Binding-time Analysis for Polymorphic Types

Rogardt Heldal and John Hughes

Chalmers University of Technology, S-41296 GÖTEBORG.
`heldal@cs.chalmers.se`, `www.cs.chalmers/~heldal`.

1 Introduction

Partial evaluation is by now a well-established technique for specialising programs [13], and practical tools have been implemented for a variety of programming languages [2, 1, 16, 5]. Our interest is in partial evaluation of modern typed functional languages, such as ML [18] or Haskell [14]. One of the key features of these languages is polymorphic typing [17], yet to date the impact of polymorphism on partial evaluation has not been studied. In this paper we explain how to extend an offline partial evaluator to handle a polymorphic language.

1.1 Background: Polymorphism

A *polymorphic function* is a function which may be applied to many different types of argument. In ML and Haskell, the types of such functions are expressed using a "forall" quantifier: for example, the well-known *map* function, which applies a function to every element of a list, has the type

$$map :: \forall \alpha_1, \alpha_2.(\alpha_1 \rightarrow \alpha_2) \rightarrow [\alpha_1] \rightarrow [\alpha_2]$$

meaning that for *any* types α_1 and α_2, *map* takes a function of type $\alpha_1 \rightarrow \alpha_2$ and a list of type $[\alpha_1]$, and produces a list of type $[\alpha_2]$. ($[\alpha]$ is our notation for a list type with elements of type α).

Polymorphic functions are heavily used in real functional programs. In particular, library functions are frequently polymorphic, since the types at which they will be needed are not known when the library is written. The standard library contains many polymorphic functions such as *map* and *foldr* (which takes a binary operator and its unit, and combines the elements of a list using the operator). These polymorphic functions greatly simplify programming, for example, the sum of a list of integers *xs* can be computed as *foldr* $(+)$ 0 *xs*, and the conjunction of a list of booleans *bs* can be computed as *foldr* $(\wedge)$ **true** *bs*.

1.2 Background: Partial Evaluation

A partial evaluator is a tool which takes a program and a *partially known* input, and performs operations in the program which depend only on the known parts, generating a specialised program which processes the remainder. For example, specialising *foldr* to the inputs *foldr* $(+)$ 0 $[x, y, z]$, where x, y and z are unknown, would generate the specialised program $x + y + z + 0$. Here the construction of the known list, and the recursion over it inside *foldr*, have been performed by the partial evaluator: only the actual computations of the sum of the unknown quantities remains in the specialised code.

Partial evaluators can be classified into *online* and *offline*. Online partial evaluators decide dynamically during specialisation which operations to perform, and which to build into the residual program: an operator is performed if its operands are known in that particular instance. An offline partial evaluator processes an *annotated* program, in which the annotations determine whether an operator is to be applied or not. Offline partial evaluators are generally more conservative, but simpler and more predictable; we focus on this type in this article.

As an example, we annotate the *power* function, which computes x^n, for specialisation with a known value for n. We annotate each operator with a *binding-time*, S (static) or D (dynamic), and we write function application explicitly as @ so that we can annotate it. Operators annotated static are performed during partial evaluation.

$$\begin{aligned} \mathit{power}\ n\ x = {} & \mathbf{if}^S\ n =^S 0 \\ & \mathbf{then}\ \mathit{Int}^{SD}\ 1 \\ & \mathbf{else}\ x \times \mathit{power}@^S(n -^S 1)@^S x \end{aligned}$$

In annotated programs we distinguish between known static values, and the corresponding dynamic code fragment; in this example, since the result of specialising *power* is code, the *coercion* Int^{SD} must be used to convert the static integer 1 to the correct type.

Annotated programs can be *interpreted* by a partial evaluator, or *compiled* into a *generating extension*. This is a program which, given the partially known input, generates a specialised version of the annotated program directly. The generating extension of this annotated *power* function is itself a recursive function, which computes the static operations directly, and generates code for the dynamic ones. Running the generating extension with the arguments 3 and "x" (a code fragment) produces the code fragment "$x \times x \times x \times 1$". Notice that, in the generating extension, a static integer and a dynamic integer are represented by different types: the former by an integer, and the latter by a code fragment — for example, an abstract syntax tree. Thus coercions do real work.

However, fixed annotations work poorly in large programs. Library functions in particular may be called in many contexts, with combinations of static and dynamic arguments which are unknown at the time the function definition is annotated. This motivates *polychronic* annotations[1] containing binding-time *variables*, which are passed as parameters to annotated functions [7]. Using polychronic annotations, we can annotate the *power* function as

$$\begin{aligned} \mathit{power}\ \beta_1\ \beta_2\ n\ x = {} & \mathbf{if}^{\beta_1}\ n =^{\beta_1} \mathit{Int}^{S\beta_1}\ 0 \\ & \mathbf{then}\ \mathit{Int}^{S(\beta_1 \sqcup \beta_2)}\ 1 \\ & \mathbf{else}\ x \times^{\beta_1 \sqcup \beta_2} \mathit{power}\ \beta_1\ \beta_2 @^S (n -^{\beta_1} \mathit{Int}^{S\beta_1}\ 1)@^S x \end{aligned}$$

[1] Also, confusingly, called "polymorphic".

where the least upper bound of two binding times is determined by $S \leq D$. This version can be specialised to any combination of known and unknown arguments, but binding-times must actually be computed and passed as parameters in the generating extension, increasing the cost of specialisation somewhat. Notice also that many more coercions are needed, now that the binding-times are no longer known a priori.

The binding-time behaviour of this function can be captured by a *binding-time type*,

$$\mathit{power} :: \forall \beta_1, \beta_2 . \mathit{Int}^{\beta_1} \rightarrow^{S} \mathit{Int}^{\beta_2} \rightarrow^{S} \mathit{Int}^{\beta_1 \sqcup \beta_2}$$

in which each type constructor is annotated to indicate whether the corresponding value is known. Program annotations can be generated by inferring these types using a binding-time type system. Types must always be *well-formed*, in the sense that no static type appears under a dynamic type constructor.

2 What About Polymorphism?

When we try to incorporate polymorphic functions into this framework, we immediately run into difficulties. Consider, for example, a possible binding-time type for the *map* function:

$$\mathit{map} :: \forall \alpha_1, \alpha_2 . \forall \beta_1, \beta_2 . (\alpha_1 \rightarrow^{\beta_1} \alpha_2) \rightarrow^{S} [\alpha_1]^{\beta_2} \rightarrow^{S} [\alpha_2]^{\beta_2}$$

But not every instantiation of this type is well-formed: if either β_1 or β_2 is D, then neither α_1 nor α_2 may be instantiated to a static type, since this would produce an ill-formed type containing a static type under a dynamic type constructor. To capture such dependencies between variables, we add *constraints* to our binding-time types, which all instantiations must satisfy. Writing $\beta \triangleright \alpha$ for the constraint that if β is D, then α must be a dynamic type, we can give a correct type for *map* as

$$\begin{aligned}\mathit{map} :: \forall \alpha_1, \alpha_2 . \forall \beta_1, \beta_2 . (\beta_1 \triangleright \alpha_1, \beta_1 \triangleright \alpha_2, \beta_2 \triangleright \alpha_1, \beta_2 \triangleright \alpha_2) \Rightarrow \\ (\alpha_1 \rightarrow^{\beta_1} \alpha_2) \rightarrow^{S} [\alpha_1]^{\beta_2} \rightarrow^{S} [\alpha_2]^{\beta_2}\end{aligned}$$

These constraints have been used before [7], but did not appear in binding-time types since that paper did not consider polymorphism.

Now consider an even simpler polymorphic function,

$$\mathit{twice}\ f\ x = f@(f@x)$$

The standard type of this function is $\forall \alpha . (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$, but for the purposes of specialisation we can be more liberal: we can allow the argument and result of f to have different binding-time types, provided the result can be coerced to the argument type. Thus we also need a *coercion* or *subtyping* constraint $\alpha_1 \leq \alpha_2$, which lets us give *twice* the binding-time type

$$\mathit{twice} :: \forall \alpha_1, \alpha_2 . \forall \beta . (\beta \triangleright \alpha_1, \beta \triangleright \alpha_2, \alpha_2 \leq \alpha_1) \Rightarrow (\alpha_1 \rightarrow^{\beta} \alpha_2) \rightarrow^{S} \alpha_1 \rightarrow^{S} \alpha_2$$

However, there is more than one way that we might choose to annotate the *definition* of *twice*.

We might expect that, just as we pass binding-times explicitly in annotated programs, we should pass types explicitly to annotated polymorphic functions. Annotating *twice* in this way would result in something like

$$\mathit{twice}\ \alpha_1\ \alpha_2\ \beta\ f\ x = f@^{\beta}([\alpha_2 \mapsto \alpha_1]\ (f@^{\beta}x))$$

But notice that we need a coercion, which we have written as $[\alpha_2 \mapsto \alpha_1]$, between two unknown types here! The compiled code for a generating extension will need to construct representations of types during specialisation, pass them as parameters, and interpret them in order to implement such coercions. Because types may be complex, this may be expensive, and in any case we prefer to avoid interpretation in generating extensions.

Therefore, we treat polymorphic functions differently. Rather than passing *types* as parameters, we pass the necessary *coercion functions*, one for each subtype constraint in the function's type. With this idea, the annotated version of *twice* becomes

$$\mathit{twice}\ \beta\ \xi\ f\ x = f@^{\beta}(\xi\ (f@^{\beta}x))$$

where ξ implements the coercion $\alpha_2 \leq \alpha_1$. At each call of *twice*, we can pass a specialised coercion function for the types which actually occur.

3 Binding-Time Analysis

Binding-time annotations are usually constructed automatically by a *binding-time analyser*. We specify our polymorphic binding-time analysis via a type system for annotated programs, which guarantees that operations annotated as static never depend on dynamic values. Given an unannotated program, the binding-time analyser finds well-typed annotations that make as many operations as possible static. This type-based approach builds on earlier work by Dussart, Henglein and Mossin [12, 7], which has been adopted for the Similix partial evaluator [2]. We favour a type-based approach because it is efficient, comprehensible, and extends naturally to handle polymorphism.

We shall specify the binding-time type system for the smallest interesting language, and then discuss how it is used to infer annotations.

3.1 The Binding-Time Type System

We consider an annotated λ-calculus with polymorphic **let** and one base type:

$$\begin{array}{lll} e[\mathit{Expression}] & ::= & c \mid x \mid \mathbf{let}\ x = e\ \mathbf{in}\ e \mid \lambda x.e \mid e@^{b}\phi\ e \mid \lambda\beta.e \mid e\ b \mid \lambda\xi.e \mid e\ \phi \\ b[\mathit{Binding\text{-}time}] & ::= & S \mid D \mid \beta \mid b \sqcup b \\ \phi[\mathit{Coercion}] & ::= & \iota \mid \xi \mid \mathit{Int}^{bb} \mid \phi \rightarrow^{bb} \phi \end{array}$$

Here β is a binding-time variable, ξ is a coercion variable, x is a program variable, and c is a constant.

In this simple language, only function application need be annotated with a binding-time, and only function arguments need be coerced. Constants and λ-expressions are always static, and are coerced to be dynamic where necessary. **let**-expressions are always dynamic, but their bodies may even so be static since we use Bondorf's CPS specialisation [3], which moves the context of a **let** into its body, where it can be specialised. Applications to binding-times and coercions always take place during specialisation, and so need no annotation.

We have already seen integer coercions. A coercion $\phi_1 \rightarrow^{b_1 b_2} \phi_2$ coerces a function with binding-time b_1 to one with binding-time b_2, applying coercion ϕ_1 to the argument and ϕ_2 to the result. ι is the identity coercion.

Binding-time types and constraints take the form

$$\begin{array}{l} \tau[\mathit{Monotype}] ::= \alpha \mid \mathit{Int}^b \mid \tau \rightarrow^b \tau \\ c[\mathit{Constraint}] ::= b \leq b \mid b \rhd \tau \mid \phi : \tau \leq \tau \end{array}$$

The complete set of binding-time type inference rules can be found in the appendix; here we focus on the rule for application:

$$\frac{\Gamma; C \vdash e_1 : (\tau_1 \rightarrow \tau_2)^b \quad \Gamma; C \vdash e_2 : \tau_3 \quad C \vdash \phi : \tau_3 \leq \tau_1 \quad C \vdash b \rhd \tau_1 \quad C \vdash b \rhd \tau_2}{\Gamma; C \vdash (e_1 \mathbin{@^b} \phi\ e_2) : \tau_2}$$

As usual in a binding-time type system, our judgements depend both on an environment Γ and a set of constraints C. Notice, however, that our subtype constraints include the coercion that maps one type to the other. Thus, from the constraint set C, we infer *which* coercion ϕ converts τ_3 to τ_1. Notice also that we include $\rhd$-constraints to guarantee that the type of the function is well-formed. Finally, the annotation on the application is taken from the type of the function.

Our constraint inference rules, with judgements of the form $C \vdash c$, can be found in the appendix. They are mostly standard, with the exception that the rules for subtyping actually construct a coercion. For example, the rule for function types

$$\frac{C \vdash \phi_1 : \tau_3 \leq \tau_1 \quad C \vdash \phi_2 : \tau_2 \leq \tau_4 \quad C \vdash b_1 \leq b_2}{C \vdash \phi_1 \rightarrow^{b_1 b_2} \phi_2 : \tau_1 \rightarrow^{b_1} \tau_2 \leq \tau_3 \rightarrow^{b_2} \tau_4}$$

constructs a coercion on functions from coercions on the argument and result. Where possible, we use the identity coercion

$$C \vdash \iota : \tau \leq \tau$$

which can be removed altogether by a post-processor. We restrict the coercions in C to be distinct coercion variables; thus we can think of C as a kind of environment, binding coercion variables to their types.

As in the Hindley-Milner type system, **let**-bound variables may have *type schemes* rather than monotypes. Type-schemes take the form

$$\begin{array}{ll} \gamma[\mathit{Qualified\ type}] & ::= \tau \mid q \Rightarrow \gamma \\ q[\mathit{Qualifier}] & ::= b \leq b \mid b \rhd \tau \mid \tau \leq \tau \\ \pi[\mathit{Polychronic\ type}] & ::= \gamma \mid \forall \beta . \pi \\ \sigma[\mathit{Polymorphic\ type}] & ::= \pi \mid \forall \alpha . \sigma \end{array}$$

We give a complete set of rules to introduce and eliminate type-schemes in the appendix; note that although our rule system is not syntax-directed, it is easy to transform it into a syntax-directed system because of the restriction on where type schemes may appear. Here we discuss only the rules which are not standard.

Notice that qualifiers are almost, but not exactly, the same as constraints. The difference is that sub-type qualifiers $\tau_1 \leq \tau_2$ do not mention a coercion. Looking at the rules for introducing and eliminating such a qualifier

$$\frac{\Gamma; C, \xi{:}\tau_1 \leq \tau_2 \vdash e : \gamma}{\Gamma; C \vdash \lambda\xi.e : \tau_1 \leq \tau_2 \Rightarrow \gamma} \qquad \frac{\Gamma; C \vdash e : \tau_1 \leq \tau_2 \Rightarrow \gamma \quad C \vdash \phi{:}\tau_1 \leq \tau_2}{\Gamma; C \vdash e\ \phi : \gamma}$$

we see why: the coercion in the constraint becomes the bound variable of a coercion abstraction; it would be unnatural to allow bound variable names in types. That we 'forget' the coercion doesn't matter: it can be recreated where it is needed by the elimination rule.

The rules for generalising and instantiating type variables are standard, except that we only allow instantiation with well-formed types. The rules for binding-time variables just introduce binding-time abstraction and application:

$$\frac{\Gamma; C \vdash e : \gamma}{\Gamma; C \vdash \lambda\beta.e : \forall\beta.\gamma}\ \beta \notin FV(C, \Gamma) \qquad \frac{\Gamma; C \vdash e : \forall\beta.\gamma}{\Gamma; C \vdash e\ b : \gamma[b/\beta]}$$

Given any unannotated expression which is well-typed in the Hindley-Milner system, we can construct a well-typed annotated expression by annotating each application with a fresh binding-time variable and a fresh coercion variable, moving constraints into qualified types, and generalising all possible variables. But this leads to polymorphic definitions with very many generalised variables, and very many qualifiers. In the remainder of this section we will see how to reduce this multitude.

3.2 Simplifying Constraints

Before generalising the type of a **let**-bound variable, it is natural to simplify the constraints as much as possible. Simplification of this kind of constraint is mostly standard [11], except that we keep track of coercions also; essentially we use the constraint inference rules in the appendix backwards, instantiating variables where necessary to make rules match. For example, we simplify the constraint $\xi : \alpha \leq Int^b$ by instantiating α to Int^β and ξ to $Int^{\beta b}$, where β is fresh, and then simplifying the constraint to $\beta \leq b$. Simplification of this kind does not change the set of solutions of the constraints.

We use two non-standard simplification rules also. Firstly, whenever we discover a cycle of binding-time variables $\beta_1 \leq \cdots \leq \beta_n \leq \beta_1$, we instantiate each β_i to the same variable. We treat cycles of type variables similarly, which much reduces the number of variables we need to quantify over. Secondly, we simplify the constraints $\{D \rhd \alpha_1, \xi : \alpha_1 \leq \alpha_2\}$ by instantiating α_2 to α_1 and ξ to ι: this preserves the set of solutions because both α_1 and α_2 have to be well-formed types annotated D at the top-level, and one such type can be a subtype of another only if they are equal.

Simplification terminates, which can be shown by a lexicographic argument: each rule reduces the size of types, the number of $\triangleright$-constraints, the number of $\sqcup$s to the left of $\leq$, or the total number of constraints.

3.3 Simplifying Polymorphic Types

The simplifications in the previous section preserve the set of instances of a polymorphic type. That is, if we simplify a type scheme σ_1 to a type scheme σ_2, then any instance τ_1 of σ_1 is guaranteed also to be an instance of σ_2. But we can go further, if we guarantee only that there is an instance τ_2 of σ_2 which is a *subtype* of τ_1. This still enables us to use a polymorphic value of type σ_2 at any instance of σ_1, provided we introduce a coercion. For example, we can simplify the type of the *power* function from $\forall\beta_1,\beta_2,\beta_3.(\beta_1 \leq \beta_3, \beta_2 \leq \beta_3) \Rightarrow \mathit{Int}^{\beta_1} \to^S \mathit{Int}^{\beta_2} \to^S \mathit{Int}^{\beta_3}$ to $\forall\beta_1,\beta_2.\mathit{Int}^{\beta_1} \to^S \mathit{Int}^{\beta_2} \to^S \mathit{Int}^{\beta_1 \sqcup \beta_2}$; these two types do not have the same instances, but any instance of the first can be derived by coercing an instance of the second. The second type has fewer quantified variables and coercions, and is therefore cheaper to specialise.

This subtype condition is guaranteed by ensuring that variables occurring negatively in the type are only instantiated to smaller quantities, while variables occurring positively are only instantiated to larger ones. Moreover, simplification must not increase the binding-time of any program annotation, otherwise it would lead to poorer specialisation. Positively occurring binding-time variables therefore cannot be instantiated at all. Dussart et al. [7] simplify by instantiating non-positive binding-time variables to the least upper bound of their lower bounds (as in the *power* example above).

In the presence of polymorphism, we instantiate type variables also. We might treat non-positive type variables in the same way that Dussart et al. treat binding-time variables, but this would introduce least upper bounds of type variables. This would be problematic for us, since we pass coercions and not types as parameters during specialisation: while it is straightforward (if expensive) to compute the least upper bound of two types, computing the least upper bound of two coercions would be far harder. But in two special cases, we can instantiate non-positive type variables to smaller types *without* needing least upper bounds.

Firstly, if $\xi : \alpha_1 \leq \alpha_2$ is the *only* constraint imposing a lower bound on α_2, and α_2 is non-positive, then we can instantiate α_2 to α_1 and ξ to ι. We also must insist that α_1 and α_2 are forced by the same set of binding-times; otherwise unifying them might make α_1 more dynamic.

Secondly, if α_1 and α_2 are *both* non-positive, have the same set of lower bounds, and are forced by the same binding-times, then they must take the same value in the least solution of the constraints, and we can unify them — even though we cannot express this least solution without least upper bound.

But there is another way to simplify constraints on type variables: we can instantiate non-negative type variables to *larger* types! This does potentially make some *types* more dynamic, but no *binding-times*, and it is the binding-time annotations which determine the quality of specialisation, not the types. We can do this in cases analogous to the two above, except that we need not be concerned with the binding-times which force type variables, since we *expect* to make type variables more dynamic. This process is specified formally in the appendix.

This form of simplification terminates since each step eliminates one variable.

For example, the type inferred for the *map* function, after simplifying binding-times, is

$$\begin{array}{l}\forall \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5 . \forall \beta_1, \beta_2 . \\ \quad (\beta_1 \rhd \alpha_1, \beta_1 \rhd \alpha_2, \beta_2 \rhd \alpha_3, \beta_2 \rhd \alpha_4, \alpha_3 \leq \alpha_1, \alpha_2 \leq \alpha_4, \alpha_5 \leq \alpha_4) \Rightarrow \\ \quad\quad (\alpha_1 \to^{\beta_1} \alpha_2) \to^S [\alpha_3]^{\beta_2} \to^S [\alpha_4]^{\beta_2}\end{array}$$

(where α_5 is a type variable internal to the definition of *map*). α_4 has two lower bounds, so cannot be reduced, while α_1 cannot be reduced to its only lower bound α_3 since $\beta_1 \rhd \alpha_1$, but β_1 does not force α_3. However, α_2, α_3, and α_5 are all non-positive and have unique upper bounds, so we can increase all three to their upper bounds and simplify the type to

$$\begin{array}{l}\forall \alpha_1, \alpha_2 . \forall \beta_1, \beta_2 . (\beta_1 \rhd \alpha_1, \beta_1 \rhd \alpha_2, \beta_2 \rhd \alpha_1, \beta_2 \rhd \alpha_2) \Rightarrow \\ \quad (\alpha_1 \to^{\beta_1} \alpha_2) \to^S [\alpha_1]^{\beta_2} \to^S [\alpha_2]^{\beta_2}\end{array}$$

The number of coercion parameters is decreased from three to zero.

4 Discussion

We have implemented this binding-time analysis in a prototype partial evaluator for polymorphic programs [10]. In practice, every binding-time analyser sometimes makes *too many* operations static, causing partial evaluation to loop, and ours is no exception. This must be prevented using user annotations, which have to be rethought in a polymorphic context. The full paper will contain details.

Polymorphism is particularly important for programs made up of many modules. In earlier work on specialising modules [8, 6, 9] we discovered we needed polymorphic binding-time analysis, which directly inspired this work.

Our analysis is built on Henglein et al's earlier polychronic analyses [12, 7]. Consel et al generalised their work in a different direction [4]. Binding-time analysers for polymorphic programs have also been developed based on abstract interpretation [15, 19], although this approach is now little used in practice.

This paper considers only parametric polymorphism, without overloading. We hope to extend our system to handle overloading based on Haskell classes [21].

Polymorphic typing is integral to widely used functional programming languages such as ML and Haskell, and has also been adopted in other languages such as Mercury [20]. Polymorphic binding-time analysis, such as ours, is vital if program specialisation is to be applied to such languages in practice.

References

1. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).
2. A. Bondorf. Automatic autoprojection of higher-order recursion equations. In N. Jones, editor, *3rd European Symposium on Programming*, LNCS, Copenhagen, 1990. Springer-Verlag.
3. A. Bondorf. Improving binding times without explicit cps-conversion. In *1992 ACM Conference on Lisp and Functional Programming. San Francisco, California*, pages 1–10, June 1992.
4. C. Consel and P. Jouvelot. Separate Polyvariant Binding-Time Analysis. Technical Report CS/E 93-006, Oregon Graduate Institute Tech, 1993.
5. Charles Consel. A tour of schism: a partial evaluation system for higher-order applicative languages. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154, June 1993.
6. D. Dussart, R. Heldal, and J. Hughes. Module-Sensitive Program Specialisation. In *Conference on Programming Language Design and Implementation*, Las Vegas, June 1997. ACM SIGPLAN.
7. D. Dussart, F. Henglein, and C. Mossin. Polymorphic Recursion and Subtype Qualifications: Polymorphic Binding-Time Analysis in Polynomial Time. In Alan Mycroft, editor, *SAS'95: 2nd Int'l Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 118–135, Glasgow, Scotland, September 1995. Springer-Verlag.
8. R. Heldal and J. Hughes. Partial Evaluation and Separate Compilation. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Amsterdam, June 1997. ACM SIGPLAN.
9. R. Heldal and J. Hughes. Extending a partial evaluator which supports separate compilation. *Theoretical Computer Science 248*, pages 99–145, 2000.
10. Rogardt Heldal. *The Treatment of Polymorphism and Modules in a Partial Evaluator*. PhD thesis, Chalmers University of Technology, April 2001.
11. F. Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *FPCA*, pages 448–472. 5th ACM Conference, Cambridge, MA, USA, Springer-Verlag, August 1991. Lecture Notes in Computer Science, Vol. 523.
12. F. Henglein and C. Mossin. Polymorphic Binding-Time Analysis. In D. Sannella, editor, *ESOP'94: European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, April 1994.
13. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
14. Simon Peyton Jones, John Hughes, (editors), Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language. Available from `http://haskell.org`, February 1999.
15. J. Launchbury. *Projection Factorisations in Partial Evaluation (PhD thesis)*, volume 1 of *Distinguished Dissertations in Computer Science*. Cambridge University Press, 1991.
16. K. Malmkjr, N. Heintze, and O. Danvy. ML partial evaluation using set-based analysis. In *Workshop on ML and its Applications*, pages 112–119. ACM SIGPLAN, 1994.

17. R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and Systems Sciences*, 17:348–375, 1978.
18. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.
19. T. Æ. Mogensen. Binding Time Analysis for Polymorphically Typed Higher Order Languages. In *Theory and Practice of Software Development*, volume 352 of *Lecture Notes in Computer Science*, pages 298–312. Springer-Verlag, March 1989.
20. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, Glenelg, Australia, February 1995.
21. P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc*. In *Proceedings 1989 Symposium Principles of Programming Languages*, pages 60–76, Austin, Texas, 1989.

A Appendix: Binding-Time Rules

$$\Gamma, x{:}\sigma, \Gamma'; C \vdash x : \sigma \qquad \Gamma; C \vdash c : Int^S \qquad \frac{C \vdash \tau_1 \text{ wft} \quad \Gamma, x{:}\tau_1; C \vdash e : \tau_2}{\Gamma; C \vdash \lambda x.e : (\tau_1 \rightarrow^S \tau_2)}$$

$$\frac{\Gamma; C \vdash e_1 : (\tau_1 \rightarrow \tau_2)^b \quad \Gamma; C \vdash e_2 : \tau_3 \quad C \vdash \phi : \tau_3 \leq \tau_1 \quad C \vdash b \rhd \tau_1 \quad C \vdash b \rhd \tau_2}{\Gamma; C \vdash (e_1 \ @^b \ \phi \ e_2) : \tau_2}$$

$$\frac{\Gamma; C \vdash e_1 : \sigma \quad \Gamma, x{:}\sigma; C \vdash e_2 : \tau}{\Gamma; C \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau}$$

Fig. 1. Syntax Directed Binding-time Rules for Expressions.

$$\frac{\Gamma; C \vdash e : \gamma}{\Gamma; C \vdash \lambda\beta.e : \forall\beta.\gamma}\ \beta \notin FV(C, \Gamma) \qquad \frac{\Gamma; C \vdash e : \forall\beta.\gamma}{\Gamma; C \vdash e\ b : \gamma[b/\beta]}$$

$$\frac{\Gamma; C, b_1 \leq b_2 \vdash e : \gamma}{\Gamma; C \vdash e : b_1 \leq b_2 \Rightarrow \gamma} \quad \frac{\Gamma; C, b \rhd \tau \vdash e : \gamma}{\Gamma; C \vdash e : b \rhd \tau \Rightarrow \gamma} \quad \frac{\Gamma; C, \xi{:}\tau_1 \leq \tau_2 \vdash e : \gamma}{\Gamma; C \vdash \lambda\xi.e : \tau_1 \leq \tau_2 \Rightarrow \gamma}$$

$$\frac{\Gamma; C \vdash e : b_1 \leq b_2 \Rightarrow \gamma \quad C \vdash b_1 \leq b_2}{\Gamma; C \vdash e : \gamma} \quad \frac{\Gamma; C \vdash e : b \rhd \tau \Rightarrow \gamma \quad C \vdash b \rhd \tau}{\Gamma; C \vdash e : \gamma}$$

$$\frac{\Gamma; C \vdash e : \tau_1 \leq \tau_2 \Rightarrow \gamma \quad C \vdash \phi{:}\tau_1 \leq \tau_2}{\Gamma; C \vdash e\ \phi : \gamma}$$

$$\frac{\Gamma; C \vdash e : \sigma}{\Gamma; C \vdash e : \forall\alpha.\sigma}\ \alpha \notin FV(C, \Gamma) \quad \frac{\Gamma; C \vdash e : \forall\alpha.\sigma \quad C \vdash \tau \text{ wft}}{\Gamma; C \vdash e : \sigma[\tau/\alpha]}$$

Fig. 2. Non-Syntax Directed Rules

$$C, c \vdash c \qquad C \vdash \iota : \tau \leq \tau \qquad \frac{C \vdash b_1 \leq b_2}{C \vdash \mathit{Int}^{b_1 b_2} : \mathit{Int}^{b_1} \leq \mathit{Int}^{b_2}}$$

$$\frac{C \vdash \phi_1{:}\tau_3 \leq \tau_1 \quad C \vdash \phi_2{:}\tau_2 \leq \tau_4 \quad C \vdash b_1 \leq b_2}{C \vdash \phi_1 \to^{b_1 b_2} \phi_2 : \tau_1 \to^{b_1} \tau_2 \leq \tau_3 \to^{b_2} \tau_4}$$

$$\frac{C \vdash b_1 \leq b_2}{C \vdash b_1 \rhd \mathit{Int}^{b_2}} \qquad C \vdash S \rhd \tau \qquad \frac{C \vdash b_1 \leq b_2}{C \vdash b_1 \rhd \tau_1 \to^{b_2} \tau_2}$$

$$C \vdash b \leq b \qquad C \vdash S \leq b \qquad C \vdash b \leq D \qquad C \vdash \beta_i \leq \sqcup \beta_i \qquad \frac{C \vdash \beta_1 \leq \beta_3 \quad C \vdash \beta_2 \leq \beta_3}{C \vdash \beta_1 \sqcup \beta_2 \leq \beta_3}$$

Fig. 3. Constraint Inference Rules

$$C \vdash \alpha \text{ wft} \qquad C \vdash \mathit{Base}^b \text{ wft} \qquad \frac{C \vdash \tau_1 \text{ wft} \quad C \vdash \tau_2 \text{ wft} \quad C \vdash b \rhd \tau_1 \quad C \vdash b \rhd \tau_2}{C \vdash \tau_1 \to^b \tau_2 \text{ wft}}$$

Fig. 4. Well-formedness of Types.

Each time one of the rules below is applied, the constraints must first be normalised and the set of force constraints must be closed using the following rule:

$$\{\beta \rhd \alpha_1, \xi : \alpha_1 \leq \alpha_2\} \leadsto \{\beta \rhd \alpha_1, \beta \rhd \alpha_2, \xi : \alpha_1 \leq \alpha_2\}$$

To simplify a type τ and constraint set C in an environment Γ:

$$\beta \notin (|\tau|^- \cup FV(\Gamma)) \Rightarrow C \leadsto C_\beta[\beta := \sqcup_{\beta' \in C_{\leq \beta}} \beta'];\, \beta := \sqcup_{\beta' \in C_{\leq \beta}} \beta'$$

$$\begin{aligned}&\alpha \notin (|\tau|^- \cup FV(\Gamma)) \ \wedge\ C_{\leq \alpha} = \{\} \ \wedge\ C_{\rhd \alpha} \subseteq C_{\rhd \alpha_1}\\ &\Rightarrow C, \xi : \alpha_1 \leq \alpha \leadsto C[\alpha := \alpha_1];\, \alpha := \alpha_1, \xi := \iota\end{aligned}$$

$$\begin{aligned}&\alpha_1, \alpha_2 \notin (|\tau|^- \cup FV(\Gamma)) \ \wedge\ C_{\leq \alpha_1} = C_{\leq \alpha_2} \ \wedge\ C_{\rhd \alpha_1} = C_{\rhd \alpha_2}\\ &\Rightarrow C \leadsto C[\alpha_1 := \alpha_2];\, \alpha_1 := \alpha_2\end{aligned}$$

$$\begin{aligned}&\alpha \notin (|\tau|^+ \cup FV(\Gamma)) \wedge\ C_{\alpha \leq} = \{\}\\ &\Rightarrow C, \xi : \alpha \leq \alpha_1 \leadsto C[\alpha := \alpha_1];\, \alpha := \alpha_1, \xi := \iota\end{aligned}$$

$$\begin{aligned}&\alpha_1, \alpha_2 \notin (|\tau|^+ \cup FV(\Gamma)) \ \wedge\ C_{\alpha_1 \leq} = C_{\alpha_2 \leq}\\ &\Rightarrow C \leadsto C[\alpha_1 := \alpha_2];\, \alpha_1 := \alpha_2\end{aligned}$$

where

$$\begin{array}{lll} C_{\leq \beta} \triangleq \{\beta_1 | \beta_1 \leq \beta \in C\} & C_\beta \triangleq C - \{\beta_1 \leq \beta \,|\beta_1 \in \mathbb{B}\} & \\ C_{\leq \alpha} \triangleq \{\alpha_1 | \xi : \alpha_1 \leq \alpha \in C\} & C_{\rhd \alpha} \triangleq \{b | b \rhd \alpha \in C\} & C_{\alpha \leq} \triangleq \{\alpha_1 | \xi : \alpha \leq \alpha_1 \in C\} \end{array}$$

Fig. 5. Increasing and Decreasing Variables.