Efficient Compilation of Polymorphic Record Calculi

Master’s in Informatics and Computer Engineering

Student: Eduardo Correia

Supervisor: Prof. Sandra Alves

U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Context

Context

λ-calculus

- First invented by **Alonzo Church** in the 1930s.
- It's a formal system in mathematical logic for expressing (Turing complete) computation, based on **function abstraction** and **application**.
- Due to its **simplicity** and **strong expressiveness**, it has been since widely used in the **study of programming languages**.

Context

Typed λ-calculus

Untyped λ-calculus

$$\lambda x.x$$

Simply-typed λ-calculus

$$\lambda x.x : (Int \to Int)$$

Polymorphic λ-calculus

$$\lambda x.x : \forall \alpha.\alpha \to \alpha$$

Context

Records

- Labeled records are **data structures** that associate **labels** with **values**.
- Widely used in various data-intensive applications.
- Akin to ***dictionaries*** in Python, or ***maps*** in Haskell.
- The following example represents a record with a student's information:

$$\{Name : \text{``Eduardo''}, Age : 23, University : \text{``FEUP''}\}$$

Context

Record Polymorphism

- In simply-typed systems, labeled records' allowable operations are restricted to **monomorphic** ones. That is, the type of a record must be specified in advance.
- **Record polymorphism** stems from the idea of having polymorphic operations over records. That is the same operation can be applied to records with different types.
- As an example, the following term, representing **label selection**, should work for any record type, containing the label $Name$:

$$\lambda x.x.\,Name$$

$$\{\,Name : \text{“Eduardo”}, Age : 23\} \quad \{\,Name : \text{“Correia”}, University : \text{“FEUP”}\}$$

Motivation and Problem

To achieve record polymorphism, some possible strategies are:

- Subtyping;
- Row variables;
- Kinds.

But these solutions either:

- Lose typing information;
- Introduce over-head, or run-time failures, that should have been caught at compile time;
- Don't allow for extensible records (add or remove labels).

Therefore, an **efficient compilation method** for a record calculus with extensible records would be a valuable contribution.

State of the Art

State of the Art

Polymorphic Record Calculus

- In 1995, **Atsushi Ohori**[1] introduced a *let*-polymorphic record calculus that extends a λ-calculus with **labeled records** and **polymorphic operations** over those records.
- In Ohori's system, the notion of a **kind** is used to specify the labels a record is expected to contain.
- An efficient compilation mechanism for this calculus was also provided.
- An **implementation** of this calculus, called **SML#**, was provided by extending the **Standard ML** language.
- However, no operations for **extensible records** were defined (due to efficiency limits).

[1] Atsushi Ohori. A polymorphic record calculus and its compilation. ACM Trans. Program.Lang. Syst., 17(6):844–895, 1995. doi:10.1145/218570.218572.

State of the Art

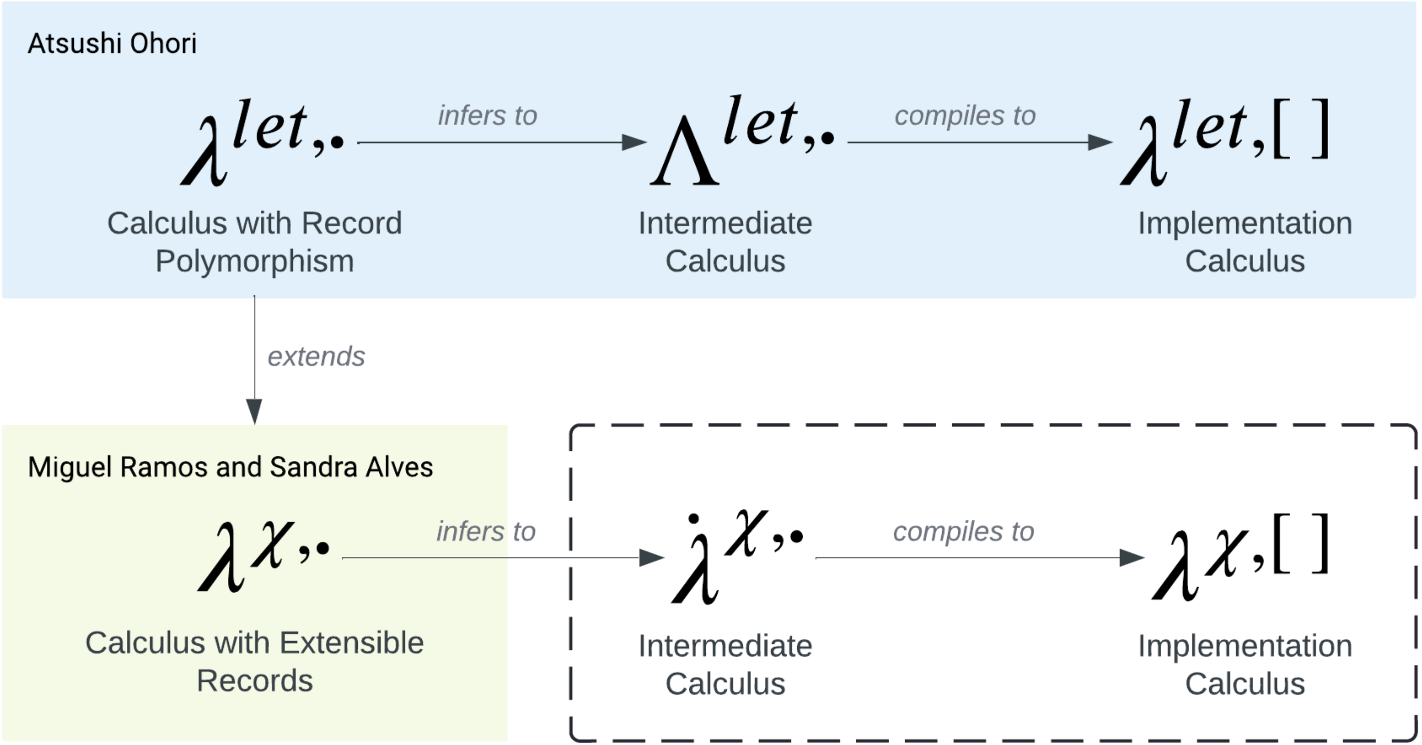
Record Calculus with Extensible Records

- In 2021, **Miguel Ramos et al**[2] developed a record calculus with **extensible records** based on Ohori's kinds.
- This calculus has a typing system, based on the notion of **kinded quantification** and a sound and complete **type inference algorithm**, based on **kinded unification**.
- Negative information was added to kinds, allowing for the specification of fields that a record should not have.
- No compilation mechanism, however, was provided.

[2] Miguel Ramos, Sandra Alves. An ML-style Record Calculus with Extensible Records. Electronic Proceedings in Theoretical Computer Science, EPTCS, 1-17, 2021. doi:10.4204/EPTCS.351.1

Developed Work

Flow



Atsushi Ohori
$\lambda^{let,\bullet}$
infers to
$\Lambda^{let,\bullet}$
compiles to
$\lambda^{let,[\,]}$
Calculus with Record Polymorphism
Intermediate Calculus
Implementation Calculus
extends
Miguel Ramos and Sandra Alves
$\lambda^{\chi,\bullet}$
infers to
$\dot{\lambda}^{\chi,\bullet}$
compiles to
$\lambda^{\chi,[\,]}$
Calculus with Extensible Records
Intermediate Calculus
Implementation Calculus

$\lambda^{\chi,\bullet}$

Record Calculus with Extensible Records

Record Calculus with Extensible Records

Terms

$$
\begin{array}{rlll}
M & ::= & c^b & \text{(constant)} \\
 & | & x & \text{(variable)} \\
 & | & \lambda x.M & \text{(function abstraction)} \\
 & | & M\ M & \text{(function application)} \\
 & | & \text{let } x = M \text{ in } M & \text{(let expression)} \\
 & | & \{l = M, \ldots, l = M\} & (\textbf{record}) \\
 & | & M.l & (\textbf{field selection}) \\
 & | & \text{modify}(M, l, M) & (\textbf{field update}) \\
 & | & M \setminus\!\setminus l & (\textbf{contraction}) \\
 & | & \text{extend}(M, l, M) & (\textbf{extension})
\end{array}
$$

Record Calculus with Extensible Records

Types

$$
\begin{array}{rcll}
\tau & ::= & b & \text{(base type)} \\
& | & \alpha & \text{(variable type)} \\
& | & \chi & \text{(extensible type)} \\
& | & \tau \to \tau & \text{(function type)} \\
\\
\sigma & ::= & \tau & \text{(monomorphic type)} \\
& | & \forall \alpha :: \kappa.\ \sigma & \text{(polymorphic type)} \\
\\
\chi & ::= & \alpha & \text{(type variable)} \\
& | & \{l : \tau, \ldots, l : \tau\} & \text{(record type)} \\
& | & \chi + \{l : \tau\} & \text{(field-addition type)} \\
& | & \chi - \{l : \tau\} & \text{(field-removal type)}
\end{array}
$$

Record Calculus with Extensible Records

Kinds

- **Kinds** restrict possible instantiations of a type.
- There are two sorts of kinds: **universal kind** and **record kinds**.

$$\mathcal{U} \mid \{\!\{ l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l \parallel l_1^r : \tau_1^r, \ldots, l_n^r : \tau_n^r \}\!\}$$

- In a record kind of the form $\{\!\{ F_l \parallel F_r \}\!\}$, F_l denotes the fields a record should at least **have**, and F_r denotes the fields a record should at least **not have**.

Record Calculus with Extensible Records

Example

- Given a term M of type τ, and kind $\{Name : String, Age : Int \,\|\, University : String\}$, the following operations are typed as follows:

$$
\begin{aligned}
M.Name &: String \\
\mathrm{modify}(M, Name, \text{``Correia''}) &: \tau \\
M \setminus\!\setminus Age &: \tau - \{Age : Int\} \\
\mathrm{extend}(M, University, \text{``FEUP''}) &: \tau + \{University : String\}
\end{aligned}
$$

$\lambda x,.$

Intermediate Calculus

Intermediate Calculus

- We define an **intermediate annotated calculus**, corresponding to an **explicitily typed** version of $\lambda^{\chi,\cdot}$.
- The additional **type information** is used to facilitate the **compilation** process.
- The type system for this calculus remains unchanged from the original calculus.
- Two new terms are added to the calculus: **polymorphic instantiation**, $(x\ \tau_1 \cdots \tau_n)$, and **polymorphic generalization**, $\mathrm{Poly}(M : \sigma)$.

$$\mathrm{let}\ x = \boxed{M}\ \mathrm{in}\ \boxed{x}$$

$$\Downarrow$$

$$\mathrm{let}\ x = \boxed{\mathrm{Poly}(M :\ \sigma)}\ \mathrm{in}\ \boxed{(x\ \tau_1 \cdots \tau_n)}$$

Intermediate Calculus

Type inference

- A **type-inference algorithm** is provided that turns an untyped term into a typed one.
- As an example, the untyped $\lambda^{\chi,\cdot}$-term

$$\begin{aligned} &\text{let } name = \lambda x.x.Name \\ &\text{in } name\,\{Name : \text{``}Eduardo\text{''}, Age : 23\} \end{aligned}$$

is inferred to

$$\begin{aligned} &\text{let } name = \mathrm{Poly}(\lambda x : \alpha_2.x.Name : \forall\alpha_1 :: \mathcal{U}.\,\forall\alpha_2 :: \{\!\{Name : \alpha_1 \mid \}\!\}.\,\alpha_2 \to \alpha_1) \\ &\text{in } (name\ \{Name : String, Age : Int\}\ Int)\,\{Name : \text{``}Eduardo\text{''}, Age : 23\} \end{aligned}$$

$\lambda^{\chi,[\,]}$

Implementation Calculus

Implementation Calculus

- We define an **implementation calculus** that will be used as an **abstract evaluation machine** for the calculus with extensible records.
- To represent labeled records in this calculus, we assume that the total **order** of the labels in a record (typically, the **lexicographic order**) is fixed.
- As such, the record type of the form $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ must satisfy the condition $l_1 \ll \cdots \ll l_n$.
- After this ordering, we can represent a record as a **list** of values, where the index of a value corresponds to the index of the label in the record type.

$$\{Name : \text{``Eduardo''}, Age : 23, University : \text{``FEUP''}\} \Rightarrow [23, \text{``Eduardo''}, \text{``FEUP''}]$$

Implementation Calculus

Indexing

- Since records are represented as lists, previously defined operations must be adapted to work with **indexes** instead of **labels**.
- There are two types of **indexes**: **index variables**, I (plus an offset n), and **index values**, i.

$$\mathcal{I} = I + n \mid i$$

- **Label operations** are then converted to **index operations**, by replacing labels with their corresponding index values.

$$M.l \Rightarrow M[\mathcal{I}]$$
$$\text{modify}(M, l, M) \Rightarrow \text{modify}(M, \mathcal{I}, M)$$
$$M \setminus\!\setminus l \Rightarrow M \setminus\!\setminus \mathcal{I}$$
$$\text{extend}(M, l, M) \Rightarrow \text{extend}(M, \mathcal{I}, M)$$

Implementation Calculus

Index Abstraction and Application

- To deal with polymorphic record operations, we introduce **index abstraction**, $\lambda I_1 \cdots I_n$, and **index application**, $x\ i_1 \cdots i_n$.
- $I_1, \ldots, I_n$ are **index variables** that represent the index values of the labels in a record type.
- $i_1, \ldots, i_n$ are **index values** that represent the index values of the labels in a record.

$$\text{let } x = \boxed{\text{Poly}(M : \ \sigma)} \text{ in } \boxed{(x\ \tau_1 \cdots \tau_n)}$$

$$\Downarrow$$

$$\text{let } x = \boxed{\lambda I_1 \cdots I_n.M} \text{ in } \boxed{x\ i_1 \cdots i_n}$$

Implementation Calculus

Index Offset

- To account for **field-removal** and **field-addition** operations, an **index offset** is added to the operations with index variables.
- Given a record type of the form $\alpha \pm_1 \{l_1 : \tau_1\} \cdots \pm_n \{l_n : \tau_n\}$, the index offset for a given label l is defined as follows:

$$0 \pm_1 \begin{cases} 0 \text{ if } l \leq l_1 \\ 1 \text{ if } l > l_1 \end{cases} \cdots \pm_n \begin{cases} 0 \text{ if } l \leq l_n \\ 1 \text{ if } l > l_n \end{cases}$$

- As an example, given a record M of the type $\{Age : Int, Name : String\}$, we have:

Implementation Calculus

Compilation

- A **compilation algorithm** is provided that turns a term in $\lambda^{\mathcal{X},\cdot}$ into a term in $\lambda^{\mathcal{X},[]}$.

As an example, the $\lambda^{\mathcal{X},\cdot}$-term

$$\text{let } name = \lambda x.x\textbf{.}Name \text{ in } name\,\{Name : \text{``}Eduardo\text{''}, Age : 23\}$$

is compiled to the following term (after type inference)

$$\text{let } name = \lambda I.\lambda x.\,x[I] \text{ in } name\,2\,\{23, \text{``João''}\}$$

Implementation

- A REPL for these calculi, named **Recording**, was implemented in **Haskell**.

Implementation

Architecture

Implementation

Demo

Conclusion

Contributions

- We’ve managed to extend Ohori’s calculus compilation relation with extensible records.
- We provide a pratical implementation of the calculus.
- Demonstration of soundness properties of the defined calculi.
- Adaptation of intermediate calculus with extended records.

Future Work

- Set operations for records (general join, intersection, difference, ...).
- Labels as first-class citizens.
- Integration of developed work in SML#.

Thank you!

Any questions?