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 6, 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 $= \langle$**name**:String; **age** : Int$\rangle$ :: 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:

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

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:

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

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:

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

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

Kinds classify types:

- Int $\to$ Int :: Type
- $\langle$**name** : String; **age** : Int$\rangle$ :: Type
- List :: **Type** $\to$ **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).

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

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

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:

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

Refinement Kinds

Specialized kinds classify types:

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

Refinement Kinds

Specialized kinds classify types:

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

Refinement Kinds:

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

Refinement Kinds

Specialized kinds classify types:

- Int $\rightarrow$ Int :: Fun
- $\langle$**name** : String; **age** : Int$\rangle$ :: 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).

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

Refinement Kinds

Specialized kinds classify types:

- Int $\to$ Int :: $\{f$ :: Fun $\mid$ auto$(f)\}$
- $\langle$**name** : String; **age** : Int$\rangle$ :: 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 constr. (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 constr. (i.e. type-level lists)

```
addFieldType ::
```

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 constr. (i.e. type-level lists)

addFieldType :: Πl::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 constr. (i.e. type-level lists)

addFieldType :: Πl::Nm. Πt::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 constr. (i.e. type-level lists)

addFieldType :: Πl::Nm. Πt::Type. Πr::$\{s$::Rec $\mid l \notin$ **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 constr. (i.e. type-level lists)

addFieldType :: Πl::Nm. Πt::Type. Πr::$\{s$::Rec $\mid l \notin$ **lab**$(s)\}$. Rec

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 constr. (i.e. type-level lists)

addFieldType :: Πl::Nm. Πt::Type. Πr::$\{s$::Rec $\mid l \notin$ **lab**$(s)\}$. Rec
addFieldType $\triangleq \lambda l$::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 constr. (i.e. type-level lists)

addFieldType :: Πl::Nm. Πt::Type. Πr::$\{s$::Rec $\mid l \notin$ **lab**$(s)\}$. Rec
addFieldType $\triangleq \lambda l$::Nm.λt::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 constr. (i.e. type-level lists)

addFieldType :: Πl::Nm. Πt::Type. Πr::$\{s$::Rec $\mid l \notin$ **lab**$(s)\}$. Rec
addFieldType $\triangleq \lambda l$::Nm.λt::Type.λr::$\{s$::Rec $\mid l \notin$ **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 constr. (i.e. type-level lists)

addFieldType :: Πl::Nm. Πt::Type. Πr::$\{s$::Rec $\mid l \notin \mathbf{lab}(s)\}$. Rec
addFieldType $\triangleq \lambda l$::Nm.λt::Type.λr::$\{s$::Rec $\mid l \notin \mathbf{lab}(s)\}$.$\langle l : t \rangle @ r$

Extending a Record Type

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

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

addField :

addField $\triangleq$

Extending a Record Type

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

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

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

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

Extending a Record Type

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

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

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

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

Extending a Record Type

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

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

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

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

Extending a Record Type

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

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

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

Extending a Record Type

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

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

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

Extending a Record Type

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

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

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

Extending a Record Type

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

addFieldType :: Πl::Nm. Πt::Type. Πr::$\{s$::Rec $\mid l \notin$ **lab**$(s)\}$. Rec

addField : $\forall l$::Nm.$\forall t$::Type.$\forall r$::$\{s$::Rec $\mid l \notin$ **lab**$(s)\}$.
$t \to r \to$ addFieldType $l\ t\ r$
addField $\triangleq$ Λl::Nm.Λt::Type.Λr::$\{s$::Rec $\mid l \notin$ **lab**$(s)\}$.
$\lambda x{:}t.\lambda y{:}r.\langle l = x\rangle$@$y$

addField *name* String $\langle age$: Int$\rangle$ "jack" $\langle age = 20\rangle$
: addFieldType *name* String $\langle age$: Int$\rangle$

Extending a Record Type

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

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

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

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

Generic Record Type Map

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

$$\begin{aligned} &\mathsf{Map} :: \\ &\mathsf{Map} \triangleq \end{aligned}$$

Generic Record Type Map

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

$$\begin{aligned} \text{Map} :: &\quad \Pi G{::}(\text{Type} \to \text{Type}). \\ \text{Map} \triangleq &\quad \lambda G{::}(\text{Type} \to \text{Type}). \end{aligned}$$

Generic Record Type Map

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

$$\begin{aligned} \text{Map} :: &\quad \Pi G{::}(\text{Type} \to \text{Type}).\, \Pi R{::}\text{Rec}. \\ \text{Map} \triangleq &\quad \lambda G{::}(\text{Type} \to \text{Type}).\lambda R{::}\text{Rec}. \end{aligned}$$

Generic Record Type Map

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

$$\begin{aligned}
&\text{Map} :: && \Pi G{::}(\text{Type} \to \text{Type}).\, \Pi R{::}\text{Rec}.\\
&\text{Map} \triangleq && \lambda G{::}(\text{Type} \to \text{Type}).\lambda R{::}\text{Rec}.\textbf{if } \neg\text{empty}(R) \textbf{ then}\\
& && \\
& && \textbf{else}
\end{aligned}$$

Generic Record Type Map

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

$$
\begin{array}{ll}
\text{Map} :: & \Pi G{::}(\text{Type} \to \text{Type}).\, \Pi R{::}\text{Rec}. \\
\text{Map} \triangleq & \lambda G{::}(\text{Type} \to \text{Type}).\lambda R{::}\text{Rec}.\textbf{if}\ \neg\text{empty}(R)\ \textbf{then} \\
& \langle \textbf{headLabel}(R) : G(\textbf{headType}(R)) \rangle @(\qquad\qquad) \\
& \textbf{else}
\end{array}
$$

Generic Record Type Map

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

$$
\begin{array}{ll}
\text{Map} :: & \Pi G{::}(\text{Type} \to \text{Type}).\, \Pi R{::}\text{Rec}. \\
\text{Map} \triangleq & \lambda G{::}(\text{Type} \to \text{Type}).\lambda R{::}\text{Rec}.\textbf{if}\ \neg\text{empty}(R)\ \textbf{then} \\
& \langle \textbf{headLabel}(R) : G(\textbf{headType}(R)) \rangle @(\qquad\qquad) \\
& \textbf{else}
\end{array}
$$

Generic Record Type Map

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

$$
\begin{array}{ll}
\text{Map} :: & \Pi G{::}(\text{Type} \to \text{Type}).\, \Pi R{::}\text{Rec}. \\
\text{Map} \triangleq & \lambda G{::}(\text{Type} \to \text{Type}).\lambda R{::}\text{Rec}.\textbf{if}\ \neg\text{empty}(R)\ \textbf{then} \\
& \langle \textbf{headLabel}(R) : G(\textbf{headType}(R)) \rangle @(\text{Map}\ G\ \textbf{tail}(R)) \\
& \textbf{else}
\end{array}
$$

Generic Record Type Map

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

$$
\begin{array}{ll}
\text{Map} :: & \Pi G{::}(\text{Type} \to \text{Type}).\, \Pi R{::}\text{Rec}.\, \{r :: \text{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\} \\
\text{Map} \triangleq & \lambda G{::}(\text{Type} \to \text{Type}).\lambda R{::}\text{Rec}.\mathbf{if}\ \neg\text{empty}(R)\ \mathbf{then} \\
& \langle \mathbf{headLabel}(R) : G(\mathbf{headType}(R)) \rangle @(\text{Map}\ G\ \mathbf{tail}(R)) \\
& \mathbf{else}\ \langle\rangle
\end{array}
$$

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

Generic Constructor Generator

Easier to use an accumulator parameter:

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

Generic Constructor Generator

Easier to use an accumulator parameter:

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

A definition for genConstr:

$$\text{genConstr} \triangleq$$

Generic Constructor Generator

Easier to use an accumulator parameter:

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

A definition for genConstr:

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

Generic Constructor Generator

Easier to use an accumulator parameter:

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

A definition for genConstr:

$$\begin{aligned}\text{genConstr} \triangleq\ & \Lambda S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.\\ & \Lambda V{::}\{v{::}\text{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(S)\}.\lambda v{:}V.\end{aligned}$$

Generic Constructor Generator

Easier to use an accumulator parameter:

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

A definition for genConstr:

$$\begin{aligned}\text{genConstr} \triangleq\ & \Lambda S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.\\ & \Lambda V{::}\{v{::}\text{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(S)\}.\lambda v{:}V.\\ & \lambda x{:}\mathbf{headType}(S).\end{aligned}$$

Generic Constructor Generator

Easier to use an accumulator parameter:

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

A definition for genConstr:

$$\begin{aligned}
\text{genConstr} \triangleq\ & \Lambda S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \\
& \Lambda V{::}\{v{::}\text{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(S)\}.\lambda v{:}V. \\
& \lambda x{:}\mathbf{headType}(S). \\
& \quad \mathbf{if} \qquad\qquad \mathbf{then} \\
& \\
& \\
& \quad \mathbf{else}
\end{aligned}$$

Generic Constructor Generator

Easier to use an accumulator parameter:

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

A definition for genConstr:

$$\begin{aligned}
\text{genConstr} \triangleq\ & \Lambda S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \\
& \Lambda V{::}\{v{::}\text{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(S)\}.\lambda v{:}V. \\
& \lambda x{:}\mathbf{headType}(S). \\
& \quad \mathbf{if}\ \neg\text{empty}(\mathbf{tail}(S))\ \mathbf{then} \\
& \\
& \\
& \quad \mathbf{else}
\end{aligned}$$

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:

$$\begin{aligned}\text{genConstr} \triangleq\ & \Lambda S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.\\ & \Lambda V{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(S)\}.\lambda v{:}V.\\ & \lambda x{:}\textbf{headType}(S).\\ & \quad \textbf{if } \neg\text{empty}(\textbf{tail}(S)) \textbf{ then } \text{genConstr}\\ & \\ & \quad \textbf{else}\end{aligned}$$

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:

$$\begin{aligned}\text{genConstr} \triangleq\ & \Lambda S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.\\ & \Lambda V{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(S)\}.\lambda v{:}V.\\ & \lambda x{:}\textbf{headType}(S).\\ & \quad \textbf{if } \neg\text{empty}(\textbf{tail}(S)) \textbf{ then } \text{genConstr } \textbf{tail}(S)\\ & \quad \textbf{else}\end{aligned}$$

Generic Constructor Generator

Easier to use an accumulator parameter:

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

A definition for genConstr:

$$\begin{aligned}
\text{genConstr} \triangleq\ & \Lambda S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \\
& \Lambda V{::}\{v{::}\text{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(S)\}.\lambda v{:}V. \\
& \lambda x{:}\mathbf{headType}(S). \\
& \quad \mathbf{if}\ \neg\text{empty}(\mathbf{tail}(S))\ \mathbf{then}\ \text{genConstr}\ \mathbf{tail}(S) \\
& \qquad \langle \mathbf{headLabel}(S) : \mathbf{ref}\, \mathbf{headType}(S)\rangle @ V \\
& \quad \mathbf{else}
\end{aligned}$$

Generic Constructor Generator

Easier to use an accumulator parameter:

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

A definition for genConstr:

$$\begin{aligned}
\text{genConstr} \triangleq\ & \Lambda S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \\
& \Lambda V{::}\{v{::}\text{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(S)\}.\lambda v{:}V. \\
& \lambda x{:}\mathbf{headType}(S). \\
& \quad \mathbf{if}\ \neg\text{empty}(\mathbf{tail}(S))\ \mathbf{then}\ \text{genConstr}\ \mathbf{tail}(S) \\
& \qquad \langle \mathbf{headLabel}(S) : \mathbf{ref}\, \mathbf{headType}(S)\rangle @ V \\
& \qquad \langle \mathbf{headLabel}(S) = \mathbf{ref}\, x\rangle @ v \\
& \quad \mathbf{else}
\end{aligned}$$

Generic Constructor Generator

Easier to use an accumulator parameter:

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

A definition for genConstr:

$$\begin{aligned}
\text{genConstr} \triangleq\ & \Lambda S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \\
& \Lambda V{::}\{v{::}\text{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(S)\}.\lambda v{:}V. \\
& \lambda x{:}\mathbf{headType}(S). \\
& \quad \mathbf{if}\ \neg\text{empty}(\mathbf{tail}(S))\ \mathbf{then}\ \text{genConstr}\ \mathbf{tail}(S) \\
& \qquad \langle \mathbf{headLabel}(S) : \mathbf{ref}\, \mathbf{headType}(S)\rangle @ V \\
& \qquad \langle \mathbf{headLabel}(S) = \mathbf{ref}\, x\rangle @ v \\
& \quad \mathbf{else}\ \langle \mathbf{headLabel}(S) = \mathbf{ref}\, x\rangle @ v
\end{aligned}$$

Generic Constructor Generator

Typing

$$\begin{aligned}\mathsf{genConstr}: \quad & \forall S{::}\{r{::}\mathsf{Rec} \mid \neg \mathsf{empty}(r)\}. \\ & \forall V{::}\{v{::}\mathsf{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(S)\}.\end{aligned}$$

Generic Constructor Generator

Typing

$$\begin{aligned}\text{genConstr}: \quad & \forall S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \\ & \forall V{::}\{v{::}\text{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(S)\}. V \to (\text{GType}\, S\, V)\end{aligned}$$

Generic Constructor Generator

Typing

genConstr : $\forall S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.$
$\forall V{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(S)\}.V \to (\text{GType}\, S\, V)$

GType :: $\Pi S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.$
$\Pi V{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(S)\}.\,\text{Fun}$

GType $\triangleq$

Generic Constructor Generator

Typing

genConstr : $\forall S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.$
$\forall V{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(S)\}.V \to (\text{GType}\, S\, V)$

GType :: $\Pi S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.$
$\Pi V{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(S)\}.$ Fun

GType $\triangleq$

Generic Constructor Generator

Typing

genConstr : $\forall S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.$
$\forall V{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(S)\}.V \to (\text{GType}\, S\, V)$

GType :: $\Pi S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.$
$\Pi V{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(S)\}.$ Fun

GType $\triangleq$

Generic Constructor Generator

Typing

$$\text{genConstr} : \forall S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.\ \forall V{::}\{v{::}\text{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(S)\}.V \to (\text{GType}\ S\ V)$$

$$\text{GType} :: \Pi S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.\ \Pi V{::}\{v{::}\text{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(S)\}.\ \text{Fun}$$

$$\text{GType} \triangleq \lambda S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.$$

Generic Constructor Generator

Typing

$$\text{genConstr} : \forall S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \; \forall V{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(S)\}. V \to (\text{GType}\, S\, V)$$

$$\text{GType} :: \; \Pi S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \; \Pi V{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(S)\}. \text{Fun}$$

$$\text{GType} \triangleq \lambda S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \; \lambda V{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(S)\}.$$

Generic Constructor Generator

Typing

$$\begin{aligned}\text{genConstr} : \quad & \forall S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \\ & \forall V{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(S)\}. V \to (\text{GType}\, S\, V)\end{aligned}$$

$$\begin{aligned}\text{GType} :: \quad & \Pi S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \\ & \Pi V{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(S)\}.\, \text{Fun} \\ \text{GType} \triangleq \quad & \lambda S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \\ & \lambda V{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(S)\}.\textbf{headType}(S) \to\end{aligned}$$

Generic Constructor Generator

Typing

$$\text{genConstr} : \forall S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.\ \forall V{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(S)\}.V \to (\text{GType}\ S\ V)$$

$$\text{GType} :: \Pi S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.\ \Pi V{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(S)\}.\ \text{Fun}$$

$$\text{GType} \triangleq \lambda S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.\ \lambda V{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(S)\}.\textbf{headType}(S) \to$$

if $\neg\text{empty}(\textbf{tail}(S))$ **then**

else

Generic Constructor Generator

Typing

$$\begin{aligned}\text{genConstr}: \quad & \forall S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \\ & \forall V{::}\{v{::}\text{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(S)\}. V \to (\text{GType}\, S\, V)\end{aligned}$$

$$\begin{aligned}\text{GType} :: \quad & \Pi S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \\ & \Pi V{::}\{v{::}\text{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(S)\}.\, \text{Fun} \\ \text{GType} \triangleq \quad & \lambda S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \\ & \lambda V{::}\{v{::}\text{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(S)\}.\mathbf{headType}(S) \to \\ & \quad \mathbf{if}\ \neg\text{empty}(\mathbf{tail}(S))\ \mathbf{then} \\ & \text{GType}\ \mathbf{tail}(S)\ \langle \mathbf{headLabel}(S) : \mathbf{ref}\,\mathbf{headType}(S)\rangle @ V \\ & \quad \mathbf{else}\end{aligned}$$

Generic Constructor Generator

Typing

$$\begin{aligned}\text{genConstr}: \quad & \forall S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \\ & \forall V{::}\{v{::}\text{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(S)\}. V \to (\text{GType}\, S\, V)\end{aligned}$$

$$\begin{aligned}\text{GType} :: \quad & \Pi S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \\ & \Pi V{::}\{v{::}\text{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(S)\}.\, \text{Fun} \\ \text{GType} \triangleq \quad & \lambda S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \\ & \lambda V{::}\{v{::}\text{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(S)\}.\mathbf{headType}(S) \to \\ & \quad \mathbf{if}\ \neg\text{empty}(\mathbf{tail}(S))\ \mathbf{then} \\ & \text{GType}\ \mathbf{tail}(S)\ \langle \mathbf{headLabel}(S) : \mathbf{ref}\, \mathbf{headType}(S) \rangle @ V \\ & \quad \mathbf{else}\end{aligned}$$

Generic Constructor Generator

Typing

$$\begin{aligned}\text{genConstr}: \quad & \forall S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \\ & \forall V{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(S)\}. V \to (\text{GType}\, S\, V)\end{aligned}$$

$$\begin{aligned}\text{GType} :: \quad & \Pi S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \\ & \Pi V{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(S)\}.\, \text{Fun} \\ \text{GType} \triangleq \quad & \lambda S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \\ & \lambda V{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(S)\}.\textbf{headType}(S) \to \\ & \quad \textbf{if}\ \neg\text{empty}(\textbf{tail}(S))\ \textbf{then} \\ & \text{GType}\ \textbf{tail}(S)\ \langle \textbf{headLabel}(S) : \textbf{ref}\,\textbf{headType}(S)\rangle @ V \\ & \quad \textbf{else}\ \langle \textbf{headLabel}(S) : \textbf{ref}\,\textbf{headType}(S)\rangle @ V\end{aligned}$$

Generic Constructor Generator

Typing

$$\text{genConstr} : \forall S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}.\ \forall V{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(S)\}.V \to (\text{GType}\, S\, V)$$

$$\begin{aligned}
\text{GType} :: \quad & \Pi S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \\
& \Pi V{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(S)\}.\,\text{Fun} \\
\text{GType} \triangleq \quad & \lambda S{::}\{r{::}\text{Rec} \mid \neg\text{empty}(r)\}. \\
& \lambda V{::}\{v{::}\text{Rec} \mid \textbf{lab}(v)\#\textbf{lab}(S)\}.\textbf{headType}(S) \to \\
& \quad \textbf{if}\ \neg\text{empty}(\textbf{tail}(S))\ \textbf{then} \\
& \quad \text{GType}\ \textbf{tail}(S)\ \langle \textbf{headLabel}(S) : \textbf{ref}\,\textbf{headType}(S)\rangle @V \\
& \quad\ \ \textbf{else}\ \langle \textbf{headLabel}(S) : \textbf{ref}\,\textbf{headType}(S)\rangle @V
\end{aligned}$$

$$\text{GType}\ \langle L_1 : T_1; \ldots; L_n : T_n\rangle\ \langle\rangle \longrightarrow^* T_1 \to \cdots \to T_n \to \langle L_n : \textbf{ref}\, T_n; \ldots; L_1 : \textbf{ref}\, T_1\rangle$$

Generic “XML” Table Generator

Generating a table-rendering function for a record type:

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

Generic “XML” 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 “XML” Table Generator

Generating a table-rendering function for a record type:

$$
\begin{array}{ll}
\text{XForm} & :: \text{Type} \to \text{Type} \\
\text{XForm} & \triangleq \lambda t{::}\text{Type}.\langle \textit{tag} : \text{String}; \textit{toStr} : t \to \text{String} \rangle \\
 & \\
\text{MkTableType} & :: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\} \\
\text{MkTableType} & \triangleq \lambda r{::}\text{Rec}.\text{Map XForm } r
\end{array}
$$

Generic “XML” Table Generator

Generating a table-rendering function for a record type:

$$
\begin{array}{ll}
\text{XForm} & :: \text{Type} \to \text{Type} \\
\text{XForm} & \triangleq \lambda t{::}\text{Type}.\langle \textit{tag} : \text{String}; \textit{toStr} : t \to \text{String}\rangle \\
 & \\
\text{MkTableType} & :: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\} \\
\text{MkTableType} & \triangleq \lambda r{::}\text{Rec}.\text{Map XForm } r
\end{array}
$$

$$
\begin{array}{ll}
\text{MkTable} & : \forall R{::}\text{Rec}.(\text{MkTableType } R) \to R \to \text{String} \\
\text{MkTable} & \triangleq
\end{array}
$$

Generic “XML” Table Generator

Generating a table-rendering function for a record type:

$$
\begin{array}{ll}
\text{XForm} & :: \text{Type} \to \text{Type} \\
\text{XForm} & \triangleq \lambda t{::}\text{Type}.\langle \textit{tag} : \text{String}; \textit{toStr} : t \to \text{String}\rangle \\
 & \\
\text{MkTableType} & :: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\} \\
\text{MkTableType} & \triangleq \lambda r{::}\text{Rec}.\text{Map XForm } r
\end{array}
$$

$$
\begin{array}{ll}
\text{MkTable} & : \forall R{::}\text{Rec}.(\text{MkTableType } R) \to R \to \text{String} \\
\text{MkTable} & \triangleq
\end{array}
$$

Generic “XML” Table Generator

Generating a table-rendering function for a record type:

$$
\begin{array}{ll}
\text{XForm} & :: \text{Type} \to \text{Type} \\
\text{XForm} & \triangleq \lambda t{::}\text{Type}.\langle \textit{tag} : \text{String}; \textit{toStr} : t \to \text{String}\rangle \\
 & \\
\text{MkTableType} & :: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\} \\
\text{MkTableType} & \triangleq \lambda r{::}\text{Rec}.\text{Map XForm } r
\end{array}
$$

$$
\begin{array}{ll}
\text{MkTable} & : \forall R{::}\text{Rec}.(\text{MkTableType } R) \to R \to \text{String} \\
\text{MkTable} & \triangleq
\end{array}
$$

Generic “XML” Table Generator

Generating a table-rendering function for a record type:

$$
\begin{array}{ll}
\text{XForm} & :: \text{Type} \to \text{Type} \\
\text{XForm} & \triangleq \lambda t{::}\text{Type}.\langle \textit{tag} : \text{String}; \textit{toStr} : t \to \text{String}\rangle \\
 & \\
\text{MkTableType} & :: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\} \\
\text{MkTableType} & \triangleq \lambda r{::}\text{Rec}.\text{Map XForm } r
\end{array}
$$

$$
\begin{array}{ll}
\text{MkTable} & : \forall R{::}\text{Rec}.(\text{MkTableType } R) \to R \to \text{String} \\
\text{MkTable} & \triangleq \Lambda R{::}\text{Rec}.
\end{array}
$$

Generic “XML” Table Generator

Generating a table-rendering function for a record type:

$$
\begin{array}{ll}
\text{XForm} & :: \text{Type} \to \text{Type} \\
\text{XForm} & \triangleq \lambda t{::}\text{Type}.\langle \textit{tag} : \text{String}; \textit{toStr} : t \to \text{String}\rangle \\
 & \\
\text{MkTableType} & :: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\} \\
\text{MkTableType} & \triangleq \lambda r{::}\text{Rec}.\text{Map XForm } r
\end{array}
$$

$$
\begin{array}{ll}
\text{MkTable} & : \forall R{::}\text{Rec}.(\text{MkTableType } R) \to R \to \text{String} \\
\text{MkTable} & \triangleq \Lambda R{::}\text{Rec}.\lambda M{:}(\text{MkTableType } R).\lambda r{:}R.
\end{array}
$$

Generic “XML” Table Generator

Generating a table-rendering function for a record type:

$$
\begin{array}{ll}
\text{XForm} & :: \text{Type} \to \text{Type} \\
\text{XForm} & \triangleq \lambda t{::}\text{Type}.\langle \textit{tag} : \text{String}; \textit{toStr} : t \to \text{String}\rangle \\
 & \\
\text{MkTableType} & :: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\} \\
\text{MkTableType} & \triangleq \lambda r{::}\text{Rec}.\text{Map XForm } r
\end{array}
$$

$$
\begin{array}{ll}
\text{MkTable} & : \forall R{::}\text{Rec}.(\text{MkTableType } R) \to R \to \text{String} \\
\text{MkTable} & \triangleq \Lambda R{::}\text{Rec}.\lambda M{:}(\text{MkTableType } R).\lambda r{:}R. \\
 & \quad \textbf{if } \neg\text{empty}(R) \textbf{ then} \\
 & \\
 & \quad \textbf{else } \text{“”}
\end{array}
$$

Generic “XML” Table Generator

Generating a table-rendering function for a record type:

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

$$
\begin{array}{ll}
\text{MkTable} & : \forall R{::}\text{Rec}.(\text{MkTableType } R) \to R \to \text{String} \\
\text{MkTable} & \triangleq \Lambda R{::}\text{Rec}.\lambda M{:}(\text{MkTableType } R).\lambda r{:}R. \\
 & \quad \textbf{if } \neg\text{empty}(R) \textbf{ then} \\
 & \quad \text{“<tr><th>”} + M.\textbf{recHeadLabel}(M).\textit{tag} + \text{“</th><td>”} + \\
 & \\
 & \quad \textbf{else “”}
\end{array}
$$

Generic “XML” Table Generator

Generating a table-rendering function for a record type:

$$
\begin{array}{ll}
\text{XForm} & :: \text{Type} \to \text{Type} \\
\text{XForm} & \triangleq \lambda t{::}\text{Type}.\langle \mathit{tag} : \text{String}; \mathit{toStr} : t \to \text{String}\rangle \\
 & \\
\text{MkTableType} & :: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\} \\
\text{MkTableType} & \triangleq \lambda r{::}\text{Rec}.\text{Map XForm } r
\end{array}
$$

$$
\begin{array}{l}
\text{MkTable} \quad : \forall R{::}\text{Rec}.(\text{MkTableType } R) \to R \to \text{String} \\
\text{MkTable} \quad \triangleq \Lambda R{::}\text{Rec}.\lambda M{:}(\text{MkTableType } R).\lambda r{:}R. \\
\qquad\qquad \textbf{if } \neg\text{empty}(R) \textbf{ then} \\
\text{“<tr><th>”} + M.\mathbf{recHeadLabel}(M).\mathit{tag} + \text{“</th><td>”} + \\
M.\mathbf{recHeadLabel}(M).\mathit{toStr}\ (r.\mathbf{recHeadLabel}(M)) + \text{“</td></tr>”} + \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } \text{“”}
\end{array}
$$

Generic “XML” Table Generator

Generating a table-rendering function for a record type:

$$
\begin{array}{ll}
\text{XForm} & :: \text{Type} \to \text{Type} \\
\text{XForm} & \triangleq \lambda t{::}\text{Type}.\langle \mathit{tag} : \text{String}; \mathit{toStr} : t \to \text{String}\rangle \\
 & \\
\text{MkTableType} & :: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\} \\
\text{MkTableType} & \triangleq \lambda r{::}\text{Rec}.\text{Map XForm } r
\end{array}
$$

$$
\begin{array}{l}
\text{MkTable} \quad : \forall R{::}\text{Rec}.(\text{MkTableType } R) \to R \to \text{String} \\
\text{MkTable} \quad \triangleq \Lambda R{::}\text{Rec}.\lambda M{:}(\text{MkTableType } R).\lambda r{:}R. \\
\qquad\qquad \textbf{if } \neg\text{empty}(R) \textbf{ then} \\
\text{“<tr><th>”} + M.\mathbf{recHeadLabel}(M).\mathit{tag} + \text{“</th><td>”} + \\
M.\mathbf{recHeadLabel}(M).\mathit{toStr}\ (r.\mathbf{recHeadLabel}(M)) + \text{“</td></tr>”} + \\
\qquad\qquad \text{MkTable } \mathbf{tail}(R)\ \mathbf{recTail}(M)\ \mathbf{recTail}(r) \quad \textbf{else “”}
\end{array}
$$

Generic “XML” Table Generator

Generating a table-rendering function for a record type:

$$
\begin{array}{ll}
\text{XForm} & :: \text{Type} \to \text{Type} \\
\text{XForm} & \triangleq \lambda t{::}\text{Type}.\langle \mathit{tag} : \text{String}; \mathit{toStr} : t \to \text{String}\rangle \\
 & \\
\text{MkTableType} & :: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\} \\
\text{MkTableType} & \triangleq \lambda r{::}\text{Rec}.\text{Map XForm } r
\end{array}
$$

$$
\begin{array}{l}
\text{MkTable} \quad : \forall R{::}\text{Rec}.(\text{MkTableType } R) \to R \to \text{String} \\
\text{MkTable} \quad \triangleq \Lambda R{::}\text{Rec}.\lambda M{:}(\text{MkTableType } R).\lambda r{:}R. \\
\qquad\qquad \textbf{if } \neg\text{empty}(R) \textbf{ then} \\
\text{“<tr><th>”} + M.\mathbf{recHeadLabel}(M).\mathit{tag} + \text{“</th><td>”} + \\
M.\mathbf{recHeadLabel}(M).\mathit{toStr}\ (r.\mathbf{recHeadLabel}(M)) + \text{“</td></tr>”} + \\
\qquad\qquad \text{MkTable } \mathbf{tail}(R)\ \mathbf{recTail}(M)\ \mathbf{recTail}(r) \quad \textbf{else “”}
\end{array}
$$

Generic “XML” Table Generator

Generating a table-rendering function for a record type:

$$
\begin{array}{ll}
\text{XForm} & :: \text{Type} \to \text{Type} \\
\text{XForm} & \triangleq \lambda t{::}\text{Type}.\langle \mathit{tag} : \text{String}; \mathit{toStr} : t \to \text{String}\rangle \\
 & \\
\text{MkTableType} & :: \Pi r{::}\text{Rec}.\{r :: \text{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\} \\
\text{MkTableType} & \triangleq \lambda r{::}\text{Rec}.\text{Map XForm } r
\end{array}
$$

$$
\begin{array}{l}
\text{MkTable} \quad : \forall R{::}\text{Rec}.(\text{MkTableType } R) \to R \to \text{String} \\
\text{MkTable} \quad \triangleq \Lambda R{::}\text{Rec}.\lambda M{:}(\text{MkTableType } R).\lambda r{:}R. \\
\qquad\qquad \textbf{if } \neg\text{empty}(R) \textbf{ then} \\
\text{“<tr><th>”} + M.\mathbf{recHeadLabel}(M).\mathit{tag} + \text{“</th><td>”} + \\
M.\mathbf{recHeadLabel}(M).\mathit{toStr}\ (r.\mathbf{recHeadLabel}(M)) + \text{“</td></tr>”} + \\
\qquad\qquad \text{MkTable } \mathbf{tail}(R)\ \mathbf{recTail}(M)\ \mathbf{recTail}(r) \quad \textbf{else “”}
\end{array}
$$

Examples

More examples in the paper…

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

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

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

$$
T, S, R \quad ::= \quad t \mid \lambda t{::}K.T \mid T\ S
$$

Kinds, Types and Refinements

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

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

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

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

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

Kinds, Types and Refinements

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

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

Kinds, Types and Refinements

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

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

Kinds, Types and Refinements

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

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

Kinds, Types and Refinements

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

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

Kinds, Types and Refinements

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

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

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

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

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

Kinds, Types and Refinements

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

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

Kinds, Types and Refinements

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

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

Kinds, Types and Refinements

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

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

$$
\begin{array}{rcl}
M, N & ::= & x \mid \lambda x{:}T.M \mid M\ N \\
& \mid & \Lambda t{::}K.M \mid M[T] \\
& \mid & \langle\rangle \mid \langle \ell = M \rangle @ N \mid \textbf{recTail}(M) \\
& \mid & \textbf{recHeadLabel}(M) \mid \textbf{recHeadTerm}(M)
\end{array}
$$

Program Terms

$$
\begin{array}{rcl}
M, N & ::= & x \mid \lambda x{:}T.M \mid M\ N \\
& \mid & \Lambda t{::}K.M \mid M[T] \\
& \mid & \langle\rangle \mid \langle \ell = M \rangle @ N \mid \textbf{recTail}(M) \\
& \mid & \textbf{recHeadLabel}(M) \mid \textbf{recHeadTerm}(M) \\
& \mid & \textbf{if}\ T :: K\ \textbf{as}\ t \Rightarrow M\ \textbf{else}\ N
\end{array}
$$

Program Terms

$$
\begin{array}{rcl}
M, N & ::= & x \mid \lambda x{:}T.M \mid M\ N \\
& \mid & \Lambda t{::}K.M \mid M[T] \\
& \mid & \langle\rangle \mid \langle \ell = M \rangle @ N \mid \mathbf{recTail}(M) \\
& \mid & \mathbf{recHeadLabel}(M) \mid \mathbf{recHeadTerm}(M) \\
& \mid & \mathbf{if}\ T :: K\ \mathbf{as}\ t \Rightarrow M\ \mathbf{else}\ N \\
& \mid & \boldsymbol{\mu} F{:}T.M \\
& \mid & \ldots
\end{array}
$$

Program Terms

$$
\begin{array}{rcl}
M, N & ::= & x \mid \lambda x{:}T.M \mid M\ N \\
& \mid & \Lambda t{::}K.M \mid M[T] \\
& \mid & \langle\rangle \mid \langle \ell = M \rangle @ N \mid \mathbf{recTail}(M) \\
& \mid & \mathbf{recHeadLabel}(M) \mid \mathbf{recHeadTerm}(M) \\
& \mid & \mathbf{if}\ T :: K\ \mathbf{as}\ t \Rightarrow M\ \mathbf{else}\ N \\
& \mid & \boldsymbol{\mu} F{:}T.M \\
& \mid & \ldots
\end{array}
$$

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

Sample Rules

Refinement Kinds and Refinement Validity:

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

Sample Rules

Refinement Kinds and Refinement Validity:

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

Sample Rules

Refinement Kinds and Refinement Validity:

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

Sample Rules

Refinement Kinds and Refinement Validity:

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}\ (\textsc{kref}) \qquad \frac{\Gamma \vdash \varphi \quad \mathsf{Valid}([\![\Gamma]\!] \Rightarrow [\![\varphi]\!])}{\Gamma \models \varphi}\ (\textsc{entails})$$

Sample Rules

Refinement Kinds and Refinement Validity:

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}\ (\textsc{kref}) \qquad \frac{\Gamma \vdash \varphi \quad \textsf{Valid}([\![\Gamma]\!] \Rightarrow [\![\varphi]\!])}{\Gamma \models \varphi}\ (\textsc{entails})$$

Sample Rules

Refinement Kinds and Refinement Validity:

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}\ (\textsc{kref}) \qquad \frac{\Gamma \vdash \varphi \quad \mathsf{Valid}([\![\Gamma]\!] \Rightarrow [\![\varphi]\!])}{\Gamma \models \varphi}\ (\textsc{entails})$$

Function Type Constructor and Destructors:

$$\frac{\Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \mathcal{K}'}{\Gamma \vdash T \to S :: \mathsf{Fun}}\ (\textsc{k-fun})$$

Sample Rules

Refinement Kinds and Refinement Validity:

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}\ (\textsc{kref}) \qquad \frac{\Gamma \vdash \varphi \quad \mathsf{Valid}([\![\Gamma]\!] \Rightarrow [\![\varphi]\!])}{\Gamma \models \varphi}\ (\textsc{entails})$$

Function Type Constructor and Destructors:

$$\frac{\Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \mathcal{K}'}{\Gamma \vdash T \to S :: \mathsf{Fun}}\ (\textsc{k-fun}) \qquad \frac{\Gamma \vdash T :: \mathsf{Fun}}{\Gamma \vdash \mathbf{dom}(T) :: \mathsf{Type}}\ (\textsc{k-dom})$$

Sample Rules

Refinement Kinds and Refinement Validity:

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}\ (\textsc{kref}) \qquad \frac{\Gamma \vdash \varphi \quad \mathsf{Valid}([\![\Gamma]\!] \Rightarrow [\![\varphi]\!])}{\Gamma \models \varphi}\ (\textsc{entails})$$

Function Type Constructor and Destructors:

$$\frac{\Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \mathcal{K}'}{\Gamma \vdash T \to S :: \mathsf{Fun}}\ (\textsc{k-fun}) \qquad \frac{\Gamma \vdash T :: \mathsf{Fun}}{\Gamma \vdash \mathbf{dom}(T) :: \mathsf{Type}}\ (\textsc{k-dom})$$

$$\frac{\Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \mathcal{K}'}{\Gamma \models \mathbf{dom}(T \to S) \equiv T :: \mathsf{Type}}\ (\textsc{eq-dom})$$

Sample Rules

Record Type Constructor:

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

Sample Rules

Record Type Constructor:

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

Sample Rules

Record Type Constructor:

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

Sample Rules

Record Type Constructor:

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

Sample Rules

Record Type Constructor:

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

Record Type Destructors:

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

Sample Rules

Record Type Constructor:

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

Record Type Destructors:

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

$$\text{(K-TAIL)}\quad \frac{\Gamma \vdash T :: \{t{::}\mathsf{Rec} \mid \neg\mathsf{empty}(t)\}}{\Gamma \vdash \mathbf{tail}(T) :: \mathsf{Rec}}$$

Sample Rules

Record Type Constructor:

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

Record Type Destructors:

$$\frac{\Gamma \vdash T :: \{t{::}\mathsf{Rec} \mid \neg\mathsf{empty}(t)\}}{\Gamma \vdash \mathbf{headType}(T) :: \mathsf{Type}} \text{ (K-HDT)} \qquad \frac{\Gamma \vdash T :: \{t{::}\mathsf{Rec} \mid \neg\mathsf{empty}(t)\}}{\Gamma \vdash \mathbf{tail}(T) :: \mathsf{Rec}} \text{ (K-TAIL)}$$

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

Sample Rules

No more kinding/equality rules for the sake of sanity:

- Type-level recursion is well-founded.
- Equalities in refinements are reflected in def. equality.
- Type-level functions are standard (with extensionality).

What about typing rules?

Typing is mostly orthogonal to ref. kinding:

What about typing rules?

Typing is mostly orthogonal to ref. kinding:

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

What about typing rules?

Typing is mostly orthogonal to ref. kinding:

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

$$(\textsc{kindcase})\quad \frac{\Gamma \vdash T :: \mathcal{K}' \quad \Gamma \vdash \mathcal{K} \quad \Gamma, t{:}\mathcal{K} \vdash M : U \quad \Gamma \vdash N : U}{\Gamma \vdash \mathbf{if}\, T :: \mathcal{K}\, \mathbf{as}\, t \Rightarrow M\, \mathbf{else}\, N : U}$$

What about typing rules?

Typing is mostly orthogonal to ref. kinding:

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

$$\frac{\Gamma \vdash T :: \mathcal{K}' \quad \Gamma \vdash \mathcal{K} \quad \Gamma, t{:}\mathcal{K} \vdash M : U \quad \Gamma \vdash N : U}{\Gamma \vdash \mathbf{if}\, T :: \mathcal{K}\, \mathbf{as}\, t \Rightarrow M\, \mathbf{else}\, N : U} \; (\textsc{kindcase})$$

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.

Meta-theoretic Results

- Type (and Kind) Preservation
  1. If $\Gamma \vdash T :: K$ and $T \to T'$ then $\Gamma \vdash T' :: K$.
  2. If $\Gamma \vdash M : T$ and $M \to M'$ then $\Gamma \vdash M' : T$

Meta-theoretic Results

- Type (and Kind) Preservation
  1. If $\Gamma \vdash T :: K$ and $T \to T'$ then $\Gamma \vdash T' :: K$.
  2. If $\Gamma \vdash M : T$ and $M \to M'$ then $\Gamma \vdash M' : T$
- Progress (Types and Terms)
- Decidable Type-checking

Meta-theoretic Results

- Type (and Kind) Preservation
  1. If $\Gamma \vdash T :: K$ and $T \to T'$ then $\Gamma \vdash T' :: K$.
  2. If $\Gamma \vdash M : T$ and $M \to M'$ then $\Gamma \vdash M' : T$
- Progress (Types and Terms)
- Decidable Type-checking

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

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).

Did not talk about / Future work:

- Ad-hoc polymorphism via kind-case
- Subkinding, subtyping

Conclusion

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

Did not talk about / Future work:

- Ad-hoc polymorphism via kind-case
- Subkinding, subtyping
- Type/Kind Reconstruction
- Typeclasses, stateful meta-programming, etc.

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:

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

(more!) Examples

A non-obviously cool function:

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

(more!) Examples

A non-obviously cool function:

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

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