Type Specialisation

John Hughes,
Department of Computer Science, Chalmers Technical University, S-41296 Göteborg, Sweden.

Categories and Subject Descriptors: D.1.1 [**Software**]: Programming Techniques—*applicative (functional) programming*; D.3.4 [**Software**]: Programming Languages—*processors: translator writing systems and compiler generators* ; F.3.3 [**Theory of Computation**]: Logics and Meanings of Programs—*studies of program constructs: type structure*; I.2.2 [**Computing Methodologies**]: Artificial Intelligence—*automatic programming: program transformation*

General Terms: Languages, Algorithms

Additional Key Words and Phrases: type specialisation, program specialisation, partial evaluation

Type specialisation is an approach to offline program specialisation in which static information is derived by a kind of type inference rather than by evaluation or reduction. Type specialisers have been implemented for an extended λ-calculus and for a subset of Haskell; their advantage is that they can achieve a better flow of static information than a partial evaluator can.

For example, consider the following term:

$$(\underline{\lambda} f.\mathbf{lift}(f \mathbin{\underline{@}} 3)) \mathbin{\underline{@}} (\underline{\lambda} x.x+1)$$

Source terms are two-level typed λ-expressions; we write application as '@' and as usual we underline dynamic operations. In this example all functions and applications are dynamic, while the arithmetic is static, and the result is converted to a dynamic integer by **lift**. *With these annotations,* this two-level term cannot be specialised by a partial evaluator, because the β-redexes are marked dynamic and so must not be reduced. In particular $f \mathbin{\underline{@}} 3$ cannot be reduced to a constant, and so cannot be a static argument to **lift**. These restrictions are often expressed by requiring two-level types to be 'well-formed', so that for example dynamic functions can only have dynamic arguments and results. The type of f here, $\mathbf{int} \mathbin{\underline{\rightarrow}} \mathbf{int}$, is a dynamic function between static types and as such is not well-formed. In order to specialise this term non-trivially with a partial evaluator, we would need to change the annotations on the λs and @s, making them static so that the β-redexes can be reduced.

In contrast the type specialiser can specialise this term as it stands. It does not

This work was supported by TFR, the Swedish Research Coucil for Engineering Sciences.

reduce static terms, instead it infers a *residual term* and *residual type* for each source term. All static information is carried by residual types; thus for example a static constant integer such as 3 has a residual type which tells us its (statically known) value. We write that type as 3, whereas a dynamic integer whose value is not known statically has a residual type of **int** – thus residual types look like mixtures of types and values. Since we can infer types for expressions without reducing them, we can derive static information even about irreducible terms such as $f \,\underline{@}\, 3$.

Residual terms need only carry dynamic information which is not deducible from their residual type. A constant such as 3 specialises to a dummy residual term $\bullet$, indicating that the dynamic content is trivial – all information in this case is captured by the residual type. We write

$$\vdash 3 : \mathbf{int} \hookrightarrow \bullet : 3$$

to mean that the source term and type on the left can be specialised to the residual term and type on the right.

Now in our example, since f is applied to a term with residual type 3, we may infer that it has a residual type of the form $3 \to \tau$, and therefore x has residual type 3. Consequently $x + 1$ has residual type 4, and, in this context,

$$\vdash \underline{\lambda} x.x + 1 : \mathbf{int} \underline{\to} \mathbf{int} \hookrightarrow \lambda x.\bullet : 3 \to 4$$

(Notice that in a different context $\underline{\lambda} x.x + 1$ might specialise differently).

Now we know that $f : 3 \to 4$, and so $f \,\underline{@}\, 3 : 4$ — we have derived a static result for a dynamic application. So we can specialise the **lift** to 4. The final result is

$$(\lambda f.4)@(\lambda x.\bullet) : \mathbf{int}$$

After specialisation a post-processor we call the *void eraser* deletes formal and actual parameters with a trivial residual type, in this case deleting f and $\lambda x.\bullet$ to obtain the final term 4.

In this example, the same final result can be achieved by a partial evaluator, by annotating the functions and applications unfoldable. But in general changing the annotations in this way is not an option: no interesting program transformer can unfold *all* function calls. The type specialiser's advantage is that even non-unfolded calls may have static arguments and results. In fact the type specialiser goes further than this: the 'well-formedness' condition on two-level types is not needed at all. Any underlining of a one-level type is a valid two-level type. For example, dynamic lists may very well have static components, provided of course that they are static when the lists are constructed or taken apart.

In recursive programs the ability to derive static information about function calls without unfolding them brings real extra power. For example, an interpreter for the λ-calculus with constants and recursion might represent values using the type

$$\mathbf{data}\ V = \mathit{Num}\ \underline{\mathbf{int}} \mid \mathit{Fun}\ (V \underline{\to} V)$$

When such an interpreter is specialised by a conventional partial evaluator, the type V must be dynamic since it is the result of the evaluation function. (If we try to unfold all calls of the evaluation function, then specialisation will loop on recursive inputs). Consequently the constructors *Num* and *Fun* appear in residual programs.

Neil Jones calls a specialiser *optimal* if it can specialise a self-interpreter to the abstract syntax tree of a program, and obtain (a renaming of) the same program as residual code. An optimal specialiser can remove a complete layer of interpretation. Since a partial evaluator cannot remove the constructors of types such as V, it cannot specialise typed interpreters optimally.

In contrast the type specialiser allows these constructors to be static. Static constructors appear in residual types, not in residual terms. For example

$$\vdash \mathit{Num}\ \underline{3} : V \hookrightarrow 3 : \mathit{Num}\ \mathbf{int}$$

Knowing such a residual type, **case** expressions over V can be specialised to the relevant branch; for example if the context tells us that x has this residual type, then

$$\vdash \mathit{Fun}\left(\begin{array}{l}\underline{\lambda}x.\ \mathbf{case}\ x\ \mathbf{of} \\ \qquad\qquad \mathit{Num}\ y \rightarrow y \underline{+} \underline{1} \\ \qquad\qquad \mathit{Fun}\ f \rightarrow \mathbf{error}\end{array}\right) : V \hookrightarrow \lambda x.x+1 : \mathit{Fun}\ (\mathit{Num}\ \mathbf{int} \rightarrow \mathit{Num}\ \mathbf{int})$$

So neither the constructors *Num* and *Fun*, nor the **case** expressions over them, appear in residual programs. Using this idea we have constructed an interpreter for typed λ-calculus which the type specialiser can specialise optimally. Neil Jones posed optimal specialisation of typed languages as a 'challenging problem' as long ago as 1988 [Jones 1988], but it has previously only been achieved for a rather restricted first-order language (by combining a partial evaluator with an abstract interpreter to predict static information about dynamic results [Dussart et al. 1995]).

The occurrences of *Num* and *Fun* in residual types carry information which is important during specialisation, but irrelevant afterwards. If they are then erased, then it is clear that any type at all built from **int** and $\rightarrow$ can be obtained as a specialisation of V. Type specialisation lifts the *inherited limit* [Mogensen 1996] that a residual program may only contain the types that appear in the source; this was another of Jones' 'challenging problems'.

If our optimal interpreter is specialised to the syntax tree of an ill-typed λ-term, then specialisation cannot be completed because the type specialiser would need to generate the same ill-typed term, and it generates only well-typed residual programs. Instead an error is reported. We believe this is inevitable: any optimal specialiser for typed languages must reject some specialisations, at least those of the interpreter which it specialises optimally to ill-typed inputs. The analogy is with compilers for typed languages, which can generate efficient type-free code only because they refuse to compile some (syntactically correct, but ill-typed) programs.

Type specialisation can be specified by *specialisation rules*, which are analogous to typing rules, but whose judgements relate both source and residual terms and types. For example, the rules for constants, variables and $\underline{\lambda}$-expressions:

$$\textit{(CON)}\quad \Gamma \vdash n : \mathbf{int} \hookrightarrow \bullet : n \qquad \textit{(VAR)}\quad \Gamma, x : \tau \hookrightarrow e : \tau' \vdash x : \tau \hookrightarrow e : \tau'$$

$$\textit{(LAM)}\quad \frac{\Gamma, x : \sigma \hookrightarrow x' : \sigma' \vdash e : \tau \hookrightarrow e' : \tau'}{\Gamma \vdash \underline{\lambda}x.e : \sigma \rightarrow \tau \hookrightarrow \lambda x'.e' : \sigma' \rightarrow \tau'}\ \ x' \text{ fresh}$$

Other rules use residual type information to guide the creation of the residual term, for example the rule for **lift**:

$$\textit{(LIFT)} \quad \frac{\Gamma \vdash e : \mathbf{int} \hookrightarrow e' : n}{\Gamma \vdash \mathbf{lift}\ e : \underline{\mathbf{int}} \hookrightarrow n : \mathbf{int}}$$

Specialisation rules constitute a modular description of type specialisation, which is easy to extend. New features can be added by adding a new source type and associated rules. One of the most important extensions is *polyvariance*. Basic type specialisation is monovariant: terms such as

$$(\underline{\lambda} f.(f \mathbin{\underline{@}} 3, f \mathbin{\underline{@}} 4)) \mathbin{\underline{@}} (\underline{\lambda} x.\mathbf{lift}\ (x+1))$$

cannot be specialised at all (since f would need to be assigned both residual types $3 \rightarrow \mathbf{int}$ and $4 \rightarrow \mathbf{int}$). A polyvariant specialiser would make two specialised versions of f, one for each call. Polyvariance can be added to the type specialiser via terms **poly** e (of type **poly** τ) which specialise to a tuple of specialisations of e, and **spec** e which specialise to an appropriate selection from the tuple. The specialisation rules are simple:

$$\textit{(POLY)} \quad \frac{\begin{array}{c}\Gamma \vdash e : \tau \hookrightarrow e_1 : \tau_1 \\ \vdots \\ \Gamma \vdash e : \tau \hookrightarrow e_n : \tau_n\end{array}}{\Gamma \vdash \mathbf{poly}\ e : \mathbf{poly}\ \tau \hookrightarrow (e_1, \ldots, e_n) : (\tau_1, \ldots, \tau_n)}$$

$$\textit{(SPEC)} \quad \frac{\Gamma \vdash e : \mathbf{poly}\ \tau \hookrightarrow e' : (\tau_1, \ldots, \tau_n)}{\Gamma \vdash \mathbf{spec}\ e : \tau \hookrightarrow \pi_i e' : \tau_i}$$

allowing **poly** to form an arbitrary number of specialisations, and **spec** to choose any one. Using them in our example gives

$$\begin{array}{l}\vdash\ (\underline{\lambda} f.(\mathbf{spec}\ f \mathbin{\underline{@}} 3, \mathbf{spec}\ f \mathbin{\underline{@}} 4)) \mathbin{\underline{@}} (\mathbf{poly}\ \underline{\lambda} x.\mathbf{lift}\ (x+1)) \ \hookrightarrow \\ \qquad (\lambda f.(\pi_1 f @ \bullet, \pi_2 f @ \bullet)) @ (\lambda x.4, \lambda x.5)\end{array}$$

or after void erasure, $(\lambda f.(\pi_1 f, \pi_2 f))@(4,5)$. When polyvariance is added, neither the other rules nor the rest of the implementation of the specialiser need be modified. (The residual tuple types here are related to the intersection types used in [Turbak et al. 1997]. They also correspond to the 'log' or 'pending list' of conventional partial evaluators. The type specialiser can handle arbitrary combinations of function values and block structure; the problems of scope discussed in [Malmkjær and Ørbæk 1995] don't arise because the log is associated with types rather than names).

Similarly the type specialiser has been extended to handle *constructor specialisation* [Mogensen 1993] and imperative features via a monad type [Dussart et al. 1997]. Easy extension has made it possible to combine features in interesting new ways; for example by using constructor specialisation in combination with higher-order functions to obtain *typed closure conversion* by specialising a suitable interpreter.

Specialisation rules are harder to implement than type inference rules because they are not purely syntax-directed. For example, when a **spec** is first encountered it may not be clear which version to choose, when a **poly** is encountered it may not be clear which versions to create. Implementations resolve such problems using mechanisms to delay parts of a specialisation until the necessary information is available.

To decide whether or not two different **spec**s refer to the same component of a **poly**, an element of search is needed. First we hypothesise that the two **spec**s have the same residual type, and then if this assumption leads to a specialisation error then we backtrack and undo it. This strategy is efficient provided specialisation errors are encountered early, which in turn depends critically on the order in which specialisation rules are applied. Implemented heuristics seem to work well, however.

In summary, type specialisation is a new approach to offline program specialisation with many advantages over partial evaluation:

—The flow of static information is improved; the restriction to 'well-formed' source types can be dropped, perhaps making some 'binding-time improvements' unnecessary.

—Types as well as terms can be specialised; residual programs can contain arbitrary types derived as specialisations of one type in the source.

—Optimal specialisation can be achieved for typed interpreters.

—A modular specification and implementation makes extensions easy; specialisation features can be combined in new ways to yield interesting results.

On the negative side, type specialisation is still immature and much less well understood than partial evaluation. Current implementations are slow prototypes which specialise programs in toy languages; the most serious limitation is currently that source and residual programs must be *simply typed*. Moreover *binding-time analysis* has proved invaluable in practice for partial evaluators, but seems to pose difficulties of principle under type specialisation. For example, if the constructors *Num* and *Fun* in the λ-interpreter above are made static, then the specialiser generates efficient residual programs without type tags, but can only specialise the interpreter to well-typed λ-terms. If they are made dynamic, then the residual programs are worse because they contain type tags, but in return the interpreter can be specialised to *any* λ-term. The binding-time of the constructors decides the static semantics of the interpreted language. It is hard to see how such a choice can be made automatically.

Type specialisation is described in [Hughes 1996b], and an informal introduction appears in [Hughes 1996a]. The extension to handle imperative features appears in [Dussart et al. 1997]. A demonstration of the prototype type specialiser is available on the web at `http://www.cs.chalmers.se/~rjmh/TypedPE/`.

REFERENCES

DUSSART, D., BEVERS, E., AND VLAMINCK, K. D. 1995. Polyvariant Constructor Specialisation. In *Proc. ACM Conference on Partial Evaluation and Program Manipulation* (La Jolla, California, 1995).

DUSSART, D., HUGHES, J., AND THIEMANN, P. 1997. Type Specialisation for Imperative Languages. In *International Conference on Functional Programming* (Amsterdam, June 1997). ACM.

HUGHES, J. 1996a. An Introduction to Program Specialisation by Type Inference. In *Functional Programming* (July 1996). Glasgow University. published electronically.

HUGHES, J. 1996b. Type Specialisation for the Lambda-calculus; or, A New Paradigm for Partial Evaluation based on Type Inference. In O. DANVY, R. GLÜCK, AND P. THIEMANN Eds., *Partial Evaluation*, Volume 1110 of *LNCS* (February 1996). Springer-Verlag.

JONES, N. D. 1988. Challenging problems in partial evaluation and mixed computation. In D. BJØRNER, A. ERSHOV, AND N. JONES Eds., *Partial Evaluation and Mixed Computation* (North-Holland, 1988), pp. 1–14. IFIP: Elsevier Science Publishers B.V.

Malmkjær, K. and Ørbæk, P. 1995. Polyvariant specialization for higher-order, block-structured languages. In W. L. Scherlis Ed., *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (La Jolla, California, June 1995), pp. 66–76.

Mogensen, T. Æ. 1993. Constructor specialization. In D. Schmidt Ed., *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (June 1993), pp. 22–32.

Mogensen, T. Æ.. 1996. Evolution of partial evaluators: Removing inherited limits. In O. Danvy, R. Glück, and P. Thiemann Eds., *Partial Evaluation. Proceedings*, Lecture Notes in Computer Science (1996), pp. to appear. Springer-Verlag.

Turbak, F., Dimock, A., Muller, R., and Wells, J. B. 1997. Compiling with Polymorphic and Polyvariant Flow Types. In *TIC'97: Types in Compilation Workshop* (Amsterdam, June 1997).