Proposal[draft]: Optimization by symbolic evaluation based program construction

December 2, 2015

Andras Slemmer

1 Overview

1.1 Idea

Traditional forward and backward analysis based optimizations do the following:

- Take a piece of code (e.g. function body) C and perform the given analysis, resulting in some information I
- Use I to transform (and justify transforming) C into more efficient code C'

The project's aim is to explore the feasibility of gaining enough information in I to enable the *construction* of some new code D that has the same semantics as C. The idea is to do this by capturing the *effect* of the reduction of C by doing symbolic evaluation, extracting the semantically relevant parts of it and constructing code that produces that same effect. The idea is that by doing so we can get rid of semantically unnecessary operations that may be present in C without having to recourse to specific types of analyses.

1.2 Motivation

Subsumption of existing optimizations

The optimization would therefore have two parts: symbolic evaluation (code $\rightarrow$ effect) and construction (effect $\rightarrow$ code), and it can be tuned by changing what the input is and what we do with the effect before construction.

Existing optimizations like strictness analysis/demand analysis, inlining or compile time garbage collection can be seen as different parameterizations.

- Strictness/demand analysis

  Strictness analysis allows compilers of lazy languages to detect possibilities of eager argument evaluation. This has the benefit that it allows the compiler to avoid "boxing" the value(allocating a thunk for it in the heap) and pass the value directly.

  We can detect these opportunities by doing symbolic evaluation that calculates what terms will definitely be reduced in the expression(code $\rightarrow$ effect). Then we can construct code that does precisely the same reductions, only this time we are in control of when and in what order these should happen for maximum efficienct. We will explore this idea later a bit more formally.

  (Cite Mycroft (Cousots?)(strictness), Peyton Jones(demand).)

- Inlining

  Because we are doing symbolic evaluation our reduction will stop and one point or another because of lack of information. An example of this is when we want to reduce a symbol that is not in scope of the evaluation. In these cases all we can do is reconstruct the original reduction. However by simply extending the input with information about symbols in scope we allow the symbolic evaluation to continue, and in effect inline the symbol.

- Compile time garbage collection

  If our runtime is garbage collected we will most probably want an intermediate representation that can directly manipulate the garbage collected heap. The effect of the reduction of this representation is then going to be allocations, dereferences etcetera. What we can do is before we do construction of the new code we can garbage collect the symbolic heap by looking at what references are in scope at the end. This way unneeded heap operations will not be present in the new code.

Lower coupling of responsibilities in compilers

Because our optimization constructs code on the same abstraction level as it started from it can be seen as an automorphism on a given intermediate representation. If we were successful in defining a general framework for evaluation-construction then compilation would become a matter of choosing the right intermediate representations until we reach machine code. The compiler writer can focus on the correctness of each translation step without having to worry about generating optimal code. Optimization would become a decoupled operation that may be performed on each representation.

Possibility of dynamic optimization

If we integrate this optimization into the runtime system we can do the same analysis extended with dynamic information about the runtime state (e.g. instead of starting with an unknown heap extend it with available information), possibly resulting in the construction of code that is specifically catered for the runtime environment at hand. An example of this would be a long running program that initially loads a configuration. Statically we have no way of knowing what the configuration is going to be, and we might miss out on chances to prune whole parts of the program (if the configuration affects branching), however at runtime we detect these long-residing objects and inline them dynamically.

2 Examples of desired results

In the following I will try to demonstrate what I mean by *effect* and *construction.* Our goal will be to take a slightly modified version of the lazy lambda calculus presented in Launchbury's seminal paper[1] and extend it so that it is capable of expressing optimized code in terms of strictness and unboxing.

We will then proceed to give operational semantics for the calculus itself and afterwards define symbolic evaluation that calculates the effect of the reduction, which, unsurprisingly, will closely follow the structure of the operational semantics. Once we have an understanding of the effect we can define the construction of optimized code producing the same effect.

Please note that my intention with this example is to demonstrate the idea rather than to provide an actual solution to the strictness optimization problem.

2.1 The calculus

The goal of strictness analysis is to allow strict evaluation of arguments to a lambda and avoid unnecessary creating of thunks in the heap. In order to capture this we need to introduce syntactic constructions that allow explicit control over reduction to normal form and allow argument passing without sharing. We will do this by introducting *strict* and *unboxed* application

$$M, N ::= x|\lambda x.M|M\ N|M\ \#\ N|M\ !\ N \tag{1}$$

$M\ \#\ N$ denotes unboxed application, $M\ !\ N$ denotes strict application.

2.2 Operational semantics

In the following we will simply extend the semantics given in [1] with one exception: In the paper the semantics of application is actually that of unboxed applicaton, that is, an argument's reduction is not going to be shared if it is used several times in the body of a lambda. Instead we will give semantics of a boxed application, that is, one that creates a new thunk in the heap for the argument. The original semantics will be that of unboxed application. Furthermore we will not consider let bindings and issues with renaming for simplicity.

$$\frac{}{\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e}\ \text{(Lambda)}$$

$$\frac{\Gamma : f \Downarrow \Gamma' : \lambda y.e \qquad \Gamma'[\nu \to x] : e[\nu/y] \Downarrow \Gamma'' : z \qquad \nu \text{ fresh}}{\Gamma : fx \Downarrow \Gamma'' : z}\ \text{(Boxed)}$$

$$\frac{\Gamma : f \Downarrow \Gamma' : \lambda y.e \qquad \Gamma' : e[x/y] \Downarrow \Gamma'' : z}{\Gamma : f \ \# \ x \Downarrow \Gamma'' : z}\ \text{(Unboxed)}$$

$$\frac{\Gamma : x \Downarrow \Gamma' : s \qquad \Gamma' : f \Downarrow \Gamma'' : \lambda y.e \qquad \Gamma'' : e[s/y] \Downarrow \Gamma''' : z}{\Gamma : f \ ! \ x \Downarrow \Gamma''' : z}\ \text{(Strict)}$$

$$\frac{\Gamma : e \Downarrow \Gamma' : z}{\Gamma[x \to e] : x \Downarrow \Gamma'[x \to z] : z}\ \text{(Variable)}$$

2.3 Symbolic reduction

Before we specify symbolic reduction we need to understand what the *effect* of reduction is. In our case it is going to be the set of variables and applications that are reduced to weak head normal form.

Our reduction context is going to be a set of reductions and a *symbolic* heap. By symbolic I mean that the reduction will finish if we try to reduce a variable that is not in the heap (free variables), much like what happens in the operational semantics in the case of lambdas. This stopping of the symbolic evaluation would be a common theme for any intermediate representation we would like to examine this way, as we can never have complete information about the runtime state.

> Sidenode: Initially I only planned to account for unboxing/strictness opportunities, but I realized that with little modification the analysis can also account for *safe* common subexpression elimination as well. By *safe* I mean that, we only extract two equivalent subexpressions into one if both are guaranteed to reduce. This way we avoid the potential memory leaks caused by a naive CSE.

The reduction set will contain either variables or applications. Note that if during symbolic evaluation we reduce $f\ x$ as well as $f\ !\ x$ then we want to represent them as the same in the reduction set. This is because at construction we will know whether we want unboxed application (if the argument was reduced as well) or simple application. In essence all strict applications will become unboxed, all unboxed applications will stay unboxed and boxed applications may or may not become unboxed. To this end we will need a helper function to disregard the type of applications in a term, in particular those that are guaranteed to reduce:

$$
\begin{aligned}
x^{\uparrow} &= x \\
(f \ \# \ x)^{\uparrow} &= f^{\uparrow} \ x \\
(f \ x)^{\uparrow} &= f^{\uparrow} \ x \\
(f \ ! \ x)^{\uparrow} &= f^{\uparrow} \ x^{\uparrow} \\
(\lambda x.M)^{\uparrow} &= \lambda x.M
\end{aligned}
$$

Our symbolic reductions will have the form $\Gamma, R : M \Downarrow \Gamma', R' : N$, where Γ is the symbolic heap, R is the reduction set and M is the term at hand.

The reductions then are:

$$\frac{}{\Gamma, R : \lambda x.e \Downarrow \Gamma, R : \lambda x.e} \ \text{(Lambda)}$$

$$\frac{\Gamma, R : e \Downarrow \Gamma', R' : z}{\Gamma[x \to e], R : x \Downarrow \Gamma'[x \to z], R' : z} \ \text{(Variable, known)}$$

$$\frac{x \notin \Gamma}{\Gamma, R : x \Downarrow \Gamma, R \cup \{x\} : x} \ \text{(Variable, unknown)}$$

$$\frac{\Gamma, R : f \Downarrow \Gamma', R' : \lambda y.e \qquad \Gamma'[w \to x], R' : e[w/y] \Downarrow \Gamma'', R'' : z}{\Gamma, R : f \ x \Downarrow \Gamma'', R'' : z} \ \text{(Boxed, known)}$$

$$\frac{\Gamma, R : f \Downarrow \Gamma', R' : g \qquad g \neq \lambda\_.\_}{\Gamma, R : f \ x \Downarrow \Gamma', R' \cup \{(g \ x)^{\uparrow}\} : g \ x} \ \text{(Boxed, unknown)}$$

$$\frac{\Gamma, R : f \Downarrow \Gamma', R' : \lambda y.e \qquad \Gamma', R' : e[x/y] \Downarrow \Gamma'', R'' : z}{\Gamma, R : f \ \# \ x \Downarrow \Gamma'', R'' : z} \ \text{(Unboxed, known)}$$

$$\frac{\Gamma, R : f \Downarrow \Gamma', R' : g \qquad g \neq \lambda\_.\_}{\Gamma, R : f \ \# \ x \Downarrow \Gamma', R' \cup \{(g \ \# \ x)^{\uparrow}\} : g \ \# \ x} \ \text{(Unboxed, unknown)}$$

$$\frac{\Gamma, R : x \Downarrow \Gamma', R' : s \qquad \Gamma', R' : f \Downarrow \Gamma'', R'' : \lambda y.e \qquad \Gamma'', R'' : e[s/y] \Downarrow \Gamma''', R''' : z}{\Gamma, R : f \ ! \ x \Downarrow \Gamma''', R''' : z} \ \text{(Strict, known)}$$

$$\frac{\Gamma, R : f \Downarrow \Gamma', R' : g \qquad g \neq \lambda\_.\_ \qquad \Gamma', R' : x \Downarrow \Gamma'', R'' : z}{\Gamma, R : f \ ! \ x \Downarrow \Gamma'', R'' \cup \{(g \ ! \ z)^{\uparrow} : g \ ! \ z} \ \text{(Strict, unknown)}$$

2.4 Construction

Our construction $\{-\}_\rho$ needs to do the same reductions as the original term and it needs to do these in an order that minimizes work. Because of this we will traverse the Reduction set in an order that has the following constraint:

$\forall M, N.(M \ N) \in R \to (M \in R \to M < (M \ N)) \wedge (N \in R \to N < (M \ N))$

In other words we want the reduction of both the function and the argument to happen before the reduction of the application does. In the construction we will assume that the input reduction list satisfies this constraint.

The construction will use an environment ρ to keep track of fresh variables corresponding to reduced variables/applications.

$$\begin{aligned}
\{[]\}_{\rho,T} &= \rho(T) && \\
\{x :: L\}_{\rho,T} &= (\lambda\nu.\{L\})_{\rho[x\to\nu],T} \ ! \ x && \nu \text{ fresh} \\
\{MN :: L\}_{\rho,T} &= (\lambda\nu.\{L\})_{\rho[M\ N\to\nu],T} \ ! \ X && \nu \text{ fresh} \\
&\text{where} && \\
&\quad X = \rho(M) \ \# \ \rho(N) && \text{if } N \in \rho \\
&\quad X = \rho(M) \ N && \text{otherwise}
\end{aligned}$$

2.5 Example

To demonstrate what the above transformation does let's examine what happens with the following term:

$M = g\ x + (g\ x + x)$

g and x are free variables and $+$ is defined as

$a + b = \mathit{plus}\ !\ a\ !\ b$

In other words it is an unknown function (free variable) that requires both arguments to be in whnf.

After symbolic reduction and construction (choosing an arbitrary ordering for construction) the new term is going to be:

$(\lambda 1.(\lambda 2.(\lambda 3.(\lambda 4.(\lambda 5.(\lambda 6.(\lambda 7.7)\ !\ (5\ \#\ 6))\ !\ (5\ \#\ 2))\ !\ (4\ \#\ 3))\ !\ \mathit{plus})\ !\ (1\ \#\ 2))\ !\ x)\ !\ g$

Arguably this is not the most beautiful term ever, the construction function could be made much smarter, but it does demonstrate that the algorithm correctly detected strictness, unboxing and CSE opportunities.

3 Project plan

3.1 Goals

By the end of the project I would like to

- Have shown that the proposed optimization can be done and works
- Have developed a theoretical framework that allows sound implementation of the technique
- Have a working compiler and runtime
- Have written a backend for an existing in-use frontend language (e.g. Haskell or ML)

3.2 Steps

Following is the layout of the bigger strokes of the project. The ordering is somewhat arbitrary and would be fixed as the project develops.

Reading

This part of the project would focus on exploring the relevant parts of compiler research. This includes papers on static analyses, semantics (e.g. abstract interpretation) and symbolic evaluation. (Papers/theses in mind?)

Toy compiler

My plan is to develop a toy compiler for experimenting purposes. I very much intend the project to have practical results, and a working implementation of the sketched optimization technique is a good coherence criterion for the idea anyway.

Theoretical basis

This would involve developing a rigorous theoretical framework in which we can examine the soundness of the optimization, as well as give optimization conditions(e.g. we are guaranteed to have less memory allocations than before). My goal is to develop a theory general enough to be agnostic with regards to the language.

The research would initially focus on a language with limited capabilities e.g. lazy lambda calculus with primitive numbers or an imperative garbage collected language working on the heap.

More expressiveness

Once we are done with the simple languages we can move onto more muddy waters and look at how to express the effect of concurrency or side effects.

For example we can represent side effecting function calls as a chain of states the runtime must be in when making the calls. Construction then needs to make sure that these states are indeed reached and the ordering is preserved.

Concurrency we may want to represent as a set of constraints on the interleaving of evaluations which the construction needs to satisfy.

Dynamic optimization

This part would be about how to incorporate the optimization into the runtime. An example use case would be detecting long residing objects in the heap and specializing closures that use it. This requires a couple of things:

The runtime must use a representation of both the code and the runtime state that is amenable to symbolic evaluation. This suggests that we either use a bytecode intepreter that is used both for execution and analysis, or think of a way to abstract the runtime state into a higher level representation.

Furthermore the runtime should be able to determine whether such an optimization is feasible at all. The above example is to detect long residing objects, but it may also be the detection of “hot” code (JVM style).

Backend for existing language

This is what would complete the practical aspect of the project. As the optimization technique is frontend-language-agnostic we can pick any language that has clear semantics. We could even bootstrap the compiler itself by targeting the language it was written in.

References

[1] John Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’93, pages 144–154, New York, NY, USA, 1993. ACM.