Universiteit Utrecht

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

A Generic Usage Analysis with Subeffect Qualifiers

Stefan Holdermans
(Joint work with Jurriaan Hage and Arie Middelkoop)

Dept. of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
E-mail: `stefan@cs.uu.nl`
Web pages: `http://people.cs.uu.nl/stefan/`

October 3, 2007
ICFP 2007

Introduction

Overview

We consider two types of usage analyses:

- sharing analysis,
- uniqueness typing.

We sketch a single, generic type and effect system of which both analyses are instances.

Its main features:

- subeffecting (but no subtyping!),
- polyvariance,
- subeffect qualification.

Sharing analysis

Sharing analysis identifies terms that may be used more than once.

$$\begin{array}{l}\textbf{let } x = 2 + 3 \\ \textbf{in } \ x + x\end{array} \qquad \begin{array}{l}\textbf{let } y = 2 + 3 \\ \textbf{in } \ 2 * y\end{array}$$

In a call-by-need context, x is updated after its first use to avoid re-evaluation of $2 + 3$.

But y is used only once: updating is unnecessary.

Type-based analyses store usage information in type derivations:

$\vdash x :^{\omega} \text{Int}$
$\vdash y :^{1} \text{Int}$

ω : possibly used more than once.
1 : used at most once.

Uniqueness typing

Uniqueness typing is used for maintaining
referential transparency in the presence of side effects:

It keeps track of which values are required to be passed around single-threadedly.

For instance: $putChar :^{\omega} \text{Char}^{\omega} \rightarrow (\text{World}^{1} \rightarrow \text{World}^{1})^{\omega}$

Ill-typed:

$$\lambda w.\ (putChar\ \texttt{'O'}\ w, putChar\ \texttt{'K'}\ w)$$

Well-typed:

$$\begin{array}{l} \lambda w.\ \textbf{let}\ w' = putChar\ \texttt{'O'}\ w \\ \qquad\qquad\ \ w'' = putChar\ \texttt{'K'}\ w' \\ \qquad\ \textbf{in}\ \ w'' \end{array}$$

Subeffecting

Differences between the two analyses

In sharing analysis, it's okay to bind a shared argument to a unique parameter.

$$double :^{\omega} \mathsf{Int}^{1} \to \mathsf{Int}^{\omega}$$
$$double = \lambda x.\ 2 * x$$

$$\begin{aligned} &\textbf{let } x :^{\omega} \mathsf{Int} \\ &\quad\ \ x = 2 \\ &\textbf{in } \ (double\ x) + x \end{aligned}$$

But: binding a unique argument to a shared parameter is not okay!

And here our analyses diverge . . .

Sharing analysis:

	1-param.	ω-param.
1-arg.	✓	✗
ω-arg.	✓	✓

Uniqueness typing:

	1-param.	ω-param.
1-arg.	✓	✓
ω-arg.	✗	✓

How do we prepare function arguments?

We introduce an order on annotations: $1 \sqsubset \omega$.
Arguments are now subjected to subeffecting:

For sharing analysis:

$$\frac{\vdash t :^{\varphi'} \tau \quad \vdash \varphi \sqsubseteq \varphi'}{\vdash t :^{\varphi} \tau} \quad \text{(T-SubDown)}$$

For uniqueness typing:

$$\frac{\vdash t :^{\varphi'} \tau \quad \vdash \varphi \sqsupseteq \varphi'}{\vdash t :^{\varphi} \tau} \quad \text{(T-SubUp)}$$

Abstracting over the direction of the inequality, we yield:

$$\frac{\vdash t :^{\varphi'} \tau \quad \vdash \varphi \diamond \varphi'}{\vdash t :^{\varphi} \tau} \quad \text{(T-SubGen)}$$

Note: these rules do not affect the annotations within types!
(We do not have a full subtyping relation.)

Polyvariance

Subeffecting is not enough

Subeffecting allows arguments to match parameters.
But we also need functions to adapt to calling contexts.

Therefore, types are polymorphic in their annotations:

$$
\begin{aligned}
&\mathit{double} :^{\omega} \forall \mathsf{p}\ \mathsf{q}.\ \mathsf{Int}^{\mathsf{p}} \to \mathsf{Int}^{\mathsf{q}} \\
&\mathit{double} = \lambda x.\ 2 * x
\end{aligned}
$$

Possible analyses for *double* are now:

$$
\begin{aligned}
&\mathit{double} :^{\omega} \mathsf{Int}^{1} \to \mathsf{Int}^{1} \\
&\mathit{double} :^{\omega} \mathsf{Int}^{1} \to \mathsf{Int}^{\omega} \\
&\mathit{double} :^{\omega} \mathsf{Int}^{\omega} \to \mathsf{Int}^{1} \\
&\mathit{double} :^{\omega} \mathsf{Int}^{\omega} \to \mathsf{Int}^{\omega}
\end{aligned}
$$

Subeffect qualification

Analyzing higher-order functions is more involved

$$apply = \lambda f.\ \lambda x.\ (f\ x)$$

Assume *apply* only operates on shared integer functions.

Possible sharing analyses are then:

$$apply :^{\omega} (\mathsf{Int}^{1} \to \mathsf{Int}^{r})^{\omega} \to (\mathsf{Int}^{1} \to \mathsf{Int}^{r})^{s}$$
$$apply :^{\omega} (\mathsf{Int}^{1} \to \mathsf{Int}^{r})^{\omega} \to (\mathsf{Int}^{\omega} \to \mathsf{Int}^{r})^{s}$$
$$apply :^{\omega} (\mathsf{Int}^{\omega} \to \mathsf{Int}^{r})^{\omega} \to (\mathsf{Int}^{\omega} \to \mathsf{Int}^{r})^{s}$$

But not:

$$apply :^{\omega} (\mathsf{Int}^{\omega} \to \mathsf{Int}^{r})^{\omega} \to (\mathsf{Int}^{1} \to \mathsf{Int}^{r})^{s}$$

Constraints are stored in qualified types

Qualified types:

- are a generalization of Haskell's type classes and
- provide a convenient framework for dealing with constraints in polymorphic types.

For sharing analysis, we can now write:

$$\mathit{apply} :^{\omega} \forall \mathsf{p}\ \mathsf{q}\ \mathsf{r}\ \mathsf{s}.\ (\mathsf{p} \sqsubseteq \mathsf{q}) \Rightarrow (\mathsf{Int}^{\mathsf{p}} \rightarrow \mathsf{Int}^{\mathsf{r}})^{\omega} \rightarrow (\mathsf{Int}^{\mathsf{q}} \rightarrow \mathsf{Int}^{\mathsf{r}})^{\mathsf{s}}$$

Or—generically:

$$\mathit{apply} :^{\omega} \forall \mathsf{p}\ \mathsf{q}\ \mathsf{r}\ \mathsf{s}.\ (\mathsf{p} \diamond \mathsf{q}) \Rightarrow (\mathsf{Int}^{\mathsf{p}} \rightarrow \mathsf{Int}^{\mathsf{r}})^{\omega} \rightarrow (\mathsf{Int}^{\mathsf{q}} \rightarrow \mathsf{Int}^{\mathsf{r}})^{\mathsf{s}}$$

Towards the complete picture

Dropping our previous assumptions complicates things a bit:

$$
\begin{aligned}
&\mathit{apply} :^{\omega} \forall \mathsf{a}\ \mathsf{b}\ \mathsf{p}\ \mathsf{q}\ \mathsf{r}\ \mathsf{s}.\ (\mathsf{p} \diamond \mathsf{q}, \mathsf{s} \sqsubseteq \mathsf{t}) \Rightarrow \\
&\qquad\qquad (\mathsf{a}^{\mathsf{p}} \to \mathsf{b}^{\mathsf{r}})^{\mathsf{t}} \to (\mathsf{a}^{\mathsf{q}} \to \mathsf{b}^{\mathsf{r}})^{\mathsf{s}} \\
&\mathit{apply} = \lambda f.\ \lambda x.\ (f\ x)
\end{aligned}
$$

Abstracting over types poses no problem.

Genericity does not amount to just flipping all inequalities. Containment: a value is used at least as often as the structure it is stored in.

Containment is tricky for uniqueness typing: s is pushed down by t but may be pulled up again through subeffecting.

Conclusion

Wrapping up

Summary:

- We have specified a generic usage analysis that can be instantiated to both sharing analysis and uniqueness typing.
- Subeffecting and polyvariance yield a high degree of context sensitivity.
- Qualified types make dealing with constraints straightforward.

Work in progress:

- Track zero-usage.
- Use subeffect qualification in other analyses.
- Compare subeffect qualification to other approaches.