



Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

APA Effects in Type Systems

Jurriaan Hage

e-mail: J.Hage@uu.nl

homepage: <http://www.cs.uu.nl/people/jur/>

Department of Information and Computing Sciences, Universiteit Utrecht

May 31, 2018

Effects

- ▶ In static analysis we compute properties of programs.
- ▶ In functional languages we tend to consider programs, expressions and values to be relatively similar.
- ▶ However, computations and values are different from an optimizer's perspective:
- ▶ Types are about properties of values (being an integer, being even, be storable in 4 bits)
- ▶ **Effects** are properties of computations
 - ▶ the maximum number of memory allocations
 - ▶ the set of functions that may be applied during evaluation
- ▶ Often come up in side-effected language, but not only there.



The Fun language

- ▶ Lambda calculus with the necessary syntactic sugar
- ▶ Arithmetic and boolean expressions as in `While`.
- ▶ ML style function declarations
 - ▶ `fn x => e` for anonymous, non-recursive functions
 - ▶ `fun f x => e` for anonymous, recursive functions
- ▶ An if-then-else construct is present.
- ▶ Example is forthcoming.



Fun with assignments and references

- ▶ Imperative constructs for Fun:

$$e ::= \dots \mid \mathbf{new}_\pi x := e_1 \mathbf{in} e_2 \mid !x \mid x := e_0 \mid e_1; e_2$$

- ▶ **new** introduces a statically scoped reference and initializes the value it refers to.
 - ▶ We need program point annotation π again.
- ▶ Deferencing the value of the reference x is via the **!** operator.
 - ▶ Explicit difference between rvalue and lvalue
- ▶ Assignments may set this value to a new one.
- ▶ Sequencing **;** first evaluates e_1 for its effect on the state, then evaluates e_2 (in this new state) and returns this value.



Example

- ▶ This variant of fibonacci uses a 'global' variable r to compute:

```
newR  $r := 0$ 
```

```
in    let fib = funF  $f z =>$  if  $z < 3$   
                                then  $r := !r + 1$   
                                else  $f(z - 1); f(z - 2)$ 
```

```
    in fib  $x; !r$ 
```

- ▶ The **fib** definition assigns to and dereferences the reference variable created at program point R.



Side Effect Analysis

- ▶ Side Effect Analysis determines
For each subexpression, which locations have been created, accessed and assigned.
- ▶ Monomorphic/monovariant, but with subeffecting.
- ▶ No algorithm.



Annotations

- ▶ Annotations are sets of effects (three kinds):

$$\varphi ::= \{!\pi\} \mid \{\pi :=\} \mid \{\mathbf{new}\pi\} \mid \varphi_1 \cup \varphi_2 \mid \emptyset$$

- ▶ $\{!\pi\}$ means that in the expression to which it is attached, a location created at program point π was accessed.
 - ▶ And similarly for the others
- ▶ We also need sets of program points:

$$\omega ::= \{\pi\} \mid \omega \cup \omega \mid \emptyset$$



- ▶ ω s (sets of locations) are **equal modulo** UCAI.
- ▶ Values are considered the same if they only differ in order, parentheses and the presence of unit or cancelling values because of idempotence.

- ▶ For example:

$$\begin{aligned} (\{\pi_1\} \cup \{\pi_2\}) \cup \emptyset &\stackrel{U}{=} \\ \{\pi_1\} \cup \{\pi_2\} &\stackrel{I}{=} \\ \{\pi_1\} \cup (\{\pi_2\} \cup \{\pi_2\}) &\stackrel{A}{=} \\ (\{\pi_1\} \cup \{\pi_2\}) \cup \{\pi_2\} &\stackrel{C}{=} \\ (\{\pi_2\} \cup \{\pi_1\}) \cup \{\pi_2\} & \end{aligned}$$

- ▶ We may simply write $\{\pi_1, \pi_2\}$.



Annotated types

- ▶ Annotated types are defined to be

$$\widehat{\tau} ::= \mathbf{int} \mid \mathbf{bool} \mid \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2 \mid \mathbf{ref}_{\omega} \widehat{\tau}$$

- ▶ Example: $\mathbf{int} \xrightarrow{\{!R, R=\}} \mathbf{int}$.



Judgments

- ▶ Judgments for Side Effect Analysis are of the form

$$\hat{\Gamma} \vdash_{\text{SE}} e : \hat{\tau} \ \& \ \varphi$$

- ▶ The name *type and effect system* should now become apparent.
 - ▶ Every expression has an (annotated) type and an effect.



Rule for let-expressions

$$\frac{\widehat{\Gamma} \vdash_{\text{SE}} e_1 : \widehat{\tau}_1 \ \& \ \varphi_1 \quad \widehat{\Gamma}[x \mapsto \widehat{\tau}_1] \vdash_{\text{SE}} e_2 : \widehat{\tau}_2 \ \& \ \varphi_2}{\widehat{\Gamma} \vdash_{\text{SE}} \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \widehat{\tau}_2 \ \& \ \varphi_1 \cup \varphi_2} \quad [\mathbf{let}]$$

Effects typically accumulate: $\varphi_1 \cup \varphi_2$.



Rule for abstraction

$$\frac{\widehat{\Gamma}[x \mapsto \widehat{\tau}_x] \vdash_{\text{SE}} e_0 : \widehat{\tau}_0 \ \& \ \varphi_0}{\widehat{\Gamma} \vdash_{\text{SE}} \mathbf{fn}_{\pi} x \Rightarrow e_0 : \widehat{\tau}_x \xrightarrow{\varphi_0} \widehat{\tau}_0 \ \& \ \emptyset} \text{ [fn]}$$

- ▶ A function body has effect, defining a function does not.
- ▶ Effects of bodies are stored on the arrows in [fn] or [fun].



Rule for application

$$\frac{\widehat{\Gamma} \vdash_{\text{SE}} e_1 : \widehat{\tau}_2 \xrightarrow{\varphi_0} \widehat{\tau}_0 \ \& \ \varphi_1 \quad \widehat{\Gamma} \vdash_{\text{SE}} e_2 : \widehat{\tau}_2 \ \& \ \varphi_2}{\widehat{\Gamma} \vdash_{\text{SE}} e_1 \ e_2 : \widehat{\tau}_0 \ \& \ \varphi_0 \cup \varphi_1 \cup \varphi_2} \quad [\text{app}]$$

- ▶ Application retrieves the effect of executing body from annotated type of function.
- ▶ Contributes it to the total effect.
- ▶ Abstraction rule stores effect on arrow type, application retrieves it.
 - ▶ Help deal with the non-compositional aspect of function definition.



Rule for dereference

$$\frac{\widehat{\Gamma}(x) = \mathbf{ref}_{\{\pi_1, \dots, \pi_n\}} \widehat{\tau}}{\widehat{\Gamma} \vdash_{\text{SE}} !x : \widehat{\tau} \ \& \ \{!\pi_1, \dots, !\pi_n\}} \quad [\text{deref}]$$

- ▶ $\{\pi_1, \dots, \pi_n\}$ describes all program points where the reference x may have been created.
- ▶ Why a set?



Rule for dereference

$$\frac{\widehat{\Gamma}(x) = \mathbf{ref}_{\{\pi_1, \dots, \pi_n\}} \widehat{\tau}}{\widehat{\Gamma} \vdash_{\text{SE}} !x : \widehat{\tau} \ \& \ \{!\pi_1, \dots, !\pi_n\}} \quad [\text{deref}]$$

- ▶ $\{\pi_1, \dots, \pi_n\}$ describes all program points where the reference x may have been created.
- ▶ Why a set?
- ▶ Various reasons:
 - ▶ Reference variables can be function arguments
 - ▶ Conditionals



Rule for new-expression

$$\frac{\hat{\Gamma} \vdash_{\text{SE}} e_1 : \hat{\tau}_1 \ \& \ \varphi_1 \quad \hat{\Gamma}[x \mapsto \mathbf{ref}_{\{\pi\}} \ \hat{\tau}_1] \vdash_{\text{SE}} e_2 : \hat{\tau}_2 \ \& \ \varphi_2}{\hat{\Gamma} \vdash_{\text{SE}} \mathbf{new}_{\pi} \ x := e_1 \ \mathbf{in} \ e_2 : \hat{\tau}_2 \ \& \ \varphi_1 \cup \varphi_2 \cup \{\mathbf{new}\pi\}} \quad [\mathbf{new}]$$

- ▶ Put the annotation into the type of x and add its effect.



Rule for assignments

$$\frac{\widehat{\Gamma} \vdash_{\text{SE}} e : \widehat{\tau} \ \& \ \varphi \quad \widehat{\Gamma}(x) = \mathbf{ref}_{\{\pi_1, \dots, \pi_n\}} \widehat{\tau}}{\widehat{\Gamma} \vdash_{\text{SE}} x := e : \widehat{\tau} \ \& \ \varphi \cup \{\pi_1 :=, \dots, \pi_n :=\}} \quad [\text{ass}]$$

- ▶ Simply add annotations to denote the fact that x has a new value.



Example

- ▶ $\text{new}_A x := 1$
 in $(\text{new}_B y := !x \text{ in } (x := !y + 1; !y + 3))$
 $+ (\text{new}_C x := !x \text{ in } (x := !x + 1; !x + 1))$
- ▶ First summand has type and effect:
 $\text{int} \ \& \ \{\text{new}_B, !A, A:=, !B\}$
- ▶ Second summand has type and effect:
 $\text{int} \ \& \ \{\text{new}_C, !A, C:=, !C\}$
 - ▶ The updated x is the local, not the global one
- ▶ Together we get
 $\text{int} \ \& \ \{\text{new}_A, !A, A:=, \text{new}_B, !B, \text{new}_C, C:=, !C\}$
- ▶ Conclusion: reference created at B is never assigned to, so could be replaced by an ordinary integer variable.



Poisoning

- ▶ $\text{new}_A x := 1$
in $(\text{fn } f \Rightarrow f (\text{fn } y \Rightarrow !x) + f(\text{fn } z \Rightarrow (x := z; z)))$
 $(\text{fn } g \Rightarrow g 1)$
- ▶ Determine that f has the type $(\text{int}^{\{!A, A=\}} \rightarrow \text{int})^{\{!A, A=\}} \text{int}$



Poisoning

- ▶ $\text{new}_A x := 1$
 $\text{in } (\text{fn } f \Rightarrow f (\text{fn } y \Rightarrow !x) + f(\text{fn } z \Rightarrow (x := z; z)))$
 $(\text{fn } g \Rightarrow g 1)$
- ▶ Determine that f has the type $(\text{int}^{\{!A, A:=\}} \rightarrow \text{int})^{\{!A, A:=\}} \text{int}$
- ▶ In the presence of poisoning, both arguments must have exactly the type of the argument to f , $(\text{int}^{\{!A, A:=\}} \rightarrow \text{int})$.
- ▶ We would prefer $(\text{fn } y \Rightarrow !x) : (\text{int}^{\{!A\}} \rightarrow \text{int})$ and
 $(\text{fn } z \Rightarrow (x := z; z)) : (\text{int}^{\{A:=\}} \rightarrow \text{int})$.
- ▶ And to weaken annotations independently and only when we must.



Subtyping

$$\frac{\widehat{\Gamma} \vdash_{\text{SE}} e : \widehat{\tau} \ \& \ \varphi \quad \widehat{\tau} \leq \widehat{\tau}' \quad \varphi \subseteq \varphi'}{\widehat{\Gamma} \vdash_{\text{SE}} e : \widehat{\tau}' \ \& \ \varphi'} \quad [\text{sub}]$$

- ▶ Subeffecting/subtyping performed by a single rule.
- ▶ The rule allows us to weaken analysis results when appropriate:
 - ▶ $\widehat{\tau} \leq \widehat{\tau}'$: $\widehat{\tau}'$ is weaker than $\widehat{\tau}$.
 - ▶ $\varphi \subseteq \varphi'$: φ' is weaker than φ .
 - ▶ In the example: large sets are weaker.
- ▶ The rule is **not** syntax directed.
 - ▶ It can always be applied, forever.
- ▶ Typically, subsumption is built into [app], [if] etc.



The example again

- ▶ $\text{new}_A x := 1$
in $(\text{fn } f \Rightarrow f (\text{fn } y \Rightarrow !x) + f(\text{fn } z \Rightarrow (x := z; z)))$
 $(\text{fn } g \Rightarrow g 1)$
- ▶ Weaken the type when necessary (when a value is “used”):

$$(\text{int}^{\{!A\}} \rightarrow \text{int}) \leq (\text{int}^{\{!A, A=\}} \rightarrow \text{int})$$

$$(\text{int}^{\{A=\}} \rightarrow \text{int}) \leq (\text{int}^{\{!A, A=\}} \rightarrow \text{int})$$

- ▶ Larger type for f does not change types of its arguments.
- ▶ Just before matching the type of an argument with the formal parameter type.
- ▶ Just before checking that the then-part and else-part have matching types.



The subtyping relation

- ▶ We should now define \leq for annotated types.
- ▶ Example (function types):

$$\frac{\hat{\tau}'_1 \leq \hat{\tau}_1 \quad \hat{\tau}_2 \leq \hat{\tau}'_2 \quad \varphi \subseteq \varphi'}{\hat{\tau}_1 \xrightarrow{\varphi} \hat{\tau}_2 \leq \hat{\tau}'_1 \xrightarrow{\varphi'} \hat{\tau}'_2}$$

- ▶ The subtyping relation is
 - ▶ covariant in the result
 - ▶ contravariant in the argument
 - ▶ and covariant in the argument of the argument, etc.
- ▶ The reference type $\mathbf{ref}_\omega \hat{\tau}$ is both covariant and contravariant (invariant) in $\hat{\tau}$.
 - ▶ A reference can be used to read from and write to.



Meanwhile, in a real life...

- ▶ Say you bought a new laptop with a six month guarantee
- ▶ What can happen?
 - ▶ The shopowner may fix a problem after seven months
 - ▶ You decide not to bring it back when it break at 5,5 months
- ▶ Effectively, each party can change the guarantee time but each in his/her own direction.
- ▶ You can not demand more than the guarantee, the shop owner cannot give you less.



Contravariance example

- ▶ Consider sets of signs as annotation and a function with analysis:

$$f :: \mathbf{int}^{\{0,+\}} \rightarrow \mathbf{int}^{\{-,0\}}$$

- ▶ This is a may-style analysis so we can weaken to

$$\mathbf{int}^{\{0,+\}} \rightarrow \mathbf{int}^{\{-,0,+\}}$$

- ▶ But what can be done with $\mathbf{int}^{\{0,+\}}$?
- ▶ If f returns a value in $\{-,0\}$ for positive arguments and zero, then it also returns such values if we restrict to $\{0\}$.
- ▶ Thus: $\mathbf{int}^{\{0\}} \rightarrow \mathbf{int}^{\{-,0\}}$ is a safe approximation of f
- ▶ Applicability of f is restricted: only for arguments 0.
- ▶ Note: growing the set on the argument may not be safe!



More covariance and contravariance

- ▶ A fact of life (with subtyping) that must be dealt with.
- ▶ Essentially it distinguishes between consuming a value and producing one.
 - ▶ And this has implications for how we should handle them.
- ▶ In **Java**: S extends T , and T extends U
 - ▶ Assume a method

T work($T t$) .

Then we may safely

- ▶ pass an S , but not a U to the method work,
- ▶ use the result of work where a U is expected, but not where we need an S .
- ▶ In other words, T work($T t$) may be weakened to U work($S t$).



More covariance and contravariance

- ▶ A fact of life (with subtyping) that must be dealt with.
- ▶ Essentially it distinguishes between consuming a value and producing one.
 - ▶ And this has implications for how we should handle them.
- ▶ In **Haskell**: $f :: \text{Eq } a \Rightarrow a \rightarrow a$
 - ▶ We may pass values b to f that have at least $\text{Eq } b$, so they may have also $\text{Ord } b$
 - ▶ We may write $\text{id } (f x)$, forgetting that a has $\text{Eq } a$
- ▶ Bottom-line: changing a value safely (weakening) is done differently depending on variance.



More?

- ▶ Call Tracking Analysis is an effect analysis that is much related to CFA.
- ▶ In Call Tracking Analysis:
 - Which functions may have been called during the evaluation of an expression.*
- ▶ In 2006, an assignment was to give a deduction system and algorithm for the monomorphic/monovariant case without subeffecting.
- ▶ Behaviours: effects are not sets but sequences.
 - ▶ Effects include information on when it happened:
Communication Analysis

