Implementing Implicit Self-Adjusting Computation

Yan Chen Joshua Dunfield Matthew A. Hammer Umut A. Acar
Max Planck Institute for Software Systems
{chenyan,joshua,hammer,umut}@mpi-sws.org

September 17, 2011

1 Synopsis

Computational problems that involve dynamic data have been an important subject of study in programming languages. Recent advances in self-adjusting computation have developed techniques that enable programs to respond automatically and efficiently to dynamic changes in their inputs. But these techniques have required an explicit programming style, where the programmer must use specific monadic types and primitives to identify, create and operate on data that can change over time. Our paper, "Implicit Self-Adjusting Computation for Purely Functional Programs" (ICFP '11), describes the theory underlying *implicit* self-adjusting computation, where the programmer need only annotate the (top-level) input types of the programs to be translated. A type-directed translation rewrites the (purely functional) source program into an explicitly self-adjusting target program. The subject of this talk is a prototype implementation (an extension of MLton) and experimental results that are competitive with explicitly self-adjusting handwritten code.

2 Implicit Self-Adjusting Computation

In self-adjusting computation, programs respond efficiently to changes in their input. The parts of the computation that could be affected by such changes are *changeable*; the parts that cannot be affected are *stable*. (Not all of a program's input need be changeable, but if we actually change the "stable" part of the input, we must run the program from scratch.)

In the information flow type system of Pottier and Simonet (2003), data that depends on *high-security* data is high-security; other data, uninfluenced by high-security data, is low-security. In our setting, data that depends on changeable data has changeable level. The resulting type system (Chen et al. 2011)[1] enriches ML types with levels on base types (e.g. integers), datatypes (e.g. lists), and functions. These types guide an automatic translation that produces explicitly self-adjusting code. Types can be polymorphic in their levels; a given function at stable type corresponds to the original source-level function, but at changeable type requires explicit primitives to access the dependency graph used by change propagation. Thus, the translation must generate monomorphic versions of level-polymorphic functions.

3 Implementation

We have implemented the proposed approach by adding level types to MLton. This integration yields direct language support for self-adjusting computation, which is necessary to infer types and to enforce numerous invariants required for self-adjusting computation.

The extended language allows the programmer to annotate ML types with the level qualifiers `$C` (changeable) and `$S` (stable). For example, "`int $S list $C`" describes lists with stable heads and changeable tails, while "`int $C list $S`" describes lists with changeable heads and stable tails (lists whose length is not expected to change).

[1] This paper, which we'll present at ICFP following the workshop, describes the underlying theory, but not the implementation or the experiments, which are the focus of this presentation.

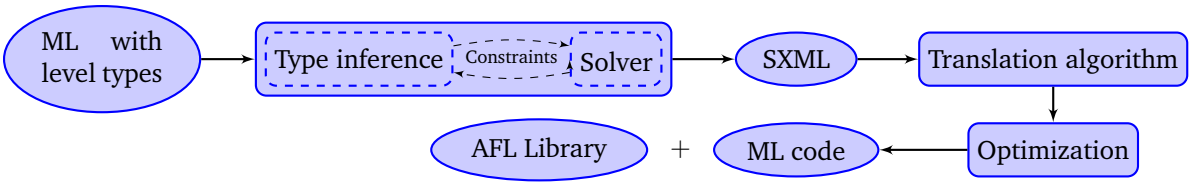

Figure 1: Structure of Implicit Self-Adjusting Computation Compiler

Figure 1 shows the structure of the compiler. Our implementation carries level type annotations through the front end of the MLton compiler up to the intermediate language SXML, where monomorphization (elimination of all Standard ML polymorphism) and A-normalization take place. We apply our translation algorithm to SXML. Apart from generalizing our algorithm to handle ML datatypes, which was relatively straightforward, our implementation closely follows the formal framework in our ICFP paper.

We extended SXML with explicit self-adjusting computation primitives. Our compiler also generates a Standard ML version of the translated SXML code, allowing us to link the translated code against an existing library (Acar et al. 2009) that defines the self-adjusting computation primitives. This enables comparing translated code directly with previous implementations. No further changes to the compiler were needed.

Our current implementation does not do type inference, relying instead on user annotations. We are in the process of implementing the type inference engine.

Compared to handwritten explicit self-adjusting code, the translation algorithm generates redundant sequences of primitives. For example, the expression `y:int $C` becomes **mod** (**read** `y` **as** `y'` **in write** `y'`), which is equivalent to `y`. We implemented an optimization phase after the translation that applies rewriting rules to eliminate many of these redundancies. We applied the optimization to several list-based benchmarks; the produced code strongly resembles explicit handwritten code in the style of AFL (Acar et al. 2009).

4 Future Work

Implicit self-adjusting computation opens up lots of opportunities for future research.

First, the level type system ensures soundness of change propagation, but to execute change propagation efficiently, we also need to memoize the computation so we can stop propagating when the updated intermediate result matches the previous one. The challenge is to know where to insert memo points, and which keys to use, taking into account the nondeterminism of memory allocation.

Second, annotations give the programmer control over granularity. For example, annotating a pair with type `(int $S * int $S) $C` means that when one part of the pair changes, all code that depends on the pair is reexecuted. This creates "extra" dependencies, but reduces memory and runtime overhead. We need to better understand the effect of granularity on both time and space.

Third, a complete and carefully engineered compiler for implicit self-adjusting computation could eventually be applied to large project (like the MLton compiler itself) to yield incremental compilation without inventing new algorithms and drastically altering existing code.

References

U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An experimental analysis of self-adjusting computation. *ACM Trans. Prog. Lang. Sys.*, 32(1):3:1–3:53, 2009.

Y. Chen, J. Dunfield, M. A. Hammer, and U. A. Acar. Implicit self-adjusting computation for purely functional programs. In *Int'l Conference on Functional Programming (ICFP '11)*, Sept. 2011.

F. Pottier and V. Simonet. Information flow inference for ML. *ACM Trans. Prog. Lang. Sys.*, 25(1):117–158, Jan. 2003.