

A Preliminary Type- and Control-Flow Analysis for System F_ω

Dongyu Wu¹ and Matthew Fluet¹ [0000–0002–4194–7618]

Rochester Institute of Technology, Rochester NY 14623, USA
dw1823@g.rit.edu, mtf@cs.rit.edu

Abstract. A type- and control-flow analysis is a program analysis that yields both type-flow information, approximating the types that may instantiate type variables, and control-flow information, approximating the values (especially λ - and Λ -expressions) that may be bound to variables. Moreover, the each of the flows informs the other; control-flow establishes the types that may instantiate Λ -bound type variables by determining the Λ -expressions that flow to a type-application expression, while type-flow filters control-flow by rejecting the flow of values with static types that are incompatible (with respect to the type-flow information) with the static type of the receiving variable.

In previous work [7, 8, 1], we introduced a (monovariant) type- and control-flow analysis for System F (with recursion). While System F has an expressive type system and has served as a useful core calculus in which to express interesting language features, it only allows abstraction over types and does not allow abstraction over type constructors. Increasingly, researchers are looking at System F_ω , with both term- and type-level abstraction over types of arbitrary kind, as a core calculus in which to explore advanced language features. Hence, we are motivated to define a type- and control-flow analysis for System F_ω that is able to analyze the rich structure of System F_ω types.

In this work, we present a preliminary type- and control-flow analysis for System F_ω (with recursion). As in previous work, we give both a specification-based formulation of the analysis, used to prove soundness of the analysis, and a flow-graph-based formulation, used to guide the implementation of an algorithm. While the macro structure of the development for System F_ω follows that for System F, moving to System F_ω introduces subtle challenges that have left some unanswered questions about the meta-theory. In particular, the decidability of type compatibility defined in terms of System F_ω 's type equivalence remains an open question. In order to obtain an algorithm, our flow-graph-based formulation uses a restricted form of type equivalence and performs a 0CFA at the type-level, yielding an analysis result that is less precise than the “best” (but as of yet, uncomputable) analysis result accepted by the specification-based formulation. Our soundness results have been formalized in the Coq proof assistant.

This work is based in part on the first author's MS thesis [43].

Keywords: Control-flow analysis · Type-flow analysis · System F_ω .

1 Introduction

Type- and control-flow analyses are a class of program analyses that combine, in a mutually beneficial manner, a control-flow analysis and a type-flow analysis. Control-flow analyses [18, 39, 38, 25, 20] are themselves a class of program analyses that approximate, at compile time, the flow of first-class functions (and other values) in a program: which first-class functions (and other values) might be bound to a given variable or returned by a given expression at run time. Similarly, type-flow analyses are a class of program analyses that approximate, at compile time, the flow of types in a program: which types might be bound to a given type variable at run time.

Control-flow analyses are an important enabling technology for the compilation and optimization of functional languages [37]. For example, control-flow analyses can be used to guide inlining [40, 21, 2], defunctionalization [4], and the specialization of high-level abstractions [32]. While type-flow analyses are a more recent development, they can similarly be used to guide defunctionalization [5], monomorphisation [16], and the optimization of intensional polymorphism [14].

In a type- and control-flow analysis, each of the underlying flow analyses informs the other. In one direction, the control-flow analysis is used to determine which first-class polymorphic functions (e.g., λ -expressions) may be applied at a type application (and, therefore, which types may instantiate which type variables). In the other direction, the type-flow analysis is used to filter the values that may be bound to variables (by rejecting the flow of values with static types that are incompatible under the type flow with the static type of the receiving variable). Critically, the soundness of a type- and control-flow analysis depends on type soundness and the well-typedness of the program under analysis. Type soundness and well-typedness ensure that, when a value is bound to a variable at run time, the actual (necessarily closed) type of the value will correspond to the actual (necessarily closed) type of the receiving variable. The type-flow information allows the analysis to over-approximate the set of possible actual (necessarily closed) types for a static (possibly open) type by (repeatedly) mapping any free type variable to one of the types to which it may be bound. Suppose the control-flow analysis proposes that a value flows to a receiving variable, but there is no possible actual type for the value that corresponds to any possible actual type of the receiving variable, in which case the types involved are said to be incompatible. Type soundness ensures that this value will never be bound to this variable at run time and the proposed flow can be rejected.

In previous work [7, 8, 1], we introduced a (monovariant) type- and control-flow analysis for System F (with recursion). The underlying control-flow analysis is 0CFA [40, 25], the classic monovariant control-flow analysis that was formulated for the untyped lambda calculus. The underlying type-flow analysis is novel, but essentially a straightforward monovariant analysis similar in spirit to 0CFA. We gave both a specification-based formulation of the type- and control-flow analysis [7, 8], used to prove soundness of the analysis, and a flow-graph-based formulation [1], used to guide the implementation of an efficient $O(l^3 + m^4)$ al-

gorithm, where l is the size of expressions in the program and m is the size of types in the program.

System F [10, 33, 30, 13], the polymorphic lambda calculus, is an important point in the space of type systems. It and simple extensions thereof have been used effectively as typed intermediate languages in compilers for functional languages [42, 27]. However, System F has only limited support for polymorphism. While it allows for abstraction over a type, it does not allow for abstraction over a *type constructor*. For example, while we can express the types of the polymorphic and higher-order functions that map over lists ($\text{listMap} : \forall \alpha. \forall \beta. (\alpha \rightarrow \beta) \rightarrow \text{List } \alpha \rightarrow \text{List } \beta$) and trees ($\text{treeMap} : \forall \alpha. \forall \beta. (\alpha \rightarrow \beta) \rightarrow \text{Tree } \alpha \rightarrow \text{Tree } \beta$), we cannot express the type of a function withMap that accepts a mapping function (either listMap or treeMap) and a data structure containing integers (either List Int or Tree Int) and returns a pair, where the first component is the result of mapping the $\text{odd} : \text{Int} \rightarrow \text{Bool}$ predicate and the second component is the result of mapping the $\text{inc} : \text{Int} \rightarrow \text{Int}$ function. Such a function wishes to abstract over List or Tree , which are type constructors and not types.

To achieve this level of abstraction, one must turn to System F_ω [10, 28, 11], the higher-order polymorphic lambda calculus. System F_ω extends System F with both expression-level abstraction over type constructors (generalizing λ -expressions and \forall -types) and type-level functions and applications. For example, System F_ω can express the type of the function withMap described above as follows:

$$\text{withMap} : \forall c::\star \Rightarrow \star. (\forall \alpha::\star. \forall \beta::\star. (\alpha \rightarrow \beta) \rightarrow c \alpha \rightarrow c \beta) \rightarrow c \text{Int} \rightarrow \text{Pair } (c \text{Bool}) (c \text{Int})$$

where c is a type variable of *kind* $\star \Rightarrow \star$, meaning that it must be instantiated by a type corresponding to a function from (proper) types to (proper) types, while α and β are type variables of *kind* \star , which must be instantiated by a (proper) type. Kinds are the “types of types” and ensure that types are used correctly.

The increased expressive power of System F_ω has made it and various restrictions and extensions thereof an attractive typed intermediate language for modern functional languages; for example, it can serve as a target language for advanced ML-like module languages [36, 34, 35], for a typed store-passing translation of general references [31], for datatype-generic programming [3]. Thus, we seek to define a type- and control-flow analysis for System F_ω that is able to analyze the rich structure of System F_ω types.

In this work (based on the first author’s MS thesis [43]), we present a preliminary type- and control-flow analysis for System F_ω . As in previous work, we give both a specification-based formulation of the analysis, used to prove soundness of the analysis, and a flow-graph-based formulation, used to guide the implementation of an algorithm. While the general structures of the specification-based and the flow-graph-based formulations of the type- and control-flow analysis for System F_ω follow the corresponding formulations of the type- and control-flow analysis for System F, as do the proofs of soundness of the formulations (via a direct proof for the specification-based formulation and via a correspondence with the specification-based formulation for the flow-graph-based formulation), moving to System F_ω introduces subtle challenges that have left some unanswered

questions about the meta-theory. Essentially, the notion of when one type “corresponds” to another (which was intentionally left vague in the discussion above about how a type-flow analysis informs the control-flow analysis) is significantly different in System F and System F_ω . In System F, simple syntactic equality (up to α -equivalence) determines when one type “corresponds” to another and this syntactic equality can be used to define when two types are compatible for the purposes of rejecting the flow of values with static types that are incompatible with the static type of the receiving variable. But, in System F_ω , *type equivalence* determines when one type “corresponds” to another; type equivalence is a non-trivial relationship, because it must judge an application of a type-level function to a type argument as equivalent to the substitution of the type argument for the function parameter in the body of the function (i.e., the β -reduction). This type equivalence can be used to define when two types are compatible, but the decidability of this definition of type compatibility in the specification-based formulation of the type- and control-flow analysis for System F_ω remains an open question (as does the existence of an efficient algorithm). In order to obtain an algorithm, our flow-graph-based formulation of the type- and control-flow analysis for System F_ω uses additional restrictions and over-approximations; while it remains sound (the analysis result obtained by the flow-graph-based formulation is accepted by the specification-based formulation), it applies to fewer programs and can be less precise (there are “better” analysis results that are accepted by the specification-based formulation). In particular, we restrict the β -reduction used in type equivalence to closed types (rejecting as ill-typed some programs that would be accepted as well-typed without the restriction) and we perform a OCFA at the type-level (introducing imprecision).

The final paper will fully develop the type- and control-flow analysis for a variant of System F_ω , extended with recursive λ - and Λ -expressions, presented in administrative normal form (ANF), and using type variables for Λ -expressions and for expression- and type-level **let**-bound types, but using a hybrid type variable / de Bruijn index for λ -types. Of note, we use an ANF representation for both expressions and types, similar to the $\lambda_{gc}^{\rightarrow\forall}$ language [22]. An operational semantics for our language can be given in the style of the ANF environment- and continuation-based C_aEK abstract machine [6] and a type system can be given by extending the standard type system for System F_ω with support for type definitions [22, 41].

Our specification-based formulation of the type- and control-flow analysis is presented as a judgment that asserts the various constraints that an acceptable analysis result must satisfy [23, 9, 26, 24, 25], where the analysis result is the pair of an abstract type environment (mapping type variables to abstract types) and an abstract value environment (mapping expression variables to abstract values). Flow soundness requires that acceptable abstract type and value environments conservatively approximate all concrete type and value environments that arise during the evaluation of the program. Soundness of the flow analysis relies crucially on soundness of the type system and the well-typedness of the program under analysis.

Our flow-graph-based formulation of the type- and control-flow analysis is presented as a collection of judgements that define a directed graph, with nodes corresponding to program constituents and edges corresponding to the flow of abstract types and values from one node to another [17, 15]. Because the flow-graph-based formulation is defined in terms of program constituents, it is difficult to handle substitutions, such as arise in the β -reduction rule for type equivalence, because variable renaming and/or de Bruijn-index shifting may yield a term that does not appear in the program. By restricting the β -reduction rule for type equivalence to closed types, we ensure that no variable renaming and/or de Bruijn-index shifting occurs and substitution yields a term that does appear in the program. Furthermore, to efficiently model the compatibility of an application of a type-level function to a type argument with the result of the substitution, we use a OCFA at the type level to obtain an over-approximation of the substitution by examining the abstract types that flow out of the body of the type-level function.

References

1. Adsit, C., Fluet, M.: An efficient type- and control-flow analysis for system f. In: Tobin-Hochstadt, S. (ed.) IFL’14: Proceedings of the 26nd International Symposium on Implementation and Application of Functional Languages. Association for Computing Machinery, Boston, MA, USA (2014)
2. Bergstrom, L., Fluet, M., Le, M., Reppey, J., Sandler, N.: Practical and effective higher-order optimizations. In: Chakravarty, M. (ed.) ICFP’14: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming. pp. 81–93. ACM, Gothenburg, Sweden (Sep 2014)
3. Cai, Y., Giarrusso, P.G., Ostermann, K.: System f-omega with equirecursive types for datatype-generic programming. In: Majumdar, R. (ed.) POPL’16: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 30–43. Association for Computing Machinery, St. Petersburg, FL, USA (Jan 2016)
4. Cejtin, H., Jagannathan, S., Weeks, S.: Flow-directed closure conversion for typed languages. In: Smolka, G. (ed.) ESOP’00: Proceedings of the Ninth European Symposium on Programming. Lecture Notes in Computer Science, vol. 1782, pp. 56–71. Springer-Verlag, Berlin, Germany (Mar 2000)
5. Contractor, M.R., Fluet, M.: Type- and control-flow directed defunctionalization. In: Chitil, O. (ed.) IFL’20: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages. Association for Computing Machinery, Canterbury, United Kingdom / virtual (2021)
6. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: Cartwright, R. (ed.) PLDI’93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Languages Design and Implementation. pp. 237–247. ACM, Albuquerque, New Mexico (Jun 1993)
7. Fluet, M.: A type- and control-flow analysis for System F. In: Hinze, R. (ed.) IFL’12: Post-Proceedings of the 24th International Symposium on Implementation and Application of Functional Languages. pp. 122–139. Lecture Notes in Computer Science, Springer-Verlag, Oxford, England (2013)

8. Fluet, M.: A type- and control-flow analysis for System F. Tech. rep., Rochester Institute of Technology (February 2013), <https://ritdml.rit.edu/handle/1850/15920>
9. Gasser, K.L.S., Nielson, F., Nielson, H.R.: Systematic realisation of control flow analyses for CML. In: Tofte, M. (ed.) ICFP'97: Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming. pp. 38–51. Association for Computing Machinery, Amsterdam, The Netherlands (Jun 1997)
10. Girard, J.Y.: Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In: Fenstad, J.E. (ed.) Proceedings of the 2nd Scandinavian Logic Symposium. Studies in Logic and the Foundations of Mathematics, vol. 63, pp. 63–92. Elsevier, Amsterdam, Netherlands (1971)
11. Harper, R.: Higher Kinds, chap. 18. In: Harper [12] (2016)
12. Harper, R.: Practical Foundations for Programming Languages (Second Edition). Cambridge University Press (2016)
13. Harper, R.: System F of Polymorphic Types, chap. 16. In: Harper [12] (2016)
14. Harper, R., Morrisett, G.: Compiling polymorphism using intensional type analysis. In: Lee [19], pp. 130–141
15. Heintze, N., McAllester, D.: On the cubic bottleneck in subtyping and flow analysis. In: Winskel, G. (ed.) Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS '97). pp. 342–351. Warsaw, Poland (Jun 1997)
16. Hoang, T., Trunov, A., Lampropoulos, L., Sergey, I.: Random testing of a higher-order blockchain language (experience report). Proceedings of the ACM on Programming Languages **6**(ICFP) (Aug 2022). <https://doi.org/10.1145/3547653>
17. Jagannathan, S., Weeks, S.: A unified treatment of flow analysis in higher-order languages. In: Lee [19], pp. 393–407
18. Jones, N.D.: Flow analysis of lambda expressions (preliminary version). In: Even, S., Kariv, O. (eds.) ICALP'81: Proceedings of the 8th International Colloquium on Automata, Languages and Programming. Lecture Notes in Computer Science, vol. 115, pp. 114–128. Springer-Verlag, Acre (Akko), Israel (Jul 1981)
19. Lee, P. (ed.): POPL'95: Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Association for Computing Machinery, San Francisco, California (Jan 1995)
20. Midtgaard, J.: Control-flow analysis of functional programs. ACM Computing Surveys **44**(3), 10:1–10:33 (Jun 2012)
21. Might, M., Shivers, O.: Environmental analysis via Δ CFA. In: Peyton Jones, S. (ed.) POPL'06: Proceedings of the 33rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 127–140. Association for Computing Machinery, Charleston, South Carolina (Jan 2006)
22. Morrisett, G., Harper, R.: Semantics of memory management for polymorphic languages. In: Gordon, A.D., Pitts, A.M. (eds.) Higher-Order Operational Techniques in Semantics, pp. 175–226. Publications of the Newton Institute, Cambridge University Press (1998)
23. Nielson, F., Nielson, H.R.: Infinitary control flow analysis: a collecting semantics for closure analysis. In: Jones, N.D. (ed.) POPL'97: Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 332–345. Association for Computing Machinery, Paris, France (Jan 1997)
24. Nielson, F., Nielson, H.R.: Interprocedural control flow analysis. In: Swierstra, S.D. (ed.) ESOP'99: Proceedings of the Eighth European Symposium on Programming. Lecture Notes in Computer Science, vol. 1576, pp. 20–39. Springer-Verlag, Amsterdam, The Netherlands (Mar 1999)

25. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag (1999)
26. Nielson, H.R., Nielson, F.: Flow logics for constraint based analysis. In: Koskimies, K. (ed.) CC'98: Proceedings of the 7th International Conference on Compiler Construction. Lecture Notes in Computer Science, vol. 1383, pp. 109–127. Springer-Verlag, London, UK (Apr 1998)
27. Peyton Jones, S.: Compiling Haskell by program transformation: A report from the trenches. In: Nielson, H.R. (ed.) ESOP'96: Proceedings of the Sixth European Symposium on Programming. Lecture Notes in Computer Science, vol. 1058, pp. 18–44. Springer-Verlag, Linköping, Sweden (Apr 1996)
28. Pierce, B.C.: Higher-Order Polymorphism, chap. 30. In: Pierce [29] (2002)
29. Pierce, B.C.: Types and Programming Languages. The MIT Press (2002)
30. Pierce, B.C.: Universal Types, chap. 23. In: Pierce [29] (2002)
31. Pottier, F.: A typed store-passing translation for general references. In: Sagiv, M. (ed.) POPL'11: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 147–158. Association for Computing Machinery, London, United Kingdom (Jan 2011)
32. Reppy, J., Xiao, Y.: Specialization of cml message-passing primitives. In: Felleisen, M. (ed.) POPL'07: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 315–326. Association for Computing Machinery, Nice, France (Jan 2007)
33. Reynolds, J.: Towards a theory of type structure. In: Robinet, B. (ed.) Proceedings of the First International Symposium on Programming. Lecture Notes in Computer Science, vol. 19, pp. 408–425. Springer-Verlag, Paris, France (Apr 1974)
34. Rossberg, A.: 1ml – core and modules united (f-ing first-class modules). In: Reppy, J. (ed.) ICFP'15: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming. pp. 35–47. ACM, Vancouver, BC, Canada (Sep 2015)
35. Rossberg, A.: 1ml – core and modules united. Journal of Functional Programming **28**, e22 (2018)
36. Rossberg, A., Russo, C., Dreyer, D.: F-ing modules. In: Benton, N. (ed.) TLDI'10: Proceedings of the Fifth ACM SIGPLAN Workshop on Types in Language Design and Implementation. pp. 89–102. Madrid, Spain (Jan 2010)
37. Serrano, M.: Control flow analysis: a functional languages compilation paradigm. In: George, K.M., Carroll, J., Oppenheim, D. (eds.) SAC'95: Proceedings of the 1995 ACM Symposium on Applied Computing. pp. 118–122. Association for Computing Machinery, Nashville, Tennessee (Feb 1995)
38. Sestoft, P.: Replacing function parameters by global variables. In: Stoy, J.E. (ed.) FPCA'89: Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture. pp. 39–53. Association for Computing Machinery, London, England (Sep 1989)
39. Shivers, O.: Control-flow analysis in Scheme. In: Schwartz, M.D. (ed.) PLDI'88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Languages Design and Implementation. pp. 164–174. Association for Computing Machinery, Atlanta, Georgia (Jun 1988)
40. Shivers, O.: Control-Flow Analysis of Higher-Order Languages or Taming Lambda. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania (May 1991), Technical Report CMU-CS-91-145
41. Stone, C.A.: Type definitions. In: Pierce, B.C. (ed.) Advanced Types and Programming Languages. The MIT Press (2005)

42. Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., Lee, P.: TIL: a type-directed optimizing compiler for ML. In: Fischer, C. (ed.) PLDI'96: Proceedings of the ACM SIGPLAN 1996 Conference on Programming Languages Design and Implementation. pp. 181–192. ACM, Philadelphia, Pennsylvania (May 1996)
43. Wu, D.: A Type- and Control-Flow Analysis for System F_ω . Master's thesis, Rochester Institute of Technology, Rochester, NY (Oct 2023), <https://www.proquest.com/docview/2884521850>