

An Introduction to Program Specialisation by Type Inference

John Hughes

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

Capsule review by Phil Wadler

About a dozen years ago, Neil Jones and colleagues set the world on fire by demonstrating the first self-applicable partial evaluator. A little later, Jones formally listed key open problems in the area, one of which was to build an optimal specialiser for a typed language, and that chestnut remained uncracked a decade later. Here John Hughes shows how a change of perspective enabled him to break it open, and split a few other tough nuts as well. This paper complements the full work published elsewhere, by conveying the insights and intuitions that can be obscured when all the i's must be dotted and all the t's crossed. The full work is destined to become a classic; this is its essential companion.

Abstract

In this paper we present a new paradigm for partial evaluation, based not on transforming terms into terms, but on transforming terms and types into terms and types. An immediate advantage is that residual programs need not involve the same types as source programs, so *type specialisation* can be accomplished naturally. Furthermore, while a conventional partial evaluator handles a static expression by reducing it to a constant residual *term*, we carry the static information in the residual *type*, which leads to improved static information flow and thereby better specialisations. Thanks to this, *optimal specialisation* of a typed interpreter is possible. Our partial evaluator is specified in a very modular fashion: each feature is associated with a type in the source language, and its specialisation is specified via corresponding introduction and elimination rules. As a result, specialisation features can be freely combined: for example, we have for the first time combined constructor specialisation with higher-order functions, deriving a firstification transformation and a closure analyses as a consequence.

This paper is an introduction to the material in [Hug96], which appeared in the Proceedings of the Dagstuhl Workshop on Partial Evaluation, 1996 (Springer LNCS).

1 Introduction

Suppose we are given a program, and values for some, but not all, of its inputs. We cannot run the program without the remaining inputs, but we may well be able to *simplify* the program using the knowledge that we have, producing a *specialised program* that is hopefully more efficient than the original. A *partial evaluator* [JGS93] performs this transformation automatically, producing a *residual program* from which all operations on the known inputs have been removed, and which need only be supplied with values for the remaining ones. Some useful terminology: operations which depend only on known values are called *static*, and the others are called *dynamic*. Here we consider *offline* partial evaluation, in which dynamic operations are labelled in the original program (by underlining).

For example, suppose we specialise the exponential function

$$\text{power } n \ x = \text{if } n = 1 \ \text{then } x \ \text{else } x \times \underline{\text{power}}(n - 1) \ x$$

to a fixed value of its first argument n . Then the test and subtraction on n are static operations, while the multiplication is dynamic, and is labelled as such. If n is known to be 3, then the residual program will be

$$\text{power}_3 \ x = x \times (x \times x)$$

in which only the dynamic operations remain.

The effect of partial evaluation is particularly striking when the original program is an interpreter. For example, consider the simple λ -calculus interpreter in Figure 1. Suppose we specialise it to interpret a particular program for

$$\begin{aligned}
eval \ \rho \ (Con \ n) &= \mathbf{lift} \ n \\
eval \ \rho \ (Var \ x) &= \rho \ x \\
eval \ \rho \ (Lam \ x \ e) &= \underline{\lambda}v. eval \ (\rho[v/x]) \ e \\
eval \ \rho \ (App \ e_1 \ e_2) &= eval \ \rho \ e_1 \ @ (eval \ \rho \ e_2)
\end{aligned}$$

Figure 1: An Interpreter for the λ -calculus

$$\begin{aligned}
\mathbf{data} \ Univ &= \underline{Num} \ \mathbf{int} \mid \underline{Fun} \ (Univ \ \Rightarrow \ Univ) \\
eval \ \rho \ (Con \ n) &= \underline{Num} \ (\mathbf{lift} \ n) \\
eval \ \rho \ (Var \ x) &= \rho \ x \\
eval \ \rho \ (Lam \ x \ e) &= \underline{Fun} \ (\underline{\lambda}v. eval \ (\rho[v/x]) \ e) \\
eval \ \rho \ (App \ e_1 \ e_2) &= \mathbf{case} \ eval \ \rho \ e_1 \ \mathbf{of} \ \underline{Fun} \ f \ \rightarrow \ f \ @ (eval \ \rho \ e_2)
\end{aligned}$$

Figure 2: A Typed Interpreter for the λ -calculus

varying arguments. The case analysis on the abstract syntax tree will be static, as will environment look-ups, but the *result* of *eval* will be dynamic. This is indicated in three ways:

- static constants are made dynamic by applying **lift**,
- the value of a *Lam*-term is a dynamic function, created by a dynamic $\underline{\lambda}$,
- we interpret an *App* by dynamically applying a function using **@**.

Now when we specialise the interpreter, the partial evaluator reduces all the redexes not marked dynamic, and only these dynamic operations contribute to the residual program:

$$eval \ \rho_0 \ (App \ (Lam \ \mathbf{f} \ (App \ (Var \ \mathbf{f}) \ (Con \ 3))) \ (Lam \ \mathbf{x} \ (Var \ \mathbf{x}))) \ \hookrightarrow \ (\underline{\lambda}v. v \ 3) (\underline{\lambda}v. v)$$

In this case the residual program is essentially the same as the interpreted one: we can regard it as a translation of the interpreted program from the language of abstract syntax trees into the real λ -calculus. Moreover, no ‘interpretive overhead’ remains: partial evaluation has removed a complete layer of interpretation. Neil Jones calls a partial evaluator which can do so ‘*optimal*’. Partial evaluators in general, and optimal ones in particular, have many applications in automatic generation of compilers [JGS93].

Unfortunately, the optimality of this partial evaluator depends crucially on the programming language being *untyped*. Look at Figure 1 again: the *eval* function sometimes returns an integer, and sometimes a function. In a typed language, we would need to inject both kinds of value into a suitable universal type, as in Figure 2. We have to tag dynamic values as either \underline{Num} or \underline{Fun} , and before applying a dynamic function we have to check and remove the tag with a **case** expression. Unfortunately, *these tags have to be dynamic*. As a result, the constructors and the **case** expressions that examine them appear in the residual program. Specialising the typed interpreter to the same example as above, we get

$$\begin{aligned}
eval \ \rho_0 \ (App \ (Lam \ \mathbf{f} \ (App \ (Var \ \mathbf{f}) \ (Con \ 3))) \ (Lam \ \mathbf{x} \ (Var \ \mathbf{x}))) \ \hookrightarrow \\
\mathbf{case} \ Fun \ \left(\begin{array}{l} \underline{\lambda}v. \ \mathbf{case} \ v \ \mathbf{of} \\ \underline{Fun} \ f \ \rightarrow \ f \ (Num \ 3) \end{array} \right) \ \mathbf{of} \\
\underline{Fun} \ f \ \rightarrow \ f \ (Fun \ (\underline{\lambda}v. v))
\end{aligned}$$

Clearly specialisation is no longer optimal: a great deal of interpretive overhead remains in the residual program.

Why can’t we just mark the *Univ* constructors static? Were we to do so, the partial evaluator would try to reduce every term of type *Univ* to one of the forms *Num e* or *Fun e*. Then every **case** on *Univ* could be simplified also. But

unfortunately, one of the expressions of this type is a dynamic application $e_1 @ e_2$ (in the *App* case of *eval*). A dynamic application should *not* be reduced by the partial evaluator, which conflicts with the need to reduce the expression to head normal form.

The classical view of partial evaluation is as a strategy for performing reductions: static values are discovered by reducing expressions to normal form. This view forces the restriction that

The result of a dynamic function must be dynamic.

And this restriction in turn makes the specialisation of typed interpreters sub-optimal: the type tags in the residual programs cannot be removed.

Can we obtain static information about the result of a dynamic function, *without* actually applying it? Our inspiration is the following: *type inference* can tell us the *type* of a function's result, without needing to apply it! By analogy, we have discovered a new paradigm for program specialisation in which static information is collected by type inference, rather than by reduction. The effect is that in examples like the above, more things can be made static, and as a result residual programs are more efficient. Specialisation is stronger, quite simply. In particular, we can specialise typed interpreters optimally.

2 Specialising Programs by Type Inference

A classical partial evaluator reduces a source term to a residual term; we write

$$e \rightarrow e'$$

In contrast we will say that a source term *and type* specializes to a residual term *and type*:

$$e : \tau \hookrightarrow e' : \tau'$$

In the classical case, static information is carried by the *residual term*: static expressions are simply reduced to normal form. For example,

$$2 + 2 \rightarrow 4$$

But with our new approach, the residual term will carry only *dynamic* information, so a purely static expression such as $2 + 2$ will be specialised to a dummy value. *All static information will be carried by the residual type!* For example, we specialise $2 + 2$ as follows:

$$2 + 2 : \mathbf{int} \hookrightarrow \bullet : 4$$

Here 4 is the residual *type*: it may seem strange that we use integers as types, but think of it as telling us how to interpret the dummy value \bullet in this case.

Since static information is carried by types, we can derive it without reduction. We just infer the residual types in the specialised program. For example, consider

$$(\underline{\lambda}x.x + 1) @ 3$$

which would be ill-formed for a traditional partial evaluator, because the addition $x + 1$ is marked static, even though the $\underline{\lambda}$ and application are marked dynamic so that the β -redex will not be reduced and the value of x will consequently be unknown. With our approach, we can specialise the actual parameter as

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

so its residual type is 3. Now ordinary type inference tells us that the $\underline{\lambda}$ -expression must have a residual type of the form $3 \rightarrow \alpha$, and so the formal parameter x must have residual type 3 also. With this information we can specialise the addition

$$x + 1 : \mathbf{int} \hookrightarrow \bullet : 4$$

which tells us that the function's result is of residual type 4. So the result of specialising the entire term is

$$(\underline{\lambda}x.x + 1) @ 3 : \mathbf{int} \leftrightarrow (\lambda x.\bullet) \bullet : 4$$

As we can see, the dynamic β -reduction has *not* been performed, but despite this the static result is known, and all the static computations have disappeared from the program.

As another example, consider

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

where none of the β -redexes is to be reduced, but even so the addition $x + 1$ is marked static. To specialise this, we first specialise the actual parameter that f is applied to:

$$3 : \mathbf{int} \leftrightarrow \bullet : 3$$

This tells us that f 's residual type has the form $3 \rightarrow \alpha$, and since f is bound to $\underline{\lambda}x.x + 1$ then x has residual type 3. We can therefore specialise the body of the second $\underline{\lambda}$ -expression as

$$x + 1 : \mathbf{int} \leftrightarrow \bullet : 4$$

which tells us that f 's result type is 4. Now we can specialise the call of f

$$f @ 3 : \mathbf{int} \leftrightarrow f \bullet : 4$$

and then the body of the first $\underline{\lambda}$ -expression:

$$\mathbf{lift} (f @ 3) : \mathbf{int} \leftrightarrow 4 : \mathbf{int}$$

Here we see that the residual type of a *dynamic* integer is of course just \mathbf{int} — there is no more static information to carry. The complete result is

$$(\underline{\lambda}f.\mathbf{lift} (f @ 3)) @ (\underline{\lambda}x.x + 1) : \mathbf{int} \leftrightarrow (\lambda f.4) (\lambda x.\bullet) : \mathbf{int}$$

in which once again, the dynamic β -redexes have not been performed, but even so the static computations have been.

The reader may object that static computations have not disappeared entirely: they leave a residue of dummy values behind them. In fact, we use a post-processor which we call the *void eraser* to recognise expressions with no dynamic content, such as the $\lambda x.\bullet$ above, and remove them. Such expressions can be recognised purely from their types, and in almost all cases can be elided. In this example we remove both the λ -expression, and the binding of f which equally has no dynamic content. The residual program after void erasure is just 4. Void erasure often brings a dramatic reduction in the size of the residual program, and is an essential stage with this approach.

The main point of these examples is to show that type inference gives *better static data flow* than reduction, and therefore stronger specialisation. We can specialise these examples *without* unfolding the β -redexes, where a classical partial evaluator would be forced either to do the unfolding, or to treat the integers, and the operations on them, as dynamic.

In these simple examples, there is no reason not to unfold, and the reader may be wondering why we go to the trouble of inventing a new paradigm, when a little unfolding would solve the same problems. The point is, of course, that no realistic partial evaluator can unfold *all* function calls, and the problem we are addressing is improving the static information about calls which are *not* unfolded.

3 Optimal Specialisation of Interpreters

Let us return to the problematic universal type *Univ* in our λ -calculus interpreter, and consider the effect of making its constructors *static*. How should a constructor application then be specialised? Of course, the static constructor ought not to appear in the residual term, which should consist just of the 'dynamic part' of value. Conversely, the residual

type ought to tell us which constructor was actually applied, since this is static information. We therefore specialise constructor applications by attaching the constructor to the residual *type*, not the term: for example,

$$Num \underline{3} : Univ \hookrightarrow 3 : Num \mathbf{int}$$

Notice that the *representation* of the result is just an **int**: the residual types **int** and *Num int* are isomorphic, but the *Num* tells us how to interpret the **int** in this case. When a **case** expression on *Univ* is specialised, the residual type of the inspected expression suffices to simplify the **case** to the appropriate branch.

To take a larger example, consider the following term of type *Univ*, which represents a function on integers:

$$Fun (\underline{\lambda}x. \mathbf{case} \ x \ \mathbf{of} \\ \quad Num \ x' \rightarrow Num \ (x' \underline{+} \underline{1}) \\ \quad Fun \ z \rightarrow \dots)$$

We might specialise this as follows:

- The outer *Fun* becomes part of the residual type, which is therefore of the form *Fun* α .
- The $\underline{\lambda}$ builds a λ expression in the residual term, which therefore is of the form $\lambda x.e$, with residual type *Fun* ($\alpha \rightarrow \beta$).
- Suppose we know from the context that the argument type is *Num int*. Then x has this residual type, and we choose the first branch of the **case**.
- The *Num* constructor in this branch becomes part of the result type.
- The dynamic addition becomes part of the residual term.

The final result of specialisation is

$$Fun \left(\begin{array}{l} \underline{\lambda}x. \mathbf{case} \ x \ \mathbf{of} \\ \quad Num \ x' \rightarrow Num \ (x' \underline{+} \underline{1}) \\ \quad Fun \ z \rightarrow \dots \end{array} \right) : Univ \hookrightarrow \lambda x.x + 1 : Fun \ (Num \ \mathbf{int} \rightarrow Num \ \mathbf{int})$$

As we can see, all the type tags are known statically and form part of the residual type, whereas they have been removed completely from the residual program.

When this same technique is applied to the λ -interpreter of Figure 2, we obtain optimal specialisations as a result.

This kind of program specialisation has the strange property that it may fail: since we propagate static information essentially by applying type inference to the residual program, we must accept that the residual program may be ill-typed! For example, we cannot specialise the term

$$\mathbf{if} \ b \ \mathbf{then} \ Num \ \underline{3} \ \mathbf{else} \ Fun \ (\underline{\lambda}x.x)$$

because the residual types of the branches of the dynamic conditional clash. This kind of situation will arise if we try to specialise the λ -interpreter to an *ill-typed* input: obviously in this case optimal specialisation, if it succeeds at all, can only produce an ill-typed result.

We are not used to thinking of program specialisers as tools which may reject some specialisations as ill-formed. But we contend that this is an inevitable property of optimal specialisers for typed languages: a compiler for typed programs can eliminate run-time tags from the code it generates precisely *because* it refuses to compile ill-typed programs, and a specialiser for typed languages should do likewise. We are used to thinking of an interpreter as defining the *dynamic semantics* of the language it interprets; in a sense when we decide what will be static and what will be dynamic, we define the static semantics also.

4 Extensions

I have presented the basic idea of program specialisation by type inference, and argued that it leads to stronger specialisation. But another advantage of the paradigm is that it is very easy to extend the specialiser with new features. We shall illustrate this with an example.

Just consider the following term,

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

which is very similar to one of the examples above, except that f is applied to two different arguments. Our specialiser cannot specialise this term at all! The problem is that f would need to be given *two different* residual types, $3 \rightarrow \mathbf{int}$ and $4 \rightarrow \mathbf{int}$. This is not allowed, and the result is a unification failure.

This example shows that our specialiser is *monovariant*: it can only construct one specialisation of each term. In this case we need *polyvariant* specialisation, to create two different specialisations of f . We have a general approach to extending our specialiser: rather than change its behaviour for existing terms, we *add a new type* to be specialised in a new way. To add polyvariance, we add types of the form **poly** τ along with operators

$$\begin{aligned} \mathbf{poly} &: \tau \rightarrow \mathbf{poly} \tau \\ \mathbf{spec} &: \mathbf{poly} \tau \rightarrow \tau \end{aligned}$$

which make a ‘polyvariant expression’ and select one of its specialisations, respectively. We specialise polyvariant expressions to a *tuple* of residual terms,

$$\mathbf{poly} e : \mathbf{poly} \tau \hookrightarrow (e_1 \dots e_n) : (\tau_1 \dots \tau_n)$$

where each e_i is a different specialisation of e , and τ_i is its residual type. We specialise applications of **spec** to select an appropriate specialisation:

$$\mathbf{spec} e : \tau \hookrightarrow \pi_i e' : \tau_i$$

In the example above, we apply **poly** to the second λ -expression so that it can be specialised twice, and apply **spec** to each occurrence of f before we call it. The result of specialisation is

$$\begin{aligned} (\lambda f.(\mathbf{spec} f \text{ @ } 3, \mathbf{spec} f \text{ @ } 4)) \text{ @ } \mathbf{poly} (\lambda x.\text{lift } (x + 1)) &: (\mathbf{int}, \mathbf{int}) \hookrightarrow \\ (\lambda f.(\pi_1 f \bullet, \pi_2 f \bullet)) (\lambda x.4, \lambda x.5) &: (\mathbf{int}, \mathbf{int}) \end{aligned}$$

which after void erasure becomes

$$(\lambda f.(\pi_1 f, \pi_2 f)) (4, 5)$$

As we can see, we have here two specialisations of f , and we select the appropriate one to use at each point.

This kind of extension is easy because we add each new type, along with its specialisation rules, *independently* of those that already exist. The extensions are very modular. One result of this is that we have been able to combine specialisation features in new ways. In fact, we have just seen an example: previous polyvariant specialisers for higher-order languages have only been able to specialise *let-bound* functions monovariantly [Bon91]. In this example, we specialised the λ -expression bound to f polyvariantly, even though it was passed as a parameter. We could apply this, for example, in an interpreter which represented the environment as a polyvariant function from identifiers to values, to be specialised to each identifier that is looked up. In residual programs such an environment would be replaced by a tuple of values of the variables actually used.

Another interesting extension I have made is to add an analogue of Mogensen’s *constructor specialisation* [Mog93]. In the source language, I added a type **sum** τ with a single constructor $\underline{In} :: \tau \rightarrow \mathbf{sum} \tau$. In the residual programs the **sum** τ type is specialised to an n -ary sum, with a specialised variant of In for each residual type that In is applied to. So for example, the program

$$\begin{aligned} \underline{\mathbf{let}} f &= \lambda x.\underline{\mathbf{case}} x \text{ of } \underline{In} y \rightarrow \underline{\mathbf{lift}} y \\ \underline{\mathbf{in}} f &3 \underline{+} f 4 \end{aligned}$$

specialises to

$$\begin{aligned} \text{let } f = \lambda x. \text{case } x \text{ of } & \text{In}_3 y \rightarrow 3 \\ & \text{In}_4 y \rightarrow 4 \\ \text{in } & f(\text{In}_3 \bullet) + f(\text{In}_4 \bullet) \end{aligned}$$

where the type of x is specialised from **sum int** to $\text{In}_3 3 \mid \text{In}_4 4$. Notice that a **case** over such a **sum** type is specialised by replicating the branches.

My implementation was actually the first implementation of constructor specialisation for a higher-order language. The combination of higher-order functions and constructor specialisation enabled me to obtain a firstification transformation as a simple application, by writing an interpreter which represented functions using the type **sum** ($\text{Univ} \rightarrow \text{Univ}$). This uses a *static function type*: static functions are unfolded by the specialiser, and static function values are represented in residual programs just by a tuple of their free variables. When this interpreter is specialised to a higher-order program, the residual code represents function values as a tag (specialised *In* constructor) identifying the function concerned, and a tuple of free variables — in other words, as a closure.

I have also performed been able to specialise of ‘imperative’ programs by adding a monadic computation type along with operations in the monad to read and write both a static and a dynamic state. The extension works smoothly, and since the semantics of the rest of the language is unaltered, no changes were needed to the existing part of the specialiser. A joint paper with Dirk Dussart and Peter Thiemann is in preparation.

Stefan Kahrs has suggested adding *subtyping* of residual types, allowing for example 4 to be a subtype of **int**. There would then be no point in distinguishing the source types **int** and **int**. The result would be an *online type specialiser* for a one-level language. The extension is not absolutely straightforward though, since online specialisers need to use some kind of ‘generalisation strategy’ to achieve termination. In this context such a strategy would need to decide when to coerce types to their supertypes (e.g. 4 to **int**), in order to reduce (to a finite number) the number of versions of polyvariant functions. It is certainly not obvious what strategy should be used.

5 Discussion

In this paper we have given an informal introduction to program specialisation by type inference. For a full paper on this work, see [Hug96]. We claim that our new framework is simple, modular, and extensible. It permits stronger specialisations than previous approaches.

Our work solves two ‘challenging problems’ posed by Neil Jones in 1988 [Jon88]. The first was to perform *type specialisation*: that is, to derive residual programs that operate on different types than the source program. Type specialisation fits badly in the ‘partial evaluation as a reduction strategy’ framework, because one of the main properties of reduction is that it does not change types! In contrast, it fits naturally into our approach, and indeed we have seen several examples of it in this paper. Limited forms of type specialisation have been used for a long time (see discussion in [Hug96]), but ours is the first that can obtain an *arbitrary* type by specialising one suitable universal type.

The second challenging problem was optimal specialisation, which we have discussed above. Optimal specialisers have existed for a long time for untyped languages, but there has been only one previous optimal specialiser for a typed language, due to Dussart et al. [DBV95]. However, this specialiser processes only a first-order language, and is optimal only if the only types are algebraic data types.

In this paper, we have stressed the necessity of collecting static information about the results of dynamic function calls. We are the first to use type inference for this purpose, but there have been earlier attempts using other methods. Dussart et al. obtained their results by using *abstract interpretation* for essentially the same purpose.

We have not shown the correctness of type specialisation. Correctness isn’t obvious. There are examples where the specialiser changes the semantics of the program, such as

$$\text{lift } (\text{fix } (\lambda x. \text{if true then } x \text{ else } 4))$$

which is specialised to the program 4, even though its semantics is \perp . The residual type of the **fix** is forced to be 4 by the **else** branch of the conditional, which allows the **lift** to be specialised even though the fixpoint is actually undefined. We believe that the specialiser is correct in the sense that it cannot change the answers that programs produce, only make

them terminate more often. However, this is yet to be proved. This weaker correctness property is actually shared by many other program specialisers, and is usually considered acceptable.

See the full paper for a more thorough comparison with related work.

6 Conclusion

Inspired by type inference, we have discovered a new paradigm for program specialisation, which has led to stronger specialisation and solves some long-standing problems. We have applied it to the λ -calculus, but we believe the basic ideas will be applicable to almost any typed programming language. Although we are aware of numerous outstanding problems with the new approach (see the full paper), we conclude that it holds much promise.

References

- [Bon91] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17(1-3):3–34, December 1991. Selected papers of ESOP '90, the 3rd European Symposium on Programming.
- [DBV95] Dirk Dussart, Eddy Bevers, and Karel De Vlaminc. Polyvariant Constructor Specialisation. In *Proc. ACM Conference on Partial Evaluation and Program Manipulation*, La Jolla, California, 1995.
- [Hug96] John Hughes. Type Specialisation for the Lambda-calculus; or, A New Paradigm for Partial Evaluation based on Type Inference. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, volume 1110 of *LNCS*. Springer-Verlag, February 1996.
- [JGS93] N. D. Jones, , C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
- [Jon88] Neil D. Jones. Challenging problems in partial evaluation and mixed computation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 1–14, North-Holland, 1988. IFIP, Elsevier Science Publishers B.V.
- [Mog93] Torben Æ. Mogensen. Constructor specialization. In David Schmidt, editor, *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 22–32, June 1993.