Efficient Strictness Analysis of Haskell in Haskell Using Abstract Reduction

Marko Schütz[1] and Manfred Schmidt-Schauß[2] and Sven Eric Panitz[2]

[1] Fachbereich Mathematik und Informatik
Institut für Informatik
Freie Universität Berlin
Takustraße 9
D-14195 Berlin
e-mail: schuetz@inf.fu-berlin.de
[2] Fachbereich Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt
Germany
e-mail:{schauss,panitz}@informatik.uni-frankfurt.de

Abstract. The extraction of strictness information marks an indispensable element of an efficient compilation of lazy functional languages like Haskell. Based on the method of abstract reduction we have developed an efficient strictness analyser for a core language of Haskell. It is completely written in Haskell and compares favourably with known implementations.
The implementation is based on the $G^{\#}$-machine, which is an extension of the G-machine that has been adapted to the needs of abstract reduction.

1 Introduction

Obtaining a maximum of strictness information is a key requirement for generating efficient code for lazy functional programming languages. Changing lazy evaluation (call-by-need) to eager evaluation (call-by-value) for specific functions has been proven to be necessary for good program performance. This has been *inter alia* pointed out as one result of doing work on benchmarking by Hartel [HFA$^+$94]. Such changes in the order of evaluation may only be made for functions, which have been proven to be strict, in order to preserve lazy semantics. Strictness information enables a safe change of the reduction ordering, i.e. the termination behaviour of the program is unaffected.
Examples of how strictness information can be used are described in [HB93, PJP]. Although a large amount of research to lay theoretical foundations for strictness analysis, see e.g. [Bur91], has been carried out since the initiating work by Mycroft [Myc80] only a few results have become known with regard to practical implementations. Today, most compilers, as e.g. by the Glasgow team [PJHH$^+$92], employ some very basic strictness analyser. The most efficient

strictness analyser for Haskell [HPW$^+$92] both in terms of efficiency and precision is claimed to be a recent implementation by Jensen et al [JHR94]. Their implementation of abstract interpretation using "chaotic" fixpoint iteration is implemented in C and has not yet been incorporated in any compiler. Another implementation has been given for Clean [NSvP91, PvE93]. This implementation is based on abstract reduction [Nöc93]. It is also written in C. Unfortunately, it is hidden in the Clean compiler and, therefore, comparisons to other implementations are hardly possible.
We have developed an adaptation of abstract reduction to Haskell and have made an implementation of a strictness analyser for the core language as it is presented in [PJL91]. Our implementation is entirely written in Haskell.
It is realised via a newly developed abstract machine, the $G^{\#}$-machine. The source gets compiled into $G^{\#}$-code. The subsequent interpretation of $G^{\#}$-code then yields strictness information. This interpretation is done in the same way as in the G-machine.
Although our implementation is not fine tuned in terms of speed, it turned out to be very good both in terms of precision and efficency. In a script of almost 290 functions all strictness information was found in a less than 18 seconds on a SPARCstation 10. A theoretical estimate of the precision and complexity of abstract reduction would be very difficult. Therefore the tests we present in section 5 which are typical cases of strictness and context analysis are of great importance. This is especially true since we can learn the dependence of precision and complexity on the value for our termination criterion. We are quite positive that it can be tailored to be incorporated in any Haskell compiler. The implementation is freely available by anonymous ftp from ftp.uni-frankfurt.de in the directory /pub/dist/kist/functional/abs-g.
A detailed description can be found in [Sch94, Sch95].
In the next section we shall give a short review of abstract reduction followed by an overview of our implementation. A further section describes the $G^{\#}$-machine with its, in contrast to Nöcker's method, non-strict order of reduction. Finally, some results of the performed tests will be given.

2 Abstract Reduction

As we cannot expect the reader to be familiar with abstract reduction, we shall give a short review of [Nöc93].
In contrast to abstract interpretation, which abstracts the denotational semantics of a functional language, abstract reduction aims at an abstraction of the operational semantics, i.e., an abstract term is reduced until it hopefully yields the term "bottom". As area of investigation, an infinite domain is defined for a language completely determined by supercombinators and their applications:

> Let S be the set of all function symbols including data constructors defined in a program script, then the abstract domain will be the set of directed graphs over the vertex set $S^{\#} = \{f^{\#} | f \in S\} \cup \{\bot^{\#}, \top^{\#}, \mathtt{Union}^{\#}\}$

where $\bot^{\#}$ and $\top^{\#}$ are constants and $\texttt{Union}^{\#}$ is of arbitrary arity. An expression $\texttt{Union}^{\#}\ x_1 \ldots x_n$ will be written as $\langle x_1, \ldots, x_n \rangle$.

Detailed discussion of this domain can be found in [GH93]. Because of the intended meaning of the $\texttt{Union}^{\#}$ operator an ordering for this domain can only be given via a concretisation map γ. In line with the literature concerning abstract interpretation we define γ as a map from abstract values to sets of values in the standard interpretation. For simplicity no type information is taken into account:

$$\begin{aligned}
\gamma(\bot^{\#}) &= \{\bot\} \\
\gamma(\top^{\#}) &= \left\{ x \middle| x \text{ is in the domain of the standard interpretation} \right\} \\
\gamma(\langle x_1, \ldots, x_n \rangle) &= \cup_{1 \le i \le n} \gamma(x_i) \\
\gamma(f^{\#}\ e_1 \ldots e_n) &= \left\{ x \middle|\ x \in \{\mathbf{S}\ [\![(f\ t_1 \ldots t_n)]\!] | \mathbf{S}\ [\![t_i]\!] = t_i', t_i' \in \gamma(e_i)\} \right\}
\end{aligned}$$

where $\mathbf{S}\ [\![exp]\!]$ denotes the standard interpretation of *exp*.
As ordering on the abstract domain we take set inclusions on the concretisations:

$$a_1 \le_\alpha a_2 := \gamma(a_1) \subseteq \gamma(a_2)$$

This ordering provides information up to how specific a value is. The least specific one is $\top^{\#}$, which expresses all concrete values, the most specific one is $\bot^{\#}$, which represents just $\bot$ expressions. Generally, it cannot be decided whether $a \le_\alpha b$. This would mean that $e \le_\alpha \bot^{\#}$ can be decided for every abstract value e, which would give a decision precedure for strictness analysis.
An approximation $\le'$ of $\le_\alpha$ with $a \le' b \Rightarrow a \le_\alpha b$ can be given as follows:

$$\begin{aligned}
\bot^{\#} &\le' t \\
t &\le' \top^{\#} \\
t &\le' t \\
\texttt{f}\ x_1, \ldots, x_n &\le' \texttt{f}\ y_1, \ldots, y_n, \text{if } \forall 1 \le i \le n : x_i \le' y_i \\
t &\le' \langle x_1, \ldots, x_n \rangle, \text{if } \exists 1 \le i \le n : t \le' x_i
\end{aligned}$$

In order to handle cyclic structures a refinement of this approximation is introduced. The reader is refered to [Nöc93].
Naturally, an equivalence relation holds between abstract values. Some important equivalences used to simplify unions are:

$$\begin{aligned}
\langle x \rangle &\equiv x \\
\langle x_1, \ldots, x_n \rangle &\equiv \langle \pi(x_1, \ldots, x_n) \rangle, \text{for a permutation } \pi \\
\langle x, x_1, \ldots, x_n \rangle &\equiv \langle x_1, \ldots, x_n \rangle, \text{if } (\exists 1 \le i \le n) : x \le' x_i \\
C(\langle x_1, \ldots, x_n \rangle) &\equiv \langle C(x_1), \ldots, C(x_n) \rangle, \text{for a context } C(\cdot) \\
\langle \langle x_1, \ldots, x_n \rangle, x_1', \ldots, x_m' \rangle &\equiv \langle x_1, \ldots, x_n, x_1', \ldots, x_m' \rangle
\end{aligned}$$

Such equivalences on unions will allow us to simplify every abstract value to an equivalent abstract value, which has at most one union construct and to shift this union operator up to the root of the value. Next, the concept of abstract reduction is introduced. An abstract value a_1 reduces to a_2, if for every expression, which abstracts a_1, a needed concrete reduction can be given, resulting in an expression that abstracts to a_2. A reduction is needed, if for all reduction sequences, which lead to head normal form, this reduction step has to be performed.
For a formal introduction of this notion the reader is referred to the original work by Nöcker [Nöc93].
The general form of abstract reduction cannot be implemented, because it is not algorithmically defined. Therefore, algorithmic reduction-steps on abstract values are defined, which approximate abstract reduction. These are two, namely abstract rewriting and reduction path analysis with bottom or cycle introduction.

– Abstract rewriting is just a sort of symbolic computation on abstract values. An expression $(f^{\#}\ a_1 \ldots a_n)$ is rewritten according to the rules defined for the supercombinator f. If more than one alternative of a function matches, the different results will be collected in a `Union`$^{\#}$ construct. The only question that arises is, how to determine which function alternatives match. Since the language Nöcker uses (Clean), includes full pattern-matching, his matching rules are more complex than ours. In the Core language only simple patterns of one constructor with pattern variables are used. This makes the rules for matching much simpler. A decision algorithm for matching of such simple patterns is given by:

$$\begin{aligned}
\text{Match}^{\#}(\top^{\#}, p) &= \text{true} \\
\text{Match}^{\#}(\bot^{\#}, p) &= \text{false} \\
\text{Match}^{\#}(C\ v_1 \ldots v_n, C'\ p_1 \ldots p_m) &= \text{false, if } C \neq C' \\
\text{Match}^{\#}(C\ v_1 \ldots v_n, C\ p_1 \ldots p_m) &= \text{true} \\
\text{Match}^{\#}(\langle e_1, \ldots, e_n \rangle, p) &= \exists 1 \leq i \leq n : \text{Match}^{\#}(e_i, p)
\end{aligned}$$

– Reduction path analysis is a kind of loop detection. If in a sequence of abstract rewritings an expression *exp* can be found, of which a generalisation *exp′* had been reduced before, we will be able to introduce a reduction cycle. Reducing *exp* can be done in the same way as reducing *exp′* $ad\ infinitum$. This cycle in the abstract reduction sequence will correspond to an infinite path in the concrete reduction, if it is needed, i.e., if a concretisation of *exp* has to be reduced in the concrete reduction. This will be the case, if *exp* is in every alternative of the reduced expression, i.e., if it is in every component of the resulting union of the case expression. A reduction cycle of a needed expression yields an infinite reduction in the concrete counterpart and therefore can be reduced to $\bot^{\#}$.

As an example for an analysis using abstract reduction, we give the iterative length function:

```
length xs n = case xs of
                   <1> -> n;
                   <2> y ys -> length ys (n+1);
```

Strictness of the second argument can be found by the following abstract reduction:

$$\begin{aligned}
&\texttt{length}\ \top^{\#}\ \bot^{\#} \\
&\to \langle \bot^{\#}, \texttt{length}\ \top^{\#}\ \texttt{(}\bot^{\#}\texttt{+1)}\rangle \\
&\equiv \texttt{length}\ \top^{\#}\ \texttt{(}\bot^{\#}\texttt{+1)} \qquad (1)\\
&\to \texttt{length}\ \top^{\#}\ \bot^{\#} \\
&\to \bot^{\#}
\end{aligned}$$

In [?] we present a reformulation of abstract reduction as a deduction calculus and prove its soundness.

3 An Overview of the Implementation

The language of discourse is a functional core language, which is assumed to be λ-lifted. We have followed the Core language in its representation from [PJL91]. The syntax of its abstraction is given in figure 1. The actual implementation can cope with an enriched syntax which includes strings. This choice was motivated by having a well documented and broadly spread source and not being bound by some special syntax of a concrete compiler. The language is parsed using the parser in [PJL91]. Then it is compiled to abstract G-code, called $G^{\#}$-code. In this step, in contrast to Nöcker's implementation, primitive values are not approximated by $\top^{\#}$. This will enable us to find strictness of functions like:

```
st a = if (1 == 0) 1 a
```

Because our work is not in the context of a real compiler, but can be used as a stand-alone system, a dependency analysis became necessary to yield information on the calling structure of the program. This enables us to analyse functions which are at the lower end of the calling hierarchy first, and use their strictness information for generating code for functions which are higher up in the calling hierarchy. Unfortunately, we found that costs for dependency analysis in most cases were more expensive than the improvements due to its information so that we turned it off. Nevertheless, in the context of a compiler, which provides dependency information further improvements can be made this way.
Strictness analysis for a function `f` in its ith argument is finally performed by starting the abstract reduction of ($\texttt{f}^{\#}\ \top^{\#} \ldots \top^{\#}\ \bot^{\#}\ \top^{\#} \ldots \top^{\#}$) and reducing until $\bot^{\#}$ is reduced or some termination criterion has been reached. The result

of the analysis is a table which gives a list for every function where the ith list element says that the function is known to be strict in the ith argument or that strictness in this argument is unknown.
Although abstract reduction enables us to find context information, our analyser in its present state has built-in support for strictness analysis only. As will be seen in section 4 the context analysis problem is easily reduced to the strictness analysis problem. Today most compilers do not use context information for code generation and the question of how to gain efficiency by context information is at least controversial, see e.g. [FB94]. Nevertheless, we made some tests for context analysis, see section 5.
For a detailed discussion of the implementational issues and the complete operational semantics of the $G^{\#}$-machine the reader is referred to [Sch95].

3.1 Open Redexes

Reduction path analysis makes it necessary to recollect the reduction history of an abstract term t,i.e. all the terms and subterms one came across while reducing to t. Therefore our implementation maintains a list where all terms which can lead to cycle or bottom introduction are recorded. We can think of these terms as those for which evaluation has started, but which have not yet been updated with a term in WHNF. We will call these terms *open redexes*. The termination criterion used in our implementation is to limit the number of open redexes.

4 List Domain and Context Information

The method is not restricted to WHNF strictness on $\perp$, i.e., more information for a function `f` than `f` $\perp = \perp$ can be generated. On recursive types like lists one might be interested in more information than the information that a list does not have a WHNF. So analysis can be made for functions, which have lists as arguments or lists as results.

4.1 4 Point List Domain

For lists there has been proposed a domain of four points, each representing a certain degree of where the first bottom of a list expression will be found. The four point domain on lists is defined as: $\{\perp, \infty, \perp\!\!\in, \top\!\!\in\}$ with the ordering $\perp \sqsubseteq \infty \sqsubseteq \perp\!\!\in \sqsubseteq \top\!\!\in$ [Wad87]. $\perp$ represents expressions with no WHNF, ∞ represents expressions with no finite list structure, $\perp\!\!\in$ represents expressions with undefined list elements, and $\top\!\!\in$ all list expressions.
Fortunately, we are able to express these abstract list values:

$$\begin{array}{rll}
\texttt{topmem} = & \top^{\#} & (\top\!\!\in) \\
\texttt{botmem} = & \texttt{<Cons }\perp^{\#}\texttt{ topmem, Cons }\top^{\#}\texttt{ botmem>;} & (\perp\!\!\in) \\
\texttt{inf} = & \texttt{Cons }\top^{\#}\texttt{ inf;} & (\infty)
\end{array}$$

4.2 Context Information

For a function f with a list result, we might be interested to know if reducing to a certain degree, e.g. to the list structure or to normal form, an application of f to some value, yields bottom. Such information is called context information [Bur91, Bur87].
We can generate context information with our machine by introducing evaluator functions [Bur91]. The only task of such evaluator functions is to force the reduction of its argument into a certain form in order to get a result in WHNF. An example of such an evaluator is[3]:

```
e_botmem xs = case xs of
                Nil -> Nil;
                (Cons  y ys) -> k1 (enf ys) y;


    k1 xs y = case xs of
                _ ->  case y of
                        _ -> y;
```

The function e_botmem requires the reduction to the complete list structure of its argument and reduction of every list element to WHNF in order to produce a WHNF. Such evaluator functions allow us to reduce context analysis to strictness analysis of a certain abstract value.
Now it is possible to generate context information quite easily. We do not have to adapt the method or to enrich the machine but simply have to analyse certain functions for WHNF strictness. For example, we can generate context information for the standard append function by analysing:

```
(e_botmem (append botmem topmem))
```

If this expression gets abstractly reduced to $\perp^{\#}$, we know that append requires the reduction of its first argument to a list of WHNFs in order to produce such a list as result.
An additional advantage of abstract reduction over abstract interpretation is that abstract reduction can be used to compute the different context information for a function seperately. This makes it possible to have the compiler analyse only those contexts which are needed. Furthermore, as we have seen for the 4-point-domain, it is straightforward to generate abstract values and evaluator functions for arbitrary algebraic types. This could even be done by the compiler itself.

5 Results

Several tests were performed on a Sun SPARCstation 10/41 with 64MB of RAM running Solaris 2.4. The analyser itself was compiled by the Glasgow Haskell

[3] For reasons of readability we use list constructors Nil and Cons in this example and not the numbered Core language constructors.

Compiler version 0.24. The Core language files for the tests were obtained by running the Gofer compiler `gofc` with a `-D` option followed by some manual touch-up to meet our syntax. A first test was analysing a program script (`t10.core`) entailing Core language versions of parts taken from the analyser itself, namely from the compiler to $G^{\#}$-code, the lexer and some auxiliary functions plus the Core language version of the Gofer [Jon94] `standard.prelude` with exception of floating point functions. In this test 287 functions with a total of 526 arguments are analysed for their strictness. To our knowledge the analyser found all strictness information in this script. This was achieved in 18 seconds user time. This time includes the times for parsing the Core language script which a real compiler will already have done.
A second test involved a Core language version (`nucleic2.core`) of the pseudoknot benchmark program [HFA+94] plus the Core language version of the `standard.prelude`. This program constitutes a widely used test for functional language compiler technology. It is a real world application from molecular biology which does not depend on laziness for correct behaviour. For this test all floating point numbers were approximated with $\top^{\#}$, as the $G^{\#}$-machine does not support floating point numbers. In this test 248 functions are analysed with a total of 493 arguments. To our knowledge here also all strictness information was found.
For a third test we analysed all of the contexts which can arise in the `standard.prelude` with respect to the 4-point-domain except for the `zip` functions with arity above 2. That is for every list valued argument we test all the combinations of the values $\top\in$, $\bot\in$ and ∞. If the result is a list as well we combine these with the evaluators $\xi_{\bot}$, $\xi_{\bot\in}$ and ξ_{∞}. Thus for `append` 27 context analyses are performed. Here 596 functions are analysed with a total of 767 arguments. 435 functions thereof are constructed to perform context analysis. These functions will be of the form:

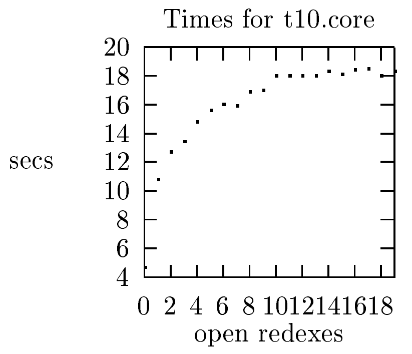
```
cAppendEInfBotmemTop xs = e_inf (append botmem topmem)
```

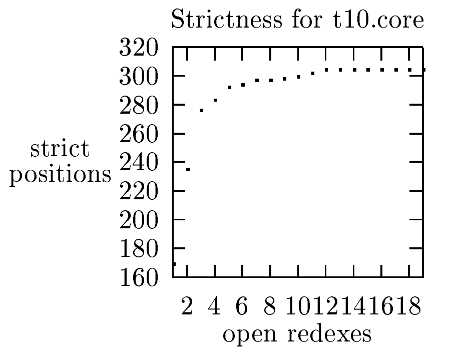
One of the most important optimizations a real compiler might make for this case is to derive new context information from context information already obtained. Based on the $\leq_{\alpha}$-relation on abstract values it suffices to abstractly reduce `e_inf (append botmem topmem)` to $\bot^{\#}$ to infer that `e_inf (append inf topmem)`, `e_inf (append botmem topmem)`, `e_inf (append botmem inf)` as well as `e_botmem (append botmem topmem)`, `e_botmem (append inf topmem)`, `e_botmem (append botmem botmem)` and `e_botmem (append botmem inf)` will also abstractly reduce to $\bot^{\#}$. We did not exclude these analyses from our input file, because with few open redexes allowed `e_inf (append inf topmem)` will not reduce to $\bot^{\#}$ and thus not allow the additional information to be derived.
The following plots show the times resp. the strict arguments found depending on the number of open redexes allowed.[4] For the `t10.core` and the `nucleic2.core`

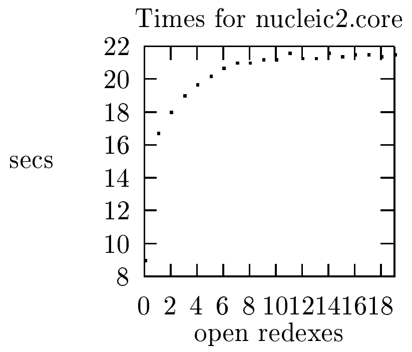
[4] The plots for time include a value for 0 open redexes. In this case the input file is parsed and compiled to $G^{\#}$-code only, but no analysis is performed.

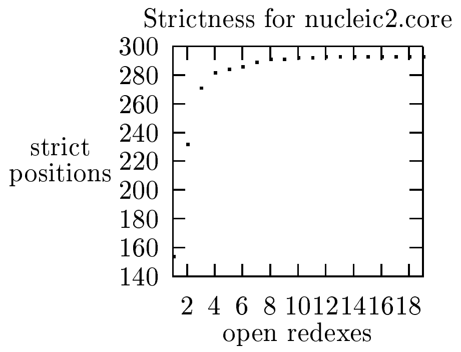
input files we see that no additional strict arguments are found once 12 or more open redexes are allowed. From this point on the running time remains constant. For the `context.core` input file, on the other hand, we see a continuing increase of running time as we increase the number of open redexes allowed. This is due to the fact, that in this analysis looping reductions are encountered which are not detected by the analyser. A better approximation of the $\leq_\alpha$-relation might remedy this situation. Two cases in which looping reductions are not detected are the context analyses:

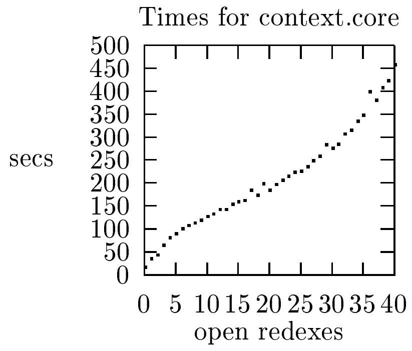
`e_inf (sums inf)` and
`e_inf (products inf)`

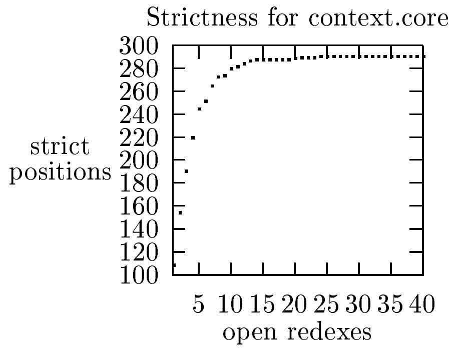












Thus for a typical strictness analysis there is no danger in specifying a very large bound for the number of open redexes: the running time will not increase

past a certain reasonable maximum. For a typical context analysis the running time is not bounded, but here also it seems that much information can be foun in a reasonable time.
As an example we include some output lines from the analysis of `context.core`:

```
...
init [strict]
cInitEBotTop [?]
cInitEBotBotmem [?]
cInitEBotInf [?]
cInitEInfTop [?]
cInitEInfBotmem [?]
cInitEInfInf [strict]
cInitEBotmemTop [?]
cInitEBotmemBotmem [?]
cInitEBotmemInf [strict]
...
```

6 Conclusion

Our implemetation efficiently approximates abstract reduction with reduction path analysis. The $G^{\#}$-machine, a new machine model based on the G-machine, systematically presents the method used. The degree of similarity with the G-machine which we were able to uphold indicates how obviously the method used and reduction in functional languages correspond.
Although the implementation favors ease of understanding over efficiency, it proves that abstract reduction with reduction path analysis is fit for every-day strictness analysis even when implemented in a functional language and that it finds strictness information which implementations of other methods do not find.
It is possible to optimize our implementation in several respects, there are even parts executed for every $G^{\#}$-machine instruction simulated where optimization is possible. Cautiously estimating, it should not be very difficult to halve the running time.

References

[Aug84] Lennart Augustsson. A compiler for Lazy ML. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 218–227, 1984.

[Bur87] Geoffrey Burn. Evaluation transformers - a model for the parallel evaluation of functional languages (extended abstract). In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 446–470. Springer, 1987.

[Bur91] Geoffrey Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation.* Pitman, London, 1991.

[FB94] Sigbjørn Finne and Geoffrey Burn. Assessing the evaluation transformer model of reduction on the Spineless G-machine. Technical report, Imperial College of Science, Technology and Medicine, Department of Computing, 1994.

[GH93] E. Goubault and C. L. Hankin. A lattice for the abstract interpretation of term graph rewriting systems. In M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors, *Term Graph Rewriting - Theory and Practice*, chapter 10. Wiley, Chichester, 1993.

[HB93] Denis B Howe and Geoffrey L Burn. Using strictness in the STG machine. In John T O'Donnel and Kevin Hammond, editors, *Functional Programming*, Workshops in Computingt, pages 127–137. Springer, 1993.

[HFA$^+$94] Pieter H. Hartel, M. Feeley, M. Alt, L. Augustsson, P. Baumann, M. Beemster, E. Chailloux, C. H. Flood, W. Grieskamp, J. H. G. van Groningen, K. Hammond, B. Hausman, M.Y. Ivory, P. Lee, X. Leroy, S: Loosemore, N. Rösjemo, M. Serrano, J.-P. Talpin, J. Thackray, P. Weiss, and P. Wentworth. Pseudoknot: a float-intensive benchmark for functional compilers. In John Glauert, editor, *Implementation of Functional Languages '94*, 1994. Draft.

[HPW$^+$92] Paul Hudak [ed.], Simon L. Peyton Jones [ed.], Philip Wadler [ed.], Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell. A non-strict purely functional language. version 1.2, 1992.

[JHR94] Kristian Damm Jensen, Peter Hjæresen, and Mads Rosendahl. Efficient strictness analysis of Haskell. In Baudouin Le Charlier, editor, *Static Analysis*, number 864 in Lecture Notes in Computer Science, pages 346–362. Springer, 1994.

[Joh84] T. Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the ACM Conference on Compiler Construction, Montreal*, pages 58–69, 1984.

[Jon94] Mark P. Jones. The implementation of the gofer functional programming system. Research Report YALEU/DCS/RR-1030, Yale University, Department of Computer Science, May 1994.

[Myc80] Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *4th International Symposium on Programming*, number 83 in Lecture Notes in Computer Science, pages 269–281. Springer, 1980.

[Nöc92] Eric Nöcker. Strictness analysis by abstract reduction in orthogonal term rewriting systems. Technical Report 92-31, University of Nijmegen, Department of Computer Science, 1992.

[Nöc93] Eric Nöcker. Strictness analysis using abstract reduction. In *Functional Programming Languages and Computer Architecture*, pages 255–265. ACM Press, 1993.

[NSvP91] E. G. J. M. H. Nöcker, J. E. W. Smetsers, M. C. J. D. van Ekelen, and M. J. Plasmeijer. Concurrent Clean. In Springer Verlag, editor, *Proc of Parallel Architecture and Languages Europe (PARLE'91)*, number 505 in Lecture Notes in Computer Science, pages 202–219, 1991.

[PJHH$^+$92] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Phil Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference*, 1992.

[PJL91] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages: a Tutorial.* Prentice-Hall International, London, 1991.

[PJP] Simon L. Peyton Jones and Will Partain. Measuring the effectiveness of a simple strictness analyser. Draft.

[PvE93] Rinus Plasmeijer and Marko van Ekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, Workingham, 1993.

[Sch94] Marko Schütz. Striktheits-Analyse mittels abstrakter Reduktion für den Sprachkern einer nicht-strikten funktionalen Programmiersprache. Master's thesis, Johann Wolfgang Goethe-Universität, Frankfurt, 1994. in German.

[Sch95] Marko Schütz. The $G^{\#}$-machine: Efficient strictness analysis in Haskell. Technical Report 1/95, Johann Wolfgang Goethe-Universität, Fachbereich Informatik, January 1995.

[Wad87] Phil Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declerative Languages*, chapter 12. Ellis Horwood Limited, Chichester, 1987.

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

[t]

Programs	$prog \rightarrow$	$sk_1; \ldots; sk_n, n \geq 1$
Super-combinators	$sk \rightarrow$	$var\ var_1 \ldots var_n = expr, n \geq 0$
Expressions	$expr \rightarrow$	$expr\ aexpr$
	$\mid$	$expr_1\ binop\ expr_2$
	$\mid$	let $defs$ in $expr$
	$\mid$	letrec $defs$ in $expr$
	$\mid$	case $expr$ of $alts$
	$\mid$	$aexpr$
	$aexpr \rightarrow$	var
	$\mid$	num
	$\mid$	Pack$\{num, num\}$
	$\mid$	($expr$)
	$\mid$	Top
	$\mid$	Bot
	$\mid$	$\langle expr_1, \ldots, expr_n \rangle$
Definitions	$defs \rightarrow$	$def_1; \ldots; def_n, n \geq 1$
	$def \rightarrow$	$var = expr$
Alternatives	$alts \rightarrow$	$alt_1; \ldots; alt_n, n \geq 1$
	$alt \rightarrow$	$< num >\ var_1 \ldots var_n\ - >\ expr, n \geq 0$
Binary operators	$binop \rightarrow$	$arop \mid relop \mid boolop$
	$arop \rightarrow$	$+ \mid - \mid * \mid /$
	$relop \rightarrow$	$< \mid <= \mid == \mid \sim= \mid >= \mid >$
	$boolop \rightarrow$	$\& \mid \mid$
Variables	$var \rightarrow$	$alpha\ varch_1 \ldots varch_n, n \geq 0$
Numbers	$num \rightarrow$	$digit_1 \ldots digit_n, n \geq 1$
	$alpha \rightarrow$	*an alphabetic character*
	$varch \rightarrow$	$alpha \mid digit \mid \_$
	$digit \rightarrow$	*a numeric character*

Fig. 1. Syntax of the abstracted Core language