Incremental Bidirectional Typing via Order Maintenance

Thomas J. Porter[1] Marisa Kirisame[2] Liam Mulcahy[1] Pavel Panchekha[2] Cyrus Omar[1]

[1]University of Michigan [2]University of Utah

Motivation

In *live programming systems*, editor services such as type checking and evaluation are continually provided while the user is editing the program. The live paradigm offers benefits to developer experience and productivity. Many editor services are most easily expressed as stateless transformations that take only the current the program text as input, and compute the result from scratch. For live programming at scale, these services cannot be implemented naively as pure functions, since the execution time will grow with the size of the program, and at some point will take longer than the time between edits. *Incremental* type checking aims to overcome this limitation by maintaining type information between states of the program. We present an algorithm for fine-grained incremental maintenance of typing information for the *marked lambda calculus* [9] across structural edits.

Problem

The marked lambda calculus [9] defines a pure, total function from an ordinary program in the typed lambda calculus to a *marked* program, which is the same as the input except for the addition of *marks*, localized type error annotations. It is these marks, as well as synthesized and analyzed types at subterms, that we to incrementally maintain.

We consider structural edit actions, such as the insertion or deletion of an AST node, as our atomic changes to the input. We make no assumptions about the sequence of edit actions, since we aim to handle cases such as collaborative editing with many cursors or compound actions.

We make no assumption about the form of the program, such as being partitioned into files. We therefore incrementalize typing at the finest granularity possible: individual syntax nodes. We make no assumptions about the size of the program, and in fact target very large scale programs. Since the typing updates incurred by an edit may be arbitrarily expensive to compute, we must not halt normal editor operation while computing these updates. This constraint fundamentally shapes our solution.

Actions and Update Propagation

Our solution maintains an enriched program data structure (EP), which stores local typing and binding information and is endowed with *update propagation dynamics.* A program edit updates the EP efficiently, after which the EP takes a sequence of efficient steps to propagate the changes generated by the edit. By separating the incremental update into an *action judgment* which updates the EP when an action is applied at a cursor location and an *update propagation judgment* which advances a not-fully propagated EP forward by one step, even those actions which necessitate long recomputation times do not block further actions from being taken; indeed, actions and update propagation steps can be applied in any order without threatening the validity of the incremental statics with respect to the from-scratch analysis.

Order Maintenance

This local propagation mechanism exploits the type system's locality, making it suitable for a bidirectional typing discipline [4]. However, even such a "local" type system exhibits nonlocality between binders and their bound variables. We opt to maintain pointers along these bindings, rather than walking the spine of the program.

To maintain these binders, we employ an *order maintenance data structure*, a totally ordered collection of elements that supports efficient comparison between elements and insertion of new elements. By annotating each AST node with an interval in the order maintenance data structure, it is possible to test whether one node is the ancestor of another in logarithmic time. This efficient test is the basis of our algorithm for maintaining binding pointers.

Related Work

This work follows the work on adaptive functional programming [1] in employing the order maintenance data structure of Dietz and Sleator [3] to maintain the dynamic dependency structure between parts of the program. The prior work on adaptive functional programming presents general translations to incremental programs, using order maintenance to prioritize the recomputation of intermediate values. On the other hand, the present work is specialized to bidirectional typing and uses order maintenance to maintain scoping data.

Prior approaches to incremental typing utilize a task engine [8], derive memoized typing rules [2], or translate typing rules to a Datalog program that can be solved incrementally [6, 7]. This last technique uses co-contextual typing [5], in which binding information is propagated bottom up rather than top down, to overcome issues related to binding structure updates. Compared to these approaches, our incremental typing system is less general, but achieves very fine granularity and direct binding updates by specializing to a bidirectionally typed lambda calculus.

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert Harper. 2002. Adaptive functional programming. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, John Launchbury and John C. Mitchell (Eds.). ACM, 247–259. https://doi.org/10.1145/503272.503296

[2] Matteo Busi, Pierpaolo Degano, and Letterio Galletta. 2019. Using Standard Typing Algorithms Incrementally. In *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11460)*, Julia M. Badger and Kristin Yvonne Rozier (Eds.). Springer, 106–122. https://doi.org/10.1007/978-3-030-20652-9_7

[3] Paul F. Dietz and Daniel Dominic Sleator. 1987. Two Algorithms for Maintaining Order in a List. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, Alfred V. Aho (Ed.). ACM, 365–372. https://doi.org/10.1145/28395.28434

[4] Jana Dunfield and Neel Krishnaswami. 2022. Bidirectional Typing. *ACM Comput. Surv.* 54, 5 (2022), 98:1–98:38. https://doi.org/10.1145/3450952

[5] Sebastian Erdweg, Oliver Bracevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. 2015. A co-contextual formulation of type rules and its application to incremental type checking. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 880–897. https://doi.org/10.1145/2814270.2814277

[6] André Pacak, Sebastian Erdweg, and Tamás Szabó. 2020. A systematic approach to deriving incremental type checkers. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 127:1–127:28. https://doi.org/10.1145/3428195

[7] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: a DSL for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 320–331. https://doi.org/10.1145/2970276.2970298

[8] Guido Wachsmuth, Gabriël D. P. Konat, Vlad A. Vergu, Danny M. Groenewegen, and Eelco Visser. 2013. A Language Independent Task Engine for Incremental Name and Type Analysis. In *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8225)*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.). Springer, 260–280. https://doi.org/10.1007/978-3-319-02654-1_15

[9] Eric Zhao, Raef Maroof, Anand Dukkipati, Andrew Blinn, Zhiyi Pan, and Cyrus Omar. 2024. Total Type Error Localization and Recovery with Holes. *Proc. ACM Program. Lang.* 8, POPL (2024), 2041–2068. https://doi.org/10.1145/3632910