Flambda2 Validator

Irene Yoon
INRIA
France
euisun.yoon@inria.fr

Chris Casinghino
Jane Street
USA
ccasinghino@janestreet.com

Abstract

This talk will describe a validation tool for Flambda2, an optimizing middle-end for OCaml. Its optimizations are centered around inlining (replacing a function call with the function body) and applying simplications that become possible after inlining. Although such a transformation sounds innocuous, it is one of the most important—and tricky to implement—optimizations in the compiler pipeline. We increase confidence in Flambda 2's optimizations by providing a relatively small and declarative definition of reduction to reduce optimized and unoptimized versions of the same program to syntactically equivalent terms. The tool is functional and can validate Flambda2's optimizations for a substantial fraction of the OCaml standard library, our main test suite.

ACM Reference Format:
Irene Yoon and Chris Casinghino. 2024. Flambda2 Validator. In *Proceedings of The 2024 OCaml Users and Developers Workshop (OCaml Workshop '24).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

This talk will describe a validation tool for Flambda2 [1], an optimizing middle-end for OCaml. Its optimizations are centered around inlining (replacing a function call with the function body) and applying simplications that become possible after inlining. Although such a transformation sounds innocuous, it is one of the most important—and tricky to implement—optimizations in the compiler pipeline. Both inlining and simplification are easy to get wrong in an effectful language like OCaml, motivating the need for a validation tool.

The core idea of our validation tool is to increase confidence in Flambda 2's optimizations by providing a relatively small and declarative definition of reduction that is sufficient to reduce optimized and unoptimized versions of the same program to syntactically equivalent terms. We define a lambda-calculus-like simplified core language for the Flambda2 IR, called *Flambda2 Core* ($F\lambda_2^C$). To validate Flambda2's optimization engine, we translate the original and optimized programs to core, normalize both, and compare them for α-equivalence. The approach is shown in Figure 1, where the double arrows illustrate components we have implemented, as described further in Section 2.

OCaml Workshop '24, September, 2024, Milan, Italy
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

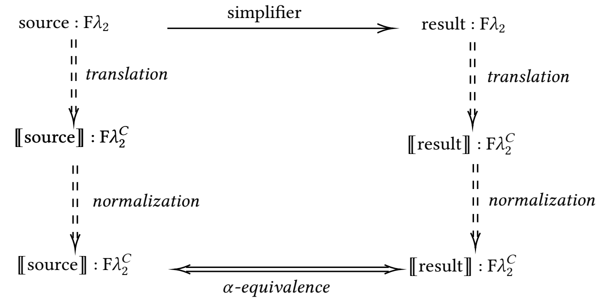

Figure 1. Semantic equivalence for Flambda2

This talk will describe work in progress, and we look forward to feedback and suggestions from the OCaml Workshop community. The tool is functional and can validate Flambda2's optimizations for a substantial fraction of the OCaml standard library, our main test suite—see Section 3 for details. The goal of providing a declarative reduction relation that captures these optimizaitons is so far a partial success, we discuss some particular tricky transformations in Section 4.

2 Methodology

We highlight the major differences between Flambda2 and $F\lambda_2^C$ and how we allow full substitution of terms in $F\lambda_2^C$.

2.1 Full expression subterms

$F\lambda_2^C$ allows for full subexpressions in argument position, so that there can be more aggressive inlining and substitution in the normalization process. Since, for example, substituting an arbitrary let-bound variable for its body is not always sound in the presence of effects, the normalization for $F\lambda_2^C$ substitutes only when it preserves the behavior of the program, which will be explained further in the talk.

2.2 Unifying closure binding of Flambda2

In Flambda2, there are two ways that a binding for a set of closures can be declared. In Flambda2 Core, there is only one syntactic form for this. In $F\lambda_2^C$, closure bindings are written `let Φ = closure f_id @f; closure g_id @g` where Φ is a variable corresponding to the whole set of closures. Each of the function slots can be referred to by using a "slot expression", which is a tuple referring to the set of closures and the function slot. To refer to the function slot that stores `f_id`, we write `slot (Φ, @f)`. Having a uniform representation

111 of closures simplifies dealing with primitives that interact
112 with them (e.g., projecting from the closure environment).
113
114

115 ### 2.3 Unifying recursive lambda expressions

116 In Flambda2, there are lambda-like expressions represent-
117 ing code blocks. Code blocks may be mutually recursive;
118 however, one can also have recursive function definitions
119 through local recursive continuations. In $F\lambda_2^C$, we unify the
120 representation of recursive lambda expressions so that the
121 normalization scheme can apply uniform treatment for deal-
122 ing with recursive sets of closures. In $F\lambda_2^C$, local continuations
123 definitions cannot be recursive; they are lifted as a recur-
124 sive form of lambda. Note that the normalization does not
125 prevent divergence, as can be observed in the next section.
126
127
128

129 ## 3 Results

130 Our primary testsuite for the validator has been the version
131 of the OCaml standard library in the Flambda2 version of
132 the OCaml compiler. We ran the compiler with validation
133 turned on for each file, subject to the limitations described
134 in Section 4. We ran the experiment on a modern AMD Epyc
135 7002 processor with a time limit of 20 minutes per file. On
136 this testsuite of 67 files, the validator successfully validates
137 45 files (67%), fails to validate 14 files (21%), and times out
138 on 8 files (12%).

139 Our experience to date suggests that the remaining valida-
140 tion failures are likely caused by missing reduction rules. The
141 most common cause has been missing rules for particular
142 OCaml primitives, but in some cases we have needed to add
143 more complex rules to account for Flambda2's optimizations,
144 and we discuss some interesting examples in Section 4.

145 The performance of the validator is bimodal. Of the 59
146 files on which it does not time out, 43 complete in under a
147 second, and only 5 require more than a minute. The speed
148 is not strongly correlated with program size—some large
149 files validate quickly and some smaller files time out. Based
150 on this performance profile, we suspect certain constructs
151 or patterns may cause the validator to loop or term size to
152 explode—further investigation is future work.

153 At present the validator comprises 7215 lines of code. This
154 includes the definition of $F\lambda_2^C$, translation from Flambda2
155 to $F\lambda_2^C$, normalization of $F\lambda_2^C$, and equivalence checking of
156 $F\lambda_2^C$ terms. In the version of the compiler we are using for
157 testing, Flambda2 itself comprises 95457 lines of code (for a
158 fair comparison, we included the translation into Flambda2
159 from the previous IR, but not the translation out of Flambda2
160 to the next IR, and did not include tests in either case).

161 At less than 10% the size of Flambda2, we believe it is
162 reasonable to consider the validator a more declarative spec-
163 ification of correctness of the middle-end, despite some of
164 the complications discussed in the next section.
165

166 ## 4 Limitations

167 The goal of this project is to increase confidence in Flambda
168 2's optimizations by providing a relatively small and declar-
169 ative definition of reduction that is sufficient to reduce op-
170 timized and unoptimized versions of the same program to
171 syntactically equivalent terms. While the results from the
172 previous section show partial success, we have also encoun-
173 tered some optimizations performed by Flambda2 that are
174 challenging to validate with this technique. This section pro-
175 vides examples of optimizations we are not yet validating
176 successfully, or where validation required a more compli-
177 cated notion of reduction.
178
179

4.1 Cross-module inlining

180 Flambda2 can perform inlining not just within a compliation
181 unit, but also between compilation units when the compiled
182 version of a library is available. This cross-module inlining
183 does not create any fundamental theoretical obstacle for our
184 approach—we could in principle fully inline every function
185 whose definition is available, even if from another module.
186 However, this would be problematic for the performance of
187 the validator, as term sizes would grow massively.

188 A more nuanced approach would be to *replicate* Flambda2's
189 cross-module inlining decisions, and only inline a function
190 body from another module if this was also done during the
191 actual optmization pass. Conveniently, Flambda2 maintains
192 an "inlining tree" data structure that records these inlining
193 decisions. However, we have not yet implemented this heuris-
194 tic, and our present results are collected with cross-module
195 inlining disabled.
196
197

198 ### 4.2 Function argument introduction/elimination

199 We have encountered two transformations applied by Flambda2
200 that modify the number of arguments to functions. If these
201 functions are not eliminated by reduction, this defeats our
202 simple syntactic equivalence check.

203 The first such example concerns invariant arguments to
204 recursive continuations. Translation from OCaml's Lambda
205 IR into Flambda2 creates recursive continuations both to
206 model recursive functions themselves and as the compilation
207 of loops. In the case that such a recursive continuation has
208 an *invariant* argument—an argument that does not change
209 from recursive call to recursive call—and the continuation
210 is applied only once, Flambda2 will eliminate the argument
211 from the continuation and simply substitute in the value
212 from the application site.

213 To handle this, we have included a "reduction" rule in our
214 system that applies a similar transformation to eliminate
215 arguments. This rule must do a deep analysis of the body
216 of the continuation to detect invariant arguments. As such,
217 this rule reduces the extent to which our reduction relation
218 serves as a simple declarative specification of correctness for
219 Flambda2.
220

The second example concerns *common subexpression elimination* (CSE), where Flambda2 attempts to eliminate duplicated computations of pure expressions. If Flambda2 achieved this goal by creating a simple let binding of the common expression, our straightforward reduction rule for let bindings would validate this optimization. However, as Flambda2 uses continuations for control flow, it instead propagates the subexpression by adding an extra argument to the continuation called by the code where the it is initially calculated. This continuation therefore has an extra argument in the optimized version.

It is possible the validator could accomodate this optimization by applying a similar rule as we used to handle the invariant arguments, where we observe this continuation is always called with the same value and simply eliminate the argument and substitute in the value. However, we have not yet implemented such an optimization, and have instead temporarily disabled the CSE optimization when running validation.

5 Acknowledgements

This work was done in collaboration with Xavier Clerc, Mark Shinwell, and Leo White at Jane Street. The authors also received guidance from the Flambda 2 team at OCamlPro, particularly Guillaume Bury, Pierre Chambart, Nathanaëlle Courant, and Vincent Laviron.

References

[1] Flambda2 Team. 2023. Efficient OCaml compilation with Flambda 2. OCaml Workshop.