Dead-code elimination for a functional language

Thijs Alkemade, Alessandro Vermeulen, Marcelo Barbosa de Sousa,
Utrecht University, 2011.

August 3, 2011

Abstract

Dead code elimination is an important part of optimizing a program, as a lot of analyses and optimizations can benefit of having a smaller input. We have implemented dead code analysis for a simple functional language, and we have investigated how it relates to strictness analysis in that language.

1 Introduction

Initially we set out to do strictness analysis and tried a Hindley-Milner like approach. However, after reading the assigned papers we chose to use Stefan Holdermans' one-pass algorithm for live variable analysis instead. The algorithm deals with a small functional language and instead of adapting it to do strictness analysis we chose to extend it with several extras instead, such as binary operators, **if** ... **then** ... **else** ... **endif**-statements, and recursive functions of the form **nfun** f $x \Rightarrow$ ... . These are similar to **fun** but they get a name. We tried adding the recursive functionality to the **let**-statement but we couldn't get it to work properly. The limitation to this is that we can only have recursive functions with one parameter.

2 The Language

The language is a functional language based on Holdermans and Hage [2010a,b] and contains the following constructs:

- Numbers
- Booleans
- Operators ($\equiv$, $+$, $-$, $*$, '*div*', '*mod*', $<$, and $>$)
- **let** ... = ... **in** ... **end**
  A non-recursive-let-statement
- **if** ... **then** ... **else** ... **endif**
  The usual if-then-else-statement
- **fun** $x \Rightarrow$ ...
  A lambda abstraction
- **nfun** f $x \Rightarrow$ ...
  A named lambda abstraction used for recursion.

Booleans, operators, named lambda abstraction and the conditional branching are our additions.

3 The 'Holdermans'-algorithm

The algorithm essentially works in the same way as the $\mathcal{W}$-algorithm by Damas-Milner. The types of expressions are calculated in exactly the same way, but in addition annotations are guessed, and constraints are generated and solved. These constraints essentially keep a record of where the annotations came from so as not to lose information. $\mathcal{R}$ is the reconstruction algorithm that is analogous to $\mathcal{W}$. The unification algorithm $\mathcal{U}$ works in same way but also takes the annotation variables into account. Next to this, the generated constraints from $\mathcal{R}$ are 'solved' by $\mathcal{S}$ in the **let**-case and as a final step for the top level of the program. The $\mathcal{S}$ (worklist-)algorithm solves the constraints by finding all constraints that influence the free annotation variables in a given constraint set.

This ensures that a term with an annotation β and an associated constraint set C is considered dead if $\neg\exists\beta' : (\beta' \sqsubseteq \beta) \in C$.

The implementation of the algorithm can be found in the directory 'src/Analysis/Inferencing/HTm/'. The file 'HH.hs' contains the algorithms itself, where 'HHLib.hs' contains helper function, and 'HHTy.hs' contains the additional types used for the inferencing, and 'HHSubs.hs' contains a type class to make applying substitutions easier.

4 Changes from the original algorithm

As mentioned before we have received the basic one-pass algorithm for determining the minimal principal types from Stefan Holdermans. When implementing the algorithm we have made a few changes to the algorithm. Next to the inevitable addition of support for our AST.

1. In the worklist algorithm we have added the statement $\mathit{infl}\ [\beta 2] := \{\}$ and lookups are replaced by *findByDefault* where the default is the empty set.
2. To ensure that the whole program is considered live, we insert α into the active variables, and also add the constraint $\alpha \sqsubseteq \alpha$ to ensure that trivial programs (such as "5") work.

5 The program explained

In principle liveness analysis and strictness analysis are closely related. A term is considered live if there is at least one path in which the term is used. A term is considered strict if it is used in every path. By default, it is assumed a term is dead, so it can be optimized away. However, for strictness analysis an argument is assumed strict, so the application can be replaced by a strict application. See Figure 1 and Figure 2 for the lattices used.

It is possible to combine the analyses to the lattice in Figure 3. Note that $\perp$ here means a term that is both dead and strict, which means it is both never used and always used, so this is impossible to occur in the result. We have not implemented or further investigated this.

In our initial implementation there was no difference between liveness and strictness analysis: if something was live, it was also necessarily strict. This was because the algorithm never needed

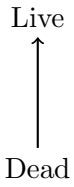

Figure 1: The lattice of liveness analysis.

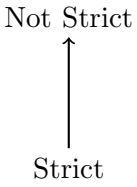

Figure 2: The lattice of strictness analysis.

to join two alternatives as we had no **if**-statements or similar constructs. This meant that every term was either live and strict or dead and not strict, and all the operations we supported only produced live and strict or dead and non-strict terms.

if-statements are different, though, as these have the following signature:

$$(L, S) \to (L, N) \to (L, N) \to \ ?$$

(The signature of the result depends only on where this statement is used, not on the statement itself.)

So we made the choice to implement **if**-statements, and not rewrite it to do strictness analysis instead of liveness analysis, as liveness analysis without adding **if**-statements would be rather uninteresting.

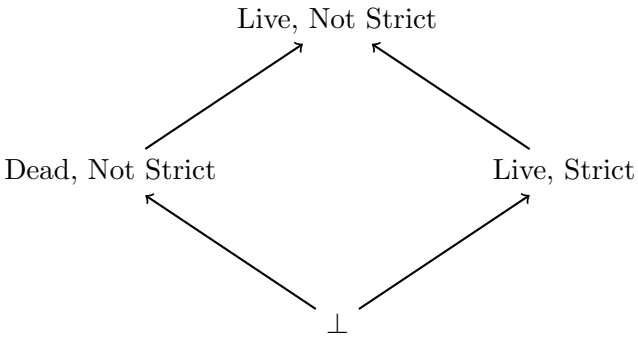

Figure 3: The lattice of combined strictness and liveness analysis.

6 Usage

Our program depends on the standard packages `base`, `containers` (for *Data.Set* etc.), `mtl` (for our *State* monad), `cmdargs` (to make it easier to pass arguments to the executable) and the Compiler Construction package `cco`[1], from which we use the pretty printing module to print

[1]Which can be downloaded from http://www.cs.uu.nl/wiki/bin/view/Cco/CourseResources#CCO_Library

the resulting program.

After `cabal build`, our program can be run with:

```
./dist/build/strictness/strictness input.fun
```

This will run the algorithm on `input.fun`, replace all dead code with $\perp$ (here standing for `undefined`, not to be confused with bottom in the lattice) and pretty print the result. Next, it will take the output and run the algorithm on it again, and compare the result. Applying the algorithm a second time should not change the result. We used this check to verify that we found the minimal solution given the constraints we can find.

We also evaluate the program and the transformed program, to again compare the result to make sure our solution is sound.

In case you were wondering where 'HH' comes from in the naming, it stands for 'Holdermans and Hage'. This is before we heard that Jurriaan had not seen this algorithm before.

7 Test cases

The sample functions to test our program with are residing in the 'test' directory. One can call the main function as shown in Figure 4 and Figure 5 or by invoking the command above with one of the files in the 'test' directory.

8 Conclusion

We have implemented liveness analysis using the algorithm by Stefan Holdermans in a simple functional language. While strictness was the complement of liveness in the original language, we have introduced **if**-statements to avoid this and make the analysis more interesting.

References

Stefan Holdermans and Jurriaan Hage. Making "stricterness" more relevant. In *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '10, pages 121–130, New York, NY, USA, 2010a. ACM. ISBN 978-1-60558-727-1. doi: http://doi.acm.org/10.1145/1706356.1706379. URL http://doi.acm.org/10.1145/1706356.1706379.

Stefan Holdermans and Jurriaan Hage. On the rôle of minimal typing derivations in type-driven program transformation. In *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, LDTA '10, pages 2:1–2:8, New York, NY, USA, 2010b. ACM. ISBN 978-1-4503-0063-6. doi: http://doi.acm.org/10.1145/1868281.1868283. URL http://doi.acm.org/10.1145/1868281.1868283.

```
*Main> :main "../test/fib.fun"
Program: let name: fib = (μfib.λx -> if (x == 0) then 0 else if (x == 1) then
          1 else ((fib (x - 1)) + (fib (x - 2))) endif endif) in (fib 10) end
Pretty printed term:
let name fib =
    nfun fib x =>
                if x == 0 then
                  0
                else
                  if x == 1 then
                    1
                  else
                    (fib (x - 1)) + (fib (x - 2))
                  endif
                endif
in fib 10
end
Pretty printed term after applying transformation twice (should be the same):
let name fib =
    nfun fib x =>
                if x == 0 then
                  0
                else
                  if x == 1 then
                    1
                  else
                    (fib (x - 1)) + (fib (x - 2))
                  endif
                endif
in fib 10
end
Equal, yay!
Eval original: 55
Eval transformed: 55
Equal eval, yay!
```

Figure 4: A sample output from running the program on the Fibonacci test.

```
*Main> :main "../test/const.fun"
Program: let name: const = (λx -> (λy -> x)) in ((const 1) 2) end
Pretty printed term:
let name const =
    fun x =>
          fun y =>
                x
in const 1 _|_
end
Pretty printed term after applying transformation twice (should be the same):
let name const =
    fun x =>
          fun y =>
                x
in const 1 _|_
end
Equal, yay!
Eval original: 1
Eval transformed: 1
Equal eval, yay!
```

Figure 5: Sample a program using *const*.