Static Analysis of the Frequency of Change

Peter Thiemann
thiemann@informatik.uni-freiburg.de

Abstract: A frequency analysis assigns to each program value an upper bound on its change frequency. We define such an analysis and prove its correctness with respect to a denotational semantics of a tiny web programming language. We sketch its use for specializing web pages.

1 Introduction

A web site with dynamic content must strike a balance between the update frequency of the content, the timeliness of the actually displayed material, and the load sustainable by the underlying application server. Typically, this balance is maintained either manually or with a dedicated content management system [Ok02]. In both approaches, the current state of the content is sampled at regular intervals and the resulting static web pages are stored on a standard web server. This procedure reduces the load of the application server and increases the effectiveness of web caching mechanisms since the latter are much better suited for static pages than for dynamic ones [BO00]. Care has to be taken during the sampling process that intrinsically dynamic content, which depends on user input, is still delivered through some dynamic execution machinery.

The frequency analysis proposed in this work aims at formalizing and automating the sampling process. Given the change frequency of a value and an up-to-dateness factor for the displayed material, a sampling frequency can be determined such that all displayed material is sufficiently timely. However, high sampling frequencies are not sensible because each sampling run produces extra load. In addition, the utility of document caching is reduced because the sampled documents expire too quickly.

Hence, we take a different approach and compute from a reasonable sampling frequency, an up-to-dateness factor, and the sources of intrinsically dynamic values a traditional binding-time division from the results of the frequency analysis. Such a binding-time division annotates each program value with a binding time: either the value is statically known or it is unknown (dynamic). Then, the sampling procedure reduces to classical program specialization extended with a backend that turns the specialized web programs into a network of static and dynamic web pages.

Due to space constraints, the present paper only covers the frequency analysis, proves its correctness, and sketches the translation to binding times. The specialization algorithm and the backend are not covered.

```
main () =
  Let today = getDate () in
  Show <html><head><title>Greeting</title></head>
         <body><p>Today is <%= today %>
                  <submit action=<% daytime (today) %> /></p>
               <p>Enter your name <input name="who" />
                  <submit action=<% greet %> parm="who" /></p>
         </body>
       </html>

daytime (date) () =
  Let currentTime = getTime () in
  Let what = greetingPhrase (currentTime) in
  Show <html><head><title>Daytime</title></head>
         <body>It's <%= what %> of <%= date %>!
         </body>
       </html>

greet (who) =
  Show <html><head><title>Greeting</title></head>
        <body>Hello, <%= who %>!
        </body>
      </html>
```

Figure 1: Example application

In Sec. 2 we give an example of the intended working of the entire translation scheme. Sec. 3 defines an abstract core language for web programming and Sec. 4 defines its denotational semantics. Sec. 5 defines precisely what we mean with timeliness and change frequency of a value. Sec. 6 presents the frequency analysis phrased as an annotated type system and Sec. 7 proves its soundness. Sec. 8 discusses related work and Sec. 9 concludes.

2 Specialization of a Web Application

This section presents a small example application that benefits from specialization. The language we use in this section is an instantiation of the λWEB calculus which is formally introduced in Sec. 3. The syntax is inspired by PHP [PHP03], JSP[PLC99], and bigwig[BMS02] and should be readable without further explanation.

The example application in Fig. 1 consists of three pages corresponding to the functions `main`, `daytime`, and `greeting`. Each function ends in a `Show` statement that terminates execution by displaying a page constructed from XHTML fragments and computed values (assuming a dynamic type discipline with automatic type conversions). The values are inserted into the generated XHTML using JSP's scriptlet notation.

All pages are dynamic to some degree because they contain computed content. However, a closer look reveals that the page generated by the `main` function only changes once a

Specialization with respect to

```
today = "May 3, 2004";  currentTime = "12:00";  what = "afternoon"

main () =
  Show <html><head><title>Greeting</title></head>
         <body><p>Today is May 3, 2004
                  <submit action=<% daytime_May_3_2004 %> /></p>
               <p>Enter your name <input name="who" />
                  <submit action=<% greet %> parm="who" /></p>
         </body>
       </html>

daytime_May_3_2004 () =
  Show <html><head><title>Daytime</title></head>
         <body>It’s afternoon of May 3, 2004!
         </body>
       </html>

greet (who) =
  Show <html><head><title>Greeting</title></head>
         <body>Hello, <%= who %>!
         </body>
       </html>
```

Figure 2: Example application, sampled at noon on May 3, 2004

day and the page corresponding to the `daytime` function changes perhaps four times per day. The only genuinely dynamic page is `greet` because it depends on user input to the previous page.

If we suppose that pages are regenerated once per day (preferably shortly after midnight), then the `main` page may be static while the others remain dynamic. If the regeneration frequency is higher than four times per day, then the `daytime` page becomes static, too. However, the `greet` page will never become static regardless of the regeneration frequency.

Hence, a web site sampling tool should take a description of a web site in the form of a program such as the above, for each page an update frequency (how quickly does the information in this page change), a sampling frequency (how often are pages regenerated), and an up-to-dateness factor. The last factor is the probability that a delivered page contains up-to-date information. The tool should proceed by determining from this information which pages may become static in the sample. Finally, it creates a correctly linked sample by specializing the script starting from the `main` function.

Figure 2 shows a sample which has been specialized as outlined above. In the final step, a compiler translates the sample into a collection of interlinked static web pages and, say, CGI scripts. The result of this tedious but straightforward step is omitted.

$$\begin{array}{rcl@{\qquad\qquad}rcl}
e & ::= & \texttt{Let}\ d\ \texttt{in}\ e & d & ::= & x = c \\
& | & \texttt{Show}\ x & & | & x = p(x\ldots) \\
& | & \texttt{If}\ x\ \texttt{then}\ e\ \texttt{else}\ e & & | & \texttt{rec}\ x(x\ldots) = e \\
& | & x(x\ldots) & & &
\end{array}$$

Figure 3: Syntax of λWEB

3 The λWEB Calculus

Many languages are deemed suitable for programming web applications. Some offer special support for creating and manipulating HTML or XML documents. Others offer a plethora of APIs for connecting to external information sources, synchronizing processes, and session management. Since the present paper is not advocating one language over another, it presents the essential techniques in terms of an abstract formal calculus that models common properties of all web programming languages. The calculus abstracts over the mentioned APIs and the generation of documents so that most web programming languages can be translated to an instance of the calculus.

To simplify the presentation, the calculus λWEB defined in Fig. 3 is an intermediate language which is obtained from a source language by a standard transformation. In particular, the XHTML fragments are translated to document constructor functions. λWEB has two syntactic categories, expressions e and declarations d. Essentially, an expression is a list of `let` declarations that ends either with `Show` x, a conditional, or a function invocation. The expression `Show` x stops execution and yields the final result x. The result must be a document suitable for display on a web browser. The conditional works as usual. All functions are tail recursive so that invocations do not return.

Each kind of declaration defines a new variable and its value. The value may be a constant, c, the result of running a primitive operation, p, or a recursively defined function. Primitive operations are expected to have unspecified side effects, *e.g.*, they may perform database operations. A recursive function is defined by formal parameter list and a body expression.

Besides basic types like integers and strings, λWEB has an abstract type `DOC` for documents. Hence, λWEB may be instantiated with an arbitrary format: HTML, PDF, plain text, etc. The operations on the type `DOC` are supposed to be free of side effects. The following primitive operations form the API for `DOC`.

empty	:	`DOC`	the empty document
$\_ + \_$	:	$(\texttt{DOC}, \texttt{DOC}) \rightarrow \texttt{DOC}$	concatenation of documents
link(_)	:	$\texttt{CONT} \rightarrow \texttt{DOC}$	create a link
value(_)	:	$B \rightarrow \texttt{DOC}$	convert a base-type value to a document

The interface abstracts from all layout considerations but allows to keep track of the dependencies of the documents from computed values (`VAL` ranges over basic type values) and of the links to other documents. A link is given by a value of type `CONT` where `CONT` is the function type `VAL` $\rightarrow$ `DOC`. The intended semantics is that clicking the link calls the

$$\begin{array}{lcl}
\mathcal{D}[\![x = c]\!]\sigma t & = & \sigma[x \mapsto [\![c]\!]] \\
\mathcal{D}[\![x = p(x_1 \ldots)]\!]\sigma t & = & \sigma[x \mapsto [\![p]\!](\sigma(x_1)\ldots)t] \\
\mathcal{D}[\![\texttt{rec}\ x(x_1 \ldots) = e]\!]\sigma t & = & \sigma[x \mapsto \mathit{fix}\lambda f.\lambda(y_1 \ldots).\mathcal{E}[\![e]\!]\sigma[x \mapsto f, x_i \mapsto y_i]] \\
\mathcal{E}[\![\texttt{Let}\ d\ \texttt{in}\ e]\!]\sigma t & = & \mathcal{E}[\![e]\!](\mathcal{D}[\![d]\!]\sigma t)t \\
\mathcal{E}[\![\texttt{Show}\ x]\!]\sigma t & = & \sigma(x) \\
\mathcal{E}[\![\texttt{If}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!]\sigma t & = & \mathit{if}\ \sigma(x)\ \mathit{then}\ \mathcal{E}[\![e_1]\!]\sigma t\ \mathit{else}\ \mathcal{E}[\![e_2]\!]\sigma t \\
\mathcal{E}[\![x(x_1 \ldots)]\!]\sigma t & = & \sigma(x)(\sigma(x_1)\ldots)t
\end{array}$$

Figure 4: Semantic equations

function with the user's inputs into the document as parameter.

The concrete example in Fig. 1 uses the scriptlet notation `<%...%>` for the function *link*(_) and the notation `<%=...%>` for embedding a value in the document by *value*(_). Concatenation is implicit in the XHTML notation.

4 Denotational Semantics of λWEB

The semantics of λWEB in this paper is special because its results are time dependent. Hence, the denotation of an expression is drawn from *Comp*, *i.e.*, a function from the current time to a value.[1]

$$\begin{array}{rcl}
\mathit{Val} & = & \mathit{Const} + \mathit{DOC} + \mathit{Fun} \\
\mathit{Comp} & = & \mathit{Time} \hookrightarrow \mathit{Val} \\
\mathit{Fun} & = & \mathit{Val}^* \hookrightarrow \mathit{Comp} \\
\mathit{Env} & = & \mathit{Var} \hookrightarrow \mathit{Val}
\end{array}$$

where *Const* is the set of interpretations of constants, c, *DOC* is the set of interpretations of documents, and *Time* is the set of real numbers. The operator $+$ stands for disjoint union, $\hookrightarrow$ for the partial function space, and X^* for $1 + X + X \times X + \ldots$. Hence, *Comp* is a pointed CPO as required for a denotational semantics.

The semantic equations in Fig. 4 define two functions

$$\begin{array}{lcl}
\mathcal{D} & : & \mathit{Decl} \to \mathit{Env} \to \mathit{Time} \hookrightarrow \mathit{Env} \\
\mathcal{E} & : & \mathit{Exp} \to \mathit{Env} \to \mathit{Comp}
\end{array}$$

where $\mathcal{D}$ transforms an environment according to a declaration and the current time and $\mathcal{E}$ computes the final value of an expression. The straightforward definition relies on predefined maps $[\![c]\!]$ and $[\![p]\!]$ that map a constant to its denotation and the name of a primitive operation to a function that takes a tuple of base type arguments and returns a time dependent value $\in \mathit{Comp}$.

[1]That is, we are describing a monadic semantics for the reader monad $M(x) = \mathit{Time} \to x$.

5 Timeliness

Since timeliness is a soft concept, we first need to define formally what it means for a document or more generally for a value to be timely.

Definition 1 Let $v = v(t)$ be a time dependent value.

The *update frequency* is the average number of changes of v per unit of time.

$$f_v = \lim_{t \to \infty} \frac{|\{t_0 \mid 0 < t_0 < t, (\exists \delta > 0)\ (\forall \epsilon < \delta)\ v(t_0 - \epsilon) \neq v(t_0 + \epsilon)\}|}{t}$$

The special value $f_v = \infty$ denotes that v changes continually.

The *sampling frequency* g is the reciprocal of the time span between two snapshots of the value. The *up-to-dateness factor* $u_v = f_v/g$ measures the minimum number of samples taken per update.

The up-to-dateness factor must be understood with a grain of salt. Even $u_v = 1$ may mean that the sampled value $v_s(t)$ is almost always different from the value $v(t)$. In the worst case, the probability that $v_s(t) = v(t)$ is $p = 1 - 1/u_v$, provided that $u_v \geq 1$.

In the typical setup, the update frequency f_v is available through estimate, measurement, or analysis and the desired freshness is given as the probability p as defined above. From these numbers, the sampling frequency may be computed as

$$g = f_v/u_v = f_v/(1/(1-p)) = f_v(1-p). \tag{1}$$

The sampling frequency computed according to that formula will usually be too high to be practical. However, we never intended to take an entirely static sample of the system. Instead, the goal is to produce a mixture of static and dynamic documents. Hence, we pick an acceptable sampling frequency g_0 and solve the formula (1) for $f_0 = g_0/(1-p)$. The resulting threshold frequency f_0 is the maximum update frequency for a value that can be considered static in the generation run.

The above consideration paves the way for computing a classical binding time from the update frequency of a value and the threshold frequency. A classical binding time distinguishes between static and dynamic values, indicated by S and D.

$$BT(f_v, f_0) = \begin{cases} S & \text{if } f_v \leq f_0 \\ D & \text{otherwise.} \end{cases} \tag{2}$$

Classical binding time information can be used to drive specialization algorithms in a well-understood way [JGS93]. One successful approach is to annotate each operation with its binding time and then specialize a program by using an interpreter that executes all operations annotated as static and generates specialized code for all operations annotated as dynamic.

$$\textit{(const)}\quad \Gamma \vdash x =^0 c \Rightarrow \Gamma(x : (B_c, 0))$$

$$\textit{(prim)}\quad \frac{\Gamma(x_i) = (B_i, \phi_i) \quad \phi = \phi_0 + \phi_1 + \ldots + \phi_n \quad \Sigma(p) = (B_1, \ldots, B_n) \to B}{\Gamma \vdash x =^\phi p^{\phi_0}(x_1, \ldots, x_n) \Rightarrow \Gamma(x : (B, \phi))}$$

$$\textit{(rec)}\quad \frac{\Gamma(x : ((\rho_1, \ldots, \rho_n) \to \phi, 0))(x_1 : \rho_1) \ldots (x_n : \rho_n) \vdash e : \phi}{\Gamma \vdash \texttt{rec}\ x(x_1, \ldots, x_n) = e \Rightarrow \Gamma(x : ((\rho_1, \ldots, \rho_n) \to \phi, 0))}$$

$$\textit{(let)}\quad \frac{\Gamma \vdash d \Rightarrow \Gamma' \qquad \Gamma' \vdash e : \phi}{\Gamma \vdash \texttt{Let}\ d\ \texttt{in}\ e : \phi} \qquad \textit{(show)}\quad \frac{\Gamma(x) = (\texttt{DOC}, \phi)}{\Gamma \vdash \texttt{Show}^\phi\ x : \phi}$$

$$\textit{(if)}\quad \frac{\Gamma(x) = (\texttt{Bool}, \phi) \qquad \Gamma \vdash e_1 : \phi' \qquad \Gamma \vdash e_2 : \phi'}{\Gamma \vdash \texttt{If}^\phi\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \phi + \phi'}$$

$$\textit{(call)}\quad \frac{\Gamma(x) = ((\rho_1, \ldots, \rho_n) \to \phi, 0) \qquad \Gamma(x_i) = \rho_i}{\Gamma \vdash x(x_1, \ldots, x_n) : \phi}$$

$$\textit{(sub)}\quad \frac{\Gamma \vdash e : \phi \qquad \phi \leq \phi'}{\Gamma \vdash e : \phi'}$$

Figure 5: Frequency Analysis

6 Frequency Analysis

The classical way of analyzing binding times is not appropriate for our task because it only distinguishes the two discrete binding times S and D, which correspond to frequencies 0 and ∞. Instead, we first perform a frequency analysis (the continuous cousin of binding-time analysis) and map the results to binding times using the function BT later on.

The first question for the frequency analysis is: Where do frequencies other than 0 and ∞ come from? In λWEB, those frequencies come from primitive operations that observe a changing global state. These operations may depend on the current time and date, they may be queries against databases, or they may be other operations that depend on the current state of the machine or the network. We assume that each such operation is annotated with an update frequency, which indicates the desired granularity of the observation of changes of the underlying state. Side-effecting operations that change the underlying state must have an update frequency of ∞ to ensure that they are always executed.

Figure 5 contains the definition of a suitable frequency analysis in terms of an annotated type system. For simplicity, the type system is based on simple types. An extension with polymorphism would be useful and would follow the path outlined elsewhere [HT04]. The analysis annotates declarations and expressions with frequencies for the sake of the succeding specialization phase.

The type language of the system is given by the grammar

$$\rho ::= (\tau, \phi) \qquad \tau ::= B \mid (\rho, \ldots, \rho) \to \phi \qquad B ::= \texttt{Bool} \mid \texttt{DOC} \mid \ldots \tag{3}$$

where ρ ranges over annotated types, which are pairs of a raw type and an update frequency ϕ, B ranges over base types, and τ is either a base type B or a function that takes as arguments a tuple of values of annotated type and terminates with a value of frequency ϕ. A separate type assignment Σ maps each name p of a primitive operation to a pair $(B_1, \dots B_n) \to B$ where the list $B_1, \dots, B_n$ determines the argument base types and B is the result base type. Type assignments are formed according to the grammar $\Gamma ::= \cdot \mid \Gamma(x : \rho)$ and are considered as finite functions.

The typing rules define two judgements, $\Gamma \vdash d \Rightarrow \Gamma'$, where declaration d transforms type assignment Γ to Γ', and $\Gamma \vdash e : \phi$, where expression e delivers a final result of frequency ϕ under type assignment Γ.

The rule *(const)* determines the base type of a constant using function *TypeOf*(_). Since constants are static, the frequency annotation is 0.

The rule *(prim)* ensures that the argument types and the result type of primitive operation p correspond to p's declaration in Σ. It computes the frequency of the result by taking the sum of the frequencies of the argument values and ϕ_0, the frequency assigned by the user to this occurrence of p. The addition yields an upper bound of the actual frequency because a value v_i at frequency ϕ_i has a number of changes proportional to ϕ_i during a sufficiently large time interval T. In the absence of further information about the values and assuming that the value of an operation p depends on all arguments and on the implicit state v_0, the number of changes of $v = p(v_1, \dots, v_n)$ during T is proportional to a number smaller than $\phi_0 + \phi_1 + \dots + \phi_n$. The actual frequency of v can be much smaller (even 0), for example, if the values v_i change in lockstep and/or the frequencies are multiples of each other. Since our model does not include such dependencies between values, our typing rule must assume the worst.

The rule *(rec)* types the declaration of recursive functions. All functions are statically present in the program, hence the frequency of a function value is 0. Since functions do not return, the system need not deal with return types.

The rule *(let)* just augments the type assignment according to the declaration and types the body. The rule *(show)* attaches the frequency of the displayed document to the occurrence of `Show` in the program. The rule *(if)* is standard: the frequency of execution of the conditional depends solely on the frequency of the condition itself. The rules *(call)* and *(sub)* are standard rules for function call and subsumption of frequencies: if a value changes at frequency ϕ it is also appropriate to view it as changing at any higher frequency ϕ'.

7 Soundness of the Analysis

This section shows that the analysis is sound with respect to the semantics given in Sec. 4. This requires to define a semantics of annotated types, to define relations between value environments and type environments, and finally to prove that the semantic equations preserve those relations.

The semantics of an annotated type is a set of time dependent values. The semantics is approximative in the sense that all frequencies are considered as upper bounds. We will see below that this approximation is unavoidable. The semantics of unannotated types is defined in the usual way. For functions, the interesting part is that whenever the frequency of the arguments conforms to their type, then so does the frequency of the result.

$$\begin{array}{lcl} [\![(\tau, \phi)]\!] & = & \{v \in \mathit{Comp} \mid f_v \leq \phi, (\forall t \in \mathit{Time})\ v(t) \in [\![\tau]\!]\} \\ [\![B]\!] & = & \mathit{Const} + \mathit{DOC} + \ldots \\ [\![(\rho_1, \ldots, \rho_n) \to \phi]\!] & = & \{g \in \mathit{Fun} \mid (\forall v_i \in [\![\rho_i]\!])\ f_{\lambda t.g(v_1(t)\ldots)t} \leq \phi\} \end{array}$$

Type environments relate variable names to annotated types whereas value environments map variable names to values. These two concepts cannot be related directly because value environments are constants. Hence, we relate type environments to value environments abstracted over time.

Definition 2 Let $S \in \mathit{Time} \to \mathit{Env}$ be a time dependent environment and Γ a type environment. $S \models \Gamma$ if $\forall (x : \rho) \in \Gamma$ the function $\lambda t.R(t)(x) \in [\![\rho]\!]$.

Thus armed, we can state and prove the soundness of the declaration transformation $\mathcal{D}$ and of the evaluation semantics $\mathcal{E}$. The proof of these two statements is by simultaneous induction because $\mathcal{D}$ is defined in terms of $\mathcal{E}$ and vice versa.

Theorem 1 *Let* $S \models \Gamma$.

1. *Suppose that* $\Gamma \vdash d \Rightarrow \Gamma'$ *and* $S'(t) = \mathcal{D}[\![d]\!](S(t))t$. *Then* $S' \models \Gamma'$.
2. *Suppose that* $\Gamma \vdash e : \phi$. *Then* $\lambda t.\mathcal{E}[\![e]\!](S(t))t \in [\![(DOC, \phi)]\!]$.

The main point of the proof is the justification of the addition of frequencies in the case of a primitive operation as outlined in the explanation of the typing rule *(prim)*.

8 Related Work

There is no direct precedent to the analysis reported in this work. However, a few papers have topics which come close and this section attempts to distinguish our work from theirs.

Ramalingam [Ra96] suggests that data flow analysis should be augmented with frequency information. His frequency information refines the typical Maybe/No answer of a program analysis by instead computing a probability for the answer. In contrast, our frequencies are not probabilities but approximations of the actual rate of change.

Ball [Ba99] introduces a frequency spectrum analysis for exploring the structure of programs. His analysis is dynamic and based on actual runtime counts, whereas ours is a static analysis.

Wu and Larus [WL94] have a framework for estimating the execution frequency of portions of a program by statically predicting both branch frequencies and a program profile. A similar framework is put forward by Wagner et al [WMGH94]. In contrast, our analysis approximates the frequency of change of data.

9 Conclusion

The present paper introduces frequency analysis as a generalization of binding-time analysis. The results of the analysis enable the generation of a collection of partially static web pages from a completely dynamic web site. This partial specialization is desirable because it reduces the load of the application server and enhances the usefulness of caching on proxy servers and in web browsers. It thus opens a new application area for program specialization.

Further work includes the formalization and correctness proof of the timely specialization. We are also exploring several implementation strategies as well as different points of view on the frequency of change of primitive operations.

References

[Ba99] Ball, T.: The concept of dynamic analysis. In: *Proceedings of the 7th European engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*. S. 216–234. Springer-Verlag. 1999.

[BMS02] Brabrand, C., Møller, A., und Schwartzbach, M.: The `<bigwig>` Project. *ACM Transactions on Internet Technology*. 2(2):79–114. 2002.

[BO00] Barish, G. und Obraczka, K.: World Wide Web caching: Trends and techniques. *IEEE Communications Magazine Internet Technology Series*. May 2000.

[HT04] Helsen, S. und Thiemann, P.: Polymorphic specialization for ML. *ACM Transactions on Programming Languages and Systems*. July 2004. To appear.

[JGS93] Jones, N., Gomard, C., und Sestoft, P.: *Partial Evaluation and Automatic Program Generation*. Prentice-Hall. 1993.

[Ok02] Okunev, V. Publish event-driven web content with jsp custom tags. `http://www.javaworld.com/javaworld/jw-04-2002/jw-0419-event.html`. April 2002.

[PHP03] PHP: Hypertext processor. `http://www.php.net/`. February 2003.

[PLC99] Peligrí-Llopart, E. und Cable, L. Java Server Pages Specification. `http://java.sun.com/products/jsp/index.html`. 1999.

[Ra96] Ramalingam, G.: Data flow frequency analysis. In: *Proc. of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*. S. 267–277. Philadelphia, PA, USA. May 1996. ACM Press.

[WL94] Wu, Y. und Larus, J. R.: Static branch frequency and program profile analysis. In: *Proceedings of the 27th annual international symposium on Microarchitecture*. S. 1–11. ACM Press. 1994.

[WMGH94] Wagner, T. A., Maverick, V., Graham, S. L., und Harrison, M. A.: Accurate static estimators for program optimization. In: *Proc. of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*. S. 85–96. Orlando, Fla, USA. June 1994. ACM Press.