Simplifying incremental code with IODyn

KYLE HEADLEY*, University of Colorado Boulder

Ad-hoc incremental computation appears in much of modern software. A program is incremental if re-executing it with changed inputs is faster than from-scratch recomputation. General purpose incremental computation solutions exist, but they require specialized code for each incremental function definition and invocation. In order to relieve this burden, we previously implemented a data structure with implicit incremental behavior. Now, we're working on a lightly annotated incremental language, IODyn, to make use of incremental data structures.

Additional Key Words and Phrases: incremental computation, data collections, IODyn, Adapton

1 INTRO

Incremental computation (IC) is used explicitly in many applications through ad-hoc caching of data that will be reused. IC can be specialized to a particular task, or be general purpose with libraries and methodologies usable in any situation. But general purpose IC can be tricky to get right in complex situations. Cache must be able to account for changing and inserting data, be able to access specific past results regardless of changes, and be tuned for performance. The available libraries for IC explicitly require this information from programmers, so they need to figure out how to handle it. This can be a time-consuming burden.

The IC engine Adapton [Hammer et al. 2015] uses "names" to handle cache requirements, and does not have a collections library. I lead a new project, IODyn, which is in development to simplify Adapton, allowing programmers to avoid names altogether. It serves as a model for how to handle implicit incrementalization with large data sets. The hope is that the model can be integrated into popular languages, providing the benefits of IC with only minor annotation.

2 IODYN

The IODyn project has three main parts: a source language, a translation to Typed Adapton [Hammer and Dunfield 2016], and a collections library. Each part has a formal definition and an implementation. The collections library is based on previous work on the Giraz [Headley 2017], an incremental data structure for sequences. IODyn defines the interface to the Giraz and additional data structures as primitive operations.

Source. The IODyn source language is based on the simply typed lambda calculus with call by push value semantics. This base is augmented with "roles" and hints. Adapton defines the roles of "editor", which makes changes to input data, and "archivist", which computes or updates output data. IODyn defines hints in lieu of names, which can suggest caching at key points or demarcate a section of code that may have unique incremental behavior. The source language has proofs showing the cases where the type of the program result is guaranteed to match the type of the program expression. These cases are: when fixed-point calculations are not used (strong normalization), or whenever the program terminates (soundness).

IODyn source uses bidirectional type checking, which supports polymorphic primitive operations without the complexity of full polymorphism. Primitive operations in IODyn support the

*PhD Student, ACM Member 1004593

Author's address: Kyle Headley, University of Colorado Boulder, kyle.headley@colorado.edu.

2017. XXXX-XXXX/2017/11-ART
https://doi.org/

interface between the core language and the incremental data structures that handle performance. A bidirectional type checker can use the type information from parts of an expression to verify other parts, like making sure two parameters have the same type. It does not create and pass around type variables, and may require annotation in key locations like function definitions.

IODyn source is implemented as a DSL within the Rust language, parsed using Rust macros. The Rust compiler processes macro invocations into a mid-level syntax with token lists inside balanced parentheses. Macro-based folds search this syntax for key symbols and recursive macros handle program structure. These operations allow a simple source syntax that's easy to extend and doesn't require any additional tooling. The source evaluation is non-incremental to provide a specification of the target incremental program.

Translation. IODyn's translation is defined by a judgment taking a typed expression of the source language to an expression of Typed Adapton. Typed Adapton statically ensures the well-formedness of the incremental program. There are projection proofs that the translation implies the typing of both source and target expressions. There is also a soundness proof, that if a translation and evaluation exist for an expression, then the translation of the evaluation and the evaluation of the translation have the same result.

3 UPCOMING ACTIVITY

At the time of this writing, we have completed most of the work described above, with the translation implementation as the next step. Typed Adapton also needs an implementation, which will become part of the IODyn project. After that, we will build tests to demonstrate the capability of the full IODyn pipeline, and evaluate incremental performance. Finally, IODyn will be enhanced with additional data structures and more intuitive translation hints.

4 RELATED WORK

There is work on "traceable data structures" [Acar et al. 2010] to build incremental collections libraries that work within the SAC [Acar et al. 2006] framework. The results look good, but the product is mostly theoretical: an implementation was not made public. SAC is similar to Adapton in purpose, though the authors focus on applying their methods to specific problems.

Janestreet has an Ocaml reinterpretation of SAC in the form of a library called "incremental". They simplify some of the annotations to make the work accessible to a wider audience. Users of "incremental" still need to understand what the library is doing, because it is optimized for performance over soundness, leading to some side cases that require additional planning.

5 CONCLUSION

The IODyn project approaches the point where it will be a full pipeline for compiling simple source code into an optimized incremental executable. It will be able to handle changes to large amounts of data, recomputing results in asymptotically less time than initial computation. Work in progress is available in two github repositories: cuplv/iodyn.rust, for data structures, and cuplv/iodyn-lang.rust, for the core language.

IODyn will allow non-experts of IC to use simple designs to build incremental applications that are type-safe, incrementally sound (identical results as from-scratch code), and with performance rivaling more complex programs. It is our goal to bring general-purpose incremental computing to a wider range of users, and IODyn is poised to do so.

REFERENCES

Umut Acar, Guy Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. 2006. A Library for Self-Adjusting Computation. *Electron. Notes Theor. Comput. Sci.* 148, 2 (March 2006), 127–154. https://doi.org/10.1016/j.entcs.2005.11.043

Umut A. Acar, Guy E. Blelloch, Ruy Ley-Wild, Kanat Tangwongsan, and Duru Türkoglu. 2010. Traceable data types for self-adjusting computation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. 483–496. https://doi.org/10.1145/1806596.1806650

Matthew A. Hammer and Joshua Dunfield. 2016. Typed Adapton: Refinement types for nominal memoization. *CoRR* abs/1610.00097 (2016). arXiv:1610.00097 http://arxiv.org/abs/1610.00097

Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael W. Hicks, and David Van Horn. 2015. Incremental computation with names. 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*. 748–766. https://doi.org/10.1145/2814270.2814305

Kyle Headley. 2017. Tuning Data and Control Structures for Incremental Computation. (2017). https://pldi17.sigplan.org/event/ic-2017-papers-tuning-data-and-control-structures-for-incremental-computation