x\rangle @ v\\ & \ \mid \langle \ell : T\rangle @ R' \Rightarrow \text{genConstr}\ R'\ (\langle \ell : \textbf{ref}\ T\rangle @ V)\end{aligned}$$

Generic Constructor Generator

Easier to use an accumulator parameter:

$$\text{genConstr Person } \langle\rangle \ \langle\rangle \to^* \lambda x{:}\text{String}.\lambda y{:}\text{Int}.\langle \textit{age} = \textbf{ref}\, y; \textit{name} = \textbf{ref}\, x\rangle$$

A definition for genConstr:

$$\begin{aligned}\text{genConstr} \triangleq\ & \Lambda R{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.\\ & \Lambda C{::}\{c{::}\text{Rec} \mid \textbf{lab}(c)\#\textbf{lab}(R)\}.\lambda v{:}C.\\ & \lambda x{:}\textbf{headType}(R).\textbf{case}\ R\ \textbf{of}\\ & \quad \langle \ell : T\rangle@\langle\rangle \Rightarrow \langle \ell = \textbf{ref}\, x\rangle@v\\ & \ \mid \langle \ell : T\rangle@R' \Rightarrow \text{genConstr}\ R'\ (\langle \ell : \textbf{ref}\, T\rangle@V)\ (\langle \ell = \textbf{ref}\, x\rangle@v)\end{aligned}$$

Generic Constructor Generator

Typing

genConstr :

Generic Constructor Generator

Typing

$$\text{genConstr} : \quad \forall R{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.$$
$$\forall C{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(R)\}.$$

Generic Constructor Generator

Typing

$$\text{genConstr} : \quad \forall R{::}\{r{::}\mathsf{Rec} \mid \neg\mathsf{empty}(r)\}.$$
$$\forall C{::}\{v{::}\mathsf{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(R)\}.C \rightarrow (\mathsf{GenConsT}\; R\; C)$$

Generic Constructor Generator

Typing

$$\text{genConstr} : \forall R{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.\ \forall C{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(R)\}.C \rightarrow (\text{GenConsT}\, R\, C)$$

$$\text{GenConsT} :: \Pi R{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.\ \Pi C{::}\{c{::}\text{Rec} \mid \textbf{lab}(c)\#\textbf{lab}(R)\}.\ \text{Fun}$$

$$\text{GenConsT} \triangleq$$

Generic Constructor Generator

Typing

$$\text{genConstr} : \forall R{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.$$
$$\forall C{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(R)\}.C \to (\text{GenConsT}\, R\, C)$$

$$\text{GenConsT} :: \Pi R{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.$$
$$\Pi C{::}\{c{::}\text{Rec} \mid \textbf{lab}(c)\#\textbf{lab}(R)\}.\,\text{Fun}$$

$$\text{GenConsT} \triangleq$$

Generic Constructor Generator
Typing

$$\text{genConstr} : \forall R{::}\{r{::}\mathsf{Rec} \mid \neg\mathsf{empty}(r)\}.$$
$$\forall C{::}\{v{::}\mathsf{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(R)\}.C \rightarrow (\mathsf{GenConsT}\,R\,C)$$

$$\mathsf{GenConsT} :: \Pi R{::}\{r{::}\mathsf{Rec} \mid \neg\mathsf{empty}(r)\}.$$
$$\Pi C{::}\{c{::}\mathsf{Rec} \mid \mathbf{lab}(c)\#\mathbf{lab}(R)\}.\,\mathsf{Fun}$$

$$\mathsf{GenConsT} \triangleq$$

Generic Constructor Generator

Typing

$$\mathsf{genConstr} : \forall R{::}\{r{::}\mathsf{Rec} \mid \neg\mathsf{empty}(r)\}. \; \forall C{::}\{v{::}\mathsf{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(R)\}.C \to (\mathsf{GenConsT}\, R\, C)$$

$$\mathsf{GenConsT} :: \Pi R{::}\{r{::}\mathsf{Rec} \mid \neg\mathsf{empty}(r)\}. \; \Pi C{::}\{c{::}\mathsf{Rec} \mid \mathbf{lab}(c)\#\mathbf{lab}(R)\}.\,\mathsf{Fun}$$

$$\mathsf{GenConsT} \triangleq \lambda R{::}\{r{::}\mathsf{Rec} \mid \neg\mathsf{empty}(r)\}. \; \lambda C{::}\{v{::}\mathsf{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(R)\}.$$

Generic Constructor Generator

Typing

$$\text{genConstr} : \forall R{::}\{r{::}\mathsf{Rec} \mid \neg\mathsf{empty}(r)\}.\ \forall C{::}\{v{::}\mathsf{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(R)\}.C \rightarrow (\mathsf{GenConsT}\ R\ C)$$

$$\mathsf{GenConsT} :: \Pi R{::}\{r{::}\mathsf{Rec} \mid \neg\mathsf{empty}(r)\}.\ \Pi C{::}\{c{::}\mathsf{Rec} \mid \mathbf{lab}(c)\#\mathbf{lab}(R)\}.\ \mathsf{Fun}$$

$$\mathsf{GenConsT} \triangleq \lambda R{::}\{r{::}\mathsf{Rec} \mid \neg\mathsf{empty}(r)\}.\ \lambda C{::}\{v{::}\mathsf{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(R)\}.\mathbf{headType}(R) \rightarrow$$

Generic Constructor Generator

Typing

$$\text{genConstr} : \quad \forall R{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.\ \forall C{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(R)\}.C \to (\text{GenConsT}\,R\,C)$$

$$\begin{aligned}
\text{GenConsT} &:: \quad \Pi R{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.\\
&\qquad \Pi C{::}\{c{::}\text{Rec} \mid \textbf{lab}(c)\#\textbf{lab}(R)\}.\,\text{Fun}\\
\text{GenConsT} &\triangleq \quad \lambda R{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.\\
&\qquad \lambda C{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(R)\}.\textbf{headType}(R) \to\\
&\qquad\quad \textbf{case}\ R\ \textbf{of}\\
&\qquad\qquad \langle \ell : T\rangle @\langle\rangle \Rightarrow\\
&\qquad\qquad \mid \langle \ell : T\rangle @R' \Rightarrow
\end{aligned}$$

Generic Constructor Generator

Typing

$$\text{genConstr} : \forall R{::}\{r{::}\mathsf{Rec} \mid \neg\mathsf{empty}(r)\}.\ \forall C{::}\{v{::}\mathsf{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(R)\}.C \to (\mathsf{GenConsT}\, R\, C)$$

$$\begin{aligned}
\mathsf{GenConsT} &:: \Pi R{::}\{r{::}\mathsf{Rec} \mid \neg\mathsf{empty}(r)\}.\ \Pi C{::}\{c{::}\mathsf{Rec} \mid \mathbf{lab}(c)\#\mathbf{lab}(R)\}.\ \mathsf{Fun} \\
\mathsf{GenConsT} &\triangleq \lambda R{::}\{r{::}\mathsf{Rec} \mid \neg\mathsf{empty}(r)\}.\ \lambda C{::}\{v{::}\mathsf{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(R)\}.\mathbf{headType}(R) \to \\
&\quad \mathbf{case}\ R\ \mathbf{of} \\
&\qquad \langle \ell : T\rangle @\langle\rangle \Rightarrow \langle \ell : \mathbf{ref}\, T\rangle @C \\
&\qquad \mid \langle \ell : T\rangle @R' \Rightarrow \mathsf{GenConsT}\, R'\, \langle \ell : \mathbf{ref}\, T\rangle @C
\end{aligned}$$

Generic Constructor Generator

Typing

$$\text{genConstr} : \forall R{::}\{r{::}\mathsf{Rec} \mid \neg\mathsf{empty}(r)\}. \forall C{::}\{v{::}\mathsf{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(R)\}.C \to (\mathsf{GenConsT}\, R\, C)$$

$$\mathsf{GenConsT} :: \Pi R{::}\{r{::}\mathsf{Rec} \mid \neg\mathsf{empty}(r)\}. \Pi C{::}\{c{::}\mathsf{Rec} \mid \mathbf{lab}(c)\#\mathbf{lab}(R)\}. \mathsf{Fun}$$

$$\begin{array}{l} \mathsf{GenConsT} \triangleq \lambda R{::}\{r{::}\mathsf{Rec} \mid \neg\mathsf{empty}(r)\}. \\ \quad \lambda C{::}\{v{::}\mathsf{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(R)\}.\mathbf{headType}(R) \to \\ \quad\quad \mathbf{case}\ R\ \mathbf{of} \\ \quad\quad\quad \langle \ell : T\rangle @\langle\rangle \Rightarrow \langle \ell : \mathbf{ref}\, T\rangle @C \\ \quad\quad\quad \mid \langle \ell : T\rangle @R' \Rightarrow \mathsf{GenConsT}\, R'\, \langle \ell : \mathbf{ref}\, T\rangle @C \end{array}$$

$$\begin{array}{l} \mathsf{GenConsT}\ \langle L_1 : T_1; \ldots; L_n : T_n\rangle\ \langle\rangle \longrightarrow^* \\ \quad T_1 \to \cdots \to T_n \to \langle L_n : \mathbf{ref}\, T_n; \ldots; L_1 : \mathbf{ref}\, T_1\rangle \end{array}$$

Generic Record Type Map

Uniformly changing the fields of a record **type**:

$$\text{Map} ::$$
$$\text{Map} \triangleq$$

Generic Record Type Map

Uniformly changing the fields of a record **type**:

$$\mathsf{Map} :: \quad \Pi G{::}(\mathsf{Type} \to \mathsf{Type}).$$
$$\mathsf{Map} \triangleq$$

Generic Record Type Map

Uniformly changing the fields of a record **type**:

$$\mathsf{Map} :: \quad \Pi G{::}(\mathsf{Type} \to \mathsf{Type}).\, \Pi R{::}\mathsf{Rec}.$$
$$\mathsf{Map} \triangleq$$

Generic Record Type Map

Uniformly changing the fields of a record **type**:

$$\mathsf{Map} :: \quad \Pi G{::}(\mathsf{Type} \to \mathsf{Type}).\, \Pi R{::}\mathsf{Rec}.\, \{r :: \mathsf{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\}$$

$$\mathsf{Map} \triangleq$$

Generic Record Type Map

Uniformly changing the fields of a record **type**:

$$\begin{aligned} \mathsf{Map} &:: \quad \Pi G{::}(\mathsf{Type} \to \mathsf{Type}).\, \Pi R{::}\mathsf{Rec}.\, \{r :: \mathsf{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\} \\ \mathsf{Map} &\triangleq \quad \lambda G{::}(\mathsf{Type} \to \mathsf{Type}). \end{aligned}$$

Generic Record Type Map

Uniformly changing the fields of a record **type**:

$$\text{Map} :: \quad \Pi G{::}(\text{Type} \to \text{Type}).\, \Pi R{::}\text{Rec}.\, \{r :: \text{Rec} \mid \textbf{lab}(r) = \textbf{lab}(R)\}$$
$$\text{Map} \triangleq \quad \lambda G{::}(\text{Type} \to \text{Type}).\lambda R{::}\text{Rec}.$$

Generic Record Type Map

Uniformly changing the fields of a record **type**:

$$
\begin{aligned}
\mathsf{Map} ::\ & \Pi G{::}(\mathsf{Type} \to \mathsf{Type}).\, \Pi R{::}\mathsf{Rec}.\, \{r :: \mathsf{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\} \\
\mathsf{Map} \triangleq\ & \lambda G{::}(\mathsf{Type} \to \mathsf{Type}).\lambda R{::}\mathsf{Rec}.\mathbf{case}\ R\ \mathbf{of} \\
& \quad\quad \langle \ell : T \rangle @ R' \Rightarrow \\
& \quad\quad \mid \langle\rangle \Rightarrow
\end{aligned}
$$

Generic Record Type Map

Uniformly changing the fields of a record **type**:

$$\begin{array}{ll} \mathsf{Map} :: & \Pi G{::}(\mathsf{Type} \to \mathsf{Type}).\, \Pi R{::}\mathsf{Rec}.\ \{r :: \mathsf{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\} \\ \mathsf{Map} \triangleq & \lambda G{::}(\mathsf{Type} \to \mathsf{Type}).\lambda R{::}\mathsf{Rec}.\mathbf{case}\ R\ \mathbf{of} \\ & \qquad \langle \ell : T \rangle @ R' \Rightarrow \langle \ell : G(T) \rangle @ (\mathsf{Map}\, G\, R') \\ & \qquad \mid \langle\rangle \Rightarrow \langle\rangle \end{array}$$

The kind $\{r :: \mathsf{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\}$ expresses the weakest invariant needed to kind Map.

Generic Record to HTML Table Generator

Generating a table-rendering function for a record type:

$$\begin{array}{ll} \mathsf{XForm} & :: \mathsf{Type} \to \mathsf{Type} \\ \mathsf{XForm} & \triangleq \lambda t{::}\mathsf{Type}.\langle \mathit{tag} : \mathsf{String}; \mathit{toStr} : t \to \mathsf{String}\rangle \end{array}$$

Generic Record to HTML Table Generator

Generating a table-rendering function for a record type:

$$\begin{array}{ll} \mathsf{XForm} & :: \mathsf{Type} \to \mathsf{Type} \\ \mathsf{XForm} & \triangleq \lambda t{::}\mathsf{Type}.\langle \mathit{tag} : \mathsf{String}; \mathit{toStr} : t \to \mathsf{String}\rangle \end{array}$$

Generic Record to HTML Table Generator

Generating a table-rendering function for a record type:

$$\begin{aligned}
&\text{XForm} && :: \text{Type} \to \text{Type} \\
&\text{XForm} && \triangleq \lambda t{::}\text{Type}.\langle \textit{tag} : \text{String}; \textit{toStr} : t \to \text{String}\rangle \\
\\
&\text{Tablifier} && :: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \textbf{lab}(r) = \textbf{lab}(R)\} \\
&\text{Tablifier} && \triangleq \lambda r{::}\text{Rec}.\text{Map } \color{red}{\text{XForm}}\ r
\end{aligned}$$

Generic Record to HTML Table Generator

Generating a table-rendering function for a record type:

$$\begin{aligned}
\text{XForm} &:: \text{Type} \to \text{Type} \\
\text{XForm} &\triangleq \lambda t{::}\text{Type}.\langle \textit{tag} : \text{String}; \textit{toStr} : t \to \text{String}\rangle
\end{aligned}$$

$$\begin{aligned}
\text{Tablifier} &:: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \textbf{lab}(r) = \textbf{lab}(R)\} \\
\text{Tablifier} &\triangleq \lambda r{::}\text{Rec}.\text{Map XForm } r
\end{aligned}$$

$$\begin{aligned}
\text{Tablify} &: \forall R{::}\text{Rec}.(\text{Tablifier } R) \to R \to \text{String} \\
\text{Tablify} &\triangleq
\end{aligned}$$

Generic Record to HTML Table Generator

Generating a table-rendering function for a record type:

$$\begin{aligned}
\text{XForm} &:: \text{Type} \to \text{Type} \\
\text{XForm} &\triangleq \lambda t{::}\text{Type}.\langle \textit{tag} : \text{String}; \textit{toStr} : t \to \text{String}\rangle
\end{aligned}$$

$$\begin{aligned}
\text{Tablifier} &:: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \textbf{lab}(r) = \textbf{lab}(R)\} \\
\text{Tablifier} &\triangleq \lambda r{::}\text{Rec}.\text{Map XForm } r
\end{aligned}$$

$$\begin{aligned}
\text{Tablify} &: \forall R{::}\text{Rec}.(\text{Tablifier } R) \to R \to \text{String} \\
\text{Tablify} &\triangleq
\end{aligned}$$

Generic Record to HTML Table Generator

Generating a table-rendering function for a record type:

$$\begin{aligned}
\text{XForm} &:: \text{Type} \to \text{Type} \\
\text{XForm} &\triangleq \lambda t{::}\text{Type}.\langle \textit{tag} : \text{String}; \textit{toStr} : t \to \text{String}\rangle
\end{aligned}$$

$$\begin{aligned}
\text{Tablifier} &:: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \textbf{lab}(r) = \textbf{lab}(R)\} \\
\text{Tablifier} &\triangleq \lambda r{::}\text{Rec}.\text{Map XForm } r
\end{aligned}$$

$$\begin{aligned}
\text{Tablify} &: \forall R{::}\text{Rec}.(\text{Tablifier } R) \to {\color{red}R} \to \text{String} \\
\text{Tablify} &\triangleq
\end{aligned}$$

Generic Record to HTML Table Generator

Generating a table-rendering function for a record type:

$$\begin{aligned}
\text{XForm} &:: \text{Type} \to \text{Type} \\
\text{XForm} &\triangleq \lambda t{::}\text{Type}.\langle \textit{tag} : \text{String}; \textit{toStr} : t \to \text{String}\rangle
\end{aligned}$$

$$\begin{aligned}
\text{Tablifier} &:: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \textbf{lab}(r) = \textbf{lab}(R)\} \\
\text{Tablifier} &\triangleq \lambda r{::}\text{Rec}.\text{Map XForm } r
\end{aligned}$$

$$\begin{aligned}
\text{Tablify} &: \forall R{::}\text{Rec}.(\text{Tablifier } R) \to R \to \text{String} \\
\text{Tablify} &\triangleq \Lambda R{::}\text{Rec}.
\end{aligned}$$

Generic Record to HTML Table Generator

Generating a table-rendering function for a record type:

$$\begin{aligned}
\text{XForm} \quad & :: \text{Type} \to \text{Type} \\
\text{XForm} \quad & \triangleq \lambda t{::}\text{Type}.\langle \textit{tag} : \text{String}; \textit{toStr} : t \to \text{String}\rangle
\end{aligned}$$

$$\begin{aligned}
\text{Tablifier} \quad & :: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\} \\
\text{Tablifier} \quad & \triangleq \lambda r{::}\text{Rec}.\text{Map XForm } r
\end{aligned}$$

$$\begin{aligned}
\text{Tablify} \quad & : \forall R{::}\text{Rec}.(\text{Tablifier } R) \to R \to \text{String} \\
\text{Tablify} \quad & \triangleq \Lambda R{::}\text{Rec}.\lambda \textit{Tab}{:}(\text{Tablifier } R).\lambda r{:}R.
\end{aligned}$$

Generic Record to HTML Table Generator

Generating a table-rendering function for a record type:

$$\begin{aligned}
\text{XForm} \quad & :: \text{Type} \rightarrow \text{Type} \\
\text{XForm} \quad & \triangleq \lambda t{::}\text{Type}.\langle \textit{tag} : \text{String}; \textit{toStr} : t \rightarrow \text{String}\rangle
\end{aligned}$$

$$\begin{aligned}
\text{Tablifier} \quad & :: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\} \\
\text{Tablifier} \quad & \triangleq \lambda r{::}\text{Rec}.\text{Map XForm } r
\end{aligned}$$

$$\begin{aligned}
\text{Tablify} \quad & : \forall R{::}\text{Rec}.(\text{Tablifier } R) \rightarrow R \rightarrow \text{String} \\
\text{Tablify} \quad & \triangleq \Lambda R{::}\text{Rec}.\lambda \textit{Tab}{:}(\text{Tablifier } R).\lambda r{:}R.\mathbf{case}\ r\ \mathbf{of} \\
& \qquad \langle\rangle \Rightarrow \\
& \quad \mid \langle \ell = v\rangle @ r' \Rightarrow
\end{aligned}$$

Generic Record to HTML Table Generator

Generating a table-rendering function for a record type:

$$\begin{array}{ll} \text{XForm} & :: \text{Type} \to \text{Type} \\ \text{XForm} & \triangleq \lambda t{::}\text{Type}.\langle \textit{tag} : \text{String}; \textit{toStr} : t \to \text{String}\rangle \end{array}$$

$$\begin{array}{ll} \text{Tablifier} & :: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \textbf{lab}(r) = \textbf{lab}(R)\} \\ \text{Tablifier} & \triangleq \lambda r{::}\text{Rec}.\text{Map XForm } r \end{array}$$

$$\begin{array}{ll} \text{Tablify} & : \forall R{::}\text{Rec}.(\text{Tablifier } R) \to R \to \text{String} \\ \text{Tablify} & \triangleq \Lambda R{::}\text{Rec}.\lambda \textit{Tab}{:}(\text{Tablifier } R).\lambda r{:}R.\textbf{case } r \textbf{ of} \\ & \quad \langle\rangle \Rightarrow \text{``''} \\ & \mid \langle \ell = v\rangle @ r' \Rightarrow \end{array}$$

Generic Record to HTML Table Generator

Generating a table-rendering function for a record type:

$$\begin{aligned}
&\text{XForm} && :: \text{Type} \to \text{Type} \\
&\text{XForm} && \triangleq \lambda t{::}\text{Type}.\langle \textit{tag} : \text{String}; \textit{toStr} : t \to \text{String}\rangle
\end{aligned}$$

$$\begin{aligned}
&\text{Tablifier} && :: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \textbf{lab}(r) = \textbf{lab}(R)\} \\
&\text{Tablifier} && \triangleq \lambda r{::}\text{Rec}.\text{Map XForm } r
\end{aligned}$$

$$\begin{aligned}
&\text{Tablify} && : \forall R{::}\text{Rec}.(\text{Tablifier } R) \to R \to \text{String} \\
&\text{Tablify} && \triangleq \Lambda R{::}\text{Rec}.\lambda \textit{Tab}{:}(\text{Tablifier } R).\lambda r{:}R.\textbf{case } r \textbf{ of} \\
& && \qquad \langle\rangle \Rightarrow \text{``''} \\
& && \mid \langle \ell = v\rangle @ r' \Rightarrow \text{``}\texttt{<tr><th>}\text{''} + \color{red}{\textit{Tab}.\ell.\textit{tag}} + \text{``}\texttt{</th><td>}\text{''} +
\end{aligned}$$

Generic Record to HTML Table Generator

Generating a table-rendering function for a record type:

$$\begin{aligned}
\text{XForm} &:: \text{Type} \to \text{Type} \\
\text{XForm} &\triangleq \lambda t{::}\text{Type}.\langle \mathit{tag} : \text{String}; \mathit{toStr} : t \to \text{String}\rangle
\end{aligned}$$

$$\begin{aligned}
\text{Tablifier} &:: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\} \\
\text{Tablifier} &\triangleq \lambda r{::}\text{Rec}.\text{Map XForm } r
\end{aligned}$$

$$\begin{aligned}
\text{Tablify} &: \forall R{::}\text{Rec}.(\text{Tablifier } R) \to R \to \text{String} \\
\text{Tablify} &\triangleq \Lambda R{::}\text{Rec}.\lambda \mathit{Tab}{:}(\text{Tablifier } R).\lambda r{:}R.\mathbf{case}\ r\ \mathbf{of} \\
&\qquad \langle\rangle \Rightarrow \text{``''} \\
&\quad \mid \langle \ell = v\rangle @ r' \Rightarrow \text{``}\texttt{<tr><th>}\text{''} + \mathit{Tab}.\ell.\mathit{tag} + \text{``}\texttt{</th><td>}\text{''} + \\
&\qquad\qquad \mathit{Tab}.\ell.\mathit{toStr}\ v + \text{``}\texttt{</td></tr>}\text{''} +
\end{aligned}$$

Generic Record to HTML Table Generator

Generating a table-rendering function for a record type:

$$\begin{aligned}
\text{XForm} &:: \text{Type} \to \text{Type} \\
\text{XForm} &\triangleq \lambda t{::}\text{Type}.\langle \mathit{tag} : \text{String}; \mathit{toStr} : t \to \text{String}\rangle
\end{aligned}$$

$$\begin{aligned}
\text{Tablifier} &:: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \textbf{lab}(r) = \textbf{lab}(R)\} \\
\text{Tablifier} &\triangleq \lambda r{::}\text{Rec}.\text{Map XForm } r
\end{aligned}$$

$$\begin{aligned}
\text{Tablify} &: \forall R{::}\text{Rec}.(\text{Tablifier } R) \to R \to \text{String} \\
\text{Tablify} &\triangleq \Lambda R{::}\text{Rec}.\lambda \mathit{Tab}{:}(\text{Tablifier } R).\lambda r{:}R.\textbf{case } r \textbf{ of} \\
&\qquad \langle\rangle \Rightarrow \text{``''} \\
&\quad \mid \langle \ell = v\rangle @ r' \Rightarrow \text{``}\texttt{<tr><th>}\text{''} + \mathit{Tab}.\ell.\mathit{tag} + \text{``}\texttt{</th><td>}\text{''} + \\
&\qquad\qquad \mathit{Tab}.\ell.\mathit{toStr}\, v + \text{``}\texttt{</td></tr>}\text{''} + \text{Tablify } \textbf{tail}(R)\ \textbf{recTail}(\mathit{Tab})\ r'
\end{aligned}$$

Generic Record to HTML Table Generator

Generating a table-rendering function for a record type:

$$\begin{array}{ll} \text{XForm} & :: \text{Type} \to \text{Type} \\ \text{XForm} & \triangleq \lambda t{::}\text{Type}.\langle \mathit{tag} : \text{String}; \mathit{toStr} : t \to \text{String}\rangle \end{array}$$

$$\begin{array}{ll} \text{Tablifier} & :: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\} \\ \text{Tablifier} & \triangleq \lambda r{::}\text{Rec}.\text{Map XForm } r \end{array}$$

$$\begin{array}{ll} \text{Tablify} & : \forall R{::}\text{Rec}.(\text{Tablifier } R) \to R \to \text{String} \\ \text{Tablify} & \triangleq \Lambda R{::}\text{Rec}.\lambda \mathit{Tab}{:}(\text{Tablifier } R).\lambda r{:}R.\mathbf{case}\ r\ \mathbf{of} \\ & \qquad \langle\rangle \Rightarrow \text{``''} \\ & \quad \mid \langle \ell = v\rangle @ r' \Rightarrow \text{``}\texttt{<tr><th>}\text{''} + \mathit{Tab}.\ell.\mathit{tag} + \text{``}\texttt{</th><td>}\text{''} + \\ & \qquad\qquad \mathit{Tab}.\ell.\mathit{toStr}\ v + \text{``}\texttt{</td></tr>}\text{''} + \text{Tablify}\ \mathbf{tail}(R)\ \mathbf{recTail}(\mathit{Tab})\ r' \end{array}$$

Generic Record to HTML Table Generator

Generating a table-rendering function for a record type:

$$\begin{aligned}
&\text{XForm} && :: \text{Type} \to \text{Type} \\
&\text{XForm} && \triangleq \lambda t{::}\text{Type}.\langle \mathit{tag} : \text{String}; \mathit{toStr} : t \to \text{String}\rangle
\end{aligned}$$

$$\begin{aligned}
&\text{Tablifier} && :: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \textbf{lab}(r) = \textbf{lab}(R)\} \\
&\text{Tablifier} && \triangleq \lambda r{::}\text{Rec}.\text{Map XForm } r
\end{aligned}$$

$$\begin{aligned}
&\text{Tablify} && : \forall R{::}\text{Rec}.(\text{Tablifier } R) \to R \to \text{String} \\
&\text{Tablify} && \triangleq \Lambda R{::}\text{Rec}.\lambda \mathit{Tab}{:}(\text{Tablifier } R).\lambda r{:}R.\textbf{case } r \textbf{ of} \\
& && \qquad \langle\rangle \Rightarrow \text{``''} \\
& && \mid \langle \ell = v\rangle @ r' \Rightarrow \text{``}\texttt{<tr><th>}\text{''} + \mathit{Tab}.\ell.\mathit{tag} + \text{``}\texttt{</th><td>}\text{''} + \\
& && \qquad \mathit{Tab}.\ell.\mathit{toStr}\, v + \text{``}\texttt{</td></tr>}\text{''} + \text{Tablify } \textbf{tail}(R)\ \textbf{recTail}(\mathit{Tab})\ {\color{red} r'}
\end{aligned}$$

Examples

More examples in the paper...

- Ad-hoc polymorphism and type transformations.
- Generic (record) table lookup.
- Object Pairing.

Outline

Kinds, Types and Refinements

$$K, K' \quad ::= \quad \mathcal{K} \mid \{t{::}\mathcal{K} \mid \varphi\} \mid \Pi t{:}K.K'$$

Kinds, Types and Refinements

$$K, K' \quad ::= \quad \mathcal{K} \mid \{t{::}\mathcal{K} \mid \varphi\} \mid \Pi t{:}K.K'$$

Kinds, Types and Refinements

$$
\begin{array}{lcl}
K, K' & ::= & \mathcal{K} \mid \{t{::}\mathcal{K} \mid \varphi\} \mid \Pi t{:}K.K' \\
\mathcal{K} & ::= & \mathsf{Rec} \mid \mathsf{Col} \mid \mathsf{Fun} \mid \mathsf{Ref} \mid \mathsf{Nm} \mid \mathsf{Type} \mid \mathsf{Gen}_K
\end{array}
$$

Kinds, Types and Refinements

$$
\begin{array}{lll}
K, K' & ::= & \mathcal{K} \mid \{t{::}\mathcal{K} \mid \varphi\} \mid \Pi t{:}K.K' \\
\mathcal{K} & ::= & \mathsf{Rec} \mid \mathsf{Col} \mid \mathsf{Fun} \mid \mathsf{Ref} \mid \mathsf{Nm} \mid \mathsf{Type} \mid \mathsf{Gen}_K \\
 & & \\
T, S, R & ::= & t \mid \lambda t{::}K.T \mid T\ S
\end{array}
$$

Kinds, Types and Refinements

$$
\begin{array}{lcl}
K, K' & ::= & \mathcal{K} \mid \{t{::}\mathcal{K} \mid \varphi\} \mid \Pi t{:}K.K' \\
\mathcal{K} & ::= & \mathsf{Rec} \mid \mathsf{Col} \mid \mathsf{Fun} \mid \mathsf{Ref} \mid \mathsf{Nm} \mid \mathsf{Type} \mid \mathsf{Gen}_K \\
\\
T, S, R & ::= & t \mid \lambda t{::}K.T \mid T\ S \\
& \mid & \boldsymbol{\mu} F : (\Pi t{:}K.\, K').\lambda t{::}K.\, T
\end{array}
$$

Kinds, Types and Refinements

$$
\begin{array}{lcl}
K, K' & ::= & \mathcal{K} \mid \{t{::}\mathcal{K} \mid \varphi\} \mid \Pi t{:}K.K' \\
\mathcal{K} & ::= & \mathsf{Rec} \mid \mathsf{Col} \mid \mathsf{Fun} \mid \mathsf{Ref} \mid \mathsf{Nm} \mid \mathsf{Type} \mid \mathsf{Gen}_K
\end{array}
$$

$$
\begin{array}{lcl}
T, S, R & ::= & t \mid \lambda t{::}K.T \mid T\ S \\
& \mid & \boldsymbol{\mu} F : (\Pi t{:}K.\, K').\lambda t{::}K.\, T \\
& \mid & \forall t{::}K.T
\end{array}
$$

Kinds, Types and Refinements

$$
\begin{array}{lcl}
K, K' & ::= & \mathcal{K} \mid \{t{::}\mathcal{K} \mid \varphi\} \mid \Pi t{:}K.K' \\
\mathcal{K} & ::= & \mathsf{Rec} \mid \mathsf{Col} \mid \mathsf{Fun} \mid \mathsf{Ref} \mid \mathsf{Nm} \mid \mathsf{Type} \mid \mathsf{Gen}_K \\
\\
T, S, R & ::= & t \mid \lambda t{::}K.T \mid T\ S \\
& \mid & \boldsymbol{\mu} F : (\Pi t{:}K.\, K').\lambda t{::}K.\, T \\
& \mid & \forall t{::}K.T \\
& \mid & L \mid \langle\rangle \mid \langle L : T\rangle @ S
\end{array}
$$

Kinds, Types and Refinements

$$
\begin{array}{lcl}
K, K' & ::= & \mathcal{K} \mid \{t{::}\mathcal{K} \mid \varphi\} \mid \Pi t{:}K.K' \\
\mathcal{K} & ::= & \textsf{Rec} \mid \textsf{Col} \mid \textsf{Fun} \mid \textsf{Ref} \mid \textsf{Nm} \mid \textsf{Type} \mid \textsf{Gen}_K \\
 & & \\
T, S, R & ::= & t \mid \lambda t{::}K.T \mid T\ S \\
 & \mid & \boldsymbol{\mu} F : (\Pi t{:}K.\, K').\lambda t{::}K.\, T \\
 & \mid & \forall t{::}K.T \\
 & \mid & L \mid {\color{red}\langle\rangle} \mid \langle L : T \rangle @ S
\end{array}
$$

Kinds, Types and Refinements

$$
\begin{array}{lcl}
K, K' & ::= & \mathcal{K} \mid \{t{::}\mathcal{K} \mid \varphi\} \mid \Pi t{:}K.K' \\
\mathcal{K} & ::= & \mathsf{Rec} \mid \mathsf{Col} \mid \mathsf{Fun} \mid \mathsf{Ref} \mid \mathsf{Nm} \mid \mathsf{Type} \mid \mathsf{Gen}_K \\
 & & \\
T, S, R & ::= & t \mid \lambda t{::}K.T \mid T\, S \\
 & \mid & \boldsymbol{\mu} F : (\Pi t{:}K.\, K').\lambda t{::}K.\, T \\
 & \mid & \forall t{::}K.T \\
 & \mid & L \mid \langle\rangle \mid \langle L : T\rangle @ S
\end{array}
$$

Kinds, Types and Refinements

$$
\begin{array}{lcl}
K, K' & ::= & \mathcal{K} \mid \{t{::}\mathcal{K} \mid \varphi\} \mid \Pi t{:}K.K' \\
\mathcal{K} & ::= & \textsf{Rec} \mid \textsf{Col} \mid \textsf{Fun} \mid \textsf{Ref} \mid \textsf{Nm} \mid \textsf{Type} \mid \textsf{Gen}_K \\
 & & \\
T, S, R & ::= & t \mid \lambda t{::}K.T \mid T\ S \\
 & \mid & \boldsymbol{\mu} F : (\Pi t{:}K.\ K').\lambda t{::}K.\ T \\
 & \mid & \forall t{::}K.T \\
 & \mid & L \mid \langle\rangle \mid \langle L : T\rangle @ S \\
 & \mid & \textbf{headLabel}(T) \mid \textbf{headType}(T) \mid \textbf{tail}(T)
\end{array}
$$

Kinds, Types and Refinements

$$
\begin{array}{lcl}
K, K' & ::= & \mathcal{K} \mid \{t{::}\mathcal{K} \mid \varphi\} \mid \Pi t{:}K.K' \\
\mathcal{K} & ::= & \mathsf{Rec} \mid \mathsf{Col} \mid \mathsf{Fun} \mid \mathsf{Ref} \mid \mathsf{Nm} \mid \mathsf{Type} \mid \mathsf{Gen}_K
\end{array}
$$

$$
\begin{array}{lcl}
T, S, R & ::= & t \mid \lambda t{::}K.T \mid T\ S \\
& \mid & \boldsymbol{\mu} F : (\Pi t{:}K.\ K').\lambda t{::}K.\ T \\
& \mid & \forall t{::}K.T \\
& \mid & L \mid \langle\rangle \mid \langle L : T \rangle @ S \\
& \mid & \textbf{headLabel}(T) \mid \textbf{headType}(T) \mid \textbf{tail}(T)
\end{array}
$$

Kinds, Types and Refinements

$$
\begin{array}{lll}
K, K' & ::= & \mathcal{K} \mid \{t{::}\mathcal{K} \mid \varphi\} \mid \Pi t{:}K.K' \\
\mathcal{K} & ::= & \textsf{Rec} \mid \textsf{Col} \mid \textsf{Fun} \mid \textsf{Ref} \mid \textsf{Nm} \mid \textsf{Type} \mid \textsf{Gen}_K
\end{array}
$$

$$
\begin{array}{lll}
T, S, R & ::= & t \mid \lambda t{::}K.T \mid T\ S \\
& \mid & \boldsymbol{\mu} F : (\Pi t{:}K.\ K').\lambda t{::}K.\ T \\
& \mid & \forall t{::}K.T \\
& \mid & L \mid \langle\rangle \mid \langle L : T\rangle @ S \\
& \mid & \textbf{headLabel}(T) \mid \textbf{headType}(T) \mid \textbf{tail}(T)
\end{array}
$$

Kinds, Types and Refinements

$$
\begin{array}{lcl}
K, K' & ::= & \mathcal{K} \mid \{t{::}\mathcal{K} \mid \varphi\} \mid \Pi t{:}K.K' \\
\mathcal{K} & ::= & \textsf{Rec} \mid \textsf{Col} \mid \textsf{Fun} \mid \textsf{Ref} \mid \textsf{Nm} \mid \textsf{Type} \mid \textsf{Gen}_K \\
\\
T, S, R & ::= & t \mid \lambda t{::}K.T \mid T\ S \\
& \mid & \boldsymbol{\mu} F : (\Pi t{:}K.\,K').\lambda t{::}K.\,T \\
& \mid & \forall t{::}K.T \\
& \mid & L \mid \langle\rangle \mid \langle L : T\rangle @ S \\
& \mid & \textbf{headLabel}(T) \mid \textbf{headType}(T) \mid \textbf{tail}(T) \\
& \mid & T \rightarrow S \mid \textbf{dom}(T) \mid \textbf{img}(T)
\end{array}
$$

Kinds, Types and Refinements

$$
\begin{array}{lcl}
K, K' & ::= & \mathcal{K} \mid \{t{::}\mathcal{K} \mid \varphi\} \mid \Pi t{:}K.K' \\
\mathcal{K} & ::= & \textsf{Rec} \mid \textsf{Col} \mid \textsf{Fun} \mid \textsf{Ref} \mid \textsf{Nm} \mid \textsf{Type} \mid \textsf{Gen}_K \\
\\
T, S, R & ::= & t \mid \lambda t{::}K.T \mid T\ S \\
& \mid & \boldsymbol{\mu} F : (\Pi t{:}K.\, K').\lambda t{::}K.\, T \\
& \mid & \forall t{::}K.T \\
& \mid & L \mid \langle\rangle \mid \langle L : T\rangle @ S \\
& \mid & \textbf{headLabel}(T) \mid \textbf{headType}(T) \mid \textbf{tail}(T) \\
& \mid & T \to S \mid \textbf{dom}(T) \mid \textbf{img}(T)
\end{array}
$$

Kinds, Types and Refinements

$$
\begin{array}{lcl}
K, K' & ::= & \mathcal{K} \mid \{t{::}\mathcal{K} \mid \varphi\} \mid \Pi t{:}K.K' \\
\mathcal{K} & ::= & \mathsf{Rec} \mid \mathsf{Col} \mid \mathsf{Fun} \mid \mathsf{Ref} \mid \mathsf{Nm} \mid \mathsf{Type} \mid \mathsf{Gen}_K
\end{array}
$$

$$
\begin{array}{lcl}
T, S, R & ::= & t \mid \lambda t{::}K.T \mid T\ S \\
& \mid & \boldsymbol{\mu} F : (\Pi t{:}K.\,K').\lambda t{::}K.\,T \\
& \mid & \forall t{::}K.T \\
& \mid & L \mid \langle\rangle \mid \langle L : T\rangle @ S \\
& \mid & \mathbf{headLabel}(T) \mid \mathbf{headType}(T) \mid \mathbf{tail}(T) \\
& \mid & T \to S \mid \mathbf{dom}(T) \mid \mathbf{img}(T)
\end{array}
$$

Kinds, Types and Refinements

$$
\begin{array}{lcl}
K, K' & ::= & \mathcal{K} \mid \{t{::}\mathcal{K} \mid \varphi\} \mid \Pi t{:}K.K' \\
\mathcal{K} & ::= & \mathsf{Rec} \mid \mathsf{Col} \mid \mathsf{Fun} \mid \mathsf{Ref} \mid \mathsf{Nm} \mid \mathsf{Type} \mid \mathsf{Gen}_K \\
 & & \\
T, S, R & ::= & t \mid \lambda t{::}K.T \mid T\ S \\
 & \mid & \boldsymbol{\mu} F : (\Pi t{:}K.\, K').\lambda t{::}K.\, T \\
 & \mid & \forall t{::}K.T \\
 & \mid & L \mid \langle\rangle \mid \langle L : T\rangle @ S \\
 & \mid & \mathbf{headLabel}(T) \mid \mathbf{headType}(T) \mid \mathbf{tail}(T) \\
 & \mid & T \to S \mid \mathbf{dom}(T) \mid \mathbf{img}(T) \\
 & \mid & \mathbf{case}\ T\ \mathbf{of}\ p_1 \Rightarrow S_1 \mid \ldots \mid p_n \Rightarrow S_n \\
 & \mid & \ldots
\end{array}
$$

Kinds, Types and Refinements

$$
\begin{array}{lcl}
K, K' & ::= & \mathcal{K} \mid \{t{::}\mathcal{K} \mid \varphi\} \mid \Pi t{:}K.K' \\
\mathcal{K} & ::= & \mathsf{Rec} \mid \mathsf{Col} \mid \mathsf{Fun} \mid \mathsf{Ref} \mid \mathsf{Nm} \mid \mathsf{Type} \mid \mathsf{Gen}_K \\
\\
T, S, R & ::= & t \mid \lambda t{::}K.T \mid T\ S \\
& \mid & \boldsymbol{\mu} F : (\Pi t{:}K.\, K').\lambda t{::}K.\, T \\
& \mid & \forall t{::}K.T \\
& \mid & L \mid \langle\rangle \mid \langle L : T\rangle @ S \\
& \mid & \mathbf{headLabel}(T) \mid \mathbf{headType}(T) \mid \mathbf{tail}(T) \\
& \mid & T \rightarrow S \mid \mathbf{dom}(T) \mid \mathbf{img}(T) \\
& \mid & \mathbf{case}\ T\ \mathbf{of}\ p_1 \Rightarrow S_1 \mid \ldots \mid p_n \Rightarrow S_n \\
& \mid & \ldots \\
\varphi, \psi & ::= & \mathsf{empty}(T) \mid T = S \mid T \in \mathbf{lab}(S) \mid T \# S
\end{array}
$$

Program Terms

$$M, N \quad ::= \quad x \mid \lambda x{:}T.M \mid M\ N$$

Program Terms

$$
\begin{array}{rcl}
M, N & ::= & x \mid \lambda x{:}T.M \mid M\ N \\
 & \mid & \Lambda t{::}K.M \mid M[T]
\end{array}
$$

Program Terms

$$
\begin{array}{lll}
M, N & ::= & x \mid \lambda x{:}T.M \mid M\ N \\
& \mid & \Lambda t{::}K.M \mid M[T] \\
& \mid & \langle\rangle \mid \langle \ell = M \rangle @ N \\
& \mid &
\end{array}
$$

Program Terms

$$
\begin{array}{rcl}
M, N & ::= & x \mid \lambda x{:}T.M \mid M\ N \\
& \mid & \Lambda t{::}K.M \mid M[T] \\
& \mid & \langle\rangle \mid \langle \ell = M \rangle @ N \\
& \mid & \textbf{case}\ M\ \textbf{of}\ p_1 \Rightarrow N_1 \mid \ldots \mid p_n \Rightarrow N_n
\end{array}
$$

Program Terms

$$
\begin{array}{rcl}
M, N & ::= & x \mid \lambda x{:}T.M \mid M\ N \\
& \mid & \Lambda t{::}K.M \mid M[T] \\
& \mid & \langle\rangle \mid \langle \ell = M \rangle @ N \\
& \mid & \textbf{case}\ M\ \textbf{of}\ p_1 \Rightarrow N_1 \mid \ldots \mid p_n \Rightarrow N_n \\
& \mid & \textbf{case}\ T\ \textbf{of}\ p_1 \Rightarrow N_1 \mid \ldots \mid p_n \Rightarrow N_n
\end{array}
$$

Program Terms

$$
\begin{array}{rcl}
M, N & ::= & x \mid \lambda x{:}T.M \mid M\ N \\
& \mid & \Lambda t{::}K.M \mid M[T] \\
& \mid & \langle\rangle \mid \langle \ell = M \rangle @ N \\
& \mid & \textbf{case}\ M\ \textbf{of}\ p_1 \Rightarrow N_1 \mid \ \ldots \ \mid p_n \Rightarrow N_n \\
& \mid & \textbf{case}\ T\ \textbf{of}\ p_1 \Rightarrow N_1 \mid \ \ldots \ \mid p_n \Rightarrow N_n \\
& \mid & \boldsymbol{\mu} F{:}T.M \\
& \mid & \ldots
\end{array}
$$

Program Terms

$$
\begin{array}{rcl}
M, N & ::= & x \mid \lambda x{:}T.M \mid M\ N \\
& \mid & \Lambda t{::}K.M \mid M[T] \\
& \mid & \langle\rangle \mid \langle \ell = M \rangle @ N \\
& \mid & \textbf{case}\ M\ \textbf{of}\ p_1 \Rightarrow N_1 \mid \ldots \mid p_n \Rightarrow N_n \\
& \mid & \textbf{case}\ T\ \textbf{of}\ p_1 \Rightarrow N_1 \mid \ldots \mid p_n \Rightarrow N_n \\
& \mid & \boldsymbol{\mu} F{:}T.M \\
& \mid & \ldots
\end{array}
$$

- ML-like core language (records, polymorphism, references, general recursion).

Sample Rules

Refinement Kinds and Refinement Validity:

(KREF)

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}$$

Sample Rules

Refinement Kinds and Refinement Validity:

(KREF)

$$\frac{\Gamma \models \varphi\{T/t\} \qquad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}$$

Sample Rules

Refinement Kinds and Refinement Validity:

(KREF)

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}$$

Sample Rules

Refinement Kinds and Refinement Validity:

(KREF)

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}$$

(ENTAILS)

$$\frac{\Gamma \vdash \varphi \quad \mathsf{Valid}([\![\Gamma]\!] \Rightarrow [\![\varphi]\!])}{\Gamma \models \varphi}$$

Sample Rules

Refinement Kinds and Refinement Validity:

(KREF)

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}$$

(ENTAILS)

$$\frac{\Gamma \vdash \varphi \quad \textsf{Valid}(\llbracket \Gamma \rrbracket \Rightarrow \llbracket \varphi \rrbracket)}{\Gamma \models \varphi}$$

Sample Rules

Refinement Kinds and Refinement Validity:

(KREF)

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}$$

(ENTAILS)

$$\frac{\Gamma \vdash \varphi \quad \mathsf{Valid}(\llbracket\Gamma\rrbracket \Rightarrow \llbracket\varphi\rrbracket)}{\Gamma \models \varphi}$$

Record Type Constructor/Destructors:

Sample Rules

Refinement Kinds and Refinement Validity:

(KREF)
$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}$$

(ENTAILS)
$$\frac{\Gamma \vdash \varphi \quad \mathsf{Valid}(\llbracket\Gamma\rrbracket \Rightarrow \llbracket\varphi\rrbracket)}{\Gamma \models \varphi}$$

Record Type Constructor/Destructors:

(K-RECCONS)
$$\frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t : \mathsf{Rec} \mid L \notin \mathbf{lab}(t)\}}{\Gamma \vdash \langle L : T\rangle @ S :: \mathsf{Rec}}$$

Sample Rules

Refinement Kinds and Refinement Validity:

(KREF)

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}$$

(ENTAILS)

$$\frac{\Gamma \vdash \varphi \quad \mathsf{Valid}(\llbracket\Gamma\rrbracket \Rightarrow \llbracket\varphi\rrbracket)}{\Gamma \models \varphi}$$

Record Type Constructor/Destructors:

(K-RECCONS)

$$\frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t : \mathsf{Rec} \mid L \notin \mathbf{lab}(t)\}}{\Gamma \vdash \langle L : T\rangle @S :: \mathsf{Rec}}$$

Sample Rules

Refinement Kinds and Refinement Validity:

(KREF)

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}$$

(ENTAILS)

$$\frac{\Gamma \vdash \varphi \quad \mathsf{Valid}(\llbracket\Gamma\rrbracket \Rightarrow \llbracket\varphi\rrbracket)}{\Gamma \models \varphi}$$

Record Type Constructor/Destructors:

(K-RECCONS)

$$\frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t : \mathsf{Rec} \mid L \notin \mathbf{lab}(t)\}}{\Gamma \vdash \langle L : T\rangle @ S :: \mathsf{Rec}}$$

Sample Rules

Refinement Kinds and Refinement Validity:

(KREF)

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}$$

(ENTAILS)

$$\frac{\Gamma \vdash \varphi \quad \mathsf{Valid}([\![\Gamma]\!] \Rightarrow [\![\varphi]\!])}{\Gamma \models \varphi}$$

Record Type Constructor/Destructors:

(K-RECCONS)

$$\frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t : \mathsf{Rec} \mid L \notin \mathbf{lab}(t)\}}{\Gamma \vdash \langle L : T \rangle @ S :: \mathsf{Rec}}$$

Sample Rules

Refinement Kinds and Refinement Validity:

(KREF)

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}$$

(ENTAILS)

$$\frac{\Gamma \vdash \varphi \quad \mathsf{Valid}(\llbracket\Gamma\rrbracket \Rightarrow \llbracket\varphi\rrbracket)}{\Gamma \models \varphi}$$

Record Type Constructor/Destructors:

(K-RECCONS)

$$\frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t : \mathsf{Rec} \mid L \notin \mathbf{lab}(t)\}}{\Gamma \vdash \langle L : T\rangle @ S :: \mathsf{Rec}}$$

(K-HDT)

$$\frac{\Gamma \vdash T :: \{t{::}\mathsf{Rec} \mid \neg\mathsf{empty}(t)\}}{\Gamma \vdash \mathbf{headType}(T) :: \mathsf{Type}}$$

Sample Rules

Refinement Kinds and Refinement Validity:

(KREF)

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}$$

(ENTAILS)

$$\frac{\Gamma \vdash \varphi \quad \mathsf{Valid}(\llbracket \Gamma \rrbracket \Rightarrow \llbracket \varphi \rrbracket)}{\Gamma \models \varphi}$$

Record Type Constructor/Destructors:

(K-RECCONS)

$$\frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t : \mathsf{Rec} \mid L \notin \mathbf{lab}(t)\}}{\Gamma \vdash \langle L : T \rangle @ S :: \mathsf{Rec}}$$

(K-HDT)

$$\frac{\Gamma \vdash T :: \{t{::}\mathsf{Rec} \mid \neg\mathsf{empty}(t)\}}{\Gamma \vdash \mathbf{headType}(T) :: \mathsf{Type}}$$

Sample Rules

Refinement Kinds and Refinement Validity:

(KREF)

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}$$

(ENTAILS)

$$\frac{\Gamma \vdash \varphi \quad \mathsf{Valid}(\llbracket \Gamma \rrbracket \Rightarrow \llbracket \varphi \rrbracket)}{\Gamma \models \varphi}$$

Record Type Constructor/Destructors:

(K-RECCONS)

$$\frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t : \mathsf{Rec} \mid L \notin \mathbf{lab}(t)\}}{\Gamma \vdash \langle L : T \rangle @ S :: \mathsf{Rec}}$$

(K-HDT)

$$\frac{\Gamma \vdash T :: \{t{::}\mathsf{Rec} \mid \neg\mathsf{empty}(t)\}}{\Gamma \vdash \mathbf{headType}(T) :: \mathsf{Type}}$$

(K-TAIL)

$$\frac{\Gamma \vdash T :: \{t{::}\mathsf{Rec} \mid \neg\mathsf{empty}(t)\}}{\Gamma \vdash \mathbf{tail}(T) :: \mathsf{Rec}}$$

Sample Rules

Refinement Kinds and Refinement Validity:

(KREF)

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}$$

(ENTAILS)

$$\frac{\Gamma \vdash \varphi \quad \mathsf{Valid}(\llbracket\Gamma\rrbracket \Rightarrow \llbracket\varphi\rrbracket)}{\Gamma \models \varphi}$$

Record Type Constructor/Destructors:

(K-RECCONS)

$$\frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t : \mathsf{Rec} \mid L \not\in \mathbf{lab}(t)\}}{\Gamma \vdash \langle L : T \rangle @ S :: \mathsf{Rec}}$$

(K-HDT)

$$\frac{\Gamma \vdash T :: \{t{::}\mathsf{Rec} \mid \neg\mathsf{empty}(t)\}}{\Gamma \vdash \mathbf{headType}(T) :: \mathsf{Type}}$$

(K-TAIL)

$$\frac{\Gamma \vdash T :: \{t{::}\mathsf{Rec} \mid \neg\mathsf{empty}(t)\}}{\Gamma \vdash \mathbf{tail}(T) :: \mathsf{Rec}}$$

(EQ-TAIL)

$$\frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t : \mathsf{Rec} \mid L \not\in \mathbf{lab}(t)\}}{\Gamma \models \mathbf{tail}(\langle L : T \rangle @ S) \equiv S :: \mathsf{Rec}}$$

What about typing rules?

Typing is mostly orthogonal to ref. kinding:

What about typing rules?

Typing is mostly orthogonal to ref. kinding:

$$(\langle\rangle I_2)\ \frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash M : T_1 \qquad \Gamma \vdash T_2 :: \{t::\mathsf{Rec} \mid L \notin \mathbf{lab}(t)\} \quad \Gamma \vdash N : T_2}{\Gamma \vdash \langle L = M\rangle @ N : \langle L : T_1\rangle @ T_2}$$

What about typing rules?

Typing is mostly orthogonal to ref. kinding:

$(\langle\rangle I_2)$

$$\frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash M : T_1 \qquad \Gamma \vdash T_2 :: \{t{::}\mathsf{Rec} \mid L \notin \textbf{lab}(t)\} \quad \Gamma \vdash N : T_2}{\Gamma \vdash \langle L = M \rangle @ N : \langle L : T_1 \rangle @ T_2}$$

(TYPECASE)

$$\frac{\Gamma \vdash T :: K \quad \mathsf{covers}(\overline{p}, T, K) \quad \Gamma \vdash (p_1 :: K) \ll N_1 : U \quad \ldots \quad \Gamma \vdash (p_n :: K) \ll N_n : U}{\Gamma \vdash \textbf{case}\, T\, \textbf{of}\, p_1 \Rightarrow N_1 \mid \ldots \mid p_n \Rightarrow N_n : U}$$

What about typing rules?

Typing is mostly orthogonal to ref. kinding:

$$(\langle\rangle I_2)\ \frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash M : T_1 \qquad \Gamma \vdash T_2 :: \{t{::}\mathsf{Rec} \mid L \notin \mathbf{lab}(t)\} \quad \Gamma \vdash N : T_2}{\Gamma \vdash \langle L = M\rangle @ N : \langle L : T_1\rangle @ T_2}$$

$$(\textsc{typecase})\ \frac{\Gamma \vdash T :: K \quad \mathsf{covers}(\overline{p}, T, K) \quad \Gamma \vdash (p_1 :: K) \ll N_1 : U \quad \ldots \quad \Gamma \vdash (p_n :: K) \ll N_n : U}{\Gamma \vdash \mathbf{case}\, T\, \mathbf{of}\, p_1 \Rightarrow N_1 \mid \ldots \mid p_n \Rightarrow N_n : U}$$

All the magic happens here:

$$(\textsc{conv})\ \frac{\Gamma \vdash M : U \quad \Gamma \models U \equiv T :: \mathsf{Type}}{\Gamma \vdash M : T}$$

Tool

Typechecker / Kindchecker + Evaluator

- Artifact implementation in OCaml ($\sim$5k LoC)
- Offloads refinements to an SMT solver (CVC4):

Tool

Typechecker / Kindchecker + Evaluator

- Artifact implementation in OCaml ($\sim$5k LoC)
- Offloads refinements to an SMT solver (CVC4):
  - Types of basic kind encoded as values of an inductive type.
  - Higher-kinded types are defunctionalized.
  - Record reasoning uses theory of finite sets.

Tool

Typechecker / Kindchecker + Evaluator

- Artifact implementation in OCaml ($\sim$5k LoC)
- Offloads refinements to an SMT solver (CVC4):
  - Types of basic kind encoded as values of an inductive type.
  - Higher-kinded types are defunctionalized.
  - Record reasoning uses theory of finite sets.
- Undergoing heavy revisions...:)

Results and Future/Ongoing Thoughts

- Reasonable type theory (Progress & Preservation, Decidable checking).

Results and Future/Ongoing Thoughts

- Reasonable type theory (Progress & Preservation, Decidable checking).

  "Well-typed programs don't go wrong."

Results and Future/Ongoing Thoughts

- Reasonable type theory (Progress & Preservation, Decidable checking).

  "Well-typed programs don't go wrong."
- Subtyping
  - E.g. Record width-subtyping
  - Subtyping in refinements (encoded as a relation in SMT)
  - Metaprograms on "inheritance" / extensible interfaces.

Results and Future/Ongoing Thoughts

- Reasonable type theory (Progress & Preservation, Decidable checking).

  "Well-typed programs don't go wrong."
- Subtyping
  - E.g. Record width-subtyping
  - Subtyping in refinements (encoded as a relation in SMT)
  - Metaprograms on "inheritance" / extensible interfaces.
- Implicits (e.g. genConstr $R' \; (\langle \ell : \textbf{ref}\, T\rangle @ V) \; (\langle \ell = \textbf{ref}\, x\rangle @ v)$

Results and Future/Ongoing Thoughts

- Reasonable type theory (Progress & Preservation, Decidable checking).

  "Well-typed programs don't go wrong."
- Subtyping
  - E.g. Record width-subtyping
  - Subtyping in refinements (encoded as a relation in SMT)
  - Metaprograms on "inheritance" / extensible interfaces.
- Implicits (e.g. genConstr $R' \; (\langle \ell : \textbf{ref}\, T \rangle @ V) \; (\langle \ell = \textbf{ref}\, x \rangle @ v)$)
- Reifying type-level functions automatically (e.g. genConstr from genConsT)
  - When can we do it?
  - Can we define a general mechanism?

Conclusion

- A framework of type-safe meta-programming.
- Uniformly view types as data with simple constructor/destructor patterns.
- Expressive, flexible and simple (enough).

Conclusion

- A framework of type-safe meta-programming.
- Uniformly view types as data with simple constructor/destructor patterns.
- Expressive, flexible and simple (enough).

Questions?

(more!) Examples

A non-obviously cool function:

$$\Lambda t{::}\{f{:}\mathsf{Fun} \mid \mathbf{dom}(f) = \mathsf{Bool}\}. \qquad :$$

(more!) Examples

A non-obviously cool function:

$$\Lambda t{::}\{f{:}\mathsf{Fun} \mid \mathbf{dom}(f) = \mathsf{Bool}\}.\lambda f{:}t.(f\ \textit{true})\ :$$

(more!) Examples

A non-obviously cool function:

$$\Lambda t{::}\{f{:}\mathsf{Fun} \mid \mathbf{dom}(f) = \mathsf{Bool}\}.\lambda f{:}t.(f\ \textit{true}) :$$
$$\forall t{::}\{f{:}\mathsf{Fun} \mid \mathbf{dom}(f) = \mathsf{Bool}\}.t \to \mathbf{img}(f)$$

(more!) Examples

A non-obviously cool function:

$$\Lambda t{::}\{f{:}\mathsf{Fun} \mid \mathbf{dom}(f) = \mathsf{Bool}\}.\lambda f{:}t.(f\ \mathit{true}) :$$
$$\forall t{::}\{f{:}\mathsf{Fun} \mid \mathbf{dom}(f) = \mathsf{Bool}\}.t \rightarrow \mathbf{img}(f)$$

- Getter/Setter generator
- Generic object pairing
- ...