Dynamic Cheap Eagerness

Karl-Filip Faxén

Dept. of Microelectronics and Information Technology, Royal Institute of Technology, Electrum 229, S-164 40 Kista, tel: +46 8 790 41 20
kff@it.kth.se

Abstract. Dynamic cheap eagerness extends cheap eagerness by allowing the decision of whether to build a thunk or speculatively evaluate its body to be deferred until run time. We have implemented this optimisation in a compiler for a simple functional language and measured its effect on a few benchmarks. It turns out that a large part of the overhead of graph reduction can be eliminated, but that run-times and instruction counts are not affected in the same degree.

1 Introduction

Cheap eagerness is an optimization, applicable to nonstrict functional languages, where expressions are evaluated before it is known that their values will actually be used. For instance, an argument in a function application may be evaluated before the call, even if it is not known that the function always uses that argument. This improves performance by eliminating a lot of book-keeping necessary for delaying and (often) later resuming the evaluation of expressions.

In order to preserve the meaning of the program, only expressions which terminate without yielding run-time errors can be evaluated speculatively. In general, global analysis must be used to find such expressions.

We have previously studied this problem in [Fax00], where we give an algorithm for detecting safe expressions for speculative evaluation. In that paper we also present experimental results showing that cheap eagerness does indeed yield a significant performance improvement, decreasing execution time by 12–58% over a range of benchmark programs. Given these numbers, and the fact that most thunks that are built are also evaluated sooner or later, we have been interested in extending cheap eagerness by speculatively evaluating even more expressions.

The technique used in our earlier work is *static* in the sense that it makes a single decision for each candidate expression in the program. An expression is speculated only if it is known that it is safe every time it is evaluated in any execution of the program. In this paper we generalize this condition to *dynamic cheap eagerness* where we may postpone the final decision to run-time by inserting a run-time test to control whether the expression is speculated or delayed. This has the consequence that, in a single execution of the program, the same expression may be speculated some of the times it is reached but not all of them.

We use such tests in two situations. First, we allow producers of lazy data structures to produce more than one element of the structure at a time. Typically, such functions contain a recursive call inside a thunk. If the thunk would be speculated, the entire, possibly infinite, structure would be produced at once, which is unacceptable. Instead we maintain a *speculation level* as part of the abstract machine state, and only speculate a thunk containing a recursive call if the speculation level is positive. The speculation level is decremented during the evaluation of the speculated thunk body. In this way, the depth of speculation can be controlled.

Second, some expressions can be statically speculated except that they evaluate some free variables which can not be statically proved to be bound to WHNFs (weak head normal forms) at run-time. In this case, the variables in question can be tested and the expression speculated when none of them is bound to a thunk. Of course the two can be combined; an expression can be speculated if the speculation level is positive *and* some variables are not bound to thunks.

Note that we still use the full static analysis machinery from [Fax00], which is based on flow analysis, and we use an almost identical intermediate language where delayed evaluation is explicitly indicated by expressions of the form $\texttt{thunk}\, e$ (where e is the delayed expression). The analysis then finds some $\texttt{thunk}\, e$ expressions which can be replaced by e alone (static speculation), or annotated by a condition s giving $\texttt{thunk}\, s\, e$ (dynamic speculation).

$$
\begin{array}{rcl}
e \in \text{Expr} & \rightarrow & x \mid x_1\, x_2 \mid b \mid op^l\, x_1 \ldots x_k \\
& \mid & \texttt{case}\ x\ \texttt{of}\ alt_1; \ldots;\ alt_n\ \texttt{end} \\
& \mid & \texttt{let}\ x = e'\ \texttt{in}\ e \\
& \mid & \texttt{letrec}\ x_1 = b_1; \ldots; x_n = b_n\ \texttt{in}\ e \\
& \mid & \texttt{thunk}^l\ s\ e \mid \texttt{eval}\ x \\
b \in \text{Build} & \rightarrow & \lambda^l x.\, e \mid C^l\, x_1 \ldots x_k \mid \texttt{thunk}^l\ e \\
alt \in \text{Alt} & \rightarrow & C\, x_1 \ldots x_k\ \texttt{->}\ e
\end{array}
\qquad
\begin{array}{l}
s \in \text{SCond} \rightarrow \mathsf{E} \mid \mathsf{R} \mid \varepsilon \\
\\
x \in \text{Var} \\
l \in \text{Label} \quad \rightarrow \underline{1}, \underline{2}, \ldots \\
C \in \text{DataCon} \cup \text{IntLit}
\end{array}
$$

Fig. 1. The syntax of Fleet

2 The Language Fleet

Figure 1 gives the syntax of Fleet. It is a simple functional language containing the constructs of the lambda calculus as well as constructors, case expressions, built-in operators and recursive and nonrecursive let. Graph reduction is handled by explicit `eval` and `thunk` constructs; the rest of the language has a call-by-value semantics. Replacing call-by-need with call-by-value is the source-to-source transformation $\texttt{thunk}^l\, e \Longrightarrow e$.

```
letrec from = λ¹n.let r = thunk² let n1 = thunk³
                                          let n'= eval n
                                          in inc⁴ n'
                                 in from n1
                  in Cons⁵ n r
in let z = 0⁶
   in from z
```

Fig. 2. The `from` program

Lambda abstractions, constructors and (unconditional) thunks are referred to as *buildable expressions*. These are the only kind of expressions allowed as right-hand-sides in `letrec` bindings.

In order to generalize static cheap eagerness to the dynamic version, we give thunk expressions a *speculation condition* s which is either

- empty, for a `thunk` that is never speculated,
- E for a `thunk` that is speculated if no free variable x which is an argument of an `eval` occurring in the `thunk` body (but not nested inside another `thunk` or abstraction) is bound to a thunk, or
- R for a `thunk` that is speculated if the E condition is true and the speculation level is positive.

The speculation conditions are chosen to be testable with just a few machine instructions; hence the restriction to `eval` arguments that are free variables of the `thunk` body (and thus available in registers for the purpose of building the thunk). Starting with a positive speculation level which is decremented similarly allows a simple test for a negative value for the R condition.

We expect the front end to produce `thunk`s with empty speculation conditions with the analysis trying to either eliminate the `thunk`s or replace the speculation conditions with E or R.

Buildable expressions and operator applications are labeled. These labels identify particular expressions and are used to convey flow information. They are also used by the cheapness analysis; its result records the labels of `thunk`s that may be statically or dynamically speculated. Figure 2 gives an example Fleet program computing the list of all natural numbers.

We give Fleet a big step operational semantics in Fig. 3. The inference rules allow us to prove statements of the form

$$\rho, \sigma \vdash e \Downarrow v$$

where ρ is an environment mapping variables to values, σ is the speculation level, e is an expression, and v a value. A value is a *closure*; a pair of an environment and a buildable expression. Note that (unconditional) thunks are also values in this semantics since they are an explicit construct in the language. We refer to

$$\rho, \sigma \vdash x \Downarrow \rho(x) \qquad \text{var}$$

$$\frac{\rho(x_1) = (\rho', \lambda^l x.\, e) \qquad \rho'[x \mapsto \rho(x_2)], \sigma \vdash e \Downarrow v}{\rho, \sigma \vdash x_1\, x_2 \Downarrow v} \qquad \text{app}$$

$$\rho, \sigma \vdash b \Downarrow (\rho, b) \qquad \text{build}$$

$$\frac{[\![op]\!]\, \rho(x_1)\ \ldots\ \rho(x_k) = C}{\rho, \sigma \vdash op^l\, x_1 \ldots x_k \Downarrow ([\,], C^l)} \qquad \text{op}$$

$$\frac{\rho(x) = (\rho', C^l\, x'_1 \ldots x'_k) \qquad \rho[\ldots, x_i \mapsto \rho'(x'_i), \ldots], \sigma \vdash e \Downarrow v}{\rho, \sigma \vdash \texttt{case}\ x\ \texttt{of}\ \ \ldots; C\, x_1 \ldots x_k\ \texttt{->}\ e; \ldots\ \ \texttt{end} \Downarrow v} \qquad \text{case}$$

$$\frac{\rho, \sigma \vdash e' \Downarrow v' \qquad \rho[x \mapsto v'], \sigma \vdash e \Downarrow v}{\rho, \sigma \vdash \texttt{let}\ x \,\texttt{=}\, e'\ \texttt{in}\ e \Downarrow v} \qquad \text{let}$$

$$\frac{\rho' = \rho[\ldots, x_i \mapsto (\rho', b_i), \ldots] \qquad \rho', \sigma \vdash e \Downarrow v}{\rho, \sigma \vdash \texttt{letrec}\ x_1 \,\texttt{=}\, b_1; \ldots; x_n \,\texttt{=}\, b_n\ \texttt{in}\ e \Downarrow v} \qquad \text{letrec}$$

$$\frac{x \in \mathit{evars}(e) \Rightarrow \rho(x) \text{ is WHNF closure} \qquad \rho, \sigma \vdash e \Downarrow v}{\rho, \sigma \vdash \texttt{thunk}^l\, \mathsf{E}\ e \Downarrow v} \qquad \text{thunk } \mathsf{E}$$

$$\frac{x \in \mathit{evars}(e) \Rightarrow \rho(x) \text{ is WHNF closure} \qquad \sigma > 0 \qquad \rho, \sigma - 1 \vdash e \Downarrow v}{\rho, \sigma \vdash \texttt{thunk}^l\, \mathsf{R}\ e \Downarrow v} \qquad \text{thunk } \mathsf{R}$$

$$\frac{\text{speculation condition } s \text{ not satisfied}}{\rho, \sigma \vdash \texttt{thunk}^l\ s\ e \Downarrow (\rho, \texttt{thunk}^l\ e)} \qquad \text{thunk}$$

$$\frac{\rho(x) = (\rho', \texttt{thunk}^l\ e) \qquad \rho', \sigma \vdash e \Downarrow v}{\rho, \sigma \vdash \texttt{eval}\, x \Downarrow v} \ \text{eval-i} \qquad\qquad \frac{\rho(x) \text{ is a WHNF closure}}{\rho, \sigma \vdash \texttt{eval}\, x \Downarrow \rho(x)} \ \text{eval-ii}$$

$$\begin{aligned}
&\mathit{evars}(\texttt{case}\ x\ \texttt{of}\ \mathit{alt}_1; \ldots;\ \mathit{alt}_n\ \texttt{end}) = \mathit{evars}_{\mathit{alt}}(\mathit{alt}_1) \cup \ldots \cup \mathit{evars}_{\mathit{alt}}(\mathit{alt}_n)\\
&\mathit{evars}(\texttt{let}\ x\ \texttt{=}\ e'\ \texttt{in}\ e) = \mathit{evars}(e') \cup (\mathit{evars}(e) \setminus \{x\})\\
&\mathit{evars}(\texttt{letrec}\ x_1\ \texttt{=}\ b_1; \ldots; x_n\ \texttt{=}\, b_n\ \texttt{in}\ e) = \mathit{evars}(e) \setminus \{x_1, \ldots,\ x_n\}\\
&\mathit{evars}(\texttt{eval}\ x) = \{x\}\\
&\mathit{evars}(e) = \emptyset \text{ (for } e \text{ other than above)}\\
&\mathit{evars}_{\mathit{alt}}(C\ x_1 \ldots x_r\ \texttt{->}\ e) = \mathit{evars}(e) \setminus \{x_1, \ldots,\ x_r\}
\end{aligned}$$

Fig. 3. Operational semantics of Fleet

closures where the expression part is an abstraction or a constructor application as *weak head normal form (WHNF) closures.*

Infinite values (rational trees where each variable binding in an environment is seen as a branch) arise in the [letrec] rule in the semantics and correspond to the cyclic structures built by an implementation.

The cheap eagerness analysis uses the results of a previous *flow analysis.* A flow analysis finds information about the data and control flow in a program [Fax95, Shi91, JW95, Ses91]. In our case, flow information takes the form of a *flow assignment* φ mapping variables (the flat syntax of Fleet ensures that every interesting expression is associated with a variable) to sets of expressions. These expressions are all *producers* of values and include buildable expressions and operator applications. Since all producers are labeled, flow information could record labels rather than expressions. We have chosen to record expressions in order to simplify the definition of the cheap eagerness analysis.

Figure 4 gives a flow assignment for the `from` program. Note that the flow information for the variable `n1`, bound to the `thunk` containing the increment, also includes the flow information for the `thunk` body. Including the flow information for the thunk body in the flow information for the thunk is necessary since the cheap eagerness transformation may replace the `thunk` with its body, and the flow information must not be invalidated by that transformation.

$\texttt{from} : \{\lambda^{\underline{1}}\texttt{n}.E_1\}$	Abbreviations:
$\texttt{n} : \{\texttt{thunk}^{\underline{3}}\, E_3,\ \texttt{inc}^{\underline{4}}\ \texttt{n}',\ \texttt{0}^{\underline{6}}\}$	$E_1 \equiv \texttt{let r = thunk}^{\underline{2}}\ E_2\ \texttt{in Cons}^{\underline{5}}\ \texttt{n r}$
$\texttt{r} : \{\texttt{thunk}^{\underline{2}}\, E_2,\ \texttt{Cons}^{\underline{5}}\ \texttt{n r}\}$	$E_2 \equiv \texttt{let n1 = thunk}^{\underline{3}}\ E_3\ \texttt{in from n1}$
$\texttt{n1} : \{\texttt{thunk}^{\underline{3}}\, E_3,\ \texttt{inc}^{\underline{4}}\ \texttt{n}'\}$	$E_3 \equiv \texttt{let n}' = \texttt{eval n in inc}^{\underline{4}}\ \texttt{n}'$
$\texttt{n}' : \{\texttt{inc}^{\underline{4}}\ \texttt{n}',\ \texttt{0}^{\underline{6}}\}$	
$\texttt{z} : \{\texttt{0}^{\underline{6}}\}$	

Fig. 4. Flow information for `from`

3 The Analysis

The task of the analysis is to find out which expressions are *cheap*, that is, guaranteed to terminate without run-time error. In a denotational framework, this corresponds to having a semantics which is not $\perp$.

3.1 Cheap expressions

We follow [Fax00] in considering most builtin operators cheap, with division and modulus being exceptions unless the divisor is a nonzero constant. Case expressions are cheap if all of their branches are. Pattern matching failure is encoded by speciall error operators in the offending branches; flow information has been used to prune these where possible. We also do not speculate `thunk`s in the right-hand-sides of `letrec` bindings since buildable expressions are required there. The differences between our previous work and the present paper lies in the treatment of recursive functions and `eval` operators.

We refer to functions where the recursive calls only occur in the bodies of `thunk` expressions, as in the `from` function above where the recursive call is in the body of the `thunk`[2], as *lazily recursive*. Calling a lazily recursive function does not lead to any recursive calls, only to the building of thunks which may perform recursive calls if and when they are evaluated (typically, lazily recursive functions produce lazy data structures). In an *eagerly recursive* function such as `length` the recursive call is not nested inside a `thunk`.

We have previously speculated calls to lazily recursive functions, but we have not speculated the `thunk`s containing the recursive calls themselves since that would make the recursion eager. We now relax that constraint by allowing such `thunk`s, which we call *cycle breakers*, to be conditionally speculated based on the speculation level.

Previously, we did not speculate `thunk`s containing `eval`s whose arguments might be thunks since these thunks would in general have been expensive (inexpensive `thunk`s are always eliminated). We now relax this constraint as well, by emitting run-time tests so that such a `thunk` may be speculated whenever the arguments to any residual `eval`s in its body are actually WHNFs. For such a test to be efficient, the tested argument must be a free variable of the `thunk`.

We note that, as in our previous work, the legality of speculating a `thunk` may in general depend on the elimination of other `thunk`s. In fact, if the body of a `thunk` contains an `eval` which might evaluate that `thunk` itself, we may have circular dependencies. These do not invalidate speculation since the elimination of the `thunk` makes the `eval` cheap in the transformed program.

3.2 Formalization

We formalize the cheapness analysis using an inference system, shown in Figs. 5 and 6, which allows us to prove judgements of the form

$$S, V \vdash^{\varphi} e : l$$

where S is a set of constraints, V is a set of thunk-local variables, φ a flow assignment, e is an expression and l is a label (in an implementation, S is the output while the other are inputs). If e occurs in the body of a `thunk`, V contains the variables that are in scope but not available for speculation tests since they are not free variables of the enclosing `thunk`. There are also auxilliary judgements of the form $S, V \vdash^{\varphi}_{R} b : l$ which express the restriction that `thunk`s in `letrec` bindings can not be eliminated.

Judgements of the form $S \vdash^{\varphi}_{l} E$, where E is an error check, ensure that expressions which may cause a run-time error are considered expensive.

$\boxed{S, V \vdash^{\varphi} e : l}$

$$\{\}, V \vdash^{\varphi} x : l \qquad \text{VAR}$$

$$\frac{S, V \cup \{x\} \vdash^{\varphi} e : l'}{S, V \vdash^{\varphi} \lambda^{l'} x.e : l} \qquad \text{ABS}$$

$$\frac{S \vdash_{l}^{\varphi} \mathsf{IsAbs}(x_1)}{S \cup \{l \rightsquigarrow_{\mathsf{c}} l' \mid \lambda^{l'} x.e \in \varphi(x_1)\}, V \vdash^{\varphi} x_1\, x_2 : l} \qquad \text{APP}$$

$$\frac{S \vdash_{l}^{\varphi} \mathsf{NoErr}(op\, x_1 \ldots x_r)}{S, V \vdash^{\varphi} op^{l'}\, x_1 \ldots x_r : l} \qquad \text{OP}$$

$$\{\}, V \vdash^{\varphi} C^{l'}\, x_1 \ldots x_r : l \qquad \text{CON}$$

$$\frac{S \vdash_{l}^{\varphi} \mathsf{OneOf}(x, C_1, \ldots, C_n) \qquad S_1, V \cup \bar{x}_1 \vdash^{\varphi} e_1 : l \ \ \ldots \ \ S_n, V \cup \bar{x}_n \vdash^{\varphi} e_n : l}{S \cup S_1 \cup \ldots \cup S_n, V \vdash^{\varphi} \texttt{case}\ x\ \texttt{of}\ \ldots; C_i\, \bar{x}_i\ \texttt{->}\ e_i; \ldots\ \texttt{end} : l} \qquad \text{CASE}$$

$$\frac{S, V \vdash^{\varphi} e : l \qquad S', V \cup \{x\} \vdash^{\varphi} e' : l}{S \cup S', V \vdash^{\varphi} \texttt{let}\ x = e\ \texttt{in}\ e' : l} \qquad \text{LET}$$

$$\frac{V' = V \cup \{x_1, \ldots, x_n\} \qquad S_1, V' \vdash_{R}^{\varphi} b_1 : l \ \ \ldots \ \ S_1, V' \vdash_{R}^{\varphi} b_n : l \qquad S, V' \vdash^{\varphi} e : l}{S \cup S_1 \cup \ldots \cup S_n, V \vdash^{\varphi} \texttt{letrec}\ x_1 = b_1; \ldots; x_n = b_n\ \texttt{in}\ e : l} \text{LETREC}$$

$$\frac{S, \emptyset \vdash^{\varphi} e : l'}{S \cup \{l \rightsquigarrow_{\mathsf{t}} l'\}, V \vdash^{\varphi} \texttt{thunk}^{l'}\ e : l} \qquad \text{THUNK}$$

$$\frac{x \in V}{\{l \rightsquigarrow_{\mathsf{e}} l' \mid \texttt{thunk}^{l'}\ e \in \varphi(x)\}, V \vdash^{\varphi} \texttt{eval}\, x : l} \qquad \text{EVAL-I}$$

$$\frac{x \notin V}{\{l \rightsquigarrow_{\mathsf{x}} l' \mid \texttt{thunk}^{l'}\ e \in \varphi(x)\}, V \vdash^{\varphi} \texttt{eval}\, x : l} \qquad \text{EVAL-II}$$

Fig. 5. Constraint derivation rules, part 1

$$\boxed{S, V \vdash^{\varphi}_{R} b : l}$$

$$\frac{b \text{ is not a } \mathtt{thunk} \qquad S, V \vdash^{\varphi} b : l}{S, V \vdash^{\varphi}_{R} b : l} \quad \text{R-WHNF}$$

$$\frac{S, \emptyset \vdash^{\varphi} e : l'}{S \cup \{l' \rightsquigarrow_{\mathsf{c}} \Omega\}, V \vdash^{\varphi}_{R} \mathtt{thunk}^{l'}\, e : l} \quad \text{R-THUNK}$$

$$\boxed{S \vdash^{\varphi}_{l} E}$$

$$\frac{\forall e \in \varphi(x) \,.\, e \text{ is an abstraction}}{\{\,\} \vdash^{\varphi}_{l} \mathsf{IsAbs}(x)} \quad \text{IS ABS}$$

$$\frac{\forall e \in \varphi(x) \,.\, e \text{ is built with one of the } C_i}{\{\,\} \vdash^{\varphi}_{l} \mathsf{OneOf}(x, C_1, \ldots, C_n)} \quad \text{ONE OF}$$

$$\frac{op\, x_1 \ldots x_r \text{ is defined if the } x_i \text{ are described by } \varphi}{\{\,\} \vdash^{\varphi}_{l} \mathsf{NoErr}(op\, x_1 \ldots x_r)} \quad \text{NO ERR}$$

$$\frac{\text{none of the above applies}}{\{l \rightsquigarrow_{\mathsf{c}} \Omega\} \vdash^{\varphi}_{l} E} \quad \text{ERROR}$$

Fig. 6. Constraint derivation rules, part 2

There are four main forms of constraints: $l \rightsquigarrow_{\mathsf{c}} l'$, $l \rightsquigarrow_{\mathsf{e}} l'$, $l \rightsquigarrow_{\mathsf{x}} l'$ and $l \rightsquigarrow_{\mathsf{t}} l'$. The labels are labels of `thunk` expressions and lambda abstractions and the constraints relate the costs of evaluating the corresponding thunk or function bodies. A fifth form of constraint is $l \rightsquigarrow_{\mathsf{c}} \Omega$, which is used to indicate that the body of the thunk or function labeled with l is expensive or unsafe to evaluate.

Figure 7 shows the constraints S derived from the `from` program in Fig. 2 using the flow assignment φ in Fig. 4 ($\underline{42}$ is an arbitrary label representing the top level context). Thus we have $S, \emptyset \vdash^{\varphi} P : \underline{42}$ where P is the `from` program.

We will solve constraints by assigning *costs* to labels in such a way that the constraints are satisfied.

Definition 1 (Costs) *A* cost Δ *is of one of the forms below. Costs are ordered by* $<$.

- *A natural number* n *is called a* finite cost*; the costs defined below are called* infinite costs*. Finite costs are ordered numerically.*

$$\underline{2} \rightsquigarrow_{\mathsf{c}} \underline{1},\ \underline{42} \rightsquigarrow_{\mathsf{c}} \underline{1},\ \underline{1} \rightsquigarrow_{\mathsf{t}} \underline{2},\ \underline{2} \rightsquigarrow_{\mathsf{t}} \underline{3},\ \underline{3} \rightsquigarrow_{\mathsf{x}} \underline{3}$$

$$\delta(\underline{1}) = 1, \quad \delta(\underline{2}) = \mathsf{R}, \quad \delta(\underline{3}) = 1, \quad \delta(\underline{42}) = 2$$

Fig. 7. The constraints derived from the `from` program and their least model

- *If n is a finite cost, then E^n is a cost. If $n < m$ then $n < \mathsf{E}^m$ and $\mathsf{E}^n < \mathsf{E}^m$.*
- *R is a cost and for all n, $\mathsf{E}^n < \mathsf{R}$.*
- *Ω is the greatest cost; for all Δ, $\Delta < \Omega$.*

We will write $\leq$ for the reflexive closure of $<$ and $\Delta \vee \Delta'$ for the least upper bound of Δ and Δ'.

Costs of the form n represent safe computations which can be statically speculated (the n is related to the height of the derivation of the evaluation relation for the computation). The costs E^n and R stand for costs which, if they are assigned to a thunk label, implies that the thunk may be dynamically speculated. Essentially, E^n means that the thunk body may be speculated when all variables which are arguments to the remaining `eval`s are bound to WHNFs, and R adds the further restriction that the speculation level must be positive (the n in E^n is analogous to the finite cost n of evaluating the thunk body if the conditions for speculation are satisfied).

Definition 2 *A* cost assignment *δ is a function from labels to costs. Further, δ is a* model *of a constraint set S, written $\delta \models S$ iff all of the following holds:*

- *For every constraint $l \rightsquigarrow_{\mathsf{c}} l' \in S$, $\delta(l) > \delta(l')$ and $\delta(l) \neq \mathsf{E}^n, \mathsf{R}$. For every constraint $l \rightsquigarrow_{\mathsf{c}} \Omega \in S$, $\delta(l) = \Omega$.*
- *For every constraint $l \rightsquigarrow_{\mathsf{e}} l' \in S$, either $\delta(l) = \Omega$ or $\delta(l')$ is finite.*
- *For every constraint $l \rightsquigarrow_{\mathsf{x}} l' \in S$, either $\delta(l) \geq \mathsf{E}^0$ or $\delta(l')$ is finite.*
- *For every constraint $l \rightsquigarrow_{\mathsf{t}} l' \in S$, either $\delta(l) > \delta(l')$ or $\delta(l') \geq \mathsf{R}$.*

A model δ of a constraint set S is minimal *iff for every δ' such that $\delta' \models S$ the following holds for all l*

- *$\delta(l)$ infinite implies $\delta'(l)$ infinite*
- *$\delta(l) = \Omega$ implies $\delta'(l) = \Omega$.*

The constraint set derived from the `from` program has a minimal model, which is shown in Fig. 7.

Looking at the inference rules in Figs. 5 and 6, we can see that constraints of the form $l \rightsquigarrow_{\mathsf{c}} l'$ correspond to function calls, where l is the label of the caller (the innermost lambda abstraction or `thunk`) and l' is the label of the callee (a lambda abstraction). If the callee (l') is associated with an infinite cost, the caller (l) must have cost Ω since the caller can not determine if the callee is actually going to do whatever possibly expensive thing its infinite costs warns of. Constraints of the form $l \leadsto_{\mathsf{c}} \Omega$ indicate that the evaluation of the thunk or function body l might cause a run-time error. Such constraints are generated by the error checking judgements $S \vdash^{\varphi}_{l} E$.

Similarly, $l \leadsto_{\mathsf{e}} l'$ and $l \leadsto_{\mathsf{x}} l'$ correspond to an `eval`, occurring in the thunk or function body l, which may evaluate a thunk labeled l'. If l' is assigned a finite cost, the thunk will be eliminated. Thus there will be no such thunk for the `eval` to find, so the cost of the thunk body does not affect the cost of the `eval`. If, on the other hand, a thunk might occur at an `eval` inside a `thunk` body, there is a difference between the case where the argument to the `eval` is available for testing outside the body ($l \leadsto_{\mathsf{x}} l'$) or not ($l \leadsto_{\mathsf{e}} l'$). In the first case, dynamic speculation is possible (the thunk label l is assigned a cost of the form E^n), otherwise it is not (l must be assigned Ω).

The fourth kind of constraint, $l \leadsto_{\mathsf{t}} l'$, corresponds to a `thunk` labeled with l' nested inside a thunk or function labeled with l. The other constraints are monotonic, in the sense that the more expensive the label on the right is, the more expensive is the label on the left. This nesting constraint is different; if l' is assigned R or Ω, l can be assigned any cost.

The motivation for this rule is as follows: Recall that the cost assigned to l' (the label of the `thunk`) is the cost associated with the thunk body; if that cost is Ω, the `thunk` is left as a thunk rather than being eliminated, and `thunk` expressions are always safe and cheap (that is their raison d'être, after all). A thunk that is assigned a finite cost, however, is replaced by the thunk body, making the enclosing expression more expensive than the body. The cost R represents bounded depth speculation; the speculation counter ensures that the cost of evaluating the `thunk` body is finite. In the case of a cost of the form E^n, the speculation counter is not consulted, so the n is used to guarantee terminating evaluation statically. This form of constraint is used to ensure that the transformation does not turn lazy recursion into eager recursion.

The nonmonotonicity of nesting constraints implies that not all constraint sets have minimal models. This happens when there is a lazily recursive cycle of otherwise safe thunks and any one could be chosen to break the cycle (by mapping it to R). In that case we will have a cycle of $\leadsto_{\mathsf{t}}$ and $\leadsto_{\mathsf{c}}$ constraints which can not all be legally assigned finite costs.

3.3 Implementation

It is easy to see how the inference system in Figs. 5 and 6 can be turned into a function $\mathsf{Constraints}$ taking a flow assignment φ, a variable set V, a label l and an expression e and producing the smallest constraint set S such that $S, V \vdash^{\varphi} e : l$. We therefore proceed to the question of how to compute a model of the constraints thus derived.

Due to the nonmonotonicity of the $\leadsto_{\mathsf{t}}$ constraints, there is room for some cunning in the constraint solving algorithm. There is sometimes a choice of which thunk labels to map to R in order to avoid turning lazy recursion into eager recursion. In these cases, it is best to choose labels that must be mapped to an infinite cost in any model of the constraints. We call such labels *forced*

Solve:
    for each $l \in \text{LabAbs}$ do
        if $l \leadsto_{\mathsf{c}}^{+} l$ or $l \leadsto_{\mathsf{c}} \Omega$ then $\delta(l) := \Omega$
    Propagate
    for each $l \in \text{LabThunk}$ do
        $R_l := \{l\} \cup \{l' \mid l' \in \text{LabThunk} \textsf{ and } l \leadsto_{\mathsf{ce}}^{+} l'\}$
        if there is a cycle C in the $\leadsto_{\mathsf{ct}}$ subgraph such that
            $C \subseteq R_l \cup \text{LabAbs}$
        then $\delta(l) := \delta(l) \vee \mathsf{E}^0$
    loop
        Propagate
        for each $l \leadsto_{\mathsf{t}} l'$ do
            if $\delta(l') \geq \mathsf{R}$ then remove $l \leadsto_{\mathsf{t}} l'$ from the graph
        if there is an $l \leadsto_{\mathsf{t}} l'$ such that $l' \leadsto_{\mathsf{ct}}^{+} l$ and preferably $\delta(l') \geq \mathsf{E}^0$
            then $\delta(l') := \mathsf{R}$
            else exit loop
    end loop
    assign finite costs to all labels not already assigned infinite cost
    adjust costs of the form E^n

Propagate:
    repeat
        if $l \leadsto_{\mathsf{c}} l'$ and $\delta(l') \geq \mathsf{E}^0$ then $\delta(l) := \Omega$
        if $l \leadsto_{\mathsf{e}} l'$ and $\delta(l') \geq \mathsf{E}^0$ then $\delta(l) := \Omega$
        if $l \leadsto_{\mathsf{x}} l'$ and $\delta(l') \geq \mathsf{E}^0$ then $\delta(l) := \delta(l) \vee \mathsf{E}^0$
    until no change

Fig. 8. Constraint solving algorithm

labels. If we succeed in solving the constraints by mapping only forced labels to R, the constraints have a minimal model and we have found it.

When discussing this algorithm, it will be helpful to view the constraints as defining a graph with labels and Ω as nodes and four different kinds of edges corresponding to the four types of constraints (this graph is a refinement of the call graph of the program). We will form sub graphs of this graph by considering only certain kinds of edges, for instance only the $\leadsto_{\mathsf{c}}$ edges or only the $\leadsto_{\mathsf{c}}$ and $\leadsto_{\mathsf{e}}$ edges. We will refer to these as the $\leadsto_{\mathsf{c}}$ and the $\leadsto_{\mathsf{ce}}$ sub graph, respectively. We will also use the transitive closure of the arrow symbols to indicate reachability (*e.g.* $l \leadsto_{\mathsf{c}}^{+} l'$ means that there is a path from l to l' in the $\leadsto_{\mathsf{c}}$ subgraph). Note that we do *not* mean the transitive and reflexive closure; $l \leadsto_{\mathsf{c}}^{+} l$ does not hold in general.

We give our constraint solving algorithm in Fig. 8. The general strategy is to systematically find the forced labels in the constraint set, using the following observations:

– Cycles in the $\rightsquigarrow_{\mathsf{c}}$ subgraph correspond to eagerly recursive functions and can only be solved by mapping all of the labels in the cycle to Ω.
– If $l \rightsquigarrow_{\mathsf{c}} l'$ or $l \rightsquigarrow_{\mathsf{e}} l'$ and l' is mapped to an infinite cost, then l must also be mapped to Ω (and similarly for $l \rightsquigarrow_{\mathsf{x}} l'$). The Propagate steps are motivated by this fact.
– If C is a cycle in the $\rightsquigarrow_{\mathsf{ct}}$ subgraph, we must map some of the thunk labels in C to R or Ω (otherwise we may turn lazy recursion into eager recursion). If there is some thunk label l from which all of the labels in C (except possibly l itself, if $l \in C$) are reachable in the $\rightsquigarrow_{\mathsf{ce}}$ subgraph, it follows that l must be mapped to an infinite cost since at least one of the thunks it may evaluate can not be statically speculated. Since l will be mapped to at least E^0, it can as well be mapped to R from the point of view of minimality.
  A special case of this is when the cycle contains only one thunk label l.

As an example of the last point, consider the following constraint set:

$$\underline{1} \rightsquigarrow_{\mathsf{t}} \underline{2},\ \underline{2} \rightsquigarrow_{\mathsf{t}} \underline{3},\ \underline{3} \rightsquigarrow_{\mathsf{c}} \underline{1},\ \underline{2} \rightsquigarrow_{\mathsf{x}} \underline{3}$$

We must map either $\underline{2}$ or $\underline{3}$ to R. If we choose $\underline{3}$, we will soon find that we will also have to map $\underline{2}$ to E^0 because of the constraint $\underline{2} \rightsquigarrow_{\mathsf{x}} \underline{3}$, but if we choose $\underline{2}$ to start with, we can assign $\underline{3}$ a finite cost.

When all forced labels have been mapped to R or Ω there may still be cycles in the $\rightsquigarrow_{\mathsf{ct}}$ graph which have not yet been broken by mapping some of the thunk labels to R or Ω. We then pick some arbitrary thunk labels from such cycles and map them to R. When no cycles are left, all nodes not mapped to infinite costs can be mapped to finite costs in a single bottom-up traversal. As we have discussed above, cycles involving $\rightsquigarrow_{\mathsf{e}}$ edges do not necessarily force any labels to infinite costs.

We conjecture that this algorithm computes a minimal model if one exists, but we have not yet proved so. The algorithm is however derived from the one in [Fax00], which has this property for static cheap eagerness.

4 Experimental Results

We have implemented dynamic cheap eagerness in an experimental compiler for a simple lazy higher order functional language called Plain. The compiler has a (rather primitive) strictness analyzer based on demand propagation and a (rather sophisticated) flow analyzer based on polymorphic subtype inference [Fax95]. Except being used for cheap eagerness, the flow information is exploited by update avoidance and representation analysis [Fax99]. The compiler also uses *cloning* to be able to generate tailor made code for different uses of the same function [Fax01]. Since several optimizations are program-wide, separate compilation is not supported. The compiler is described more fully in the author's PhD thesis [Fax97, chapter 3].

The compiler can be instructed to generate code counting various events, including the construction and evaluation of thunks. We have measured user level (machine) instruction counts using the `icount` analyzer included in the Shade distribution. The execution times measured are the sum of user and system times, as reported by the `clock()` function, and is the average of four runs. Both times and instruction counts include garbage collection. All measurements were performed on a lightly loaded Sun Ultra 5 workstation with 128MB of memory and a 270MHz UltraIIi processor.

We have made preliminary measurements of the effectiveness of dynamic cheap eagerness using a set of small test programs (the largest is ≈ 700 lines of code). The small size of the programs makes any claims based on these experiments very tentative. With this caveat, we present the programs:

nqh The N-Queens program written with higher order functions wherever possible (`append` is defined in terms of `foldr` etc), run with input 11.
q1 Same as `nqh`, but with all higher order functions manually removed by specialization, run with input 11.
nrev Naive reverse, a *very* silly program, but it shows a case where dynamic cheap eagerness really pays off, run on a list of length 4141 elements.
event A small event driven simulator from Hartel and Langendoen's benchmark suite [HL93], run with input 400000.
sched A job scheduler, also from that benchmark suite, run with input 14.
infer A simple polymorphic type checker from [HL93][1], run with input 600.

4.1 Strategies

We have measured several strategies which differ in which `thunk`s they are willing to dynamically speculate.

s: Only static cheap eagerness, corresponding to the most aggressive strategy in [Fax00].
e: Thunks with `eval`s of free variables (speculation condition E) are speculated, but not cycle breakers. The speculation depth is not used.
c: Cycle breakers with no `eval`s are speculated (speculation condition R with an empty *evars* set), but no thunks with `eval`s are speculated. Run with speculation depth 1, 2 and 9.
ce+c: Cycle breakers and thunks with conservative `eval`s are speculated. A conservative `eval` is an `eval` which is guaranteed never to evaluate a `thunk` with an empty speculation condition. Effectively, we add the condition that if $l \rightsquigarrow_{\mathsf{x}} l'$ and $\delta(l') = \Omega$ then $\delta(l) = \Omega$. Speculation conditions of both E and R are used. Run with speculation depth 1, 2 and 9.
e+c: Cycle breakers and `thunk`s with `eval`s of free variables are speculated, as described in Sec. 3. Run with speculation level 1, 2 and 9.

The motivation of the **ce+c** strategy is to decrease the number of failed speculation tests. Typically, such a test costs 7–8 instructions, including branches and two `nop`s in delay slots (we statically predict that the speculation will be done).

[1] It is called `typecheck` there.

Since dynamic cheap eagerness duplicates the bodies of conditionally speculated `thunks`, code growth is potentially an issue. Since `thunks` may be nested inside `thunk` bodies, there is in principle a risk of exponential code growth. In our measurements, however, we have not seen code growth above 10%, with 1–2% being more common.

4.2 Execution Times and Operation Counts

Looking at the results in Table 1, we immediately see that for most programs, and in particular for the somewhat larger programs `event`, `sched` and `typecheck`, dynamic cheap eagerness has very little to offer beyond its static version in terms of reductions of execution time and instruction count. No optimization strategy is even able to consistently improve performance, although for all of the programs there is some startegy that can reduce instruction counts, albeit often only a by tiny amount.

In `nrev`, we see an example where dynamic cheap eagerness works as intended. This program is dominated by calls to `append` with large left arguments which themselves are results from `append`.

On the other hand, dynamic cheap eagerness is successful in reducing the number of thunks built and evaluated, which are the operations targeted by speculative evaluation. Again, `nrev` stands out with up to 90% of these operations eliminated with the **ce+c** and **e+c** strategies and a speculation depth of 9. The larger programs also get some reductions, ranging from 4% of thunks built and 5% of thunks evaluated for `sched` (the **c**, **ce+c** and **e+c** strategies with depth 9) to 15% and 27%, respectively, for `infer` (the **e+c** strategy with depth 9).

One of the causes of these results is apparent if we study Table 2 which gives the average number of thunks built and thunks evaluated for every 1000 instructions executed. Given that building or evaluating a thunk yields an overhead of very roughly ten instructions, the numbers can be interpreted as percentages of the executed instructions that are spent on these operations. It is then clear that the larger programs spend at most some 13–15% of their instructions on the operations which are targeted by this optimization. The program with the best speedup, `nrev`, spends the largest amount of instructions, some 36%, on the targeted operations.

4.3 Speculation Statistics

To further understand the results, we can study Table 3 which give other statistics relevant to speculative evaluation. For `nqh`, we see that all of the strategies which speculate cycle breakers yield the same results, with deeper speculation yielding improved speedup. This is not surprising since effectively all thunks built are also evaluated. The same behaviour is shown by `q1`.

For `nrev` we see that cycle breaking must be combined with speculation of evaluation since the cycle-breaking `thunk` in `append` also contains an `eval`. Otherwise no speculation is performed.

Table 1. Execution times and operation counts.

		s	e	c			ce+c			e+c		
				1	2	9	1	2	9	1	2	9
`nqh`	t	1.60	1.60	1.60	1.60	1.50	1.60	1.60	1.50	1.60	1.60	1.50
	I	418.0	418.0	420.3	412.9	405.8	420.3	412.9	405.8	420.3	412.9	405.8
	%I	100	100	101	99	97	101	99	97	101	99	97
	T	3979	3979	2994	2665	2337	2994	2665	2337	2994	2665	2337
	%T	100	100	75	67	59	75	67	59	75	67	59
	E	3979	3979	2994	2665	2337	2994	2665	2337	2994	2665	2337
	%E	100	100	75	67	59	75	67	59	75	67	59
`q1`	t	1.20	1.20	1.17	1.10	1.00	1.13	1.10	1.00	1.13	1.10	1.00
	I	314.3	314.3	316.7	309.4	302.1	316.7	309.4	302.1	316.7	309.4	302.1
	%I	100	100	101	98	96	101	98	96	101	98	96
	T	2170	2170	1185	856	528	1185	856	528	1185	856	528
	%T	100	100	55	39	24	55	39	24	55	39	24
	E	2170	2170	1185	856	528	1185	856	528	1185	856	528
	%E	100	100	55	39	24	55	39	24	55	39	24
`nrev`	t	3.60	3.63	3.60	3.63	3.67	2.90	2.50	1.93	2.87	2.50	1.93
	I	472.1	472.1	472.1	472.1	472.1	465.9	422.5	363.2	465.9	422.5	363.2
	%I	100	100	100	100	100	99	89	77	99	89	77
	T	8576	8576	8576	8576	8576	4289	2860	859	4289	2860	859
	%T	100	100	100	100	100	50	33	10	50	33	10
	E	8576	8576	8576	8576	8576	4289	2860	859	4289	2860	859
	%E	100	100	100	100	100	50	33	10	50	33	10
`event`	t	2.20	2.20	2.20	2.20	2.20	2.20	2.23	2.23	2.20	2.30	2.23
	I	315.6	316.1	315.5	315.4	315.4	314.0	317.0	316.1	330.5	333.8	331.8
	%I	100	100	100	100	100	99	100	100	105	106	105
	T	2222	2222	2219	2218	2217	2122	2091	2078	1946	1929	1902
	%T	100	100	100	100	100	95	94	94	88	87	86
	E	2096	2096	2093	2092	2091	1996	1965	1952	1820	1803	1776
	%E	100	100	100	100	100	95	94	93	87	86	85
`sched`	t	27.13	26.77	26.70	26.50	26.53	26.67	26.50	26.50	26.37	26.13	26.20
	I	5120.8	5151.1	5115.4	5100.7	5089.0	5115.4	5100.7	5089.0	5145.7	5131.0	5119.2
	%I	100	101	100	100	99	100	100	99	100	100	100
	T	42293	42293	41204	40881	40650	41204	40881	40650	41204	40881	40650
	%T	100	100	97	97	96	97	97	96	97	97	96
	E	35619	35619	34531	34207	33977	34531	34207	33977	34531	34207	33977
	%E	100	100	97	96	95	97	96	95	97	96	95
`infer`	t	4.00	3.97	3.93	4.00	4.00	4.00	4.00	4.07	4.03	3.90	4.07
	I	710.3	710.1	711.8	712.5	725.3	712.0	705.4	727.5	713.9	707.2	728.8
	%I	100	100	100	100	102	100	99	102	101	100	103
	T	4727	4643	4588	4569	4729	4157	3926	4142	4052	3815	4023
	%T	100	98	97	97	100	88	83	88	86	81	85
	E	4241	4149	4071	4021	3942	3609	3337	3204	3522	3245	3104
	%E	100	98	96	95	93	85	79	76	83	77	73

Legend: t is CPU time, I is millions of instructions, T is thousands of thunks built, E is thousands of thunks evaluated, %I,%T,%E are percentages relative to the **s** column (100 means no change).

Table 2. Average number of thunks built or evaluated per 1000 instructions executed.

Program	thunks built	thunks evaluated
`nqh`	9.5	9.5
`q1`	6.9	6.9
`nrev`	18.2	18.2
`event`	7.0	6.6
`sched`	8.3	7.0
`infer`	6.7	6.0

As for `sched`, there are few of the thunks built that are candidates for speculative evaluation at all, and combined with the small amount of instructions spent on building and evaluating thunks, it is unsurprising that the effects, positive or negative, are small. The conservative evaluation speculation strategy **ce+c** is clearly beneficial, as the strategies with aggressive speculation of evaluation gets the worst slowdowns.

Finally, `infer` is the only program where speculation regularly creates more work by speculating thunks which would otherwise not have been evaluated, leading to slowdown for deep speculation of cycle breakers. It is interresting to note that in this case both the number of thunks built and speculation conditions evaluated increase, which is possible if the speculation of one `thunk` body leads to other thunks being built or more speculation conditions being evaluated.

5 Conclusions and Further Work

We have implemented a generalization of the static cheap eagerness optimization discussed in [Fax00] and measured its effectiveness. The good news is that speculative evaluation can further reduce the number of thunks built and evaluated by between 5 and 90%. The bad news is that this hardly improves run-times and instructions counts at all for most programs since the gains are offset by losses in terms of the costs of the dynamic tests and wasted work due to speculation.

Another reason why going from no speculation to static speculation gave better speedup than going from static to dynamic speculation is that static speculation enables other optimizations such as unboxing and removal of redundant `eval` operations. This is not the case for dynamic speculation since other passes must still assume that conditionally speculated thunks might occur, only less often.

We believe that other techniques can be used to exploit dynamic cheap eagerness. Specifically, partial unrolling of recursion can eliminate the need for maintaining the speculation counter at run-time and it also opens up possibilities for e.g. sharing heap checks between several allocations. Eventually, list unrolling [SRA94, CV94] can be used to replace several cells in a data structure

Table 3. Speculative evaluation statistics.

		s	e	c			ce+c			e+c		
				1	2	9	1	2	9	1	2	9
`nqh`	%I	100	100	101	99	97	101	99	97	101	99	97
	T	3979	3979	2994	2665	2337	2994	2665	2337	2994	2665	2337
	%TE	100	100	100	100	100	100	100	100	100	100	100
	C	0	0	1807	1807	1807	1807	1807	1807	1807	1807	1807
	%CS	0	0	55	73	91	55	73	91	55	73	91
	%SE	-	-	99	100	100	99	100	100	99	100	100
`q1`	%I	100	100	101	98	96	101	98	96	101	98	96
	T	2170	2170	1185	856	528	1185	856	528	1185	856	528
	%TE	100	100	100	100	100	100	100	100	100	100	100
	C	0	0	1807	1807	1807	1807	1807	1807	1807	1807	1807
	%CS	0	0	55	73	91	55	73	91	55	73	91
	%SE	-	-	99	100	100	99	100	100	99	100	100
`nrev`	%I	100	100	100	100	100	99	89	77	99	89	77
	T	8576	8576	8576	8576	8576	4289	2860	859	4289	2860	859
	%TE	100	100	100	100	100	100	100	100	100	100	100
	C	0	0	0	0	0	8572	8572	8572	8572	8572	8572
	%CS	0	0	0	0	0	50	67	90	50	67	90
	%SE	-	-	-	-	-	100	100	100	100	100	100
`event`	%I	100	100	100	100	100	99	100	100	105	106	105
	T	2222	2222	2219	2218	2217	2122	2091	2078	1946	1929	1902
	%TE	94	94	94	94	94	94	94	94	94	93	93
	C	0	92	6	6	6	146	146	146	1598	1598	1598
	%CS	0	0	50	67	90	68	90	99	17	18	20
	%SE	-	-	99	99	92	100	100	100	100	100	100
`sched`	%I	100	101	100	100	99	100	100	99	100	100	100
	T	42293	42293	41204	40881	40650	41204	40881	40650	41204	40881	40650
	%TE	84	84	84	84	84	84	84	84	84	84	84
	C	0	4017	1642	1642	1642	1642	1642	1642	5660	5660	5660
	%CS	0	0	66	86	100	66	86	100	19	25	29
	%SE	-	-	100	100	100	100	100	100	100	100	100
`infer`	%I	100	100	100	100	102	100	99	102	101	100	103
	T	4727	4643	4588	4569	4729	4157	3926	4142	4052	3815	4023
	%TE	90	89	89	88	83	87	85	77	87	85	77
	C	0	206	374	406	646	1315	1368	1732	1715	1769	2133
	%CS	0	67	54	71	94	57	79	92	50	68	81
	%SE	-	67	84	76	49	84	84	65	84	83	66

Legend: %I is relative instruction count as percentage (100 means same as with the **s** strategy), T is thousands of thunks built, %TE is the percentage of thunks built that are also evaluated, C is thousands of speculation conditions evaluated, %CS is percentage of conditions that selected speculation, %SE is percentage of speculated thunks which would have been evaluated (- means no thunks were speculated).

with a single larger cell so that a data structure is in effect produced a few items at a time. This optimisation is correct in the same cases that dynamic cheap eagerness is correct, that is, when each item of the structure, but not necessarily the entire structure, can be computed in advance. Thus our results indicate that list unrolling is often valid in a lazy functional language. It would be interesting to explore these possibilities in the future.

References

[CV94] Hall Cordelia V. Using Hindley-Milner type inference to optimise list representation. In *Lisp and Functional Programming*, June 1994.

[Fax95] Karl-Filip Faxén. Optimizing lazy functional programs using flow inference. In Allan Mycroft, editor, *Proceedings of the Second International Symposium on Static Analysis*, pages 136–153, Glasgow, UK, September 1995. Springer-Verlag.

[Fax97] Karl-Filip Faxén. *Analysing, Transforming and Compiling Lazy Functional Programs.* PhD thesis, Department of Teleinformatics, Royal Institute of Technology, June 1997.

[Fax99] Karl-Filip Faxén. Representation analysis for coercion placement. In Konstantinos Sagonas and Paul Tarau, editors, *Proceedings of the International Workshop on Implementation of Declarative Languages*, September 1999.

[Fax00] Karl-Filip Faxén. Cheap eagerness: Speculative evaluation in a lazy functional language. In Philip Wadler, editor, *Proceedings of the 2000 International Conference on Functional Programming*, September 2000.

[Fax01] Karl-Filip Faxén. The costs and benefits of cloning in a lazy functional language. In Stephen Gilmore, editor, *Trends in Functional Programming, volume 2*, pages 1–12. Intellect, 2001. Proc. of Scottish Functional Programming Workshop, 2000.

[HL93] Pieter Hartel and Koen Langendoen. Benchmarking implementations of lazy functional languages. In *Functional Programming & Computer Architecture*, pages 341–349, Copenhagen, June 93.

[JW95] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Principles of Programming Languages*, 1995.

[Ses91] Peter Sestoft. *Analysis and efficient implementation of functional programs.* PhD thesis, DIKU, University of Copenhagen, Denmark, October 1991.

[Shi91] O. Shivers. The semantics of Scheme control-flow analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, volume 26, pages 190–198, New Haven, CN, June 1991.

[SRA94] Zhong Shao, John H. Reppy, and Andrew W. Appel. Unrolling lists. In *Lisp and Functional Programming*, June 1994.

This article was processed using the LaTeX macro package with LLNCS style