Type Inference for a Tuple Concatenation Operator

Kent D. Lee

Computer Science

Luther College
Overview

- Motivation for tuple concatenation
- A Little Action Notation
- Type Inference Algorithm
- An example from Action Semantics
- Conclusion
Motivation for Tuple Concatenation

- Most of my research has been in the area of Action Semantics. Specifically I’ve worked on automated compiler generation based on Action Semantic Descriptions.

- Sorts in Action Semantics come from Unified Algebras and are similar to types in other languages.

- As part of this research I extended the Actress type inference algorithm to support tuple concatenation.
Genesis: An action semantics-based compiler generator

Based on the Actress Design
Extending the Research

• I would like a simple way to write and perform actions interactively as a means of studying type inference over actions and for teaching action semantics.

• I have two alternatives
  
  – Write a full-blown interpreter of actions with a scanner, parser, and a type inference algorithm and interpret the resulting action.

  – Extend ML with functions that implement Action Semantics within ML. To do this implies that ML would do the type inference over the actions.
A Long-term Goal

- A long-term goal of this work is to see if the Hindley-Milner type inference algorithm can be extended to support action notation to enhance compiler generation. Right now Genesis’ type inference algorithm works most of the time, but fails to correctly check some actions because it is not fully polymorphic.
Data Notation of Action Semantics

- Integers
- Truth-Values
- Cells
- Tuples
- Maps
- Abstractions
Actions

- Primitive Actions
  - give _, bind _ to _, store _ in _, allocate _, enact _, unfold
  - no ‘give _ label _’ as Actress implemented

- Combinators
  - and, and then, then and others
  - else implements deterministic choice
  - recursively expresses mutually recursive bindings that are close together (mutual recursion)
Yielders

• the given _, the given _ # _, _ bound to _

• rest yields all but first element of a tuple
  – Useful in ‘give the rest of the given data’

• Tupling constructor - (_,_): tuples are flat.

• Incomes and Outcomes
Action Types

- An Action’s type may be described by its incoming and outgoing transients and bindings.

- Storage is ignored since the set of allocated cells can’t be known statically in general.

- Bindings are represented as a map of Identifiers to types.

- Transients are represented as a map of natural numbers to types.

- Records as proposed by Even and Schmidt serve both purposes.

- In other words tuples are represented as records.
Tuple Record Schemes

- $\{1 : \theta_1, 3 : \theta_4\} \gamma_5$
  - $[\theta_1 \mapsto \text{integer}, \theta_4 \mapsto \text{integer}]$
  - $\gamma_5$ represents as yet unknown fields that are ignored by an action

- $\{1 : \theta_1, 3 : \theta_4\} \rho_5$
  - $[\theta_1 \mapsto \text{truth-value}, \theta_4 \mapsto \text{integer}]$
  - $\rho_5$ represents as yet known fields that are propagated by an action

- $\{1 : \theta_1, 2 : \theta_4\}$
  - No row variable needed when all fields are known.
Tuple Concatenation in Records

• Previously not thought possible to carry out tuple concatenation of records

• It is possible in some cases

  – $concat(\{\}, R) = R$

  – $concat(R, \{\}) = R$

  – $concat(\{1 : \sigma_1, 2 : \sigma_2\}, \{1 : \sigma_3\}) = \{1 : \sigma_1, 2 : \sigma_2, 3 : \sigma_3\}$

• Introduce a new type constructor in other cases

  – $R_1 \land R_2$ constructs a new tuple without carrying out the $concat$ operation
Type Inference Algorithm used in Genesis

Determine the type of an action as follows:

1. Annotate the action by a bottom-up traversal of its tree. During the traversal unify type schemes according to a set of type inference rules

   (a) According to the type inference rules, append to a list a set of constraints that must hold at the end of type inference.

   (b) Introduce a new unity constraint for each tuple concatenation.

2. After annotation is complete, carry out the constraint satisfaction until all constraints are satisfied or until no further satisfaction is possible.
Example

Consider the Small program:

\[
\text{let fun } f(x,y) = x+y \\
\text{in} \\
\quad \text{output}(f(4,5)) \\
\text{end}
\]
The Program’s Action

bind "output" to native abstraction of an action
[ using a given (integer|truth-value) ] [ giving () ]
before
bind "input" to native abstraction of an action
[ giving an integer ] [ using the given () ]
hence

furthermore
recursively
bind "f" to closure of the abstraction of
furthermore

bind "x" to the given (integer|truth-value)≠1
and then
give the rest of the given data
then
bind "y" to the given (integer|truth-value)
then
give (integer|truth-value) bound to "x"
or
give [(integer|truth-value)]cell bound to "x"
and then
give (integer|truth-value) bound to "y"
or
give [(integer|truth-value)]cell bound to "y"
then
give the sum of the given data

hence

give 4 and then give 5
then
enact application of the abstraction
bound to "f" to the given data
then
enact application of the abstraction
bound to "output" to the given data
Annotating ‘give 4 and then give 5’

• Annotate ‘give 4’ and ‘give 5’

\[
give\ 4
: \langle\{\gamma_{154},\{\}\\gamma_{155}\}\rangle \leftrightarrow \langle\{1:\theta_{45}\},\{\}\rangle
\]

\[
give\ 5
: \langle\{\gamma_{156},\{\}\\gamma_{157}\}\rangle \leftrightarrow \langle\{1:\theta_{46}\},\{\}\rangle
\]

\[
[\theta_{45} \mapsto 4, \theta_{46} \mapsto 5]
\]

• Annotate ‘give 4 and then give 5’

\[
give\ 4
and\ then
\]

\[
give\ 5
: \langle\{\gamma_{154},\{\}\\gamma_{155}\}\rangle \leftrightarrow \langle\{\}\gamma_{158},\{\}\rangle
\]

\[
[\gamma_{156} \mapsto \{\{}\gamma_{154},\gamma_{157} \mapsto \{\}\\gamma_{155}\}
constraint:\ unity(\{\}\gamma_{158},\{1:\theta_{45}\}\wedge\{1:\theta_{46}\})
\]

16
Annotating More of the Action

- Annotate ‘bind “x”...’ and ‘give the rest’. The $\theta_{13}$ is there because the rest yields all but the first element of a tuple.

\[
\text{bind “x” to the given (integer|truth-value)} \#1 \\
: (\{1: \theta_{12}\} \gamma_{59}, \{\} \gamma_{60}) \mapsto (\{\}, \{x: \theta_{12}\})
\]

\[
\text{give the rest of the given data} \\
: (\{\} \rho_{71}, \{\} \gamma_{69}) \mapsto (\{\} \rho_{70}, \{\})
\]

$[\theta_{12} \mapsto (\text{integer|truth-value}), \theta_{13} \mapsto \text{datum}]$
constraint: $\text{unity}(\{1: \theta_{13}\} \land \{\} \rho_{70}, \{\} \rho_{71})$

- Annotate the combined action

\[
\mid \text{bind “x” to the given (integer|truth-value)} \#1 \\
\text{and then} \\
\mid \text{give the rest of the given data} \\
: (\{1: \theta_{12}\} \rho_{72}, \{\} \gamma_{60}) \mapsto (\{\} \rho_{73}, \{x: \theta_{12}\})
\]

$[\gamma_{59} \mapsto \{\} \rho_{72}, \rho_{71} \mapsto \{1: \theta_{12}\} \rho_{72}, \gamma_{69} \mapsto \{\} \gamma_{60}]$
constraint: $\text{unity}(\{\} \land \{\} \rho_{70}, \{\} \rho_{73})$
And Finally,

- **Annotate ‘bind “y” ...’**

  bind “y” to the given (integer|truth-value)
  : ([1:θ₁₆], {γ₇₅}) ↦ ([], {y:θ₁₆})
  
  [θ₁₆ ↦ (integer|truth-value)]

- **The Last Annotation**

  bind “x” to the given (integer|truth-value) #1
  and then
  give the rest of the given data
  then
  bind “y” to the given (integer|truth-value)
  : ([1:θ₁₂], ρ₇₂, {γ₆₀}) ↦ ([], {x:θ₁₂, y:θ₁₆})
  
  [γ₇₅ ↦ {}γ₆₀, ρ₇₃ ↦ {1:θ₁₆}]
1\textsuperscript{st} Pass of Constraint Satisfaction

- After applying the substitution, the set of constraints is
  
  1. $\text{unity} (\{\} \gamma_{158}, \{1: \theta_{45}\} \land \{1: \theta_{46}\})$
  2. $\text{unity} (\{1: \theta_{13}\} \land \{\} \rho_{70}, \{1: \theta_{12}\} \rho_{72})$
  3. $\text{unity} (\{\} \land \{\} \rho_{70}, \{1: \theta_{16}\})$

  $[\theta_{12} \mapsto (\text{integer}\text{|truth-value}), \theta_{13} \mapsto \text{datum}, \theta_{16} \mapsto (\text{integer}\text{|truth-value}), \theta_{45} \mapsto 4, \theta_{46} \mapsto 5]$

- The first tuple concatenation can be performed immediately, the second concatenation is undefined and results in \texttt{concatFailure}, and the third concatenation can be performed

- The resulting substitution is

  $[\gamma_{158} \mapsto \{1: \theta_{45}, 2: \theta_{46}\}, \rho_{70} \mapsto \{1: \theta_{16}\}]$
2nd Pass of Constraint Satisfaction

- Applying the substitution to the remaining constraint results in
  \[ \text{unity}(\{1:\theta_{13}\}\land\{1:\theta_{16}\},\{1:\theta_{12}\}\rho_{72}) \]

- Now tuple concatenation can be carried out to yield the substitution

\[
[\theta_{12} \mapsto \theta_{50}, \theta_{13} \mapsto \theta_{50}, \theta_{50} \mapsto \text{integer|truth-value}, \\
\rho_{72} \mapsto \{1:\theta_{16}\}]\]
Resulting Actions

- Concatenation to construct a tuple

\[
give 4 \\
and then \\
give 5 \\
: (\{}\gamma_{154},\{}\gamma_{155}\) \mapsto (\{1:\theta_{45},2:\theta_{46}\},\{\})
\]

- Concatenation to select from a tuple

\[
\begin{align*}
\text{bind } "x" \text{ to the given (integer|truth-value)} & \#1 \\
\text{and then} \\
\text{give the rest of the given data} \\
\text{then} \\
\text{bind } "y" \text{ to the given (integer|truth-value)} \\
: (\{1:\theta_{50},2:\theta_{16}\},\{\}\gamma_{60}) \mapsto (\{\},\{x:\theta_{50},y:\theta_{16}\})
\end{align*}
\]
Using ML as a Meta-language for Actions

If tuple concatenation were possible in ML, then ML could be used as a meta-language for Action Notation. Consider the following action

```
give number(4)
and_then
give number(5)
then
give the sum of the given_data
```
Action Notation in ML

ML functions could be written to support Action Notation. An action is a function that takes as input a tuple of transients and bindings and produces a tuple of transients and bindings. The transients would have to be of type dataTuple?

- Yielders

```ml
fun of x = x

fun the x = x

fun number x _ = dataTuple(x)

fun sum y d =

let val (a,b) = y d in dataTuple(a+b) end

fun given_data(t,b) = tupleOf(t)
```
The give action and Combinator Functions

fun give x (t,b) = (x (t,b), nobindings)

fun and_then (x,y) (t,b) =

let

    val (t',b') = x (t,b)
    val (t'',b'') = y (t,b)

in

    (concat(t', t''), merge(b',b''))

end

fun then (x,y) (t,b) =

let val (t',b') = x (t,b)

in y (t',b') end
Conclusion

- Tuple concatenation can be performed in the context of unification over records.

- A set of type inference rules govern how the record type schemes are combined to derive the type of an action.

- The type inference algorithm handles both constructing tuples and selecting from tuples by virtue of it being based on unification.

- A few functions have been written as a proof of concept using an int list to simulate the new dataTuple type.

- Can Hindley-Milner type inference be applied to type checking a dataTuple type?
Some sort inference rules for tuples

(And-Then)

\[ \varepsilon \vdash a_1 : (\tau_1, \beta_1) \iff (\tau'_1, \beta'_1); \varepsilon \vdash a_2 : (\tau_2, \beta_2) \iff (\tau'_2, \beta'_2) \]

\[ \varepsilon \vdash a_1 \text{ and then } a_2 : (\text{unify } \tau_1 \tau_2, \text{unify } \beta_1 \beta_2) \iff (\text{concat } \tau'_1 \tau'_2, \text{merge } \beta'_1 \beta'_2) \]

(Then)

\[ \varepsilon \vdash a_1 : (\tau_1, \beta_1) \iff (\tau'_1, \beta'_1); \varepsilon \vdash a_2 : (\tau_2, \beta_2) \iff (\tau'_2, \beta'_2); \text{unify } \tau_1 \tau_2 \neq \text{nothing} \]

\[ \varepsilon \vdash a_1 \text{ then } a_2 : (\tau_1, \text{unify } \beta_1 \beta_2) \iff (\tau'_2, \text{merge } \beta'_1 \beta'_2) \]

(Give-Individual)

\[ \varepsilon \vdash y : (\tau, \beta) \sim \sigma; \sigma \& I \neq \text{nothing} \]

\[ \varepsilon \vdash \text{give } y : (\tau, \beta) \iff (\{1 : \sigma\}, \{\}) \]

(Give-Data)

\[ \varepsilon \vdash y : (\tau, \beta) \sim \sigma; \sigma \& \Gamma \neq \text{nothing} \]

\[ \varepsilon \vdash \text{give } y : (\tau, \beta) \iff (\sigma, \{\}) \]

(Rest)

\[ \varepsilon \vdash y : (\tau, \beta) \sim \sigma; \theta \& \text{datum} \neq \text{nothing}; \text{unify } \sigma(\text{concat } \{1 : \theta\}\{\} \rho) \neq \text{nothing} \]

\[ \varepsilon \vdash \text{rest } y : (\tau, \beta) \sim \{\} \rho \]

(Given-Data)

\[ \varepsilon \vdash \text{given data} : (\tau, \beta) \sim \tau \]

(Given)

\[ \varepsilon \vdash s : \sigma; \theta \& \sigma \neq \text{nothing} \]

\[ \varepsilon \vdash \text{given } s : (\{1 : \theta\}, \{\} \gamma_1) \sim \theta \]

(Given#)

\[ \varepsilon \vdash s : \sigma; \theta \& \sigma \neq \text{nothing} \]

\[ \varepsilon \vdash \text{given } s \# n : (\{n : \theta\} \gamma_1, \{\} \gamma_2) \sim \theta \]

(Sum)

\[ \varepsilon \vdash y : (\tau, \beta) \sim \sigma; \theta_1, \theta_2, \theta_3 \& \text{integer} \neq \text{nothing}; \text{unify } \sigma\{1 : \theta_1, 2 : \theta_2\} \neq \text{nothing}; \theta_3 \leq [(\theta_1 | \theta_2)] \]

\[ \varepsilon \vdash \text{sum } y : (\tau, \beta) \sim \theta_3 \]
Some SML Code

val nobindings = []:(string * int) list
fun tuple0f([x,y]) = (x,y)

fun give x (t,b) = (x (t,b), nobindings)
fun 0f x = x
fun The x = x
fun number x _ = [x]

fun sum y d =
  let val (a,b) = y d
  in
    [a+b]
  end;

fun given_data(t,b) = tuple0f(t);

infix and_then;
fun op and_then (x,y) (t,b) =
let val (t’,b’) = x (t,b)
  val (t’’,b’’) = y (t,b)
in
  (t’@t’’,b’@b’’)
end;

infix Then;
fun op Then (x,y) ((t,b) : int list * (string * int) list) =
let val (t’,b’) = x (t,b)
in
  y (t’,b’)
end;

((give (number 4)) and_then (give (number 5)))
  Then
  (give (The (sum (0f (The given_data)))))) ([4,5],nobindings)