Polyvariant Cardinality Analysis for Non-strict Higher-order Functional Languages

Brief Announcement

Hidde Verstoep Jurriaan Hage

Department of Computing and Information Sciences
Utrecht University
hiddeverstoep@gmail.com,J.Hage@uu.nl

Abstract

In this brief announcement we outline work we have done on defining a polyvariant cardinality analysis for a non-strict higher-order language equipped with user defined datatypes and explicit strictness that includes sharing analysis, absence analysis, strictness analysis and uniqueness typing as a special case. It aims to be the basis of an analysis framework in a compiler for a Haskell-like language, that does not have access to an aggressive inliner such as the one present in GHC.

Categories and Subject Descriptors D.3 [*Software*]: Applicative (functional) languages; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism

General Terms Languages, Theory, Verification

Keywords type-based program analysis, cardinality analysis, uniqueness typing, polyvariance, absence analysis, sharing analysis, usage analysis

1. Introduction

In this paper we motivate and shortly describe a type system for cardinality analysis for a non-strict, higher-order functional language: an analysis that counts how often a value may be used. Cardinality analysis is an analysis that generalizes uniqueness typing (unique values are never used more than once), sharing analysis (which values are used at most once), strictness analysis (which values are used at least once) and absence analysis (which values are not needed at all). In combining multiple analyses, our work has a similarity in spirit with work by Hage et al. [6] and Sergey et al. [10].

In the former, a usage analysis is defined, that can be instantiated to uniqueness typing and sharing analysis, making precise in which way the two differ. In this case, genericity is restricted to the *specification* of the analysis: if you want to compute both uniqueness and sharing information, two different instantiations of the generic analysis have to be run in separation. The analysis we consider in this paper is run once, and afterwards strictness, sharing and absence information can be read off (uniqueness information can also be read off, but this is more subtle). Moreover, [6] does not include absence and strictness analysis. Absence information in particular can be useful to improve also the precision for uniqueness typing and sharing analysis. Consider the following function:

$$
\begin{aligned}
g\ xs = \mathbf{let}\ & \\
& len = length\ xs \\
\mathbf{in}\ & \\
& map\ (\lambda x \rightarrow x + len)\ xs
\end{aligned}
$$

To discover that the elements xs are accessed exactly once, we need to determine that $length$ uses the spine of xs once, but does not access the elements of xs at all. Having derived for map that it uses the spine and elements of xs exactly once, we can then derive a type for g that specifies that the spine of its first argument xs is used more than once, but elements of the list are used only once.

Sergey et al. [10] integrates sharing analysis, strictness analysis and absence analysis for a higher-order language in a single analysis. Their work is tailored to the back-end of GHC, and is more akin to abstract interpretation than a type and effect system. Their analysis is not polyvariant. The rationale for a monovariant analysis is that due to GHC's aggressive inliner, not much is to be gained from complicating the analysis by introducing polyvariance. Our motivation is to help those without access to such an inliner achieve good precision; the Utrecht Haskell Compiler (UHC) [3] and Mozilla's Rust language (`www.rust-lang.org`) are typical examples.

It makes sense to want to combine analyses into a single specification, since specifications, implementations, and meta-theory can be shared. On the other hand, since our analysis is polyvariant, and includes subtyping, running it does not come cheap. But since we need to run the analysis only once to obtain the results of what otherwise would be four different analyses that are run in separation, we may hope to recoup some of this additional cost, and gain some precision as well, as our example above shows.

In the remainder of the paper we sketch the ideas behind our analysis, leaving a description of the exact details of the analysis and the solver, and an empirical evaluation of the cost and benefits of the analysis within the UHC compiler for later. For more details, the interested reader is directed to [11]. A prototype implementation of the algorithm that implements the analysis (written in Haskell), and a partial proof of the soundness of the full type system can be downloaded from `http://www.cs.uu.nl/wiki/Hage/Downloads`.

PEPM '15, January 13-14 2015, Mumbai, India.

http://dx.doi.org/10.1145/2678015.2682536

2. Approach

Sharing analysis [4, 6, 9] is an analysis that determines whether or not an expression is used at most once. Consider the expression $(\lambda x \to x+x)\ (1+1)$. In a lazy setting, the value of the expression $(1+1)$ is needed twice, but only evaluated once. After it has been evaluated, the thunk that represents the expression is "updated", overwriting the thunk by the computed value, 2, for immediate access later on. But in $(\lambda x \to x)\ (1+1)$ the value of $(1+1)$ is only used once, and the thunk update could have been omitted. To avoid the update we need to find out which values are used at most once.

In type and effect systems sharing analysis is often formulated using the lattice $1 \sqsubset \omega$. The annotation 1 says that the expression is guaranteed to be used at most once, while ω says that an expression can be used more than once. A subeffecting rule will allow the value of an expression $\mathbf{e}$ for which we have inferred annotation ω to be considered as a value that is used at most once:

$$\frac{\vdash \mathbf{e} : \tau^{\varphi'} \quad \vdash \varphi \sqsubseteq \varphi'}{\vdash \mathbf{e} : \tau^{\varphi}}$$

The thunk created for $\mathbf{e}$ may then superfluously perform an update, but that is not unsound (it won't crash the program).

Uniqueness Uniqueness analysis [1, 2, 6] determines whether values are used at most once, or more than once, just like sharing analysis. The purpose of uniqueness typing is however quite different.

Consider the function: $writeFile :: String \to File \to File$ that writes data to a file, thereby exhibiting a side effect. Then

$$\lambda f \to (writeFile\ \texttt{"O"}\ f, writeFile\ \texttt{"K"}\ f) \qquad (1)$$

can be evaluated in two ways when applied: the file may contain either `"OK"` or `"KO"`, depending on the order in which the writeFile functions are evaluated; it is obviously not a pure function.

Uniqueness typing provides us with a safe way to use functions with side effects. With uniqueness analysis, we can give $writeFile$ a type such that the uniqueness type system guarantees that it has access to an unduplicated $File$. Under that regime, expression (1) would be rejected. Uniqueness analysis can automatically annotate expressions with 1 and ω and will reject a program when this is not possible. A 1 annotation indicates that an expression must be unique. A ω annotation indicates that an expression is not necessarily unique. The subeffecting rule that should then be used is:

$$\frac{\vdash \mathbf{e} : \tau^{\varphi'} \quad \vdash \varphi' \sqsubseteq \varphi}{\vdash \mathbf{e} : \tau^{\varphi}}$$

Note that the condition on the annotation is swapped with respect to the rule for sharing analysis. The result is that an expression which is marked unique can still be used as a parameter to a function which accepts non-unique parameters.

A valid annotated type for the $writeFile$ function:

$$writeFile :: (String^{\omega} \to (File^{\mathbf{1}} \to File^{\mathbf{1}})^{\omega})^{\omega}$$

The ω annotations mean that the first parameter, the entire function and its partial applications may be shared. However, the second parameter, a $File$, must be unique, and the resulting $File$, may also not be duplicated. That is clearly not the case in expression (1). A valid expression, in which the file is used uniquely, would be:

$$\lambda f \to writeFile\ \texttt{"2"}\ (writeFile\ \texttt{"1"}\ f)$$

Uniqueness analysis is tricky. Consider the function

$$writeFile' :: (File^{\mathbf{1}} \to (String^{\omega} \to File^{\mathbf{1}})^{\mathbf{1}})^{\omega}$$

which is like $writeFile$ but with its parameters swapped. A consequence is that the partial application may no longer be shared. If allowed, the following expression (2) would become typeable, but its behavior would be like that of expression (1).

$$\lambda f \to (\lambda w \to (w\ \texttt{"1"}, w\ \texttt{"2"}))\ (writeFile'\ f) \qquad (2)$$

So, any partial application that has access to unique values must also be unique. Although this might seem to solve the problem, when the type and effect system includes subeffecting it is possible to change the annotation on $(writeFile'\ f)$ from 1 to ω. This makes expression (2) type correct, something that we do not want.

Strictness Strictness analysis [7, 12] is an analysis that determines whether expressions are used at least once. If a parameter to a function is guaranteed to be used at least once, its value could be calculated immediately instead of passing an unevaluated thunk. This leads to performance benefits: passing a value on the stack is more efficient than creating a thunk – which might even refer to more thunks – on the heap.

A possible lattice to use with strictness is: $S \sqsubset L$. The meaning of the elements is: (i) an expression is used at least once, or evaluated strictly, for S; and (ii) an expression is used any number of times, or possibly evaluated lazily, for L. Even if a function has a parameter annotated S, we want to accept an argument annotated L. The subeffecting rule that that implies is exactly the same as the one for sharing analysis.

Absence Absence analysis is an analysis that determines whether expressions are used or not. It is similar to dead or unreachable code elimination. For the standard Haskell function $const\ x\ y = x$, it is easy to see that its second argument is not used.

In human-written code, there are typically not many functions that do not use their arguments. There are however many frequently used functions that use only a part of their argument, e.g., $length$, fst and snd. So absence analysis does make sense, particularly when (user defined) datatypes are present. Other code that might benefit from this analysis is computer generated code. Computer generated code is often less "smart" and might contain a lot of dead code this analysis can detect.

2.1 Similarities

The questions that these analyses try to answer are very similar, and involve counting in some way how often values are used: (i) sharing and uniqueness analysis ask which values are used at most once, (ii) strictness analysis asks which value is used at least once, and (iii) absence analysis asks which values are not used at all.

The analyses differ only in the way subeffecting is achieved, and the difficulties that arise when uniqueness is considered in the presence of partial application. Also, uniqueness is different from all the other analyses in that it can lead to the rejection of programs; the other analyses provide information about the program for the purpose of optimization.

Our aim in the following, where we make our ideas more precise, is to compute an approximation of how often a value is used, or demanded. Essentially, the abstract values in our domain are sets over $0, 1$ and ∞. In that case, a set like $\{0\}$, describes a value that is not used, and $\{0, 1\}$ a value that can be used zero times, or once. Because language constructs differ in the way they place demands on their subexpressions, we introduce a number of abstract operators to cover the variety of ways in which they behave. For example, if in application $f\ \mathbf{e}$, both f and $\mathbf{e}$ use a given value, then $f\ \mathbf{e}$ uses the value at least twice, adding them together as it were. But if the uses of the value are found in the then-part and the else-part of a single conditional, then we should not add them together, but combine them in some other way. We shall also need additional operators to deal with partial application, and with conditional use.

3. The analysis lattice

In annotated type systems, types have annotations attached to them. The (concrete) annotations in our annotated type system are

$$\begin{array}{rcl} \varpi & ::= & \emptyset \mid \{\pi\} \mid \varpi_1 \cup \varpi_2 \\ \pi & ::= & 0 \mid 1 \mid \infty \end{array}$$

In other words, annotations are (isomorphic to) the powerset of $\{0, 1, \infty\}$. In the following we write $\bot$ for $\emptyset$, the bottom of the lattice, $\mathbf{0}$ for $\{0\}$, $\mathbf{1}$ for $\{1\}$, ω for $\{\infty\}$ and $\top$ for $\{0, 1, \infty\}$.

Since we aim for a polyvariant analysis, we add annotation variables to the mix:

$$\varphi \quad ::= \quad \beta \mid \varpi$$

For clarity, we shall distinguish between *use annotation* ν that tells us how often a value is used, and *demand annotations* δ that tell us how often a variable is demanded.

$$\begin{array}{rcl} \nu & ::= & \varphi \\ \delta & ::= & \varphi \end{array}$$

Note that even though the definitions are the same they describe different things ("use" and "demand" respectively) and are used differently in the type system.

A *type* can be a type variable α, a fully applied datatype, or a function type; as usual, we write $\overline{\varphi_l}$ for a sequence of l φs:

$$\begin{array}{rcl} \tau & ::= & \alpha \mid T\ \overline{\varphi_l}\ \overline{\tau_k} \mid \rho_\tau \to \eta_\tau \\ \eta_\mu & ::= & \mu^\nu \quad \text{where } \mu \in \{\tau, \sigma\} \\ \rho_\mu & ::= & \mu^{\nu,\delta} \quad \text{where } \mu \in \{\tau, \sigma\} \end{array}$$

Definitions η_μ and ρ_μ are indexed by μ, which can be either the symbol τ, or (defined below) the symbol σ. Since a function produces a value, we use η_τ to attach a use annotation to its result type. Because the argument is available to the body of the function as a variable we attach both use and demand annotations to the argument, and use ρ_τ for the argument. The slogan: η for use only, ρ for use and demand both. Datatypes will be explained in more detail later on.

With types now defined, we can provide the usual definitions for annotation and type polymorphic type schemes σ, and environments Γ, in which we store not only the type scheme associated with each variable, but also its use and demand. Constraints C will be defined later on.

$$\begin{array}{rcl} \sigma & ::= & \gamma \mid \forall \overline{\upsilon}.\ C \Rightarrow \tau \\ \upsilon & ::= & \alpha \mid \beta \\ \Gamma & ::= & \epsilon \mid \Gamma, x : \rho_\sigma \end{array}$$

We now define the explicitly annotated term language for our analysis. Before analysis, annotations are still unavailable; we use annotation variables to make terms valid. Terms consist of (in that order) variables, functions, applications, mutually recursive lets, *seq* $\mathbf{e_1}$ $\mathbf{e_2}$ expressions to force evaluation of $\mathbf{e_1}$ before returning the value of $\mathbf{e_2}$, constructor expressions, and case expressions:

$$\begin{array}{rcl} \mathbf{e} & ::= & x \mid \lambda^\nu\ x \to \mathbf{e} \mid \mathbf{e}\ x \\ & \mid & \mathbf{let}\ \overline{x_i =^{\nu,\delta} \mathbf{e_i}}\ \mathbf{in}\ \mathbf{e} \mid \mathit{seq}\ \mathbf{e_1}\ \mathbf{e_2} \\ & \mid & K^\nu\ \overline{x_i} \mid \mathbf{case}\ \mathbf{e}\ \mathbf{of}\ \overline{K_i\ \overline{x_{ij}} \to \mathbf{e_i}} \end{array}$$

To simplify analysis, and without loss of generality, function and constructor application can only take variables as arguments; constructors must always be fully applied.

Constraints

Many type systems encode operations on annotations in the type rules. Since our annotations are quite complex, we first define some operations on annotations using annotation operators and introduce some additional notation. This will simplify the type rules significantly.

We start out by motivating the (four) operations we shall need:

1. To perform the addition of two numbers $\mathbf{e_1} + \mathbf{e_2}$, it is clear that both expressions are needed to calculate a result. Suppose a variable x occurs in both $\mathbf{e_1}$ and $\mathbf{e_2}$, then we need a way to combine the uses of x in both branches to find the use of x in the expression $\mathbf{e_1} + \mathbf{e_2}$. The way the uses are combined must reflect that *both* expressions are used.
2. Given the expression $\mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e_1}\ \mathbf{else}\ \mathbf{e_2}$ it is clear that either $\mathbf{e_1}$ or $\mathbf{e_2}$ will be evaluated depending on $\mathbf{e}$, but certainly not both. We need a way to combine the uses associated with variables in the expressions $\mathbf{e_1}$ and $\mathbf{e_2}$ that reflects that only one of the branches will be evaluated.
3. A function $\lambda x \to \mathbf{e}$ can be applied multiple times. However, all variables occurring in $\mathbf{e}$ that are defined outside of the function are used repeatedly for every application of the function. It might even be the case that one of these variables is never used at all if the function is never applied. So, we need to be able to express repeated use.
4. Consider the expression $\mathbf{let}\ b = 0; a = b\ \mathbf{in}\ f\ a$. The value of b is only used if a is used, and a is used only if f uses its argument. So, we need to be able to express conditional use.

Now that we have a basic understanding of the operations we will be needing, we can define them as operators (Figure 1). The operator $\oplus$ expresses the combination of branches that are both used (case 1). The operator $\sqcup$ expresses the combination of branches of which only one is used (case 2). The operator $\cdot$ expresses repeated use (case 3). The formula that defines the operator can be explained as follows: for every element m in ϖ_1, for all combinations of m elements from ϖ_2, take the sum of the m elements. For example, suppose a function (which is applied twice) uses a value at most once, then the value is used: $\{\omega\} \cdot \{\mathbf{0}, \mathbf{1}\} = \{\mathbf{0}+\mathbf{0}, \mathbf{0}+\mathbf{1}, \mathbf{1}+\mathbf{0}, \mathbf{1}+\mathbf{1}\} = \{\mathbf{0}, \mathbf{1}, \omega\}$. This is as it should be: neither call may use the value ($\mathbf{0}$), one call uses it, but the other does not ($\mathbf{1}$), or both may use it (ω). The operator $\triangleright$ expresses conditional use (case 4). The result of this operator is equal to ϖ_2 unless ϖ_1 includes 0.

$$\begin{array}{rcl} \varpi_1 \oplus \varpi_2 & ::= & \{m + n \mid m \in \varpi_1, n \in \varpi_2\} \\ \varpi_1 \sqcup \varpi_2 & ::= & \varpi_1 \cup \varpi_2 \\ \varpi_1 \cdot \varpi_2 & ::= & \{\sum_{i=1}^{min(m,2)} n_i \mid m \in \varpi_1, \forall i.n_i \in \varpi_2\} \\ \varpi_1 \triangleright \varpi_2 & ::= & \bigcup_{m \in \varpi_1} (m \equiv 0\ ?\ \mathbf{0} : \varpi_2) \end{array}$$

Figure 1. Annotation value operators

Defining the operators for annotation values ϖ is relatively easy since they do not include variables. However, we want to lift the operators to annotations, types, type schemes and environments and these can include annotation variables. An operator requires two concrete annotation values to directly calculate a result. Since we might encounter annotation variables we cannot assume this is always the case. The usual solution, and one we also use, is to employ constraints to keep track of the relations between annotations (whether they are variables or values).

Figure 3 contains an overview of all different kinds of constraints. A constraint C can be an equality, $\oplus$, $\sqcup$, $\cdot$ or $\triangleright$ constraint for annotations, types and type schemes. We also need instantiation and generalization constraints to deal with polyvariance and polymorphism. We forego the definitions that rephrase our operators in terms of constraints, for annotations, types, type schemes, and type environments.

$$\frac{\Gamma_1 \vdash_{\sqsubseteq} \mathbf{e} : (\eta_2{}^{\delta_2} \to \eta_3)^{\mathbf{1}} \rightsquigarrow C_1 \qquad x : \eta_4{}^{\mathbf{1}} \vdash_{\diamond} x : \eta_2 \rightsquigarrow C_2 \qquad \Gamma_1 \oplus x : \eta_4{}^{\delta_2} = \Gamma_2 \rightsquigarrow C_3}{\Gamma_2 \vdash \mathbf{e}\ x : \eta_3 \rightsquigarrow C_1 \cup C_2 \cup C_3}\ \textsc{App}$$

$$\frac{\begin{array}{c}\mathbf{data}\ T\ \overline{u_l}\ \overline{\alpha_k} = \overline{K_i\ \overline{\rho_{ij}}} \qquad \tau_{ij}{}^{\nu_{ij},\delta_{ij}} = \rho_{ij}\ [\overline{\varphi_l}/_{\overline{u_l}}, \overline{\tau_k}/_{\overline{\alpha_k}}] \\ \Gamma_0 \vdash_{\sqsubseteq} \mathbf{e} : (T\ \overline{\varphi_l}\ \overline{\tau_k})^{\mathbf{1}} \rightsquigarrow C_1 \qquad \overline{\Gamma_i, x_{ij} : (\forall\, \emptyset\ .\ \emptyset \Rightarrow \tau_{ij})^{\nu_{ij},\delta_{ij}} \vdash \mathbf{e_i} : \eta \rightsquigarrow C_2} \qquad \Gamma_0 \oplus (\bigsqcup_i \Gamma_i) = \Gamma \rightsquigarrow C_3\end{array}}{\Gamma \vdash \mathbf{case}\ \mathbf{e}\ \mathbf{of}\ \overline{K_i\ \overline{x_{ij}} \to \mathbf{e_i}} : \eta \rightsquigarrow C_1 \cup C_2 \cup C_3}\ \textsc{Case}$$

Figure 2. A few rules of the static semantics

$$\begin{array}{rcl} C & ::= & \emptyset \mid \varphi_1 \equiv \varphi_2 \mid \varphi \equiv \varphi_1 \oplus \varphi_2 \mid \varphi \equiv \varphi_1 \sqcup \varphi_2 \\ & \mid & \varphi \equiv \varphi_1 \cdot \varphi_2 \mid \varphi \equiv \varphi_1 \rhd \varphi_2 \mid \tau_1 \equiv \tau_2 \\ & \mid & \tau \equiv \tau_1 \oplus \tau_2 \mid \tau \equiv \tau_1 \sqcup \tau_2 \mid \tau \equiv \varphi_1 \cdot \tau_2 \\ & \mid & \tau \equiv \varphi_1 \rhd \tau_2 \mid \sigma_1 \equiv \sigma_2 \mid \sigma \equiv \sigma_1 \oplus \sigma_2 \\ & \mid & \sigma \equiv \sigma_1 \sqcup \sigma_2 \mid \sigma \equiv \varphi_1 \cdot \sigma_2 \mid \sigma \equiv \varphi_1 \rhd \sigma_2 \\ & \mid & \mathit{inst}\ (\sigma) \equiv \tau \mid \mathit{gen}\ (\rho_\tau, C, \Gamma) \equiv \rho_\sigma \mid C_1 \cup C_2 \end{array}$$

Figure 3. Constraints

Usually, algebraic datatypes are defined like so:

$$\mathbf{data}\ T\ \overline{\alpha} = \overline{K_i\ \overline{\tau_{ij}}},$$

where T is the name of the datatype, $\overline{\alpha}$ is a sequence of mutually distinct type parameters, $\overline{K_i}$ are the constructors and each τ_{ij} refers to the unannotated type of a field of a constructor. To be able to express for the example in the introduction that the spine of the list can be used at most once, while the elements are not used at all, we have to annotate the elements and the spine of the list in separation. This leads to reserving extra space for annotation variables in datatype definitions: $\mathbf{data}\ T\ \overline{\beta}\ \overline{\alpha} = \overline{K_i\ \overline{\tau_{ij}}}$ The problem of deciding what to annotate and to which extent is a hard one; [12] provides a detailed account.

4. A fragment of the type system

In this section we discuss a few of the type rules that make up our type system. The main judgement is $\Gamma \vdash \mathbf{e} : \eta_\tau \rightsquigarrow C$, under environment Γ, $\mathbf{e}$ has demand η_τ, subjected to the constraints that remain in C.

The remaining rules, and the constraint solver to solve the generated constraints can be found in [11].

The App-rule ensures the function and argument are type correct. The function is used exactly once due to the application. However, since the function may be shared it is important to apply subeffecting here. Since the use should include one, we fix the way of subeffecting to $\sqsubseteq$. When typing the argument, which is a variable by definition, the variable will be found to be demanded exactly once. However, the function specifies (using δ_2) how often it demands an argument, so we set the demand to δ_2 when creating the result environment. This is also the only place where it is necessary to use the analysis dependent subeffecting parameter $\diamond$.

The Case-rule starts with creating a fresh instance of the annotated datatype, similar to the Con-rule. We make sure to type the expression we match on with use $\mathbf{1}$ and subeffecting set to $\sqsubseteq$, similar to the function expression in the App-rule. The result type is the same as that of each of the case-arms. The result environment is built by adding Γ_0 to the combined environment of all the arms, Γ_i. Since only one of the case-arms will be executed combining is done using the $\sqcup$ operator.

5. Conclusion and Future Work

We have sketched a polyvariant cardinality analysis for a non-strict higher-order functional language, that includes as a special case absence analysis, sharing analysis, strictness analysis and uniqueness typing.

In the short term, we shall build the analysis into the UHC compiler so that we can experiment with its effectiveness [3]. This also paves the way for introducing the heap-recycling directives of [5] for writing in-place algorithms in a pure setting; these heavily depend on a uniqueness type system. Other tasks for the future are to complete our soundness proof in Coq, and to experiment with formulation that supports higher-ranked polyvariance [8].

References

[1] E. Barendsen, S. Smetsers, et al. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6(6):579–612, 1996.

[2] E. De Vries, R. Plasmeijer, and D. Abrahamson. Uniqueness typing simplified. *Implementation and Application of Functional Languages*, pages 201–218, 2008.

[3] A. Dijkstra, J. Fokker, and S. D. Swierstra. The architecture of the Utrecht Haskell Compiler. In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, pages 93–104, New York, NY, USA, 2009. ACM.

[4] J. Gustavsson. *A type based sharing analysis for update avoidance and optimisation*, volume 34. ACM, 1998.

[5] J. Hage and S. Holdermans. Heap recycling for lazy languages. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 189–197. ACM, 2008.

[6] J. Hage, S. Holdermans, and A. Middelkoop. A generic usage analysis with subeffect qualifiers. In *ACM SIGPLAN Notices*, volume 42, pages 235–246. ACM, 2007.

[7] S. Holdermans and J. Hage. Making stricterness more relevant. In *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 121–130. ACM, 2010.

[8] S. Holdermans and J. Hage. Polyvariant flow analysis with higher-ranked polymorphic types and higher-order effect operators. In *Proceedings of the 15th ACM SIGPLAN 2010 International Conference on Functional Programming (ICFP '10)*, pages 63–74. ACM Press, 2010.

[9] J. Launchbury, A. Gill, J. Hughes, S. Marlow, S. Jones, and P. Wadler. Avoiding unnecessary updates. *Functional Programming, Glasgow*, pages 144–153, 1992.

[10] I. Sergey, D. Vytiniotis, and S. Peyton Jones. Modular, higher-order cardinality analysis in theory and practice. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 335–347, New York, NY, USA, 2014. ACM.

[11] H. Verstoep. Counting analyses, 2013. MSc thesis, `http://www.cs.uu.nl/wiki/Hage/CountingAnalyses`.

[12] K. Wansbrough. *Simple polymorphic usage analysis*. PhD thesis, University of Cambridge, 2002.