Type and Effect System Project Report

Samuel Klumpers (6057314) Philipp Zander (7983034)

2022-07-01

Contents

1 Overview

In this project we implemented Control Flow Analysis for the Fun language, by extending algorithm $\mathcal{W}$ to track annotations. We augment the Fun language with pairs, lists, and general datatypes, and provide new rules for the annotations in these cases.

We furthermore extend the Control Flow Analysis to Call Tracking Analysis by inserting `&` effects for calls into the rules, and implement subtyping to replace subeffecting.

See the haddocks too.

Compile the project by executing `stack build`. Run both implemented analyses on `examples/loop.fun` by executing `stack run -- loop`.

The output can be read as ordinary Haskell types with annotations. The annotated type τ `&` ψ indicates that evaluating may call all functions at program points in the set ψ. In the arrow type τ_0 `-`φ`;`ψ`->` τ_1 of a function f, the φ annotation is the set of locations f may have been defined, while ψ is the set of functions that may be called when applying f. The annotation φ in the tuple type `pair` φ`(`τ_0`,` τ_1`)` and in the list type τ `list` φ represent the sets of locations the tuple or list may have been created.

2 Deviation from the slides/book

Algorithm $\mathcal{W}$ as presented in slides and book does not allow implementing the following poison free lambda rule as is demonstrated by `examples/poison.fun`.

$$\frac{\Gamma[\mathtt{a} \mapsto \tau_0] \ \vdash \ e \ : \ \tau_1}{\Gamma \ \vdash \ \lambda_\pi \mathtt{a}. \ e \ : \ \tau_0 \xrightarrow{\{\pi\}} \tau_1} \text{lam}$$

So we are implementing the following rule instead keeping $\varphi = \emptyset$ whenever the design of $\mathcal{W}$ allows it.

$$\frac{\Gamma[\mathtt{a} \mapsto \tau_0] \ \vdash \ e \ : \ \tau_1}{\Gamma \ \vdash \ \lambda_\pi \mathtt{a}. \ e \ : \ \tau_0 \xrightarrow{\{\pi\} \cup \varphi} \tau_1} \text{lam}$$

3 Rules for datatypes

In the following sections, we omit the hats (e.g. $\hat{\tau}$) on the types.

To implement product types into the Fun language, we add the pair introduction and elimination rules to the type and effect axioms.

$$\frac{\Gamma \vdash e_0 \;:\; \tau_0 \qquad \Gamma \vdash e_1 \;:\; \tau_1}{\Gamma \vdash \texttt{Pair}_\pi\texttt{(}e_0\texttt{, }e_1\texttt{)} \;:\; \tau_0 \times^{\{\pi\}} \tau_1}\ \text{pair introduction}$$

The annotation on the `Pair` constructor denotes the set of program points the pair may have been created.

$$\frac{\Gamma \vdash e_0 \;:\; \tau_0 \times^{\varphi} \tau_1 \qquad \Gamma[\texttt{x0} \mapsto \tau_0][\texttt{x1} \mapsto \tau_1] \vdash e_1 \;:\; \tau_2}{\Gamma \vdash \texttt{pcase}\ e_0\ \texttt{of Pair(x0, x1) =>}\ e_1 \;:\; \tau_2}\ \text{pair elimination}$$

The type parameters of the product type ensure that the values passed to pair constructors retain their annotations when they are extracted by pattern matching.

In a similar fashion we introduce rules for the introduction and elimination of lists.

$$\frac{\Gamma \vdash e_0 \;:\; \tau \qquad \Gamma \vdash e_1 \;:\; \tau\ \texttt{list}^{\varphi}}{\Gamma \vdash \texttt{Cons}_\pi\texttt{(}e_0\texttt{, }e_1\texttt{)} \;:\; \tau\ \texttt{list}^{\varphi \cup \{\pi\}}}\ \text{list introduction}$$

$$\frac{}{\Gamma \vdash \texttt{Nil}_\pi \;:\; \tau\ \texttt{list}^{\{\pi\}}}\ \text{list introduction}$$

$$\frac{\Gamma \vdash e_0 \;:\; \tau_0\ \texttt{list}^{\varphi} \qquad \Gamma[\texttt{x0} \mapsto \tau_0][\texttt{x1} \mapsto \tau_0\ \texttt{list}^{\varphi}] \vdash e_1 \;:\; \tau_1 \qquad \Gamma \vdash e_2 \;:\; \tau_1}{\Gamma \vdash \texttt{lcase}\ e_0\ \texttt{of Cons(x0, x1) =>}\ e_1\ \texttt{or}\ e_2 \;:\; \tau_1}\ \text{list elimination}$$

Unlike in the case of pairs, the second field of the `Cons` shares its only type variable with the first field, so we must lose some information of one or both values inserted in the fields. We choose to merge the annotation of the `Cons` constructor with the annotation on the tail. This means we forget that e.g. $\texttt{Cons}_1\texttt{(1, Nil}_3\texttt{)}$ is evidently defined at 1, and record it as $\{1, 3\}$, but we do retain that in pattern matching `Cons(x, y) => y`, `y` could have been defined at 3.

For general data types we give introduction and elimination rules as follows. For all data constructors $\texttt{C}_\pi$ with data fields $\pi_1, \ldots, \pi_n$, type constructors `D` with type parameters $\texttt{a}_1, \ldots, \texttt{a}_k$, if $\texttt{C}_\pi$ is a constructor of `D` and $\forall\, 1 \leq i \leq n$: $(\forall\, 1 \leq j \leq k \colon \pi_i = \texttt{a}_j \implies \tau_i = \tau'_j) \wedge ((\nexists\, 1 \leq j \leq k \colon \pi_i = \texttt{a}_j) \implies \tau_i = \pi_i)$ then

$$\frac{\Gamma \vdash e_1 \;:\; \tau_1 \qquad \cdots \qquad \Gamma \vdash e_n \;:\; \tau_n}{\Gamma \vdash \texttt{C}_\pi\texttt{(}e_1\texttt{, ..., }e_n\texttt{)} \;:\; \texttt{D}\ \tau'_1\ \texttt{...}\ \tau'_k}\ \text{data introduction}$$

$$\frac{\Gamma \vdash e_0 \;:\; \texttt{D}\ \tau_{i_1}\ \texttt{...}\ \tau_{i_k} \qquad \Gamma[\texttt{x1} \mapsto \tau_1] \cdots [\texttt{xn} \mapsto \tau_n] \vdash e_1 \;:\; \tau_0 \qquad \Gamma \vdash e_2 \;:\; \tau_0}{\Gamma \vdash \texttt{dcase}\ e_0\ \texttt{of Cons(x1,..., xn) =>}\ e_1\ \texttt{or}\ e_2 \;:\; \tau_0}\ \text{data elimination}$$

4 Rules for Call Tracking Analysis

We adapt the rules for Control Flow Analysis to Call Tracking Analysis. Most of the changes are unremarkable, and propagate and combine sets of call labels through the side-effect. Notably, the arrow type gets a new annotation, so that in $\texttt{f : } \tau_0 \xrightarrow{\varphi, \psi} \tau_1$, φ still represents the program points the function may have been defined, but ψ now records the functions that might be called when we call `f`.

We will write out most resulting effects in terms of the effects in the hypotheses, but this is only for emphasis on the flow of the side-effects. In fact, we could replace most instances of ψ_i with simply ψ here: subtyping would take care of almost all unions for us!

$$\frac{\Gamma[\texttt{a} \mapsto \tau_0] \vdash e \texttt{ : } \tau_1 \texttt{ \& } \psi}{\Gamma \vdash \lambda_\pi \texttt{a. } e \texttt{ : } \tau_0 \xrightarrow{\{\pi\}, \psi} \tau_1 \texttt{ \& } \emptyset}\ \text{lam}$$

$$\frac{\Gamma \vdash e \texttt{ : } \tau \texttt{ \& } \psi \qquad \tau \leq \tau' \qquad \psi \leq \psi'}{\Gamma \vdash e \texttt{ : } \tau' \texttt{ \& } \psi'}\ \text{sub}$$

$$\frac{\Gamma[\texttt{f} \mapsto \tau_0 \xrightarrow{\{\pi\}, \psi} \tau_1][\texttt{a} \mapsto \tau_0] \vdash e \texttt{ : } \tau_1 \texttt{ \& } \psi}{\Gamma \vdash \mu \texttt{f. } \lambda_\pi \texttt{a. } e \texttt{ : } \tau_0 \xrightarrow{\{\pi\}, \psi} \tau_1 \texttt{ \& } \emptyset}\ \text{mu}$$

$$\frac{\Gamma \vdash e_0 \texttt{ : } \tau^0_\oplus \texttt{ \& } \psi_0 \qquad \Gamma \vdash e_1 \texttt{ : } \tau^1_\oplus \texttt{ \& } \psi_1}{\Gamma \vdash e_0 \oplus e_1 \texttt{ : } \tau_\oplus \texttt{ \& } \psi_0 \cup \psi_1}\ \text{op}$$

$$\frac{\Gamma \vdash e_0 \texttt{ : } \sigma \texttt{ \& } \psi_0 \qquad \Gamma[\texttt{a} \mapsto \sigma] \vdash e_1 \texttt{ : } \tau \texttt{ \& } \psi_1}{\Gamma \vdash \texttt{let a = } e_0 \texttt{ in } e_1 \texttt{ : } \tau \texttt{ \& } \psi_0 \cup \psi_1}\ \text{let}$$

$$\frac{\Gamma \vdash e_0 \texttt{ : Bool \& } \psi_0 \qquad \Gamma \vdash e_1 \texttt{ : } \tau_1 \texttt{ \& } \psi_1 \qquad \Gamma \vdash e_2 \texttt{ : } \tau_1 \texttt{ \& } \psi_2}{\Gamma \vdash \texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2 \texttt{ : } \tau_1 \texttt{ \& } \psi_0 \cup \psi_1 \cup \psi_2}\ \text{if}$$

$$\frac{\Gamma \vdash e_0 \texttt{ : } \tau_0 \xrightarrow{\varphi, \psi_0} \tau_1 \texttt{ \& } \psi_1 \qquad \Gamma \vdash e_1 \texttt{ : } \tau_0 \texttt{ \& } \psi_2}{\Gamma \vdash e_0\ e_1 \texttt{ : } \tau_1 \texttt{ \& } \varphi \cup \psi_0 \cup \psi_1 \cup \psi_2}\ \text{app}$$

$$\frac{\Gamma(\texttt{a}) = \sigma}{\Gamma \vdash \texttt{a : } \sigma \texttt{ \& } \emptyset}\ \text{var}$$

$$\frac{}{\Gamma \vdash n \texttt{ : Int \& } \emptyset}\ \text{num}$$

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool} \ \& \ \emptyset}\ \text{false}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool} \ \& \ \emptyset}\ \text{true}$$

$$\frac{\Gamma \vdash e_0 : \tau_0 \ \& \ \psi_0 \qquad \Gamma \vdash e_1 : \tau_1 \ \& \ \psi_1}{\Gamma \vdash \texttt{Pair}_\pi(e_0\texttt{,}\ e_1) : \tau_0 \times^{\{\pi\}} \tau_1 \ \& \ \psi_0 \cup \psi_1}\ \text{pair introduction}$$

$$\frac{\Gamma \vdash e_0 : \tau_0 \times^{\varphi} \tau_1 \ \& \ \psi_0 \qquad \Gamma[\texttt{x0} \mapsto \tau_0][\texttt{x1} \mapsto \tau_1] \vdash e_1 : \tau_2 \ \& \ \psi_1}{\Gamma \vdash \texttt{pcase}\ e_0\ \texttt{of Pair(x0, x1) =>}\ e_1 : \tau_2 \ \& \ \psi_0 \cup \psi_1}\ \text{pair elimination}$$

$$\frac{\Gamma \vdash e_0 : \tau \ \& \ \psi_0 \qquad \Gamma \vdash e_1 : \tau\ \texttt{list}^{\varphi} \ \& \ \psi_1}{\Gamma \vdash \texttt{Cons}_\pi(e_0\texttt{,}\ e_1) : \tau\ \texttt{list}^{\varphi \cup \{\pi\}} \ \& \ \psi_0 \cup \psi_1}\ \text{list introduction}$$

$$\frac{}{\Gamma \vdash \texttt{Nil}_\pi : \tau\ \texttt{list}^{\{\pi\}} \ \& \ \emptyset}\ \text{list introduction}$$

$$\frac{\Gamma \vdash e_0 : \tau_0\ \texttt{list}^{\varphi} \ \& \ \psi_0 \qquad \Gamma[\texttt{x0} \mapsto \tau_0][\texttt{x1} \mapsto \tau_0\ \texttt{list}^{\varphi}] \vdash e_1 : \tau_1 \ \& \ \psi_1 \qquad \Gamma \vdash e_2 : \tau_1 \ \& \ \psi_2}{\Gamma \vdash \texttt{lcase}\ e_0\ \texttt{of Cons(x0, x1) =>}\ e_1\ \texttt{or}\ e_2 : \tau_1 \ \& \ \psi_0 \cup \psi_1 \cup \psi_2}\ \text{list elimination}$$

5 Subtyping

5.1 Rules

$$\frac{\Gamma \vdash e : \tau \ \& \ \psi \qquad \tau \leq \tau' \qquad \psi \subseteq \psi'}{\Gamma \vdash e : \tau' \ \& \ \psi'}\ \text{sub}$$

$$\frac{\tau_0' \leq \tau_0 \qquad \tau_1 \leq \tau_1' \qquad \varphi \subseteq \varphi' \qquad \psi \subseteq \psi'}{\tau_0 \xrightarrow{\varphi, \psi} \tau_1 \leq \tau_0' \xrightarrow{\varphi', \psi'} \tau_1'}\ \text{sub}$$

5.2 Examples

The following was inferred with subtyping.

```
Expression:
let f = fn₁ a => a 0 in
let x = f (fn₂ a => a) in
let g = fn₃ a => a 0 in
let y = g (fn₄ a => a) in
let z = if false then  f else  g in
f

Type:
forall a₆₄.(Int -{2};{}-> a₆₄) -{1};{2}-> a₆₄ & {1,2,3,4}
```

The following was inferred with subeffecting only demonstrating poisoning.

```
Expression:
let f = fn_1 a => a 0 in
let x = f (fn_2 a => a) in
let g = fn_3 a => a 0 in
let y = g (fn_4 a => a) in
let z = if  false then  f else  g in
f

Type:
forall a_56.(Int -{2,4};{}-> a_56) -{1};{2,4}-> a_56 & {1,2,3,4}
```

The following was inferred with subtyping.

```
Expression:
let f = fn_1 a => a 0 in
let x = f (fn_2 a => a) in
let g = fn_3 a => a 0 in
let y = g (fn_4 a => a) in
let z = if  false then  f else  g in
z

Type:
forall a_64.(Int -{};{}-> a_64) -{1,3};{2,4}-> a_64 & {1,2,3,4}
```

The following was inferred with subeffecting only.

```
Expression:
let f = fn_1 a => a 0 in
let x = f (fn_2 a => a) in
let g = fn_3 a => a 0 in
let y = g (fn_4 a => a) in
let z = if  false then  f else  g in
z

Type:
forall a_56.(Int -{2,4};{}-> a_56) -{1,3};{2,4}-> a_56 & {1,2,3,4}
```

The following was inferred with subtyping.

```
let unify = fn_1 a => fn_2 b => if  false then  a else  b in
```

```
let f = fn_3 a => a 0 in
let x = f (fn_4 a => a) in
let g = fn_5 a => a 0 in
let y = g (fn_6 a => a) in
let z = unify f g in
f

Type:
forall a_96.(Int -{4};{}-> a_96) -{3};{4}-> a_96 & {1,2,3,4,5,6}
```

The following was inferred with subeffecting only only demonstrating poisoning.

```
Expression:
let unify = fn_1 a => fn_2 b => if  false then  a else  b in
let f = fn_3 a => a 0 in
let x = f (fn_4 a => a) in
let g = fn_5 a => a 0 in
let y = g (fn_6 a => a) in
let z = unify f g in
f

Type:
forall a_78.(Int -{4,6};{}-> a_78) -{3};{4,6}-> a_78 & {1,2,3,4,5,6}
```

The following was inferred with subtyping.

```
Expression:
let unify = fn_1 a => fn_2 b => if  false then  a else  b in
let f = fn_3 a => a 0 in
let x = f (fn_4 a => a) in
let g = fn_5 a => a 0 in
let y = g (fn_6 a => a) in
let z = unify f g in
z

Type:
forall a_96.(Int -{};{}-> a_96) -{3,5};{4,6}-> a_96 & {1,2,3,4,5,6}
```

The following was inferred with subeffecting only.

```
Expression:
```

```
let unify = fn_1 a => fn_2 b => if  false then  a else  b in
let f = fn_3 a => a 0 in
let x = f (fn_4 a => a) in
let g = fn_5 a => a 0 in
let y = g (fn_6 a => a) in
let z = unify f g in
z

Type:
forall a_78.(Int -{4,6};{}-> a_78) -{3,5};{4,6}-> a_78 & {1,2,3,4,5,6}
```

The following was inferred with subtyping.

```
Expression:
let f = fn_1 a => a (fn_2 a => a) in
let g = fn_3 a => a (fn_4 a => a) in
let z = if  false then  f else  g in
f

Type:
forall a_48 a_49.((a_48 -{2};{}-> a_48) -{};{}-> a_49) -{1};{}-> a_49 & {}
```

The following was inferred with subeffecting only only demonstrating poisoning.

```
Expression:
let f = fn_1 a => a (fn_2 a => a) in
let g = fn_3 a => a (fn_4 a => a) in
let z = if  false then  f else  g in
f

Type:
forall a_40 a_41.((a_40 -{2,4};{}-> a_40) -{};{}-> a_41) -{1};{}-> a_41 & {}
```

The following was inferred with subtyping.

```
Expression:
let f = fn_1 a => a (fn_2 a => a) in
let g = fn_3 a => a (fn_4 a => a) in
let z = if  false then  f else  g in
z
```

Type:
forall a_{48} a_{49}.((a_{48} -{2,4};{}-> a_{48}) -{};{}-> a_{49}) -{1,3};{}-> a_{49} & {}

The following was inferred with subeffecting only.

Expression:
let f = $\mathtt{fn}_1$ a => a ($\mathtt{fn}_2$ a => a) in
let g = $\mathtt{fn}_3$ a => a ($\mathtt{fn}_4$ a => a) in
let z = if false then f else g in
z

Type:
forall a_{40} a_{41}.((a_{40} -{2,4};{}-> a_{40}) -{};{}-> a_{41}) -{1,3};{}-> a_{41} & {}

The following was inferred with subtyping.

let unify = $\mathtt{fn}_1$ a => $\mathtt{fn}_2$ b => if false then a else b in
let f = $\mathtt{fn}_3$ a => a ($\mathtt{fn}_4$ a => a) in
let g = $\mathtt{fn}_5$ a => a ($\mathtt{fn}_6$ a => a) in
let z = unify f g in
f

Type:
forall a_{84} a_{85}.((a_{84} -{4};{}-> a_{84}) -{};{}-> a_{85}) -{3};{}-> a_{85} & {1,2}

The following was inferred with subeffecting only only demonstrating poisoning.

let unify = $\mathtt{fn}_1$ a => $\mathtt{fn}_2$ b => if false then a else b in
let f = $\mathtt{fn}_3$ a => a ($\mathtt{fn}_4$ a => a) in
let g = $\mathtt{fn}_5$ a => a ($\mathtt{fn}_6$ a => a) in
let z = unify f g in
f

Type:
forall a_{62} a_{63}.((a_{62} -{4,6};{}-> a_{62}) -{};{}-> a_{63}) -{3};{}-> a_{63} & {1,2}

The following was inferred with subtyping.

let unify = $\mathtt{fn}_1$ a => $\mathtt{fn}_2$ b => if false then a else b in
let f = $\mathtt{fn}_3$ a => a ($\mathtt{fn}_4$ a => a) in
let g = $\mathtt{fn}_5$ a => a ($\mathtt{fn}_6$ a => a) in
let z = unify f g in

```
z
```

Type:
```
forall a_{84} a_{85}.((a_{84} -{4,6};{}-> a_{84}) -{};{}-> a_{85}) -{3,5};{}-> a_{85} & {1,2}
```

The following was inferred with subeffecting only.

```
let unify = fn_1 a => fn_2 b => if  false then  a else  b in
let f = fn_3 a => a (fn_4 a => a) in
let g = fn_5 a => a (fn_6 a => a) in
let z = unify f g in
z
```

Type:
```
forall a_{62} a_{63}.((a_{62} -{4,6};{}-> a_{62}) -{};{}-> a_{63}) -{3,5};{}-> a_{63} & {1,2}
```

Our implementation can be switched from subtyping to subeffecting by replacing `subtype` with the alternative from line 204.