October 1997

On type-directed partial evaluation

Walid Taha

On Type-Directed Partial Evaluation

Walid Taha

October 13, 1997

Type Directed Partial Evaluation (TDPE) is a new development in partial evaluation that has two properties that make it attractive for formal investigation: First, it is concise; it is defined in about six lines [Dan96b]. Second, it is easy to implement; the definition can be coded directly in a functional language like Scheme, yielding a demonstrably efficient and effective partial evaluator [DV96, Ves97]. In this note, we present a taste of TDPE, review the theoretical foundations and developments relating to TDPE, and highlight some major open research questions.

Type-Directed Partial Evaluation: We begin with an example of using TDPE in an interactive loop based on a compiler, similar to that of SML/NJ. Let `residualize` represent the main function of TDPE, which takes a (representation of) a type, a *compiled* object, and returns a normalized term based on that object:

```
- val compiled_0 = fn f => fn x:int => fn y:int => fn z:int => f (f (x,y),z);
  val compiled_0 = fn : (int * int -> int) -> int -> int -> int -> int
- val compiled_1 = compiled_0 (fn (x,y) => x);
  val compiled_1 = fn : int -> int -> int -> int

- val term_1 = residualize (int -> int -> int -> int) compiled_1
  val term_1 = "fn x' => fn y' => fn z' => x'" : term
```

Note that the syntax of the new term (or program) `term_1` has been derived from the `compiled_1` object code, and that the expression `f (f (x,y),z)` has been replaced (working under the λ-abstractions) by a renaming of `x`. The object passed to `residualize` must be the compiled form of a closed term. Free variables, primitives, and recursion can all handled by λ-abstracting them from the original term (using an explicit Y-combinator for recursion).

The significance of being able to take compiled code as input is two fold: First, it alleviates the need for the symbolic execution of the source program, and hence, can be done more efficiently. Second, in standard partial evaluation we sometimes find that we need to insert a "static" function value into a "dynamic" context [JGS93]. Traditionally, this meant replacing the static value by the text of the computation that produced it. In the above example, it would mean symbolically replacing both occurances of `f` in the expression `f (f (x,y),z)` with the syntax syntax `(fn (x,y) => x)`. With `residualize`, we arrive at the simpler expression `x` directly.

Theoretical Foundations: There appear to be some strong ties between TDPE and some new developments in proof theory [Dan96b]. Berger and Schwichtenberg developed a mathematical theory that could take advantage of modern functional language compilers to perform efficient proof normalization [BS91]. They presented an inversion algorithm for deriving a unique λ-term

given a semantic value, and proved that this new term is always in long normal form[1]. They also show that this algorithm has the (rather strong property) that the inverse image of any two semantically equal values yields the same term up to α-renaming. Their theory is valid in a large class of models, and for semantic values originating from closed terms. The inversion construction is also valid in other settings such as combinatory logic [CD97] and has applications in proof extraction and interactive computer-aided proof systems [Ber93, Coq93]. An insightful feature of Coquand's more recent presentation [CD97] is the use of an interpretation function that computes both the value and its representation in long normal form simultaneously.

Applying TDPE: TDPE has been used to partially evaluate an interpreter for a Pascal-like language, deriving an efficient automatically generated compiler [DV96]. It has also been applied to Paulson's Tiny interpreter written in both direct and CPS style. The code generated by starting with both styles is strikingly similar [Dan96a]. These two examples suggest that TDPE may alleviate the need for writing interpreters in CPS style in order to attain good generated code, which often seemed to be the case with standard partial evaluation techniques [JGS93]. Sheard integrated TDPE into a typed language with type inference in the context of an interpreter-based implementation, and used TDPE to specialize a small term-rewriting engine (in direct style) yielding compact code [She97]. Attaining the same compact code by a staged version of the same term-rewriting system (in direct style) did not seem possible [TS97].

Open Problems: While TDPE is concise, it is not conceptually simple: The TDPE translations use powerful language constructs such as type-dependence and multi-level η-expansions, [Car86, Coq96, DMP95, NN92, GJ95, TS97] the *combination* of which is not yet a well-explored area. Open theoretical problems include:

- Polymorphism, inductive types, and polyvariant specialization [Dan96b]. One particular polyvariance [JGS93] problem is that λ-abstracting primitives forces them to be monovariant. While progress has been made at the implementation level [She97], many important theoretical properties such as completeness and strong-normalization are still to be established.
- Effects. Duplication of computation can be controlled by means of let-insertion. This has been done in the context of TDP using Danvy and Filiniski's shift and reset [Dan96a, DF90]. Is it still possible to formally verify the correctness of TDPE after extending it using shift and reset?
- α-equivalence and gensym [Sta94]. Berger and Schwichtenberg's formal treatment of the inversion construction used an explicit representation of α-equivalent families of terms [BS91]. Implementations of TDPE use gensym. Can both treatments be formally reconciled?
- Type systems. A wealth of type-theoretic questions arise naturally when we consider TDPE [Ves97]. For example, a first cut at assigning a type to `residualize` yields $Type \rightarrow Value \rightarrow$

[1] **Remark on Terminology:** Berger and Schwichtenberg use the term "long β-normal form", and Coquand uses the the term "long η-normal form". Hindley [Hin97, Pg. 110] remarks that both are used in the literature interchangeably, but also that such terms are β-normal and not necessarily η-normal. Berger uses the neutral term "long normal form".

$Term$ [Dan96b]. But this type does not explicitly reflect the relationships between the parameters. A second attempt at assigning a type to `residualize` can be $\Pi\, x{:}Type.\ F(x) \rightarrow$ `<`$F(x)$`>`, where `<`y`>` is the type of code with type y in a statically-typed two-level λ-calculus, and F is some function from $Type$ (the representation of types in the object language) to the meta-language *type*. On close inspection of the implementation of `residualize` [Dan96b, Ves97], we see that F cannot be the identity coercion. What then is F?

There are also many interesting challenges relating to implementation, including:

- Long normal forms can be quite large in practice [She97]. Generating short normal form directly is theoretically possible [Ber93]. Can this be done efficiently in practice?
- Coquand remarks that the evaluation strategy for normalization follows exactly the evaluation strategy for the meta-language [CD97], which suggests that in a compiled implementation maintaining both the semantic and syntactic representation, the overhead of maintaining the syntactic one can be reduced if a lazy meta-language is employed. Can we completely eliminate the cost of representation if it is never "demanded"?

Concluding Remarks: TDPE holds great promise as a powerful method for partial evaluation; it leverages on the mature compilation technology for functional languages to achieve remarkably efficient symbolic manipulation. At the same time, it raises many interesting research questions and challenges.

Acknowledgements: This note has benefited greatly from discussions, suggestions, and pointers from Thierry Coquand, Olivier Danvy and my advisor Tim Sheard. I am grateful to Tim, Lisa Walton, John Launchbury, Jim Hook, Byron Cook, Sherri Shulman, Thomas Nordin and Riccardo Pucella for valuable comments on earlier drafts.

References

[Ber93] U. Berger. Program extraction from normalization proofs. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, 1993.

[BS91] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ–calculus. In Rao Vemuri, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, Loss Alamitos, 1991.

[Car86] L. Cardelli. A polymorphic λ-calculus with Type:Type, 1986.

[CD97] T. Coquand and P. Dybjer. Intiuitionistic model constructions and normalization proofs. *Math. Struct. in Comp. Science*, 1997.

[Coq93] C. Coquand. From semantics to rules: a machine assisted analysis. *LNCS*, 832, 1993.

[Coq96] T. Coquand. An algorithm for type-checking dependent types, 1996.

[Dan96a] O. Danvy. Pragmatics of type-directed partial evaluation. *LNCS*, 1110, 1996.

[Dan96b] O. Danvy. Type-directed partial evaluation. In *POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg, Florida, January 1996*. ACM, 1996.

[DF90] O. Danvy and A. Filinski. Abstracting control. In *1990 ACM Conference on Lisp and Functional Programming90*. ACM, 1990.

[DMP95] O. Danvy, K. Malmkjaer, and J. Palsberg. The essence of eta-expansion in partial evaluation. *LISP and Symbolic Computation*, 1995.

[DV96] O. Danvy and R. Vestergaard. Semantics-based compiling: A case study in type-directed partial evaluation. *LNCS*, 1140, 1996.

[GJ95] A. Glück and J. Jørgensen. Efficient multi-level generating extensions for program specialization. *LNCS*, 982, 1995.

[JGS93] N. D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentic Hall, 1993.

[NN92] F. Nielson and H. R. Nielson. *Two-Level Functional Programming Languages*. Cambridge University Press, 1992.

[She97] T. Sheard. A type-directed, on-line, partial evaluator for a polymorphic language. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM, 1997.

[Sta94] I. Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, December 1994. Also published as Technical Report 363, University of Cambridge Computer Laboratory.

[TS97] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM, 1997.

[Ves97] R. Vestergaard. From proof normalization to compiler generation and type-directed change-of-representation. Masters Thesis, 1997.