Alternate Control-Flow Analyses for Defunctionalization in the MLton Compiler

Ryan Shea (rps3099@rit.edu) · Advisor: Dr. Matthew Fluet (mtf@cs.rit.edu)

Background

Defunctionalization is a program transformation in MLton that converts a higher-order program into a first-order program. Control-flow analysis (CFA) is the algorithm that identifies the set of functions that could be invoked at each call site in the program. This analysis is used to guide defunctionalization in the creation of flow sets that replace higher-order functions in the new first-order program. The motivation of this project was to increase the precision of the CFA so that flow sets created by defunctionalization are as small as possible.

Example Program

```plaintext
val fid = fn (f: int -> int) => f
val inc = fn (x: int) => x + 1
val dec = fn (y: int) => y - 1
val neg = fn (z: int) => -z

val a1 = fid inc (* a1 = inc *)
val b1 = a1 1 (* b1 = 2 * *)
val a2 = fid dec (* a2 = dec *)
val b2 = a2 1 (* b2 = 0 * *)
val a3 = neg (* a3 = neg *)
val b3 = a3 1 (* b3 = -1 * *)
```

Simply-Typed Analysis (STA)

The Simply-Typed Analysis uses only the type of a function to determine what could be invoked at a call site. Any function in the program matching the sites expected input and output types are added to the flow set. This analysis was used as a proof of concept for implementing new CFAs in MLton.

STA Output

\[
\begin{align*}
F(fid) &= \{ f \} \\
F(f) &= \{ x, y, z \} \\
F(inc) &= \{ x, y, z \} \\
F(dec) &= \{ y \} \\
F(neg) &= \{ x, y, z \} \\
F(a1) &= \{ x, y, z \} \\
F(a2) &= \{ x, y, z \} \\
F(a3) &= \{ x, y, z \}
\end{align*}
\]

0-CFA (MLton’s Current CFA)

0-CFA generates more precise flow sets than STA by using control-flow information to determine which functions could flow to a call site. 0-CFA gets the zero in its name from the fact that it is context insensitive and does not consider the environment when performing its analysis.

0-CFA Output

\[
\begin{align*}
F(fid) &= \{ f \} \\
F(f) &= \{ x, y, z \} \\
F(inc) &= \{ x, y, z \} \\
F(dec) &= \{ y \} \\
F(neg) &= \{ x, y, z \} \\
F(a1) &= \{ x, y, z \} \\
F(a2) &= \{ x, y, z \} \\
F(a3) &= \{ x, y, z \}
\end{align*}
\]

M-CFA (Proposed New CFA)

M-CFA considers the previous \( M \) calling contexts at each call site when creating a flow set. This leads to an improvement in precision because only functions that flow into each call site from the \( M \)-th calling context get grouped together. This project used an \( M \) of one because it becomes unnecessarily computationally expensive to increase the context factor past one or two in most situations.

M-CFA Output

\[
\begin{align*}
F(fid) &= \{ f \} \\
F(f) &= \{ x, y, z, f_1 \} \\
F(inc) &= \{ x, y, z \} \\
F(dec) &= \{ y \} \\
F(neg) &= \{ x, y, z \} \\
F(a1) &= \{ x, y, z \} \\
F(a2) &= \{ y \} \\
F(a3) &= \{ x, y, z \}
\end{align*}
\]

Results

The results below show that M-CFA has about 99% set distribution between flow sets with zero or one members. M-CFA is also capable of identifying a large amount of functions that are never called, which the other two analyses struggle with.

Conclusion

The goal of this project was to implement an alternative CFA that improved the precision of the MLton defunctionalization pass. M-CFA accomplishes this goal by improving the precision of the CFA pass in MLton considerably compared to 0-CFA.

References

1. Henry Cejtin et al., Flow-Directed Closure Conversion for Typed Languages, Proceedings of the 9th European Symposium on Programming Languages and Systems, p.56-71, April 02, 2000
2. Matthew Might, Yannis Smaragdakis, and David Van Horn. 2010. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. SIGPLAN Not. 45, 6 (June 2010), 305-315. DOI=http://dx.doi.org/10.1145/1809028.1806631