Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

APA
Effects in Type Systems

Jurriaan Hage
e-mail: jur@cs.uu.nl
homepage: http://www.cs.uu.nl/people/jur/

Department of Information and Computing Sciences, Universiteit Utrecht

May 22, 2012

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.

Adding assignments and references to `Fun`

- Imperative constructs for `Fun`:

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

- **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:

$$
\begin{array}{lll}
\textbf{new}_{\mathsf{R}}\ r := 0 & & \\
\textbf{in} & \textbf{let fib} = \textbf{fun}_{\mathsf{F}}\ f\ z => \textbf{if}\ z < 3 & \\
 & & \textbf{then}\ r := !r + 1 \\
 & & \textbf{else}\ f(z-1); f(z-2) \\
 & \textbf{in fib}\ x; !r &
\end{array}
$$

- The **fib** definition assigns to and references 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.

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Annotations

- Annotations are sets of effects (three kinds):

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

- $\{!\pi\}$ 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:

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

UCAI

- ωs (sets of locations) are equal modulo UCAI.
- Values are considered the same if they only differ in order, parenthesis and the presence of unit or cancelling values because of idempotence.
- For example:
  $$(\{\pi_1\} \cup \{\pi_2\}) \cup \emptyset \stackrel{U}{=}$$
  $$\{\pi_1\} \cup \{\pi_2\} \stackrel{I}{=}$$
  $$\{\pi_1\} \cup (\{\pi_2\} \cup \{\pi_2\}) \stackrel{A}{=}$$
  $$(\{\pi_1\} \cup \{\pi_2\}) \cup \{\pi_2\} \stackrel{C}{=}$$
  $$(\{\pi_2\} \cup \{\pi_1\}) \cup \{\pi_2\}$$
- We may simply write $\{\pi_1, \pi_2\}$.

Annotated types

- Annotated types are defined to be

$$\widehat{\tau} ::= \textbf{int} \mid \textbf{bool} \mid \widehat{\tau_1} \xrightarrow{\varphi} \widehat{\tau_2} \mid \textbf{ref}_\omega \widehat{\tau}$$

- Example: $\textbf{int} \xrightarrow{\{!\mathsf{R}, \mathsf{R}{:=}\}} \textbf{int}$.

Example revisited

- $\textbf{new}_{\mathsf{R}}\ r := 0$
  $\textbf{in}\quad \textbf{let fib} = \textbf{fun}_{\mathsf{F}}\ f\ z \Rightarrow \textbf{if}\ z < 3$
  $\qquad\qquad \textbf{then}\ r := !r + 1$
  $\qquad\qquad \textbf{else}\ f(z-1); f(z-2)$
  $\quad \textbf{in fib}\ x; !r$
- A reference variable like r has type $\textbf{ref}_{\{\mathsf{R}\}}\ \textbf{int}$
- The function **fib** has type $\textbf{int} \xrightarrow{\{!\mathsf{R},\mathsf{R}:=\}} \textbf{int}$.
  - It is obviously a function from **int** to **int**.
  - Which may, as a side effect, access and update a reference created at R.

Judgments

- Judgments for Side Effect Analysis are of the form

$$\widehat{\Gamma} \vdash_{\mathrm{SE}} e : \widehat{\tau}\ \&\ \varphi$$

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

Rule for let-expressions

$$\frac{\widehat{\Gamma} \vdash_{\text{SE}} e_1 : \widehat{\tau_1} \ \& \ \varphi_1 \qquad \widehat{\Gamma}[x \mapsto \widehat{\tau_1}] \vdash_{\text{SE}} e_2 : \widehat{\tau_2} \ \& \ \varphi_2}{\widehat{\Gamma} \vdash_{\text{SE}} \textbf{let } x \ = \ e_1 \textbf{ in } e_2 : \widehat{\tau_2} \ \& \ \varphi_1 \cup \varphi_2} \ [\text{let}]$$

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

Rule for abstraction

$$\frac{\widehat{\Gamma}[x \mapsto \widehat{\tau}_x] \vdash_{\mathrm{SE}} e_0 : \widehat{\tau}_0 \ \&\ \varphi_0}{\widehat{\Gamma} \vdash_{\mathrm{SE}} \textbf{fn}_\pi\ x\ \ \texttt{=>}\ \ e_0 : \widehat{\tau}_x \xrightarrow{\varphi_0} \widehat{\tau}_0 \ \&\ \emptyset}\ [\mathsf{fn}]$$

- 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

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

- 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

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

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

Rule for dereference

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

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

Rule for new-expression

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

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

Rule for assignments

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

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

Example

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

Poisoning

- $\textbf{new}_{\mathsf{A}}\ x :=\ 1$
  $\textbf{in}\quad (\textbf{fn}\ f \Rightarrow f\ (\textbf{fn}\ y \Rightarrow !x) + f(\textbf{fn}\ z \Rightarrow (x := z; z)))$
  $\quad\quad (\textbf{fn}\ g \Rightarrow g\ 1)$
- Determine that f has the type $(\textbf{int} \overset{\{!\mathsf{A},\mathsf{A}{:=}\}}{\rightarrow} \textbf{int}) \overset{\{!\mathsf{A},\mathsf{A}{:=}\}}{\rightarrow} \textbf{int}$?

Poisoning

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

Subtyping

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

- 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

- $\textbf{new}_{\mathsf{A}}\ x := \ 1$
  $\textbf{in} \quad (\textbf{fn}\ f \Rightarrow f\ (\textbf{fn}\ y \Rightarrow\ !x) + f(\textbf{fn}\ z \Rightarrow (x := z; z)))$
  $\quad (\textbf{fn}\ g \Rightarrow g\ 1)$
- Weaken the type when necessary (when a value is "used"):

$$(\textbf{int} \xrightarrow{\{!\mathsf{A}\}} \textbf{int}) \leq (\textbf{int} \xrightarrow{\{!\mathsf{A},\mathsf{A}{:=}\}} \textbf{int})$$

$$(\textbf{int} \xrightarrow{\{\mathsf{A}{:=}\}} \textbf{int}) \leq (\textbf{int} \xrightarrow{\{!\mathsf{A},\mathsf{A}{:=}\}} \textbf{int})$$

- 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):

$$\frac{\widehat{\tau_1'} \leq \widehat{\tau_1} \qquad \widehat{\tau_2} \leq \widehat{\tau_2'} \qquad \varphi \subseteq \varphi'}{\widehat{\tau_1} \xrightarrow{\varphi} \widehat{\tau_2} \leq \widehat{\tau_1'} \xrightarrow{\varphi'} \widehat{\tau_2'}}$$

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

Contravariance example

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

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

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

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

- 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: $\mathbf{int}^{\{0\}} \to \mathbf{int}^{\{-,0\}}$ 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

    $$T\ \texttt{work}(T\ t)\ .$$

    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, $T\ \texttt{work}(T\ t)$ may be weakened to $U\ \texttt{work}(S\ t)$.

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 :: $\texttt{Eq}\ a \Rightarrow a \rightarrow a$
  - We may pass values b to f that have at least $\texttt{Eq}\ b$, so they may have also $\texttt{Ord}\ b$
  - We may write $\texttt{id}\ (f\ x)$, forgetting that a has $\texttt{Eq}\ a$
- 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