Polymorphism, Subtyping, Whole Program Analysis and Accurate Data Types in Usage Analysis

Tobias Gedell[1], Jörgen Gustavsson[2], and Josef Svenningsson[1]

[1] Department of Computing Science,
Chalmers University of Technology and Göteborg University,
{gedell,josefs}@cs.chalmers.se
[2] Spotfire, Inc.

Abstract. There are a number of choices to be made in the design of a type based usage analysis. Some of these are: Should the analysis be monomorphic or have some degree of polymorphism? What about subtyping? How should the analysis deal with user defined algebraic data types? Should it be a whole program analysis?
Several researchers have speculated that these features are important but there has been a lack of empirical evidence. In this paper we present a systematic evaluation of each of these features in the context of a full scale implementation of a usage analysis for Haskell.
Our measurements show that all features increase the precision. It is, however, not necessary to have them all to obtain an acceptable precision.

1 Introduction

In this article we study the impact of polymorphism, subtyping, whole program analysis and accurate data types on type based *usage analysis*. Usage analysis is an analysis for lazy functional languages that aims to predict whether an argument of a function is used at most once. The information can be used to reduce some of the costly overhead associated with call-by-need and perform various optimizing program transformations. The focus of this paper is however solely on improving the precision of usage analysis, not on its uses.

Polymorphism Polymorphism is the primary mechanism for increasing the precision of a type based analysis and achieving a degree of context sensitivity.

Previous work by Peyton Jones and Wansbrough has indicated that polymorphism is important for usage analyses. Convinced that polymorphism could be dispensed with they made a full scale implementation of a completely monomorphic usage analysis. However, it turned out that it was "almost useless in practice" [WPJ99]. They drew the conclusion that the reason was the lack of polymorphism. In the end they implemented an improved analysis with a simple form of polymorphism that also incorporated other improvements [Wan02]. The resulting analysis gave a reasonable precision but there is no evidence that polymorphism was the crucial feature.

Studies of other program analyses have come to a different conclusion about polymorphism. On example is points-to analysis for C for which several studies have shown that monomorphic analyses [FFA00,HT01,FRD00,Das00,DLFR01] give adequate precision for the purpose of an optimizing compiler [DLFR01]. Moreover, extending these analyses with polymorphism seem to have only a moderate effect [FFA00,DLFR01].

Point-to analysis may not be directly relevant for usage analysis but it still begs the question of how much polymorphism really can contribute to the precision of an analysis. One of the goals of this paper has been to shed some light on this question.

Subtyping Another important feature in type based analysis is subtyping. It provides a mechanism for approximating a type by a less informative super type. This gives a form of context sensitivity since a type may have different super types at different call sites. It also provides a mechanism for combining two types, such as the types of the branches of an if expression, by a common super type. Thus, the effects of subtyping and polymorphism overlap.

This raises a number of questions. Does it suffice with only polymorphism or only subtyping? How much is gained by having the combination?

Whole program analysis Another issue that also concerns context sensitivity is whole program analysis versus modular program analysis. A modular analysis which considers each module in isolation must make a worst case assumption about the context in which it appears.

This will clearly degrade the precision of the analysis. But how much? Is whole program analysis a crucial feature? And how does it interact with the choice of monomorphism versus polymorphism?

Data types Another important design choice in a type based analysis is how to deal with user defined data types. The intuitive and accurate approach may require that the number of annotations on a type is exponential in the size of the type definitions of the analyzed program. The common solution to the problem is to limit the number of annotations on a type in some way, which can lead to loss of precision. The question is how big the loss is in practice.

Contributions In order to evaluate the above features, we have implemented a range of usage analyses:

- With different degrees of polymorphism (Section 3)
- With and without subtyping (Section 4)
- Using different treatments of data types (Section 5)
- As whole program analyses and as modular analyses (Section 6)

All analyses have been implemented in the GHC compiler and have been measured with GHC's optimizing program transformations both enabled and

disabled. We present figures summarizing (the arithmetic mean of) the effectiveness of each of the different features. More detailed figures for each of the programs we've analyzed can be found in the first authors licentiate thesis [Ged06].

We have not measured every combination of the above features. Instead we have started with a very precise analysis and successively turned off various features to see how much precision is lost. The initial analysis is the most precise in all but one aspect. It doesn't use whole program analysis. Our reason for that is that we wanted to stay close to how we would have implemented the analysis in GHC. Since GHC supports separate compilation so does our base line analysis.

Our systematic evaluation shows that each of these features has a significant impact on the precision of the analysis. Especially, it is clear that some kind of context sensitivity is needed through polymorphism or subtyping. Our results also show that the different features partly overlap. The combined effect of polymorphism and subtyping is for example not very dramatic although each one of them has a large effect on the accuracy. Another example is that whole program analysis is more important for monomorphic analysis than polymorphic analysis.

2 Usage Analysis

Implementations of lazy functional languages maintain sharing of evaluation by updating. For example, the evaluation of

$$(\lambda x.x + x)\ (1 + 2)$$

proceeds as follows. First, a closure for $1 + 2$ is built in the heap and a reference to the closure is passed to the abstraction. Second, to evaluate $x + x$ the value of x is required. Thus, the closure is fetched from the heap and evaluated. Third, the closure is updated (i.e., overwritten) with the result so that when the value of x is required again, the expression needs not be recomputed.

The same mechanism is used to implement lazy data structures such as potentially infinite lists.

The sharing of evaluation is crucial for the efficiency of lazy languages. However, it also carries a substantial overhead which is often not needed. For example, if we evaluate

$$(\lambda x.x + 1)\ (1 + 2)$$

then the update of the closure is unnecessary because the argument is only used once.

The aim of usage analysis is to detect such cases. The output of the analysis is an annotated program. Each point in the program that allocates a closure in the heap is annotated with 1 if the closure that is created at that point is always used at most once. It is annotated with ω if the closure is possibly used more than once or if the analysis cannot ensure that the closure is used at most once.

The annotations allow a compiler to generate code where the closures are not updated and thus effectively turning call-by-need into call-by-name. Usage analysis also enables a number of program transformations [PJPS96,PJM99].

Usage analysis has been studied by a number of researchers [LGH+92,Mar93,TWM95,Fax95,Gus98,WPJ99,WPJ00,GS00,Wan02].

2.1 Measuring the Effectiveness

We measured the effectiveness of the analyses by running them on the programs from the *nofib* suite [Par93] which is a benchmarking suite designed to evaluate the Glasgow Haskell Compiler (GHC). We excluded the toy programs and ran our analysis on the programs classified in the category *real* but had to exclude the following three programs: *HMMS* did not compile with GHC on our test system, *ebnf2ps* is dependent on a version of Happy that we could not get to work with our version of GHC, and *veritas* because many analyses ran out of memory when analyzing it.

Despite the name of the category, the average size of the programs is unfortunately quite small, ranging from 74 to 2,391 lines of code, libraries excluded.

The notion of effectiveness When measuring the effectiveness it is natural to do so by modifying the runtime system of GHC. The runtime system is modified to collect the data needed to compute the effectiveness during a program's execution.

The easiest way is to count how many created closures that are only used once and how many of those closures that were detected by the analysis. This can be implemented by adding three counters to the runtime system: one that is incremented as soon as an updatable closure is created, one that is incremented each time a closure is used a second time, and one that is incremented as soon as a closure annotated with 1 is created. With these counters one can compute an effectiveness of an analysis:

$$\frac{\textit{closures annotated with } 1}{\textit{created closures} - \textit{closures used twice}}$$

This is the metric used by Wansbrough [Wan02].

A drawback of this approach is that it does not take into account that each program point can only have one annotation – if any of the closures allocated at a program point is used more than once, that program point has to be annotated with ω for the analysis to be sound. Thus, any program point which has some closures used more than once and some used at most once would make even a perfect analysis get less than a 100 percent effectiveness. And such program points are common.

What we would like to do is to compute the effectiveness by measuring the proportion of *program points* that are correctly annotated instead of the proportion of *updates* that are avoided. We, therefore, modified the run time system to compute the best possible annotations which are consistent with the observed run time behavior. I.e., if all the closures allocated at a specific program point is used at most once during the execution, that program point could be annotated with 1 otherwise ω. We did this by, for each closure, keeping track of at which

program point it was created. When a closure is used a second time we add its program point to the set of program points that need to be annotated with ω. We were careful to exclude dead code i.e. code that was not executed in the executions such as parts of imported libraries which were not used. It is important to note that this way of measuring is still based on running the program on a particular input and a perfect analysis may still get an effectiveness which is less than 100 percent.

Wansbrough's and our metrics differ also at another crucial point. The former metric depends very much on how many times each program point that allocates closures is executed. If a single program point allocates a majority of all closures, the computed effectiveness will depend very much on whether that single program point was correctly annotated by the analysis. In contrast, the effectiveness computed with the latter measurement will hardly be affected by one conservative annotation.

We think that our metric is more informative and have, therefore, used it for all our measurements.

Optimizing program transformations Our implementation is based on GHC which is a state of the art Haskell implementation. The specific version of GHC we have used is 5.04.3. GHC parses the programs and translates them into the intermediate language Core, which is essentially System F [PJPS96]. When GHC is run with optimizations turned on (i.e. given the flag `-O`), it performs aggressive program transformation on Core before it is translated further. We inserted our analyses after GHC's program transformations just before the translation to lower level representations.

We ran the analysis with GHC's program transforming optimizations both enabled and disabled. The latter gives us a measure of the effectiveness of an analysis on code prior to program transformations. This is relevant because usage information can be used to guide the program transformations themselves.

2.2 Implementation

Actually implementing all the analyses we report on in this paper would have been a daunting task. To get around this problem we used the following trick: The only analysis we actually implemented was the most precise analysis, with polymorphism, polymorphic recursion, subtyping and whole program analysis. This analysis generated constraints, in the form of Constraint Abstractions [GS01]. These constraints have enough structure preserved from the original program to enable us to identify precisely where we can change them to correspond to a lesser precise analysis. We implemented several transformations on our constraints which effectively removed polymorphism, polymorphic recursion, subtyping, whole program analysis and which mimicked various ways of handling data types, respectively.

Although this trick helped us greatly in performing the measurements it had an unfortunate drawback. The transformed constraints, although semantically equivalent to a less precise analysis, was very remote from what an actual analysis

would have generated. Several of our translations produced constraints that were very hard for the constraint solver. Therefore, any timings that we might have reported on would have been highly misleading. This is the reason why we have chosen to exclude them from this paper.

3 Polymorphism

We start by evaluating usage polymorphism. To see why it can be a useful feature, consider the function that adds up three integers.[3]

$$\mathit{plus3}\ x\ y\ z = x + y + z$$

Which usage type should we give to this function? Since the function uses all its arguments just once, it seems reasonable to give it the following type.

$$\mathit{Int}^1 \to \mathit{Int}^1 \to \mathit{Int}^1 \to \mathit{Int}^\omega$$

The annotations on the type express that all three arguments are used just once by the function and that the result may be used several times. However, this type is not correct. The problem is that the function may be partially applied:

$$\mathit{map}\ (\mathit{plus3}\ (1+2)\ (3+4))\ \mathit{xs}$$

If xs has at least two elements then $\mathit{plus3}\ (1+2)\ (3+4)$ is used more than once. As a consequence, so is also $(1+2)$ and $(3+4)$.

To express that functions may be used several times we need to annotate also function arrows. A possible type for *plus3* could be:

$$\mathit{Int}^\omega \to^\omega \mathit{Int}^\omega \to^\omega \mathit{Int}^1 \to^\omega \mathit{Int}^\omega$$

The function arrows are annotated with ω which indicates that *plus3* and its partial applications may be used several times. The price we pay is that the first and the second argument are given the type Int^ω. This type is sound but it is clearly not a good one for call sites where *plus3* is not partially applied. What is needed is a mechanism for separating call sites with different usage.

The solution to the problem is to give the function a usage polymorphic type:

$$\forall\ u_0\ u_1\ u_2\ u_3 \mid u_2 \le u_0, u_3 \le u_0, u_3 \le u_1 . \mathit{Int}^{u_0} \to^\omega \mathit{Int}^{u_1} \to^{u_2} \mathit{Int}^1 \to^{u_3} \mathit{Int}^\omega$$

The type is annotated with usage variables and the type schema contains a set of constraints which restrict how the annotations can be instantiated. A constraint $u \le u'$ simply specifies that the values instantiated for u must be smaller than or equal to the values instantiated for u' where we have the ordering that $1 < \omega$. This form of polymorphism is usually referred to as constrained polymorphism or bounded polymorphism.

In our example, $u_2 \le u_0$ enforces that if a partial application of *plus3* to one argument is used more than once then that first argument is also used more than once. Similarly, $u_3 \le u_0$ and $u_3 \le u_1$ makes sure that if we partially apply *plus3* to two arguments and use it more than once then both these arguments are used more than once.

[3] This example is due to Wansbrough and Peyton Jones [WPJ00]

3.1 Degrees of Polymorphism

There are many different forms of parametric polymorphism. In this paper we consider three different systems where usage generalization takes place at let-bindings.

- An analysis with monomorphic recursion in the style of ML. Intuitively, this gives the effect of a monomorphic analysis where all non-recursive calls have been unwound.
- An analysis with polymorphic recursion [Myc84,Hen93,DHM95]. Intuitively, this gives the effect of the previous analysis where recursion has been (infinitely) unwound.
- An analysis where the form of type schemas are restricted so that generalized usage variables may not be constrained. A consequence of the restriction is that an implementation need not instantiate (i.e., copy) a potentially large constraint set whenever the type is instantiated. Wansbrough and Peyton Jones [WPJ00] suggested this in the context of usage analysis and called it *simple usage polymorphism*.
  With simple usage polymorphism it is not possible to give *plus3* the type

  $$\forall\, u_0\, u_1\, u_2\, u_3 \mid u_2 \leq u_0, u_3 \leq u_0, u_3 \leq u_1.\mathit{Int}^{u_0} \rightarrow^{\omega} \mathit{Int}^{u_1} \rightarrow^{u_2} \mathit{Int}^{1} \rightarrow^{u_3} \mathit{Int}^{\omega}$$

  because the generalized variables u_0, u_1, u_2, u_3 are all constrained. Instead we can give it the type

  $$\forall\, u.\mathit{Int}^{u} \rightarrow^{\omega} \mathit{Int}^{u} \rightarrow^{u} \mathit{Int}^{1} \rightarrow^{u} \mathit{Int}^{\omega}$$

  where we have unified the generalized variables into one. This type is clearly worse but it gives a degree of context sensitivity. An alternative is to give it a monomorphic type. For example

  $$\mathit{Int}^{\omega} \rightarrow^{\omega} \mathit{Int}^{1} \rightarrow^{\omega} \mathit{Int}^{1} \rightarrow^{1} \mathit{Int}^{\omega}.$$

  These types are incomparable and an implementation needs to make a heuristic choice. We use the heuristic proposed by Wansbrough [Wan02] to generalize the types of all exported functions and give local functions monomorphic types.

The analyses include usage subtyping; use an aggressive treatment of algebraic data types and are compatible with separate compilation (i.e., we analyze the modules of the program one by one in the same order as GHC). We discuss and evaluate all these features later on.

3.2 Evaluation

The results are shown in Figure 1, which shows the average effectiveness of each analysis.

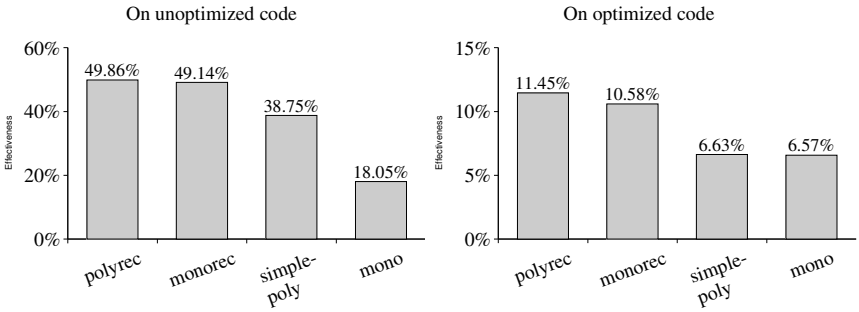

Fig. 1. Measurements of polymorphism

The most striking observation is that the results are very different depending on whether GHC's optimizing program transformations are turned on or off. The effectiveness is much lower with program transformations turned on. While we have yet to make any detailed studies of this phenomenon we here suggest some possible explanations. Firstly, one possible contributor to this phenomenon is GHC's aggressive inliner [PJM99]. There is no need to create closures for the arguments of inlined function calls and thus many targets for the analysis disappears. The net effect is that the proportion of difficult cases (such as closures in data structures and calls to unknown functions) increases which reduces the effectiveness.

Another explanation is strictness analysis [Myc82]. Strictness analysis can decide that the argument of a function is guaranteed to be used at least once (in any terminating computation). In those cases there is no need to suspend the evaluation of that argument. If an argument is used exactly once then it is a target for both strictness and usage analysis. When the strictness analysis (as part of GHC's program transformation) is run first it removes some easy targets.

Our measurements also show that the polymorphic analyses are significantly better than the monomorphic one. Polymorphic recursion turns out to have hardly any effect compared to monomorphic recursion. Simple polymorphism comes half way on unoptimized code – it is significantly better than monomorphism but significantly worse than constrained polymorphism, which shows that it can serve as a good compromise. This is, however, not the case for optimized code.

The largest surprise to us was that the accuracy of the monomorphic analysis is relatively good. This seems to contradict the results reported by Wansbrough and Peyton Jones [WPJ00] who implemented and evaluated the monomorphic analysis from [WPJ99]. They found that the analysis was almost useless in practice and concluded that it was the lack of polymorphism that caused the poor results. We do not have a satisfactory explanation for this discrepancy.

4 Subtyping

Consider the following code fragment.

$$\textbf{let } x =^{u} 1 + 2 \textbf{ in } \ldots$$

Here u is the usage annotation associated with the closure for $1 + 2$.

The analysis can take u to be 1 if and only if x is used at most once. That is assured by giving x the type Int^1. The type system then makes sure that the program is well typed only if x is actually used at most once.

If we on the other hand take u to be ω then x has the type Int^ω. It is always sound to annotate a closure with ω regardless of how many times it is used. We, therefore, want the term to be well typed regardless of how many times x is actually used. The solution is to let Int^ω be a *subtype* of Int^1. That is, if a term has the type Int^ω we may also consider it to have the type Int^1.

Subtyping makes the system more precise. Consider the function f.

$$f\ x\ y\ = \textbf{if } x * x > 100 \textbf{ then } x \textbf{ else } y$$

It seems reasonable that we should be able to give it, for example, the type

$$Int^\omega \rightarrow^\omega Int^1 \rightarrow^\omega Int^1.$$

This type expresses that if the result of the function is used at most once then the second argument is used only once. The first argument is, however, used at least twice regardless of how many times the result is used.

To derive this type we must have usage subtyping. Otherwise, the types of the branches of the conditional would be incompatible – x has type Int^ω and y has the type Int^1. With subtyping we can consider x to have the type Int^1.

Without subtyping x and y has to have the same type and the type of the function must be

$$Int^\omega \rightarrow^\omega Int^\omega \rightarrow^\omega Int^\omega$$

which puts unnecessary demands on y.

Subtyping can also give a degree of context sensitivity. Consider, for example, the following program.

$$\begin{array}{ll} \textbf{let} & f\ x = x + 1 \\ & a = 1 + 2 \\ & b = 3 + 4 \\ \textbf{in} & f\ a + f\ b + b \end{array}$$

Here, b is used several times and is given the type Int^ω. Without subtyping nor polymorphism we would have to give a the same type and the two call sites would pollute each other.

When subtyping is combined with polymorphism it naturally leads to constrained polymorphism. Note, however, that subtyping is not the only source of inequality constraints in a usage analysis. Inequality constraints are also used for the correct treatment of partial application (see Section 3) and data structures. Thus, we use constrained polymorphism also in the systems without subtyping.

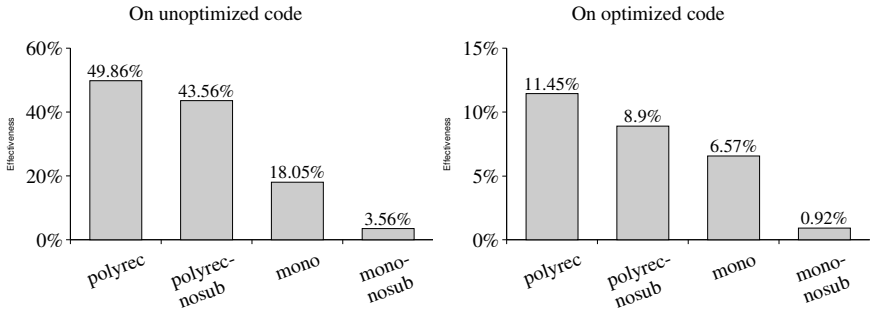

Fig. 2. Measurements of subtyping

4.1 Evaluation

We have evaluated two systems without subtyping – a polymorphicly recursive and a monomorphic analysis. Both analyses use an aggressive treatment of data types and are compatible with separate compilation. Figure 2 shows the average effectiveness of each analysis. We have included the system with polymorphic recursion and subtyping and the monomorphic system with subtyping from Section 3 for an easy comparison.

The results show that the accuracy of the monomorphic system without subtyping is poor. The precision is dramatically improved if we add subtyping or polymorphism. Our explanation is that both polymorphism and subtyping gives a degree of context sensitivity which is crucial.

The polymorphic system without subtyping is in principle incomparable to the monomorphic system with subtyping. However, in practice the polymorphic system outperforms the monomorphic one. The difference is much smaller when the analyses are run on optimized code which is consistent with our earlier observation that context sensitivity becomes less important because of inlining.

The combination of subtyping and polymorphism has a moderate but significant effect when compared to polymorphic analysis without subtyping. The effect is relatively larger on optimized code. The explanation we can provide is that the proportion of hard cases - which requires the combination – is larger because the optimizer has already dealt with many simple cases.

5 Algebraic data types

An important issue is how to deal with data structures such as lists and user defined data types. In this section we evaluate some different approaches.

Let us first consider the obvious method. The process starts with the user defined data types which only depend on predefined types. Suppose T is such a type.

$$data\, T\, \boldsymbol{\alpha} = C_1\, \boldsymbol{\tau} \mid \ldots \mid C_n\, \boldsymbol{\tau}$$

The types on the right hand side are annotated with fresh usage variables. If there are any recursive occurrences they are ignored. The type is then parameterized on these usage variables, $\boldsymbol{u}$.

$$data\ T\ \boldsymbol{u}\ \boldsymbol{\alpha} = C_1\ \boldsymbol{\tau}'_1 \mid \ldots \mid C_n\ \boldsymbol{\tau}'_n$$

Finally, any recursive occurrence of T is replaced with $T\ \boldsymbol{u}$. The process continues with the remaining types in the type dependency order and when T is encountered it is replaced with $T\ \boldsymbol{u}'$ where $\boldsymbol{u}'$ is a vector of fresh variables. If there are any mutually recursive data types they are annotated simultaneously.

As an example consider the following data type for binary trees:

$$data\ Tree\ \alpha = Node\ (Tree\ \alpha)\ (Tree\ \alpha) \mid Leaf\ \alpha$$

When annotated, it contains three annotation variables:

$$\begin{aligned} data\ Tree\ \langle k_0, k_1, k_2\rangle\ \alpha = {} & Node\ (Tree\ \langle k_0, k_1, k_2\rangle\ \alpha)^{k_0}\ (Tree\ \langle k_0, k_1, k_2\rangle\ \alpha)^{k_1} \\ & \mid\ Leaf\ \alpha^{k_2} \end{aligned}$$

This approach is simple and accurate and we used it in all the analyses in the previous sections. The net effect is equivalent to a method where all non-recursive occurrences in a type are first unwound. As a result the number of annotation variables can grow exponentially. An example of this is the following data type:

$$\begin{aligned} & data\ T_0\ \langle k_0\rangle\ = C\ Int^{k_0} \\ & data\ T_1\ \langle k_0, k_1, k_2, k_3\rangle = C'\ (T_0\ \langle k_1\rangle)^{k_0} \mid C''\ (T_0\ \langle k_3\rangle)^{k_2} \\ & \ldots \\ & data\ T_n\ \langle k_0, \ldots, k_m\rangle = C'_n\ (T_{n-1}\ \langle \ldots\rangle)^{k_0} \mid C''_n\ (T_{n-1}\ \langle \ldots\rangle)^{k_{m/2}} \end{aligned}$$

Here T_n will contain $O(2^n)$ usage variables.

In practice, the number of required variables sometimes grows very large. The largest number we have encountered was a type in the Glasgow Haskell Compiler which required over two million usage annotations. As a consequence a single subtyping step leads to over two million inequality constraints and our implementation simply could not deal with all those constraints. This problem was the reason for why we had to exclude the program veritas from our study. It is clear that an alternative is needed and we tried two different ones.

The first approach was to put a limit on the number of usage variables which are used to annotate a type. If the limit is exceeded then we simply use each variable several times on the right hand side of the type. We do not try to do anything clever and when we exceed the limit we simply recycle the variables in a round robin manner. This approach leads to ad-hoc spurious behavior of the analysis when the limit is exceeded but maintains good accuracy for small types. We tried this approach with a limit of 100, 10 and 1.

The second approach was to simply annotate all types on the right hand side with only ω. The effect is that information is lost when something is inserted

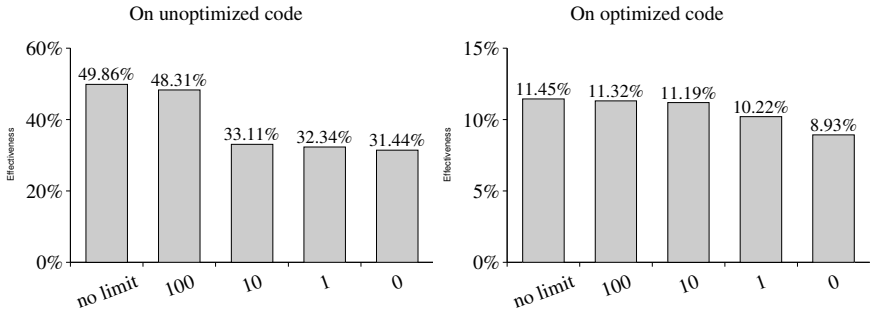

Fig. 3. Measurements of treatments of data types

into a data structure – the analysis simply assumes the worst about its usage. Intuitively this can be thought of as a special case of the approach above where the limit is zero.

All the analyses used for measuring the treatment of data types have subtyping and polymorphic recursion and are compatible with separate compilation.

5.1 Evaluation

The average effectiveness of each analysis is shown in Figure 3.

The results are quite different for optimized and unoptimized code. In the case of unoptimized code there is a clear loss in precision when we limit the number of annotation variables. The loss is quite small when the limit is 100 but quite dramatic when the limit is only 10. Going further and annotating with only one or no variables has a smaller effect.

The situation is different for optimized code. Here there is only a small difference when the number of variables are limited to 100 or 10. But there is a noticeable effect when one or no variables are used.

We believe that this effect stems from Haskell's class system. When Haskell programs are translated into Core each class context is translated to a so called dictionary parameter. A dictionary is simply a record of the functions in an instance of a class. Large classes leads to large records of functions which are passed around at run time. When the number of annotations are limited, it substantially degrades the precision for these records. Presumably, most dictionaries require more than 10 variables but less than 100 which explains the effect for unoptimized code.

These records are often eliminated by GHC's program transformations which tries to specialize functions for each particular instance [Jon94,Aug93]. Thus, in optimized code there are not so many large types which explains why the effect of limiting the number of variables to 10 is quite small.

6 Whole Program Analysis

So far all the analyses have been compatible with separate compilation. In this section we consider whole program analysis.

Suppose that f is an exported library function where the closure created for x' is annotated with u.

$$f\ x = \mathbf{let}\ x' =^{u} x + 1\ \mathbf{in}\ \lambda y.x' + y$$

In the setting of separate compilation we have to decide which value u should take without knowledge of how f is called. In the worst case, f is applied to one argument and the resulting function is applied repeatedly. The closure of x' is then used repeatedly so we must assume the worst and let u be equal to ω. We can then give f the type

$$Int^{1} \rightarrow^{\omega} Int^{1} \rightarrow^{\omega} Int^{\omega}$$

With separate compilation we must make sure that the types of exported functions are general enough to be applicable in all contexts. That is, it must still be possible to annotate the remaining modules such that the resulting program is well typed. Luckily, this is always possible if we ensure that the types of all exported functions have an instance where the positive (covariant) positions in the type are annotated with ω. In the type of f this is reflected in that the function arrows and the resulting integer are annotated with ω. Wansbrough and Peyton Jones [WPJ00] calls this process pessimization. Further discussion can be found in Wansbrough's thesis [Wan02].

In the setting of whole program analysis this process in unnecessary which improves the result of the analysis. We have chosen to evaluate the effect on two analyses, the polymorphicly recursive analysis with subtyping and the monomorphic analysis with subtyping. Both analyses use the aggressive treatment of data types.

6.1 Evaluation

The average effectiveness for each analysis is shown in Figure 4. They show that whole program analysis improves both analyses significantly on both unoptimized and optimized code.

The effect is greater for the monomorphic analysis. The explanation is that the inaccuracies that are introduced by the pessimization, needed for separate compilation, spreads further in the monomorphic analysis due to the lack of context sensitivity. One can think of pessimization as simulating the worst possible calling context which then spreads to all call sites.

An interesting observation is that there is only a small difference between the polymorphic and the monomorphic whole program analysis for optimized code. The combination of aggressive inlining and whole program analysis almost cancels out the effect of polymorphism.

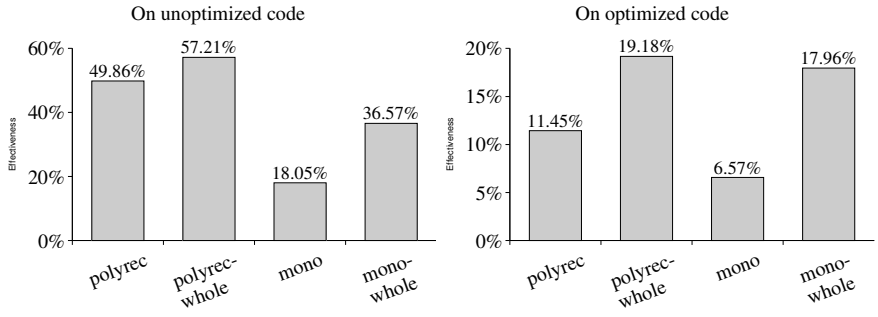

Fig. 4. Measurements of whole program analysis

7 Related Work

The usage analyses in this paper build on the type based analyses in [TWM95,Gus98,WPJ99,WPJ00,GS00,Wan02]. The use of polymorphism in usage analysis was first sketched in [TWM95] and was developed further in [GS00] and [WPJ00,Wan02] where simple polymorphism was proposed. Usage subtyping was introduced in [Gus98,WPJ99]. The method for dealing with data types was suggested independently by Wansbrough [Wan02] and ourselves [Ged03]. The method for dealing with separate compilation is due to Wansbrough and Peyton Jones [WPJ99].

The measurements of Wansbrough and Peyton Jones on their monomorphic analysis with subtyping and a limited treatment of data types showed that is was "almost useless in practice". Wansbrough later made thorough measurements of the precision of simple usage polymorphism with some different treatments of data types in [Wan02]. He concludes that the accuracy of the simple usage polymorphism with a good treatment of data types is reasonable which is consistent with our findings. He also compares the accuracy with a monomorphic usage analysis but the comparison is incomplete – the monomorphic analysis only has a very coarse treatment of data types.

Foster et al [FFA00] evaluate the effect of polymorphism and monomorphism on Steensgaard's equality based points-to analysis [Ste96] as well as Andersen's inclusion based points-to analysis [And94]. Their results show that the inclusion based analysis is substantially better than the unification based. Adding polymorphism to the equality based analysis also has a substantial effect but adding polymorphism to the inclusion based analysis gives only a small improvement.

There are clear analogies between Steensgaard's equality based analysis and usage analysis without subtyping. Andersen's inclusion based analysis relates to usage analysis with subtyping. Given these relationships, our results are consistent with the results of Foster et al with one exception – the combination of polymorphism and subtyping has a significant effect in our setting. However, when we apply aggressive program transformations prior to the analysis and run it in whole program analysis mode then our results coincide.

8 Conclusions

We have performed a systematic evaluation of the impact on the accuracy of four dimensions in the design space of a type based usage analyses for Haskell. We evaluated

- different degrees of polymorphism: polymorphic recursion, monomorphic recursion, simple polymorphism and monomorphism,
- subtyping versus no subtyping,
- different treatments of user defined types, and
- whole program analysis versus analysis compatible with separate compilation.

Our results show that all of these features individually have a significant effect on the accuracy. A striking outcome was that the results depended very much on whether the analyzed programs were first subject to aggressively optimizing program transformations. A topic for future work would be to investigate how much each optimization affects the analysis result.

Our evaluation of polymorphism and subtyping showed that the polymorphic analyses clearly outperform their monomorphic counterparts. The effect was larger when the analyses did not incorporate subtyping. This is not surprising given that subtyping gives a degree of context sensitivity and, thus, partially overlaps with polymorphism. Polymorphic recursion turned out to give very little when compared to monomorphic recursion. For unoptimized code, simple polymorphism (where variables in types schemas cannot be constrained) was shown to lie in between monomorphism and constrained polymorphism.

The measurements also showed that the treatment of data types is important. The effectiveness of the different alternatives turned out to depend on whether the code was optimized or not. We believe that the explanation is coupled to the implementation of Haskell's class system and, thus, that this observation might be rather Haskell specific.

Whole program analysis turned out to have a rather large impact. The effect was greater for monomorphic analysis. The reason is that the conservative assumptions, that have to be made in the setting of separate compilation, have larger impact due to the lack of context sensitivity in monomorphic analysis. In fact, the whole program monomorphic analysis with subtyping was almost as good as the whole program polymorphic analysis with subtyping on optimized programs.

Finally we note that the effectiveness of even the most precise analysis seems quite poor. For unoptimized code the best figure is 57% and for optimized code the top effectiveness is a poor 19%. Is this because we have used an imprecise measure or because of fundamental limitations of the form of usage analysis used in this paper? We leave this question for future investigation.

References

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[Aug93] Lennart Augustsson. Implementing haskell overloading. In *Functional Programming Languages and Computer Architecture*, pages 65–73, 1993.

[Das00] Manuvir Das. Unification-based pointer analysis with directional assignments. In *PLDI'00*, pages 35–46. ACM Press, June 2000.

[DHM95] D. Dussart, F. Henglein, and C. Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In *SAS'95*, September 1995.

[DLFR01] Manuvir Das, Ben Liblit, Manuel Fähndrich, and Jakob Rehof. Estimating the impact of scalable pointer analysis on optimization. In *SAS'01*. Springer LNCS 2126, 2001.

[Fax95] Karl-Filip Faxén. Optimizing lazy functional programs using flow inference. In *Proc. of SAS'95*, pages 136–153. Springer-Verlag, LNCS 983, September 1995.

[FFA00] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for c. In *SAS'00*, pages 175–198, 2000.

[FRD00] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable Context-Sensitive Flow Analysis using Instantiation Constraints. In *PLDI'00*, Vancouver B.C., Canada, June 2000.

[Ged03] Tobias Gedell. A Case Study on the Scalability of a Constraint Solving Algorithm: Polymorphic Usage Analysis with Subtyping. Master thesis, October 2003.

[Ged06] Tobias Gedell. Static analysis and deductive verification of programs. Licentiate thesis, 2006.

[GS00] Jörgen Gustavsson and Josef Svenningsson. A usage analysis with bounded usage polymorphism and subtyping. In Markus Mohnen and Pieter W. M. Koopman, editors, *IFL*, volume 2011 of *Lecture Notes in Computer Science*, pages 140–157. Springer, 2000.

[GS01] Jörgen Gustavsson and Josef Svenningsson. Constraint abstractions. In *PADO II*, pages 63–83. Springer Verlag LNCS 2053, 2001.

[Gus98] Jörgen Gustavsson. A type based sharing analysis for update avoidance and optimisation. In *ICFP*, pages 39–50. ACM, SIGPLAN Notices 34(1), 1998.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[HT01] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *PLDI'01*, pages 254–263. ACMPress, June 2001.

[Jon94] Mark P. Jones. Dictionary-free overloading by partial evaluation. In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)*, pages 107–117, 1994.

[LGH+92] J. Launchbury, A. Gill, J. Hughes, S. Marlow, S. L. Peyton Jones, and P. Wadler. Avoiding Unnecessary Updates. In J. Launchbury and P. M. Sansom, editors, *Functional Programming*, Workshops in Computing, Glasgow, 1992. Springer.

[Mar93] S. Marlow. Update Avoidance Analysis by Abstract Interpretation. In *Proc. 1993 Glasgow Workshop on Functional Programming*, Workshops in Computing. Springer–Verlag, 1993.

[Myc82] Alan Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicative Programs*. PhD thesis, University of Edinburg, 1982.

[Myc84] Alan Mycroft. Polymorphic Type Schemes and Recursive Definitions. In *Proceedings 6th International Symposium on Programming, Lecture Notes in Computer Science*, Toulouse, July 1984. Springer Verlag.

[Par93] W. Partain. The nofib benchmark suite of haskell programs, 1993.

[PJM99] S. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell compiler inliner. In *Workshop on Implementing Declarative Languages*, 1999.

[PJPS96] S. Peyton Jones, W. Partain, and A. Santos. Let-floating: moving bindings to give faster programs. In *Proc. of ICFP'96*, pages 1–12. ACM, SIGPLAN Notices 31(6), May 1996.

[Ste96] B. Steensgaard. Points-to analysis in almost linear time. In *POPL'96*, pages 32–41. ACM Press, January 1996.

[TWM95] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *Proc. of FPCA*, La Jolla, 1995. ACM Press, ISBN 0-89791-7.

[Wan02] Keith Wansbrough. *Simple Polymorphic Usage Analysis*. PhD thesis, Computer Laboratory, Cambridge University, England, March 2002.

[WPJ99] Keith Wansbrough and Simon Peyton Jones. Once Upon a Polymorphic Type. In *Proc. of POPL'99*. ACM Press, January 1999.

[WPJ00] Keith Wansbrough and Simon Peyton Jones. Simple Usage Polymorphism. In *ACM SIGPLAN Workshop on Types in Compilation*. Springer-Verlag, September 2000.