Cheap Eagerness: Speculative Evaluation in a Lazy Functional Language

Karl-Filip Faxén
Dept. of Teleinformatics, KTH, Stockholm
kff@it.kth.se

ABSTRACT

Cheap eagerness is an optimization where cheap and safe expressions are evaluated before it is known that their values are needed. Many compilers for lazy functional languages implement this optimization, but they are limited by a lack of information about the global flow of control and about which variables are already evaluated. Without this information, even a variable reference is a potentially unsafe expression!

In this paper we show that significant speedups are achievable by cheap eagerness. Our cheapness analysis uses the results of a program-wide data and control flow analysis to find out which variables may be unevaluated and which variables may be bound to functions which are dangerous to call.

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. Some cheap expressions, such as those entirely consisting of constructors, are evaluated simply by allocating and initializing memory, and may therefore always be evaluated speculatively. The natural next step is to evaluate simple arithmetic expressions speculatively, but here we encounter a problem: Since arithmetic operators are strict, any variables occurring in an arithmetic expression must be evaluated, something that can take an arbitrary amount of time (the same problem occurs with case expressions).

Sometimes one can see that a variable is already evaluated by looking at the scope of the variable. This typically happens for variables with several occurrences, at least one of which is strict. Both the Chalmers LML/Haskell compiler [9] and the Glasgow Haskell compiler [14] include this optimization

In order to go further than this, a global analysis is needed. The first such analysis was defined by Mycroft [10], although only for a first-order language. His analysis is based on abstract interpretation of a denotational semantics and has since been extended to a higher order totality analysis by Abramsky [1]. More recently, Solberg [17] has presented an analysis based on an annotated type system, which can also handle higher-order programs. Neither of these analyses handle data structures like lists or pairs well, and neither of them has been implemented in a compiler (Mycroft's analysis was implemented in an interpreter, though).

In this paper we present a somewhat different approach. Our algorithm takes as input both the program to be optimized and information derived from it by an earlier flow analysis. Thus the cheapness analysis becomes simpler because it does not have to compute the control and data flow of the program. In this way the analysis is applicable to any functional language for which there is a flow analyser, so we could handle for instance a dynamically typed language (which is not possible with the analyses of Mycroft, Abramsky and Solberg).

Any flow analyser can be used together with our cheapness analysis; the correctness of flow information is formulated relative to the operational semantics of the analysed language. Thus we will have very little to say about flow analysis in this paper; we will only use its result, flow information. The flow analyser we use is based on polymorphic subtype inference and is presented in [2].

Both Mycroft's and Solberg's analysers are based on finding when an expression is guaranteed not to diverge. If the conditions (definedness of free variables) are satisfied, these expressions can be evaluated speculatively. Our analysis takes a more direct approach by working over a language where delayed evaluation is explicitly indicated by expressions of the form ${\tt thunk}\,e$ where e is the delayed expression. The analysis then finds some ${\tt thunk}\,e$ expressions which can be replaced by e alone, taking into account all of the global consequences of the replacement.

```
e \in \operatorname{Expr} \to x \mid x_1 x_2 \mid b \mid op^l x_1 \dots x_k \\ \mid \operatorname{case} x \text{ of } alt_1; \dots; \ alt_n \text{ end} \\ \mid \operatorname{let} x = e' \text{ in } e \\ \mid \operatorname{letrec} x_1 = b_1; \dots; x_n = b_n \text{ in } e \\ \mid \operatorname{eval} x \\ b \in \operatorname{Build} \to \lambda^l x. e \mid C^l x_1 \dots x_k \mid \operatorname{thunk}^l e \\ alt \in \operatorname{Alt} \to C x_1 \dots x_k \to e \\ x \in \operatorname{Var} \\ l \in \operatorname{Label} \to \underline{1}, \underline{2}, \dots \\ C \in \operatorname{DataCon} \cup \operatorname{IntLit} \\
```

Figure 1: The syntax of Fleet

Cheap eagerness is a complement to transformations based on strictness analysis [10; 19]. Both techniques allow call-by-value to be used instead of call-by-need, thereby evaluating expressions earlier than in an unoptimized implementation. The difference is in the correctness criteria; where strictness analysis finds expressions that would have been evaluated anyway, cheap eagerness finds expressions which are harmless to evaluate even if they would not have been evaluated under call-by-need. This has the important implication that cheap eagerness can find some opportunities for optimization that strictness analysis can never find, and vice versa.

Replacing call-by-need with call-by-value improves performance in several ways: Fewer thunks need to be constructed and (often) later evaluated and unboxed data representations can be used in more places [12; 4]. In addition, some compiler back-ends have separate eval operations, for instance the Charlmes LML/Haskell compiler [9] and our own compiler [3]. Such an explicit eval operation can be eliminated if it can be proved that its argument will never be a thunk. Eliminating thunks increases the opportunities for this transformation.

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 a source-to-source transformation:

$$\mathtt{thunk}^l \, e \Longrightarrow e$$

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

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 is a set of labels of thunks that may

```
letrec from = \lambda^{\pm}n.let r = thunk<sup>2</sup>

let n1 = thunk<sup>3</sup>

let n'= eval n

in inc<sup>4</sup> n'

in from n1

in let z = 0^{\frac{6}{2}}

in from z
```

Figure 2: The from program

be eliminated.

Figure 2 gives an example Fleet program computing the list of all natural numbers.

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

$$\rho \vdash e \Downarrow v$$

where ρ is an environment mapping variables to values, e is an expression, and v a value. A value is a closure; a pair of an environment and a buildable expression. Note that thunks are also values in this semantics since they are an explicit construct in the language. We refer to closures where the expression part is an abstraction or a constructor application as $weak\ head\ normal\ form\ (whnf)\ closures$.

An unusual feature of our semantics is that we allow values to be infinite (seen as trees where each variable binding in an environment is a branch). See Appendix A for a formal definition of infinite values using path expressions. Infinite values 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 [2; 16; 8; 15]. In our case, flow information takes the form of a flow assignment φ mapping variables 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 the body.

¹In a higher order language these are intertwined since functions are first class data. This is reflected in the nomenclature of flow analysis, as the term control flow analysis is also used [16]. Since data is often represented as closures, both in an implementation and in the semantics, the term closure analysis can also be found in the literature [15].

$$\frac{\rho \vdash e' \Downarrow v' \qquad \rho[x \mapsto v'] \vdash e \Downarrow v}{\rho \vdash \mathsf{let} \ x = e' \ \mathsf{in} \ e \Downarrow v} \qquad \mathsf{let}$$

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

$$\frac{\rho(x) = (\rho', \mathtt{thunk}^l \; e) \qquad \rho' \; \vdash \; e \Downarrow v}{\rho \; \vdash \; \mathtt{eval} \; x \Downarrow v} \qquad \qquad \mathtt{eval}\text{-}\mathrm{i}$$

$$\frac{\rho(x) \text{ is a whnf closure}}{\rho \vdash \text{eval } x \Downarrow \rho(x)}$$
 eval-ii

Figure 3: Operational semantics of Fleet

 $egin{array}{lll} ext{from} &:& \{\lambda^{rac{1}{2}} n. E_1\} \ & ext{n} &:& \{ ext{thunk}^{rac{3}{2}} E_3, & ext{inc}^{rac{4}{2}} ext{n}', & 0^{rac{6}{2}} \} \ & ext{r} &:& \{ ext{thunk}^{rac{2}{2}} E_2, & ext{Cons}^{rac{5}{2}} ext{n} ext{r}\} \ & ext{n} &:& \{ ext{thunk}^{rac{3}{2}} E_3, & ext{inc}^{rac{4}{2}} ext{n}'\} \ & ext{n}' &:& \{ ext{inc}^{rac{4}{2}} ext{n}', & 0^{rac{6}{2}} \} \ & ext{z} &:& \{0^{rac{6}{2}} \} \end{array}$

Abbreviations:

Figure 4: Flow information for from

3. THE ANALYSIS

The first question that must be answered when formalizing the cheapness analysis is what kinds of expressions should be considered expensive or unsafe, thus making them unfit for speculative evaluation.

Calls of recursive functions: Not all recursive functions are expensive. If the recursive call only occurs nested inside the body of a thunk expression, as in the from function above where the recursive call is in the body of the thunk², a call to the function will not lead to a recursive call, only to the construction of a thunk that, if and when it is evaluated, will perform the recursive call. Thus we will distinguish between functions that are lazily recursive (recursive calls only in thunk bodies) and those that are eagerly recursive. A typical eagerly recursive function is length.

Case: If the scrutinized variable in a case expression is bound to a constructor not in any alternative, a runtime error will occur (in the semantics, this shows up as the impossibility of making a derivation of the evaluation relation; there is no rule for pattern matching failure). Fortunately, flow information can be used to find out whether this may happen or not.

Operators: Some operators, for instance division, are undefined for some operands. Other operators, for instance fixed precision integer addition, may overflow. If such overflow is trapped, integer addition is an unsafe operation, but if overflow just wraps around, as we will assume in this paper, it is safe. In both cases, flow information could be used to find out whether an exception is possible or not.

Evaluation: We are going to eliminate all thunks which are cheap and safe. If an eval might actually find its argument a thunk, it is because that thunk is expensive or unsafe. Thus an eval is safe iff every thunk that might be bound to its argument is eliminated, information that is readily available from flow analysis.

It is easy to see that elimination of one thunk may depend on elimination of another (this happens if the body of the first thunk contains an eval which may evaluate closures built with the second thunk). In fact, there can even be circular dependencies. Consider the thunk labeled with $\underline{3}$ in the from program in Figure 2, reproduced below:

thunk
$$\frac{3}{2}$$
 let n'= eval n in inc $\frac{4}{2}$ n'

The body of this thunk is cheap and safe except for the subexpression eval n. The only thunk that can be bound to n is, according to the flow information in Figure 4, the very thunk containing the eval.

In order to understand if, and in that case why, a thunk depending on itself can be eliminated or not, it is important to

 $^{^2}$ Although this is not entirely satisfactory, it is consistent with e.g. the definition of Haskell 98 [11, section 6.4].

note that the necessary condition is that none of the speculatively evaluated expressions in the final program execute an expensive operation, in this case find a thunk in an eval. But if we eliminate the thunk³ in the example, there will be no thunk bound to n in the final program. Thus we are allowed to eliminate that thunk.

A final restriction is that thunks in the right-hand-sides of letrec bindings can not be eliminated since buildable expressions are required in this position. This reflects both the semantics, which puts the right-hand-sides in closures, and real implementations which build cyclic data structures for letrec bindings.

3.1 Formalization

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

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

where S is a set of constraints, φ a flow assignment, e is an expression and l is a label. There are also auxilliary judgements of the form $S \vdash_R^{\varphi} b: l$ which express the restriction that thunks in letrec bindings can not be eliminated.

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

There are three main forms of constraints: $l \leadsto_{\mathsf{c}} l'$, $l \leadsto_{\mathsf{c}} l'$ and $l \leadsto_{\mathsf{t}} l'$. 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 fourth form of constraint is $l \leadsto_{\mathsf{c}} \Omega$, which is used to indicate that the body of the thunk or function labeled with l is expensive or unsafe to evaluate.

Figure 6 shows the constraints S derived from the from program in Figure 2 using the flow assignment φ in Figure 4 (<u>42</u> is an arbitrary label representing the top level context). Thus we have $S \vdash^{\varphi} e : \underline{42}$ where e is the from program.

Definition 1 — (Cost assignment is a function δ mapping labels to costs, which are either natural numbers or the special cost Ω . We will let Δ range over costs.

The cost Ω represents a computation which may loop or raise an exception. The other costs all represent terminating and nonexcepting computations; we call these *finite costs*. The output of the analysis is a cost assignment; all thunks whose labels are mapped to finite costs will be eliminated.

DEFINITION 2. A cost assignment δ is a model of a constraint set S, written $\delta \models S$ iff all of the following holds:

- For every constraint $l \leadsto_{\mathsf{c}} l' \in S$, either $\delta(l) > \delta(l')$ or $\delta(l) = \Omega$. For every constraint $l \leadsto_{\mathsf{c}} \Omega \in S$, $\delta(l) = \Omega$.
- For every constraint $l \rightsquigarrow_{\mathbf{e}} l' \in S$, either $\delta(l) = \Omega$ or $\delta(l') \neq \Omega$.

$$S \vdash^{\varphi} e : l$$
 $S \vdash^{\varphi} x : l$ VAR

$$\frac{S \vdash^{\varphi} e \ : \ l'}{S \vdash^{\varphi} \lambda^{l'} x. e \ : l}$$
 ABS

$$\frac{S \vdash^{\varphi}_{l} \mathsf{IsAbs}(x_{1})}{S \cup \{l \leadsto_{\mathbf{C}} l' \mid \lambda^{l'} x. \, e \in \varphi(x_{1})\} \, \vdash^{\varphi} \, x_{1} \, x_{2} \, : \, l} \qquad \qquad \mathsf{APP}$$

$$\frac{S \vdash^{\varphi}_{l} \mathsf{NoErr}(op \ x_{1} \dots x_{r})}{S \vdash^{\varphi} op^{l'} x_{1} \dots x_{r} \ : \ l}$$
 OP

$$S \vdash^{\varphi} C^{l'} x_1 \dots x_r : l$$
 CON

$$\frac{S \vdash^{\varphi}_{l} \mathsf{O} \, \mathsf{neOf}(x, C_1, \ldots, C_n)}{S \vdash^{\varphi} \mathsf{case} \, x \, \, \mathsf{of} \, \ldots; C_l \, x_{i1} \ldots x_{ir_i} \, \xrightarrow{} \, e_i; \ldots \, \, \mathsf{end} \, : \, l} \quad \text{\tiny CASE}$$

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

$$\frac{S \vdash_{R}^{\varphi} b_{1} : l \quad \dots \quad S \vdash_{R}^{\varphi} b_{n} : l \qquad S \vdash^{\varphi} e : l}{S \vdash^{\varphi} \text{letrec } x_{1} = b_{1}; \dots; x_{n} = b_{n} \text{ in } e : l}$$
 LETREC

$$\frac{S \vdash^{\varphi} e : l'}{S \cup \{l \leadsto_{\mathbf{t}} l'\} \vdash^{\varphi} \mathsf{thunk}^{l'} e : l}$$
 THUNK

$$S \cup \{l \leadsto_{\mathsf{e}} l' \mid \mathtt{thunk}^{l'} e \in \varphi(x)\} \vdash^{\varphi} \mathtt{eval} x : l$$
 EVAL

$$\begin{array}{c|c} S \vdash^{\varphi}_{R} b : l \\ \hline \\ b \text{ is not a thunk} & S \vdash^{\varphi} b : l \\ \hline \\ S \vdash^{\varphi}_{R} b : l \\ \hline \\ S \cup \{l' \leadsto_{\mathbf{C}} \Omega\} \vdash^{\varphi}_{R} \text{ thunk}^{l'} e : l \end{array} \qquad \text{R-WHNF}$$

$$\frac{\forall e \in \varphi(x) \ . \ e \ \text{is an abstraction}}{S \ \vdash_{l}^{\varphi} \ \mathsf{IsAbs}(x)} \qquad \qquad \mathsf{IS \ ABS}$$

$$\frac{\forall e \in \varphi(x). e \text{ is built with one of the } C_i}{S \vdash_l^{\varphi} \mathsf{OneOf}(x, C_1, \dots, C_n)}$$
 ONE OF

$$\frac{op \, x_1 \dots x_r \text{ is defined if the } x_i \text{ are described by } arphi}{S dash_i^{arphi} \, \, \mathsf{NoErr}(op \, x_1 \dots x_r)}$$

$$S \cup \{l \leadsto_{\mathbf{c}} \Omega\} \vdash^{\varphi}_{l} E$$
 Error

Figure 5: Constraint derivation rules

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

Figure 6: Constraints derived from the from program

$$\delta(\underline{1}) = 1$$
, $\delta(\underline{2}) = \Omega$, $\delta(\underline{3}) = 1$, $\delta(\underline{42}) = 2$

Figure 7: Model of the constraints derived from from

• For every constraint $l \leadsto_{\mathsf{t}} l' \in S$, either $\delta(l) > \delta(l')$ or $\delta(l) = \Omega$ or $\delta(l') = \Omega$.

A model δ of a constraint set S is minimal iff $\delta(l) = \Omega$ and $\delta' \models S$ implies that $\delta'(l) = \Omega$ (every label mapped to Ω by δ must be mapped to Ω by any model of S). Note that a minimal model is unique if it exists.

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

Looking at the inference rules, we can see that constraints of the form $l \leadsto_{\mathsf{c}} l'$ 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). 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 $S \vdash_{\mathsf{c}}^{\varphi} E$.

Similarly, $l \sim_e l'$ corresponds 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.

The third kind of constraint, $l \sim_t l'$, 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 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'ètre, 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. 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. Consider the following example:

letrec f =
$$\lambda^{\underline{1}}x$$
.let t = thunk² g x
in Pair³ t t
g = $\lambda^{\underline{4}}y$.let u = thunk⁵ f y
in Pair⁶ u u
in ...

We get the following constraint set (the rather trivial flow information is not shown):

$$1 \sim_{\mathsf{t}} 2$$
, $2 \sim_{\mathsf{c}} 4$, $4 \sim_{\mathsf{t}} 5$, $5 \sim_{\mathsf{c}} 1$

If we try to assign finite costs to all of the labels, we have a cycle of strict inequalities, so we have to assign Ω to either label $\underline{2}$ or label $\underline{5}$. If we choose label $\underline{2}$, we get the following legal cost assignment:

$$\delta(\underline{1}) = 1, \ \delta(\underline{2}) = \Omega, \ \delta(\underline{4}) = 3, \ \delta(\underline{5}) = 2$$

We could equally well have chosen to assign Ω to label $\underline{5}$ and 2 to label $\underline{2}$. Operationally, this is not surprising; it is ok to eliminate one of the thunk expressions in the example since the recursion is indirect. Eliminating both, however, would turn the lazy recursion of the original program into eager recursion.

3.2 Monovariance and Cloning

Analyses where a single property is computed for every program point are generally called *monovariant* (with the opposite being a *polyvariant* analysis). Our cheap eagerness analysis is a monovariant analysis since every label is assigned only one cost. In large programs, this is potentially a serious limitation: Functions called from different sites may be cheap at some sites and expensive at other.

Consider for instance a strict function which is applied to an evaluated argument at some sites and to a thunk at other. Consequently, this function will sometimes evaluate thunks, and the label of its lambda abstraction must therefore be mapped to Ω , making all calls to this function expensive.

The monovariance may even confuse the analyser into believing that more functions are recursive than is actually the case. Consider the following code fragment:

let apx =
$$\lambda^{\frac{1}{2}}$$
f.fx
in let g = apx apx
in ...g...

The flow information for f clearly must include the $\lambda^{\underline{1}}$ f abstraction, but then we will get the constraint $\underline{1} \leadsto_{\mathsf{c}} \underline{1}$ which will force $\delta(\underline{1}) = \Omega$.

One way of dealing with these problems is to clone (make several copies of) each function before the flow analysis (this technique has been used by e.g. Heintze [7] to increase the precision of a monovariant flow analysis). We clone global let and letrec bindings where all right-hand-sides are lambda abstractions. Our cloning strategy is deep in the sense that we apply cloning to the body of a let(rec) expression before making one copy of the bindings for every occurrence of any of the bound variables in the (cloned) body. The alternative is shallow cloning where one copy is made for every occurrence in the original body. Deep cloning may lead to exponential code growth, but guarantees that there is only one nonrecursive occurrence of a variable bound in a clonable let(rec). Cloning of the expression above yields

```
Solve
       for each l \in \mathsf{LabAbs} do
               if l \leadsto_{\mathsf{c}}^+ l or l \leadsto_{\mathsf{c}} \Omega then \delta(l) := \Omega
       Propagate-\Omega
       for each l \in \mathsf{LabThunk} do
                R_l := \{l\} \cup \{l' \mid l' \in \mathsf{LabThunk} \text{ and } l \leadsto_{\mathsf{ce}}^+ l'\}
                if there is a cycle C in the \rightarrow_{ct} subgraph such that
                       C \subseteq R_l \cup \mathsf{LabAbs}
                then \delta(l) := \Omega
       loop
                {\sf Propagate-}\Omega
                for each l \leadsto_{\mathsf{t}} l' do
                       if \delta(l') = \Omega then remove l \leadsto_{\mathsf{t}} l' from the graph
                if there is an l \rightsquigarrow_{\mathsf{t}} l' such that l' \rightsquigarrow_{\mathsf{ct}}^+ l
                       then \delta(l') := \Omega
                        else exit loop
       end loop
       assign finite costs to all labels such that \delta(l) \neq \Omega
Propagate-\Omega:
       for each \it l do
               if l \rightsquigarrow_{\mathsf{ce}}^+ l' and \delta(l') = \Omega then \delta(l) := \Omega
```

Figure 8: Constraint solving algorithm

```
let apx = \lambda^{\frac{1}{2}}f.fx
in let apx' = \lambda^{\frac{2}{2}}f'.f'x
in let g = apx apx'
in ...g...
```

if x is not bound in a clonable let(rec). Now the flow information for f includes the $\lambda^2 f'$ abstraction, so we get $\underline{1} \leadsto_c \underline{2}$ instead.

3.3 Implementation

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

Due to the nonmonotonicity of the $\sim_{\rm t}$ constraints, there is room for some cunning in the constraint solving algorithm. There is sometimes a choice of which thunk labels to map to Ω in order to avoid turning lazy recursion into eager recursion. In these cases, it is best to choose labels that must be mapped to Ω in any model of the constraints. We call such labels forced labels. If we succeed in solving the constraints by mapping only forced labels to Ω , the constraints have a minimal model and we have found it. The algorithm we present below always finds the minimal model if one exists.

When discussing this algorithm, it will be helpful to view the constraints as defining a graph with labels and Ω as nodes and three different kinds of edges corresponding to the three types of constraints³ (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_{c} edges or only the \leadsto_{c} and \leadsto_{e} edges. We will refer to these as the \leadsto_{c} and the \leadsto_{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_{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 Figure 8. The general strategy is to systematically find the forced labels in the constraint set, using the following observations:

- Cycles in the ~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 →_c l' or l →_e l' and l' is mapped to Ω, then l
 must also be mapped to Ω. The Propagate steps are
 motivated by this fact.
- If C is a cycle in the \sim_{ct} subgraph, we must map some of the thunk labels in C to 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 \sim_{ce} subgraph, it follows that l must be mapped to Ω .

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:

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

We must map either $\underline{2}$ or $\underline{3}$ to Ω . If we choose $\underline{3}$, we will soon find that we will also have to map $\underline{2}$ to Ω because of the constraint $\underline{2} \sim_{\mathsf{e}} \underline{3}$, 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 Ω there may still be cycles in the $\sim_{\rm ct}$ graph which have not yet been broken by mapping some of the thunk labels to Ω . We then pick some arbitrary thunk labels from such cycles and map them to Ω . When no cycles are left, all nodes not mapped to Ω can be mapped to finite costs in a single bottom-up traversal. As we have discussed above, cycles involving $\sim_{\rm e}$ edges do not necessarily force any labels to Ω .

Lemma 1. For every constraint set S, Solve computes a δ such that $\delta \models S$. Furthermore, if S has a minimal model, then δ is this minimal model.

Proof: We will not give a formal proof of the first part of the lemma, since we think that the algorithm is relatively straightforward. The second part is less obvious, though. Clearly, if the loop ... end loop exits on its first iteration, the algorithm will only map forced labels to Ω and hence δ will be the minimal model of S. We will now prove that if this does not happen, then S has no minimal model.

When the algorithm reaches the if in the first iteration of the loop ... end loop we have the following situation: Take

³The choice of symbols for the constraints was motivated by this analogy.

any thunk label l which is not yet mapped to Ω . For every cycle C in the \leadsto_{ct} subgraph there is some thunk label $l' \in C$ (distinct from l) which is not reachable from l in the \leadsto_{ce} subgraph (otherwise C would have been contained in $R_l \cup \mathsf{LabAbs}$ and l would have been mapped to Ω). Thus we can break C by mapping l' to Ω without having to map l to Ω . This means that we can pick an arbitrary label l and break every cycle in the \leadsto_{ct} subgraph without mapping l to Ω .

If the algorithm does not exit the loop ... end loop at this point, there is still some (unbroken) cycle in the \sim_{ct} subgraph. Thus any model δ of S maps at least one more label l to Ω , but because of the above argument, we know that there must also exist some other model δ' of S which does not map l to Ω . Hence δ is not minimal.

4. EXPERIMENTAL RESULTS

We have implemented 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 [2]. Except being used for cheap eagerness, the flow information is exploited by update avoidance and representation analysis [4]. Since several optimizations are programwide, separate compilation is not supported. The compiler is described more fully in the author's PhD thesis [3, 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 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.
- qf Same as q1, but with deforestration (see Wadler [18]) applied (also manually), run with input 11.
- sort Sorts a list of 6000 integers using insertion sort.
- factor Factorizes a number into primes in a very inefficient way, run with input 1073602561.
- event A small event driven simulator from Hartel and Langendoens benchmark suite [6], run with input 400000.

- sched A job scheduler, also from that benchmark suite, run with input 12.
- tc A simple polymorphic type checker from the same source⁴, run with input 600.

There are two sets of measurements, with and without cloning (see Section 3.2). The programs which only occur without cloning were not affected by it; the numbers are all the same.

In Table 1 we have measured the execution of the programs with different levels of aggressiveness for the cheap eagerness optimization:

- No cheap eagerness at all, except for buildable expressions.
- All evals and function calls (except partial applications) considered expensive. This option was implemented by marking the label of every thunk body containing an eval and the label of every lambda abstraction whose body was not another abstraction with Ω. This alternative corresponds roughly to what one can accomplish without a special cheap eagerness analysis.
- As above, but without the restriction on evals. This alternative corresponds roughly to the analysis (very) briefly presented in [2].
- Full cheap eagerness as described in Section 3.3.

While one must be very careful in drawing conclusions from such a small set of not-very-large programs, it seems clear that cheap eagerness pays off. For no program does execution time or instruction count increase. The benefit varies from essentially none (sort), where the analysis does not find any important cheap and safe thunks to eliminate, to dramatic (better than a factor of two) for qf which is dominated by a tail recursive function (inner loop) with a non-strict, accumulating parameter.

The larger programs, event, sched and tc, all get some benefit, with a reduction in the number of executed instructions varying from 5% (sched) to 26% (event). The smaller reductions for these programs have two causes; first, they excute call-by-need-related operations less frequently than sort and the N-Queens variants, as can be seen from the leftmost set of columns, and second, a smaller part of those operations are eliminated. The latter effect is partially due to monovariance, as the improvement achieved by cloning⁵ shows.

The improvements in instruction count and execution time are due not only to the thunk allocations avoided, but also to

⁴It is called typecheck there.

⁵ Although the cloned programs execute fewer instructions, their run-times are longer. This effect is due to an increase in the number of misses in the instruction cache, caused by the enormous code growth associated with our aggressive cloning. We have recently solved this problem [5], but not in time to rerun all the tests. The cloned versions are now faster than those without cloning.

	No cheap						All evals expensive							Some calls cheap										
	Th	$\mathbf{E}\mathbf{v}$	$\mathbf{E}\mathbf{c}$	\mathbf{Co}	In	\mathbf{Ti}	Th	$\mathbf{E}\mathbf{v}$	\mathbf{Ec}	\mathbf{Co}	In	\mathbf{Ti}	Th	$\mathbf{E}\mathbf{v}$	\mathbf{Ec}	Co	In	Ti	Th	$\mathbf{E}\mathbf{v}$	\mathbf{Ec}	\mathbf{Co}	In	\mathbf{Ti}
<u> </u>	Without cloning																							
nqh	12	24	11	27	777	3.6	47	58	52	37	71	70	47	58	52	37	71	70	47	58	52	37	71	70
q1	13	14	11	25	541	2.3	32	28	36	0	58	55	32	28	36	0	58	55	32	28	36	0	58	55
qf	11	13	9	30	432	1.7	4	4	5	0	48	42	4	4	5	0	48	42	4	4	5	0	48	42
sort	12	35	12	23	778	5.4	100	100	100	100	100	99	100	100	100	100	100	99	100	100	100	100	100	99
factor	6	18	6	12	90	0.5	100	100	100	100	100	100	99	33	99	0	92	91	99	33	99	0	92	93
event	9	18	8	12	504	3.9	67	90	61	44	84	83	63	75	55	3	74	72	63	75	55	3	74	73
sched	11	22	8	7	108	0.6	98	100	98	98	99	98	95	98	93	75	97	98	90	93	88	75	95	94
tc	8	35	7	11	913	6.7	89	68	89	100	92	79	86	67	86	100	91	74	79	64	77	100	89	72
With deep cloning																								
nqh	13	22	11	18	724	3.4	47	64	52	0	72	67	43	24	48	0	58	57	43	24	48	0	58	57
sort	12	35	12	23	776	5.4	100	100	100	100	100	99	100	100	100	100	100	99	100	100	100	100	100	99
event	11	19	9	14	437	3.8	67	88	61	41	83	82	63	72	55	2	77	80	61	69	54	2	77	76
sched	11	20	8	7	108	0.9	98	100	98	98	99	102	95	93	93	86	96	108	81	85	75	85	91	88
tc	9	35	7	11	843	7.1	89	69	89	4	90	91	86	61	85	0	87	91	73	54	70	0	84	80

No cheap: Number of Thunks built and Evals, Ecalls (evals applied to thunks) and Coercions (boxings and unboxings) executed per 1000 instructions executed, followed by millions of Instructions executed and execution Time in seconds with cheap eagerness disabled.

Other options: Relative operation and instruction counts as well as relative execution time, all as perentages:

 $\frac{\text{Optimized operation count or time}}{\text{Unoptimized operation count or time}} \times 100$

Table 1: Different versions of cheap eagerness (with strictness analysis)

the enabling of other optimizations. In particular, eliminating thunks often enables the elimination of eval operations (which can be elided if the compiler can prove that the argument will always be a whnf), and it also makes it possible to use unboxed data representations more often.

Interestingly, the reduction in execution time is (almost) always larger than the reduction in instruction counts, indicating that "expensive" instructions are eliminated.

If we compare the reductions in the number of thunks built and the number of thunks evaluated, we see that cheap eagerness finds thunks which have an about average probability of being evaluated; it does not just find thunks which would not have been evaluated anyway. Combining this information with a comparison of the ratios of thunks built to instructions executed and thunks evaluated to instructions executed, we see that both before and after optimization, most thunks are evaluated. We also see that the reduction in instruction counts increase when we move towards more aggressive optimization. Speculative evaluation pays off even for function calls!

We have also compared strictness analysis and cheap eagerness since they are both optimizations which replace call-by-need with call-by-value. The results are shown in Table 2. The strictness analysis which is turned on or off is the interprocedural part of the analysis described in [3]. With the analysis turned off, all functions are considered nonstrict. Strictness is still propagated intraprocedurally; a let where

the bound variable is used strictly in the body will *not* get a thunk wrapped around the right hand side.

The baseline of the comparison is the performance with neither strictness analysis nor cheap eagerness turned on, and the other alternatives are with only strictness analysis, only cheap eagerness or both turned on (the cheap eagerness is the full version presented in the rightmost column in Table 1).

The results indicate that cheap eagerness makes a bigger difference in instruction count than strictness analysis for all programs except sched. This is a slightly surprising result, given that cheap eagerness is so much less studied in the literature. One must not forget, however, that in these tests we are pitting a very simple strictness analyser against a rather sophisticated cheapness analysis (especially when the cheapness analyser is helped by cloning).

5. RELATED WORK

Earlier work on cheap eagerness basically falls in to groups. First, there is the work on termination (later totality) analysis, pioneered by Mycroft [10], where the objective is to formulate an analysis which is able to guarantee that the semantics of an expression is not \bot . Mycroft's first-order analysis was later extended to a higher-order language by Abramsky [1] and subsequently reformulated as a totality type system by Solberg [17].

None of these analyses handle lazy data structures; the base

	Neither cheap nor strict						Strict but not cheap						Ch	ict	Cheap and strict									
	Th	$\mathbf{E}\mathbf{v}$	$\mathbf{E}\mathbf{c}$	\mathbf{Co}	In	\mathbf{Ti}	Th	$\mathbf{E}\mathbf{v}$	\mathbf{Ec}	\mathbf{Co}	In	Ti	Th	$\mathbf{E}\mathbf{v}$	$\mathbf{E}\mathbf{c}$	\mathbf{Co}	In	Ti	Th	$\mathbf{E}\mathbf{v}$	\mathbf{Ec}	\mathbf{Co}	In	Ti
	Without cloning																							
nqh	13	27	12	25	861	4.2	80	80	78	97	90	87	37	48	40	37	64	60	37	47	40	35	64	61
q1	15	17	13	31	631	2.8	75	75	73	69	86	83	27	27	30	2	51	48	24	21	26	0	50	45
qf	13	16	12	41	518	2.2	69	70	65	61	83	77	5	7	6	10	42	35	3	2	3	0	40	32
sort	13	37	13	25	744	6.0	99	99	99	99	104	90	98	99	98	98	99	98	98	99	98	98	104	89
factor	6	19	6	13	87	0.5	100	100	100	100	103	98	99	33	99	100	93	98	98	33	98	0	95	91
event	11	17	10	17	688	5.1	63	76	59	53	73	77	49	61	44	2	57	57	40	56	33	2	54	56
sched	13	26	11	6	142	0.9	65	62	57	99	76	70	94	96	93	102	97	93	59	58	50	74	73	66
tc	9	31	8	10	1114	7.4	72	92	68	90	82	91	83	56	82	90	78	74	57	59	52	90	73	65
With deep cloning																								
nqh	14	25	13	23	838	3.7	80	79	78	67	86	91	36	22	39	2	51	50	35	19	37	0	50	52
sort	13	37	13	25	744	6.0	99	99	99	99	104	90	98	99	98	98	99	99	98	99	98	98	104	89
event	13	21	12	21	568	5.4	63	69	59	51	77	71	49	60	44	1	64	58	39	48	32	1	59	53
sched	13	26	11	6	142	1.2	65	59	57	87	76	69	90	94	88	102	96	88	53	50	43	74	69	61
tc	11	34	9	11	973	9.9	72	90	68	87	87	72	78	53	77	0	79	65	53	49	48	0	72	57

Neither cheap nor strict: Number of Thunks built and Evals, Ecalls (evals applied to thunks) and Coercions (boxings and unboxings) executed per 1000 instructions executed, followed by millions of Instructions executed and execution Time in seconds with cheap eagerness and strictness analysis disabled.

Other options: Relative operation and instruction counts as well as relative execution time, all as percentages:

 $\frac{\text{Optimized operation count or time}}{\text{Unoptimized operation count or time}} \times 100$

Table 2: Comparing cheap eagerness with strictness analysis

domains are all flat. Very little is also said about how the information is to be used to transform programs. In the case of recursive programs, it is far from clear how to go from an analysis result to a correct program transformation. Consider the from program: If we have non flat domains for lists, we clearly have $\mathcal{E}[\texttt{from }e]\rho \neq \bot$ for all expressions e and environments ρ mapping from to the semantics of its definition, but evaluating the recursive call speculatively still changes the termination behaviour of the program. This is an example of a situation where a locally meaning preserving transformation changes the semantics of a program on the global level, a problem that has been studied by for instance Sands [13].

On the other hand, we have the more practically oriented work of Johnsson [9] and Santos [14]. They both exploit cases where all of the free variables of an expression have been evaluated because of strict occurrences not in the speculated expression. Consider e.g. the expression $\mathbf{x+g(x+1,y)}$ where \mathbf{x} is clearly used strictly. Since \mathbf{x} must be evaluated anyway, the other occurrence of \mathbf{x} requires no further evaluation, making $\mathbf{x+1}$ a cheap and safe expression. Johnsson is also able to speculate some expressions conditionally based on run-time information, a generalization we believe might be worthwile also in our case. It is however orthogonal with respect to the use of global information.

It would be interesting to see the performance improvements they get, but unfortunately neither author present numbers for this particular transformation.

6. CONCLUSIONS AND FURTHER WORK

We have presented what to our knowledge is the first global cheapness analysis actually implemented in a compiler. Our experimental results, although they are based on not-very-large programs, indicate that cheap eagerness can cut instruction counts significantly and that our global analysis is a clear improvement on purely local transformations.

The analyser can handle data structures and higher order functions and does not depend on the intermediate language being typed. It is not perfect, however. The main disadvantage is that the analysis is not compositional; separate compilation is impossible. A compositional analysis is clearly desirable, and is sketched in the author's PhD thesis [3, chapter 6].

An especially encouraging aspect of our results is that the more willing the optimizer is to consider thunks cheap, the larger is the performance improvement. This indicates that it would be useful to consider even more aggressive versions of cheap eagerness, especially as most thunks still remain in several programs.

 Currently, we leave a thunk if eliminating it would change lazy recursion into eager recursion. This is necessary to avoid an unbounded amount of extra work; if we could bound this amount, we could lift the restriction. One way of achieving this is to maintain a speculation level as a new abstract machine register, initialized to a small positive integer (e.g. 4). When reaching such a cycle-breaking thunk, the speculation counter is checked: If it is positive, it is decremented and the thunk body is speculatively evaluated (and then the speculation counter is incremented), otherwise a thunk is built.

- Thunks which contain operations which may cause runtime errors, such as divides where the divisor can not be proved to be nonzero, are never speculated in our current system. If the dangerous arguments are available (i.e. as free variables of the thunk), we could test whether for instance a divisor really is zero and only build a thunk in that case.
- If we speculate more thunks conditionally in this way, many evals which we can not prove will never see a thunk, will see a whnf most of the time. Thus we could check if the argument to an eval in a thunk is a whnf and only build the thunk in that case.

Given the large part of the thunks built by a program that are also evaluated, and the ideas above, we think that cheap eagerness, already quite worth while, has a lot more to give in the future!

7. ACKNOWLEDGEMENTS

Thanks to Alan Mycroft for an enlightning email discussion about cheap eagerness and to all of those who tried to help me find out who coined the term "cheap eagerness" (it surfaced some years after Alan's work). If you know who it was, please send me an email!

8. REFERENCES

- S. Abramsky. Abstract interpretation, logical relations and Kan extensions. *Journal of Logic and Computation*, 1(1):5-39, 1990.
- [2] 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.
- [3] Karl-Filip Faxén. Analysing, Transforming and Compiling Lazy Functional Programs. PhD thesis, Department of Teleinformatics, Royal Institute of Technology, June 1997.
- [4] 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.
- [5] Karl-Filip Faxén. The costs and benefits of cloning in a lazy functional language. To appear in: Draft Procs, 2nd Scottish Workshop on Functional Programming (Eds. Stephen Gilmore and Kevin Hammond), July 2000
- [6] Pieter Hartel and Koen Langendoen. Benchmarking implementations of lazy functional languages. In Functional Programming & Computer Architecture, pages 341-349, Copenhagen, June 93.

- [7] Nevin Heintze. Set-based analysis of ML programs. In Proc. ACM Conference on LISP and Functional Programming, 1994.
- [8] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Principles of Programming Languages*, 1995.
- [9] T. Johnsson. Efficient compilation of lazy evaluation. In M. Van Deusen, editor, Compiler construction: Proceedings of the ACM SIGPLAN '84 symposium (Montreal, Canada, June 17-22, 1984), volume 19(6) of ACM SIGPLAN Notices, pages 58-69, New York, NY, USA, June 1984. ACM Press.
- [10] Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of the* 4th International Symposium on Programming, pages 269–281. Springer Verlag, April 1980. LNCS 83.
- [11] Simon Peyton Jones and John Hughes. Report on the programming language Haskell 98. Downloadable from www.haskell.org, February 1999.
- [12] Simon L Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In John Hughes, editor, FPCA '91, pages 636-666. Springer Verlag, 1991. LNCS 523.
- [13] David Sands. Total correctness by local improvement in the transformation of functional programs. ACM Transactions on Programming Languages and Systems, 18(2):175–234, March 1996.
- [14] André Santos. Compilation by Transformation in Non-Strict Functional Languages. PhD thesis, Glasgow University, Department of Computing Science, 1995.
- [15] Peter Sestoft. Analysis and efficient implementation of functional programs. PhD thesis, DIKU, University of Copenhagen, Denmark, October 1991.
- [16] 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.
- [17] Kirsten Lackner Solberg. Strictness and totality analysis with conjunction. In *Proceedings of TAPSOFT'95*, pages 501–515. Springer Verlag, 1995. LNCS 915.
- [18] Philip Wadler. Deforestration: Transforming programs to eliminate trees. In Harald Ganzinger, editor, ESOP '88, pages 344–358. Springer Verlag, 1989.
- [19] Philip Wadler and R.J.M. Hughes. Projections for strictness analysis. In Functional Programming & Computer Architecture, Portland, Oregon, September 1987.

$$\begin{array}{c} \rho \vdash^{\varphi} e' \Downarrow \text{ wrong} \\ \hline \rho \vdash^{\varphi} \text{ let } x = e' \text{ in } e \Downarrow \text{ wrong} \end{array} \qquad \text{w-let} \\ \\ \frac{\rho \vdash^{\varphi} e \Downarrow v}{\rho \vdash^{\varphi} \text{ thunk}^{l} e \Downarrow v} \qquad \text{spec-thunk} \\ \\ \underline{\rho(x) = (\rho', b) \qquad b \not :/ \varphi(x)}_{\rho \vdash^{\varphi} e \Downarrow \text{ wrong}} \qquad \text{wrong} \end{array}$$

 $b::B \text{ iff } b \in B \text{ or } b = C^l \text{ and } op^l x_1 \dots x_r \in B$

Figure 9: Extra rules for checking flow information

APPENDIX

A. INFINITE VALUES

The semantics uses infinitely nested closures. We will here give a formal definition of these. Let \vec{x} range over finite sequences of variables and let $x \cdot \vec{x}$ be the x followed by the sequence \vec{x} and $\vec{x} \cdot x$ be the sequence \vec{x} followed by x. The empty sequence is written as ϵ .

DEFINITION 3. A value is a function f from sequences of variables to buildable expressions such that if $f(\vec{x}) = b$ then $f(\vec{x} \cdot x)$ is defined iff $x \in FV(b)$. Further, $f(\epsilon)$ is always defined and if $f(\vec{x} \cdot x)$ is defined, then so is $f(\vec{x})$ (the domain of f is closed under taking of prefixes).

If ρ is an environment and $FV(b) \subseteq Dom(\rho)$ we will write (ρ,b) for the value f such that $f(\epsilon)=b$ and $f(x\cdot\vec{x})=\rho(x)(\vec{x})$ iff $x\in FV(b)$ and $\vec{x}\in Dom(\rho(x))$.

The following lemma, which we state without proof, tells us that the semantics of letrec expressions makes sense since the equation which occurs there has a unique solution.

Lemma 2. An equation of the form

$$\rho' = \rho[x_1 \mapsto (\rho', b_1), \dots, x_n \mapsto (\rho', b_n)]$$

has a unique solution ρ' given by

$$\rho'(x)(\vec{x}) = \rho(x)(\vec{x}) \quad \text{if } x \notin \{x_1, \dots, x_n\}$$

$$\rho'(x_i)(x \cdot \vec{x}) = \rho'(x)(\vec{x}) \quad \text{if } x \in FV(b_i)$$

$$\rho'(x_i)(\epsilon) = b_i$$

where the induction in the middle line is well-founded since \vec{x} is always of finite length.

B. SAFE FLOW ASSIGNMENTS

In this Appendix we make good on our promise to give a formal definition of correct flow information by defining an extension to the operational semantics of Fleet presented in Section 2. This new semantics, which consists of all of the rules in the standard semantics plus three new rules (given in Figure 9) is parameterized with respect to a flow assignment φ . Thus the judgements in the extended system have the form

$$\rho \vdash^{\varphi} e \Downarrow v$$

where ρ , e and v are as in the original system except that there is a distinguished error value wrong, which is generated and propagated by the new rules.

The flow assignment is checked in the [wrong] rule. If the value of any variable in the environment fails to match the flow information for this variable, wrong is generated and propagates through the [w-let] rule to the root of the derivation. The matching predicate, ::, is satisfied if the expression part of the closure is included in the flow information or if the expression part is a constructor produced by an operator application (which imparted its label to the constructor) included in the flow information.

The [spec-thunk] rule is needed since we will speculatively evaluate a thunk body which would not have been evaluated in the original program. We must thus catch any violations of the flow assignment during the evaluation of the thunk body. We must also take account of the fact that when we have eliminated a thunk, some variables which had been originally bound to the thunk may now become bound to the value the thunk body evaluates to, so we must not accept as safe flow information which does not take this possibility into account.

DEFINITION 4. We say that φ is safe with respect to ρ , e iff $\rho \vdash^{\varphi} e \downarrow$ wrong is not derivable in the extended system.

C. CORRECTNESS

This section sketches a correctness proof for the cheap eagerness transformation. We do not give all the details of the proof, but we have chopped the proof into small enough pieces that we hope that the interested reader can reconstruct it.

We will need a number of definitions for stating the correctness result and its supporting lemmas. First, a notation for the transformation to perform:

Definition 5. We will write $e \setminus \delta$ where δ is a cost assignemnt, for the expression e with any occurrences of a subexpression thunk e' such that $\delta(e') \neq 0$ replaced by e'.

A program will not compute exactly the same value after cheap eagerness as before since some thunks will be replaced by weak head normal forms. Rather, the values will be related by the following relation:

Definition 6 (Approximation). We say that a value (ρ, b) i, δ -approximates a value (ρ_*, b_*) , written $(\rho, b) \prec_{\delta}^{i} (\rho_*, b_*)$ iff

- i = 0, or
- $b \setminus \delta = b_*$ and $b_* \setminus \delta = b_*$ and $\rho \prec_{\delta}^{i-1} \rho_*$, or
- $b = \operatorname{thunk}^l e \text{ and } \delta(l) \neq \Omega \text{ and if } \rho \vdash e \Downarrow v \text{ then } v \prec_{\delta}^i (\rho_*, b_*).$

We say that ρ i, δ -approximates ρ_* iff $Dom(\rho) = Dom(\rho_*)$ and $\rho(x) \prec_{\delta}^i \rho_*(x)$ for every $x \in Dom(\rho)$.

Further, we say that v δ -approximates v_* , written $v \prec_{\delta} v_*$ iff for all i > 0 $v \prec_{\delta}^i v_*$ and similarly for $\rho \prec_{\delta} \rho_*$.

DEFINITION 7 (δ -FREENESS). We say that an environment ρ_* is i, δ -free iff i = 0 or, for every $x \in Dom(\rho_*)$ there are b_* and ρ'_* such that $\rho_*(x) = (\rho'_*, b_*)$ and ρ'_* is $(i \perp 1), \delta$ -free and $b_* \setminus \delta = b_*$. We say that ρ_* is δ -free if for all i > 0 ρ_* is i, δ -free.

Note the use of induction in these definitions. This is necessary since closures may be infinitely nested. Therefore, we will typically rely on induction over i in $v \prec_{\delta}^{i} v_{*}$. We will follow the convention of using v_{*}, ρ_{*}, \ldots for more more evaluated values.

Definition 8. We will write $\delta \models^{\varphi} e : \Delta$ to mean that

- there are S and l such that $S \vdash^{\varphi} e : l$ and $\delta \models S$ and $\delta(l) = \Delta$, and
- for every $x \in Dom(\varphi)$ and $e \in \varphi(x)$ there are S and l such that $S \vdash^{\varphi} e : l$ and $\delta \models S$.

When evaluating an optimized program, we do of course not get a completely different value, a fact we formalize in the lemma below. If we had used a denotational semantics instead of an operational one, we would have gotten this result for free, as it where. The result is really very obvious; if we speculate some thunks and the evaluation of the program still terminates, we will get essentially the same value.

LEMMA 3 (MONOTONICITY). If $\rho \vdash e \Downarrow v$ and $\rho \prec_{\delta} \rho_*$ and $\rho_* \vdash e \setminus \delta \Downarrow v_*$, then $v \prec_{\delta} v_*$.

Proof sketch: By induction on the height of the derivation of $\rho_* \vdash e \setminus \delta \Downarrow v_*$.

We are now ready to state the crucial lemma; that an expression the analysis thinks is cheap and safe really is cheap and safe. The condition that the environment be δ -free is related to correctness of eliminating sets of mutually evaluating thunks.

LEMMA 4 (TERMINATION). If φ is safe with respect to ρ , e and $\delta \models^{\varphi} e : \Delta$, where $\Delta \neq \Omega$, and ρ_* is δ -free then $\rho_* \vdash e \setminus \delta \Downarrow v_*$ for some value v_* .

Proof sketch: By induction on k in $\delta \models^{\varphi} e : k$ and on the size of e.

We finally arrive at the main correctness lemma, which says that if we eliminate some thunks which the analysis thinks are safe to eliminate, and if the original program terminated, then the new one will terminate too.

LEMMA 5. If φ is safe with respect to ρ , e and $\delta \models^{\varphi} e : \Delta$ and $\rho \vdash e \Downarrow v$ and $\rho \prec_{\delta} \rho_*$ then $\rho_* \vdash e \setminus \delta \Downarrow v_*$ for some value v_* .

Proof sketch: By induction on the height of the derivation of $\rho \vdash e \Downarrow v$. The proof uses both of the above lemmas. \square

COROLLARY 6. If e is closed and φ is safe with respect to $[\]$, e and $\delta \models^{\varphi} e : \Delta$ and $[\] \vdash e \Downarrow v$ then $[\] \vdash e \backslash \delta \Downarrow v_*$ for some v_* such that $v \prec_{\delta} v_*$.

We will now turn our attention to ensuring that the conditions in the above lemma can be met. In particular, our aim is to prove that the analysis computes a δ such that $\delta \models^{\varphi} e : \Delta$ for some cost Δ . To do this, we need an additional condition on the flow assignment φ .

DEFINITION 9. If e is a closed expression we say that φ is parsimonious with respect to e iff for every $x \in Dom(\varphi)$ and $e' \in \varphi(x)$ we have that e' is a subexpression of e.

Note that it is easy to see that for every closed expression e and flow assignment φ which is safe with respect to e, [] (the empty environment) there is a pasimonious and safe flow assignment φ' with the same domain as φ and such that $\varphi'(x) \subseteq \varphi(x)$ for all $x \in Dom(\varphi)$.

LEMMA 7. If e is a closed expression and φ is parsimonious with respect to e and $S \vdash^{\varphi} e : l$ and $\delta \models S$, then $\delta \models^{\varphi} e : \delta(l)$.

Proof: Since every expression e' in the range of φ is a subexpression of e and $S \vdash^{\varphi} e : l$ we have $S \vdash^{\varphi} e' : l'$ where l' is the label of immediately enclosing thunk or function body of e' in e.

LEMMA 8 (CORRECTNESS). If e is a closed expression, φ is parsimonious with respect to e and safe with respect to e, [], l is an arbitrary label not occurring in e, $\delta = \mathsf{Solve}(S)$, $S = \mathsf{Constraints}(\varphi, l, e)$ and [] $\vdash e \Downarrow v$ then [] $\vdash e \backslash \delta \Downarrow v_*$ for some v_* such that $v \prec_{\delta} v_*$.

Proof: Combine Lemma 1, Lemma 7 and Corollary 6.