Polymorphic Recursion and Subtype Qualifications: Polymorphic Binding-Time Analysis in Polynomial Time*

Dirk Dussart[1], Fritz Henglein[2] and Christian Mossin[2]

[1] K.U.Leuven, Belgium***
[2] DIKU, Denmark†

Abstract. The combination of *parameter polymorphism*, *subtyping* extended to *qualified* and polymorphic types, and *polymorphic recursion* is useful in standard type inference and gives expressive type-based program analyses, but raises difficult algorithmic problems.
In a program analysis context we show how Mycroft's iterative method of computing principal types for a type system with polymorphic recursion can be generalized and adapted to work in a setting with subtyping. This does not only yield a proof of existence of *principal types* (most general properties), but also an algorithm for computing them. The punch-line of the development is that a very simple modification of the basic algorithm reduces its computational complexity from exponential time to polynomial time relative to the size of the given, explicitly typed program.
This solves the open problem of finding an inference algorithm for *polymorphic binding-time analysis* [7].

1 Introduction

This paper deals with the interaction of two important typing disciplines: *subtyping* and *polymorphic typing* with *polymorphic recursion*. We will study this interaction in the context of *binding-time analysis*.

Polymorphic recursion was introduced in [12] as an extension of ML style polymorphism that allows recursive definitions to be used at different types inside the definition. Mycroft was the first to give a (*semi*)-algorithm for computing principal types for his system. His algorithm basically works by iterating algorithm $\mathcal{W}$. It later turned out that the type inference problem associated with Mycroft's type system is undecidable, see [10]. The extension is, however, desirable both in the context of standard typing and program analysis [14, 7, 17].

Binding-time analysis is an analysis used in *partial evaluation* to separate those program expressions that can be safely evaluated at *specialization-time*

* To appear in Proc. 2nd Int'l Static Analysis Symposium (SAS), Glasgow, Scotland, Sep. 25–27, 1995, Lecture Notes in Computer Science, Springer-Verlag
*** Supported by the Belgian National Fund for Scientific Research
† Supported by the Danish Technical Research Council

from those that cannot. This separation is made on the basis of information about which parts of the program input will be available at specialization-time only. The program parts that only depend on known data and therefore can be safely evaluated at specialization-time will be marked (annotated) *static*. Those parts that possibly depend on unknown data will be marked *dynamic*.

Binding-time analysis can be specified by *binding-time logics*, type inference systems with a simple subtyping theory. Most binding-time logics used in practice are *monomorphic* or *monovariant* in the sense that each function must be assigned a single argument/result pair of binding times. This is a severe restriction, since every function must be assigned a binding that is a conservative approximation of all uses of the function. The results of monovariant analyses can be improved by either manually or automatically copying the functions in as many variants as there are different binding-time calling contexts. The downside of the copy approach is that, first of all, the size of annotated program can be exponential in the size of the original program and hence also the complexity of the analysis; furthermore, it is difficult to provide feedback to the programmer about the binding-time properties of the original program.

In [7] it is demonstrated that polymorphism can be used to obtain the same results without copying. The idea is that functions may be assigned *polymorphic* binding-time properties; that is, they may have binding-time *parameters* that are instantiated independently in each actual use context of a function. Instantiation and copying is deferred to specialization time — it is performed lazily. It is shown that polymorphism subsumes copying in the sense that unfolding or unrolling of function definitions, whether recursive or not, does not improve the results of the binding time analysis. Furthermore, since every expression has a *principal* (most general) binding-time property, the analysis is modular. To achieve a high degree of polyvariance for recursively defined functions the binding-time logic uses polymorphic recursion over binding-time properties.

Example 1. Consider the following term

$$\mathrm{fix\,f.}\ \lambda \mathrm{x}.\ \lambda \mathrm{y}.\ \mathrm{if\ x} = 0\ \mathrm{then}\ 1\ \mathrm{else}\ (\mathrm{f@y})@(\mathrm{x}-1)$$

where the recursive call permutes its arguments. This implies that with a monovariant binding-time analysis both formal parameters will be made dynamic when one of these turns out to be dynamic. With binding-time type polymorphism, on the other hand, we are able to use the same definition at different binding-time type instances. The result of annotating this function would be[5]:

$$\mathrm{fix\,f.}\ \Lambda\beta_{\mathrm{x}}.\Lambda\beta_{\mathrm{y}}.\Lambda\beta.\lambda \mathrm{x}.\ \lambda \mathrm{y}.\ \mathrm{if}^{\beta_{\mathrm{x}}}\ \mathrm{x} =^{\beta_{\mathrm{x}}} 0\ \mathrm{then}\ [\mathrm{S} \leadsto \beta]1\ \mathrm{else}\ (\mathrm{f}\{\beta_{\mathrm{y}}\}\{\beta_{\mathrm{x}}\}\{\beta\}@^{\mathrm{S}}\mathrm{y})@^{\mathrm{S}}(\mathrm{x} -^{\beta_{\mathrm{x}}} 1)$$

where $\Lambda\beta.e$ is used for binding-time abstraction and $e\{b\}$ is used for application for such abstractions. Assume that we apply this function to the value 3 as first and dynamic second argument, that is $(\mathrm{fix\,f.}\ \ldots)\{\mathrm{S}\}\{\mathrm{D}\}\{\mathrm{D}\}@^{\mathrm{S}}3@^{\mathrm{S}}\mathrm{d}$ where d is a dynamic variable. Now unfolding yields (since the conditional is static every second time):

$$\mathrm{if\ y} = 0\ \mathrm{then}\ 1\ \mathrm{else\ if}\ (\mathrm{y}-1) = 0\ \mathrm{then}\ 1\ \mathrm{else\ if}\ (\mathrm{y}-2) = 0\ \mathrm{then}\ 1\ \mathrm{else}\ 1$$

[5] note that even though we do not include annotations in this paper, annotated programs are trivially derived from the inference tree.

We will return to this example later showing how to compute its principal type.

Section 2 provides the basic definitions of the elements of our binding-time analysis: the syntax of our higher-order typed programming language, formal languages for expressing primitive and structured binding-time properties, and a formalization of the subtyping (implication) relation that holds between binding-time properties. In Section 3 we start with a logical specification of our binding-time analysis and transform it in several steps to a first algorithm for computing principal binding-time properties. In Section 4 we improve the algorithm to run in polynomial time instead of exponential time. Section 5 provides a brief summary of closely related work, and Section 6 concludes with a summary and possible extensions and future work.

2 Annotations and Types

This section provides the basic definition for the binding-time analyses that will be used in later sections.

2.1 Language

As in [7] we consider a simply-typed lambda-calculus extended with booleans ('true', 'false', and 'if e then e′ else e″'), let and fix-point operator. Let and fix are used as definition mechanism for named non-recursive and recursive functions, respectively. The typing rules of the language are standard and hence omitted.

2.2 Binding-time annotations

The aim of binding time analysis is to annotate every expression as being either *static* (definitely evaluable at specialisation time) or *dynamic* (possibly not evaluable until run time). We use the *primitive binding time values* S, D to denote "static", respectively "dynamic". We also add *binding-time variables* ranged over by β, and a formal *least upper bound* operator $\vee$ on binding-time values. Here we differ from [7] where the domain of binding times consists of constants and variables. Having $\vee$ allows us to express that an expression depends on more than one piece of data in a single binding-time annotation, avoiding the introduction of additional binding-time variables and constraints. This does, however, defer some of the binding-time computations until specialization-time[6].

The language of *binding-time annotations* is ranged over by b and generated by the production

$$b \quad ::= \quad \beta \mid S \mid D \mid b \vee b'.$$

We use C to denote a set of *binding-time assumptions* of the form $\beta \leq_a \beta'$ between binding-time variables β, β'. Finally, we define an ordering on binding

[6] This computation is simple, and the cost should be weighed against the cost of passing extra binding-time parameters

time annotations, which captures the extension of $S \leq D$ (S is an "earlier" binding time than D) to binding-time annotations, including binding-time variables and and (formal) disjunctions. We write $C \vdash C'$ iff for all $\beta \leq_a \beta'$ in C' we have that $C \vdash \beta \leq_a \beta'$.

$$C \vdash S \leq_a b \qquad C \vdash b \leq_a D \qquad C \vdash b \leq_a b$$

$$\frac{C \vdash b \leq_a b' \quad C \vdash b' \leq_a b''}{C \vdash b \leq_a b''} \qquad C, \beta \leq_a \beta', C' \vdash \beta \leq_a \beta'$$

$$C \vdash b \leq_a b \vee b' \qquad C \vdash b' \leq_a b \vee b' \qquad \frac{C \vdash b \leq_a b'' \quad C \vdash b' \leq_a b''}{C \vdash b \vee b' \leq_a b''}$$

2.3 Binding-Time Types

A binding-time value such as S or D expresses only the "top-level" information for structured or higher-order expressions. For example, if an expression of type bool $\rightarrow$ bool has binding-time S then we know it will be "known" at specialization-time. What we do not know, however, is anything about the binding times of the function's argument or result. These are *structured* binding-time properties that are required to find anything out about the binding time of the result of applying such a function, *if* the function itself is static. For example it could map static arguments to static results or dynamic arguments to dynamic results. If it the function expression is dynamic, however — that is, it has top-level binding-time value D — then it is senseless to ask about its structured binding-time property: any application will be dynamic, and any argument has to be dynamic.

Structured binding-time properties are expressed by *binding-time types*, a language of types generated from κ by

$$\kappa ::= b \mid \kappa' \xrightarrow{b} \kappa''.$$

The binding-time type of an expression reflects the structure of its (standard) type. For example, Boolean expressions have only a top-level binding-time annotation b. Function expressions have binding-time properties $\kappa \xrightarrow{b} \kappa'$. Here b denotes the top-level binding-time value of the expression, and $\kappa \rightarrow \kappa'$ is the structural binding-time information, which is *only* relevant if b is S. If b is D then both κ and κ' must have top-level binding-time annotations D – and so on recursively; that is, *every* annotation in κ and κ must be D. A binding-time type such as $S \xrightarrow{D} S$ is not meaningful.

Since binding-time annotations are not simply S or D, but may contain binding-time variables, it is actually not trivial to characterize the meaningful binding-time types. What we must ensure is that the binding-time annotations occurring in κ and κ' must be at least as dynamic as the top-level annotation b in $\kappa \xrightarrow{b} \kappa'$. To do this we define judgements '$C \vdash \kappa$ wft' expressing that κ is a *well-formed* binding-time type, given the subtyping assumptions C.

$$C \vdash b \text{ wft} \qquad \frac{\begin{array}{cc} C \vdash b \leq_f \kappa & C \vdash \kappa \text{ wft} \\ C \vdash b \leq_f \kappa' & C \vdash \kappa' \text{ wft} \end{array}}{\mathrm{C} \vdash \kappa \xrightarrow{\mathrm{b}} \kappa' \text{ wft}}$$

The same well-formedness conditions for binding-time types are used in the Nielsons' work on TML in [13], only they are formulated differently. In the above we used relation $\beta \leq_f \kappa$ which express that annotation β is smaller than then top-level annotation on binding-time type κ:

$$\frac{C \vdash b \leq_a b'}{C \vdash b \leq_f b'} \qquad \frac{C \vdash b \leq_a b'}{C \vdash b \leq_f \kappa \xrightarrow{b'} \kappa'}$$

Finally, the subtyping relation $\leq_a$ on annotations induces a subtyping relation $\leq_s$ on binding-time types:

$$\frac{C \vdash b \leq_a b'}{C \vdash b \leq_s b'} \qquad \frac{C \vdash \kappa_1' \leq_s \kappa_1 \quad C \vdash \kappa_2 \leq_s \kappa_2' \quad C \vdash b \leq_a b'}{C \vdash \kappa_1 \xrightarrow{b} \kappa_2 \leq_s \kappa_1' \xrightarrow{b'} \kappa_2'}$$

The subtyping rule for function types is contravariant in the domain and covariant in the range part. One major difference compared to previous binding-time logics is the ability to use higher-order coercions. This increases the precision of the analysis but poses greater demands on the specializer.

2.4 Polymorphic Binding-Time Types

Binding-time types are extended to include *qualified (binding-time) types* ρ and *(qualified binding-time) type schemes* σ:

$$\rho ::= \kappa \mid b \leq b' \Rightarrow \rho$$
$$\sigma ::= \rho \mid \forall \beta . \sigma$$

If C is a set $\{b_{11} \leq b_{12}, \ldots, b_{n1} \leq b_{n2}\}$ of constraints and $\vec{\beta}$ is a vector $(\beta_1, \ldots, \beta_m)$ of binding-time variables, we use $\forall \vec{\beta}. C \Rightarrow \kappa$ as a shorthand for $\forall \beta_1. \cdots \forall \beta_m . b_{11} \leq b_{12} \Rightarrow \cdots \Rightarrow b_{n1} \leq b_{n2} \Rightarrow \kappa$.

Finally, well-formedness of binding-time types is extended to type schemes:

Definition 1. (Well-formedness)

- $C \vdash C' \Rightarrow \kappa$ wft iff for all C'' such that $C'' \vdash C'$ we have $C, C'' \vdash \kappa$ wft.
- $C \vdash \forall \vec{\beta}. C' \Rightarrow \kappa$ wft iff for all substitutions S and C'' such that $C, C'' \vdash S(C')$ we have $C, C'' \vdash S(C') \Rightarrow S(\kappa)$ wft.

3 Polymorphic Binding-time Analysis with Polymorphic Recursion

Having introduced the programming language of discourse and a formal language of binding-time properties we are now ready to present the binding-time analysis itself. It is given in the form of an inference system for *binding-time judgements* $C, A \vdash e : \sigma$. Here C is a sequence of subtyping assumptions between binding-time variables, A associates program variables (free in e) with binding-time types, e is a program expression, and σ is a (possibly polymorphic) binding-time type. Intuitively, such a judgement expresses that, for any assignment of binding-time values S or D to binding-time variables satisfying the constraints in C, if the program variables have binding times as prescribed by A, then e has binding time σ.

The inference rules describe how binding-time judgements can be derived formally. We shall not concern ourselves with the (semantic) correctness of the rules here, but take it for granted and view the type inference system as our (given) *problem specification*. The rules are correct in the sense that they can be used to drive a specializer to give only correct results [7].

The salient feature of this inference system is that it *combines* polymorphic subtyping with polymorphic recursion. This gives it considerable expressive strength that is useful and relevant in practice, but raises serious algorithmic questions as to how inference is to be performed.

In this and the following section we develop step by step, starting from the specification of the binding-time analysis itself, a polynomial-time algorithm for computing *principal* binding-time types. The first step consists of making the inference system syntax-directed by building the logical rules into the non-logical rules, as required. This not only simplifies the following steps, but also leads to practically better algorithms. We then show how the principal typing property for our type system can be established by a generalization of Mycroft's method [12] we call *Kleene-Mycroft Iteration (KMI)*. The availability of disjunctive binding-time annotations is used to eliminate *neutral* binding-time variables; that is, binding-time variables occurring in C in a judgement $C, A \vdash \kappa$, but not in A or κ. This in turn establishes that the ascending chains constructed by KMI are (polynomially) bounded.

This method of establishing the principal typing property is constructive in the sense that it yields an algorithm for actually computing principal types. Whilst natural and easy to explain, its run-time is potentially prohibitive since it is, in the worst case, exponential in the *nesting depth* of fix-expressions. This is due to the fact that every fix-expression $\mathrm{fix}\,\mathrm{x}.\ \mathrm{e}$ induces an iterative loop in the inference algorithm, where the type of x is initialized to "bottom" every time the loop is executed. A single, simple, but crucial change to KMI, which we call *accelerated Kleene-Mycroft Iteration*, makes the algorithm polynomial, however: instead of initializing x to bottom every time we execute the (inner) loop corresponding to $\mathrm{fix}\,\mathrm{x}.\ \mathrm{e}$ we initialize it to the *result of the most recent execution of the loop*.

The resulting algorithm is simple to understand and implement, easy to prove correct, and executes in polynomial time in the size of the given (explicitly typed!) program expression. It still contains some redundancy in that it follows the syntactic structure of the subexpressions. Using sophisticated constraint-solving methods it is possible to develop an improved, practically fast inference algorithm. This is not described here, however.

3.1 Binding-time logic

Program expressions are assumed to be well-typed and given by their typing derivation. The *non-logical rules* defining the binding-time properties for the constructs of our programming language are given in Figure 1. The *logical* rules defining the fundamental properties of qualification, quantification and *coercing* the binding-time type of an expression are displayed in Figure 2. Together they constitute a *binding-time theory (logic)* for our programming language.[7]

$$C, A \vdash x : A(x)$$

$$\frac{C \vdash \kappa \text{ wft} \quad \mathrm{C, A, x : \kappa \vdash e : \kappa'}}{C, A \vdash \lambda x.\, e : \kappa \xrightarrow{S} \kappa'} \qquad \frac{C, A \vdash e : \kappa' \xrightarrow{b} \kappa'' \quad C, A \vdash e' : \kappa'}{C, A \vdash e@e' : \kappa''}$$

$$C, A \vdash \mathrm{true} : \mathrm{S} \qquad \mathrm{C, A \vdash false : S} \qquad \frac{\mathrm{C, A \vdash e : b} \quad \begin{array}{c} C, A \vdash e' : \kappa \\ C, A \vdash e'' : \kappa \end{array} \quad \mathrm{C \vdash b \leq_f} \kappa}{\mathrm{C, A \vdash if\ e\ then\ e'\ else\ e''} : \kappa}$$

$$\frac{C, A \vdash e : \sigma \quad C, A, x : \sigma \vdash e' : \sigma'}{C, A \vdash \mathrm{let\ x = e\ in\ e'} : \sigma'} \qquad \frac{C \vdash \sigma \text{ wft} \quad \mathrm{C, A, f} : \sigma \vdash \mathrm{e} : \sigma}{C, A \vdash \mathrm{fix\,f.\,e} : \sigma}$$

Fig. 1. Binding-time rules: non-logical part

The rules contain well-formedness requirements for binding-time types in sufficiently many places to guarantee that derivations only produce well-formed binding-time types:

Proposition 2. *Let C be a set of subtyping assumptions and A a set of assumptions such that $C \vdash A(x)$ wft for all x in the domain of A.*
Then $C, A \vdash e : \sigma$ implies $C \vdash \sigma$ wft.

Proof. By induction on the derivation of $C, A \vdash e : \sigma$.

[7] In contrast to other type based binding-time analyses the terms of our type system are not *annotated*. This is, however, only a cosmetic difference since annotations are uniquely determined by the type derivation.

$$\frac{C, \beta \leq \beta', C', A \vdash e : \rho}{C, C', A \vdash e : \beta \leq \beta' \Rightarrow \rho} \qquad \frac{C, A \vdash e : b \leq b' \Rightarrow \rho \quad C \vdash b \leq b'}{C, A \vdash e : \rho}$$

$$\frac{C, A \vdash e : \sigma}{C, A \vdash e : \forall \beta . \sigma} \quad (\beta \notin FV(C, A)) \qquad \frac{C, A \vdash e : \forall \beta . \sigma}{C, A \vdash e : \sigma[b/\beta]}$$

$$\frac{C, A \vdash e : \kappa \quad C \vdash \kappa' \text{ wft} \quad \mathrm{C} \vdash \kappa \leq_s \kappa'}{C, A \vdash e : \kappa'}$$

Fig. 2. Introduction and elimination rules for qualified types and type schemes

Note that this proposition "breaks" if we remove any one of the well-formedness requirements on binding-time types in the rules.

A judgement can be weakened by instantiating its free binding-time variables and/or strengthening its assumptions (whatever occurs to the left of the turnstile) and/or weakening its conclusion (to the right of the turnstile). The result is called a lazy instance of the original judgement.

Definition 3. (Lazy instance) A binding-time statement $C', A' \vdash e' : \kappa'$ is a *lazy instance* of $C, A \vdash e : \kappa$ if $e = e'$ and

1. $C' \vdash S(C)$,
2. $C', A' \vdash S(A)$, and
3. $C' \vdash S(\kappa) \leq_s \kappa'$.

If $C, A \vdash e : \kappa$ is also a lazy instance of $C', A' \vdash e' : \kappa'$ then the two binding-time judgements are *equivalent*.

Even though there is no explicit rule for weakening a judgement, any lazy instance is nonetheless derivable:

Proposition 4. *Derivable binding-time judgements are closed under the lazy instance relation; that is, if $C, A \vdash e : \kappa$ is derivable, then so is any lazy instance of it.*

Proof. By rule induction on $C, A \vdash e : \kappa$.

3.2 Normalized derivations

The logical rules in Figure 2 can be applied to any subexpression in a derivation of a binding-time type for an expression e. Thus the structure of a derivation for e is not determined by the syntactic structure of e itself.

It is possible to *normalize* any derivation of $C, A \vdash e : \kappa$, however, such that the coercion rule is only applied to

1. arguments in function calls,
2. the expressions in the then- and else-branches of a conditional, and

3. to the whole expression e.

Likewise, the other non-logical rules can be restricted to be applicable only in connection with certain syntactic constructs. Consequently we can build the applications of the logical rules into the non-logical rules. In this fashion we arrive at a strictly syntax-directed inference system, given by the rules in Figure 3. A *normalized* judgement $C, A \vdash_n e : \kappa$ is derived using only the rules of Figure 3 instead of those of Figures 1 and 2.

$$\frac{C \vdash C'[\vec{b}/\vec{\alpha}]}{C, A, x : \forall\vec{\alpha}.C' \Rightarrow \kappa \vdash_n x : \kappa[\vec{b}/\vec{\alpha}]}$$

$$\frac{C \vdash \kappa \text{ wft} \quad \mathrm{C, A, x} : \kappa \vdash \mathrm{e} : \kappa'}{C, A \vdash \lambda x.\ e : \kappa \xrightarrow{S} \kappa'} \qquad \frac{C, A \vdash_n e : \kappa' \xrightarrow{b} \kappa'' \quad C, A \vdash_n e' : \kappa \quad C \vdash \kappa \leq_s \kappa'}{C, A \vdash_n e@e' : \kappa''}$$

$$C, A \vdash \text{true} : \mathrm{S} \qquad \mathrm{C, A} \vdash \text{false} : \mathrm{S}$$

$$\frac{C, A \vdash_n e : b \quad \begin{array}{c} C, A \vdash_n e' : \kappa' \quad C \vdash \kappa' \leq_s \kappa \\ C, A \vdash_n e'' : \kappa'' \ C \vdash \kappa'' \leq_s \kappa \end{array} \quad C \vdash \kappa \text{ wft} \quad \mathrm{C} \vdash \mathrm{b} \leq_{\mathrm{f}} \kappa}{C, A \vdash_n \text{if e then e}' \text{ else e}'' : \kappa}$$

$$\frac{C, A \vdash_n e : \kappa \quad C', A, x : \forall\vec{\alpha}.C \Rightarrow \kappa \vdash_n e' : \kappa'}{C', A \vdash_n \text{let x} = \text{e in e}' : \kappa'} \quad (\vec{\alpha} \cap FV(A) = \emptyset)$$

$$\frac{C \vdash \kappa \text{ wft} \quad \mathrm{C} \vdash \kappa' \leq_s \kappa \quad \mathrm{C, A, f} : \forall\vec{\alpha}.\mathrm{C} \Rightarrow \kappa \vdash_{\mathrm{n}} \mathrm{e} : \kappa' \quad \mathrm{C}' \vdash_{\mathrm{n}} \mathrm{C}[\vec{\mathrm{b}}/\vec{\alpha}]}{C', A \vdash_n \text{fix f. e} : \kappa[\vec{\mathrm{b}}/\vec{\alpha}]} \quad (\vec{\alpha} \cap FV(A) = \emptyset)$$

Fig. 3. Normalized binding-time analysis rules

Normalized judgements are judgements constructed by certain derivations in the original inference system. Their main point is that they let us restrict our attention to derivations that have a certain form determined by the given program expression, without, however, restricting which judgements we can derive in the end (modulo a "final" subtyping step):

Theorem 5 Soundness and completeness. *The normalized binding-time inference system (Figure 3) is* sound *and* complete *with respect to the original binding-time inference system (Figures 1 and 2):*

Soundness. *If* $C, A \vdash_n e : \kappa$ *then* $C, A \vdash e : \kappa$.
Completeness. *If* $C, A \vdash e : \kappa$ *then there exists* κ' *such that* $C, A \vdash_n e : \kappa'$ *and* $C \vdash \kappa' \leq_s \kappa$.

Proof. **Soundness.** By simple inspection we can see that every rule in Figure 3 is derivable in the canonical system of Figure 1 together with Figure 2.

Completeness. We need to prove a stronger statement: if $C, A \vdash e : \forall\vec{\alpha}.C' \Rightarrow \kappa$ then, for all substitutions S such that $C \vdash S(C')$ there exists κ' with $C, A \vdash_n e : \kappa'$ and $C \vdash \kappa' \leq S(\kappa)$. This is done by rule induction on $C, A \vdash e : \kappa$.

3.3 Eliminating Binding-Time Variables in Assumptions

The subtyping assumptions C in $C, A \vdash_n e : \kappa$ may contain binding-time variables that occur (freely) neither in A nor in κ. These can always be eliminated: we can construct C' only containing binding-time variables occurring free in A or κ such that $C', A \vdash_n e : \kappa$ and $C, A \vdash_n e : \kappa$ are equivalent.

We can write a binding-time type uniquely as an *annotated* standard type:

$$b = \mathrm{bool}[\mathrm{b}]$$
$$\tau[\vec{b}^-, \vec{b}^+] \xrightarrow{b} \tau'[\vec{b}'^-, \vec{b}'^+] = (\tau \to \tau')[\vec{b}^+\vec{b}'^-, \vec{b}^-\vec{b}'^+ b]$$

We say the binding-time annotations in the vectors $\vec{b}^-$ and $\vec{b}^+$ occur in *negative*, respectively *positive* position in $\tau[\vec{b}^-, \vec{b}^+]$. In this fashion we can assign a polarity to every occurrence of a binding-time annotation in a binding-time type. Note that a particular binding-time annotation can have both positive and negative occurrences in a binding-time type.

Let $A = \{x_1 : \forall\vec{\beta_1}.C_1 \Rightarrow \kappa_1, \ldots, x_k : \forall\vec{\beta_k}.C_k \Rightarrow \kappa_k\}$. A binding-time annotation occurring freely in A or κ occurs positively (negatively) in $C, A \vdash e : \kappa$ if it occurs positively (negatively) in $\kappa_1 \xrightarrow{S} \ldots \xrightarrow{S} \kappa_k \xrightarrow{S} \kappa$. So, if a binding-time annotation occurs negatively in some binding-time type $A(x)$ then it occurs *positively* in any judgement with assumptions A. Note that binding-time variables occurring in C may or may not occur in A or κ. Those that do, will occur positively, negatively, or both. Those that do not, have *no* polarity: they are *neutral*. Non-negative binding-time variables can be eliminated as expressed in the following lemma.

Lemma 6. *Let β be a binding-time variable with no negative occurrence in $C, A \vdash e : \kappa$.*

Let $\beta_1 \leq_a \beta, \ldots, \beta_k \leq_a \beta$ be all the containments in C with β on the right-hand side. We delete these and replace every containment $\beta \leq_a \gamma$ in C by $\beta_1 \leq_a \gamma, \ldots, \beta_k \leq_a \gamma$ and call the resulting containments C'. Let $S = \{\beta \mapsto \bigvee_{i=1}^{k} \beta_i\}$.

Then $C, A \vdash e : \kappa$ and $C', S(A) \vdash e : S(\kappa)$ are equivalent, i.e. lazy instances of each other.

Restricting application of Lemma 6 to neutral binding-time variables — those occurring only in C — we can see that the substitution S is the identity on A and κ, resulting in the following theorem.

Theorem 7. *For every binding-time judgement $C, A \vdash e : \kappa$ there is an equivalent $C', A \vdash e : \kappa$ where C' contains only binding-time variables occurring freely in A or κ.*

3.4 Polymorphic Recursion Revisited: Kleene-Mycroft Iteration

Consider an ML-polymorphic type system [11, 5, 4], extended with a rule for *polymorphic recursion* [12]:

$$\frac{A, f : \sigma \vdash e : \sigma}{A \vdash \mathrm{fix\,f.\,e} : \sigma}$$

where σ can be a *type scheme (polymorphic type)*; that is, a type of the form $\forall\vec{\alpha}.\tau$, where τ is simple type.[8]

Mycroft has shown that this strengthened type system has the principal typing property: every expression e typable under assumptions A has a principal type $\sigma = \forall\vec{\alpha}.\tau$ such that any type $\sigma' = \forall\vec{\beta}.\tau'$ is a *generic instance* of σ; that is, $\vec{\beta} \cap FV(\sigma) = \emptyset$ and $\tau' = S(\tau)$ for some substitution S with domain contained in $\vec{\alpha}$. His proof also yields a "natural" algorithm for computing the principal type.

The algorithm and its correctness are corollaries of an elegant general argument couched in terms of notions of domain theory. Let us recapitulate Mycroft's argument since it is very general and since it is at the heart of our adaptation to polymorphic binding-time analysis, which combines polymorphic recursion with qualified types.

Mycroft shows that the set of α-equivalence classes of type schemes $\forall\vec{\alpha}.\tau$, together with an artificial top element $\top$ forms a cpo (complete partial order) under the generic instance ordering $\sqsubseteq$, where, with the exception of $\top$, all elements of the cpo are *compact*.

Let $\mathrm{fix\,f.\,e}$ be a fix-expression. For now, let us assume that it is closed and contains no other fix-expressions. Mycroft defines a function $F_{A,e}$ on α-equivalence classes of type schemes, by

$$F_{A,e}(\sigma) = \sigma' \Leftrightarrow A, f : \sigma \vdash e : \sigma'$$

where σ' is the *principal type* of e under $A, f : \sigma$. If $\sigma = \top$ or if e has no type under $A, f : \sigma$, then $F_{A,e}(\sigma) = \top$. He shows that $F_{A,e}$ is well-defined and continuous.

From these general properties, all desired results follow: the principal typing property as well as a natural algorithm for computing principal types of fix-expressions. By standard fixed-point theory for cpo's we know that $F_{A,e}$ has a least fixed point σ_p. If $\sigma_p = \top$ then $\mathrm{fix\,f.\,e}$ is not typable. If $\sigma_p \neq \top$, then:

1. σ_p is a type of $\mathrm{fix\,f.\,e}$ since $A, f : \sigma_p \vdash e : \sigma_p$; this follows from the fact that σ_p is a fixed point;
2. σ_p is a *principal* type of $\mathrm{fix\,f.\,e}$; this follows from the fact that any other type of $\mathrm{fix\,f.\,e}$ is a fixed-point of $F_{A,e}$, and σ_p is the least fixed point with respect to the generic instance ordering $\sqsubseteq$;
3. σ_p can be computed by constructing a *Kleene sequence*; that is, there exists $k < \omega$ such that $\sigma_p = F^k_{A,e}(\bot)$; this follows from the fact that σ_p is compact.

[8] This typing calculus has also been termed *Milner-Mycroft Calculus* [6].

In summary, if fix f. e has a type under assumptions A, then it has a *principal* type that can be computed by iterating $F_{A,e}$ on $\bot = \forall\alpha.\alpha$ until we obtain a fixed point; that is until $F^k_{A,e}(\bot) = F^{k+1}_{A,e}(\bot)$. Recall that equality here means α-equivalence. We call the computation of $F^k_{A,e}(\bot)$ *Kleene-Mycroft Iteration (KMI)*.

3.5 Principality: Kleene-Mycroft Iteration for Qualified Type Schemes

We now generalize Kleene-Mycroft Iteration to our setting with subtype qualified type schemes.

The generic instance preordering on qualified type schemes is defined by $\sigma \sqsubseteq \sigma' \Leftrightarrow x : \sigma \vdash x : \sigma'$. We call $\sigma_0, \sigma_1, \ldots, \sigma_i, \ldots$ a *Kleene-Mycroft sequence* for fix f. e if:

1. $\sigma_0 \sqsubseteq \sigma_1 \sqsubseteq \ldots \sqsubseteq \sigma_i \sqsubseteq \ldots$,
2. $f : \sigma_i \vdash e : \sigma_{i+1}$, and
3. $\sigma_i \sqsubseteq \sigma$ for all i and any type σ of fix f. e

We call the sequence *finite with limit* σ_p if $\sigma_p = \sigma_{i+1} \sqsubseteq \sigma_i$ for some i.

Proposition 8. *If* fix f. e *has a finite Kleene-Mycroft sequence with limit* σ_p *then* σ_p *is a principal type for* fix f. e.

For simplicity's sake, let as assume that e does not contain any fix-expressions and has only one free variable, x. How can we compute a principal type of fix x. e (if it even has one)?

Note that for every type τ there is a well-formed binding-time type $\bot_\tau$ that is least at that type: $\bot_\tau = \forall\vec{\alpha}\vec{\beta}.C(\tau[\vec{\alpha},\vec{\beta}]) \Rightarrow \tau[\vec{\alpha},\vec{\beta}]$. Letting $\sigma_0 = \bot_\tau$ we can construct a Kleene-Mycroft sequence for fix f. e by computing $f : \sigma_i \vdash e : \sigma_{i+1}$, where σ_{i+1} is a principal type for e under assumption $f : \sigma_i$. This yields a principal type for fix f. e if

1. e has a principal type under assumption $f : \sigma_i$, for all i;
2. the Kleene-Mycroft sequence $\sigma_0 \sqsubseteq \sigma_1 \sqsubseteq \ldots \sigma_i \sqsubseteq \ldots$ is finite.

The first property is easy to establish and follows from work on polymorphic subtype inference without polymorphic recursion (recall that we have assumed that e does not contain fix) [9, 8, 15, 1].

For the second property we exploit Theorem 7: we can choose σ_{i+1} to be *minimal*; that is, such that its subtyping constraints only contain variables that also occur in the underlying simple binding-time type. In doing so, we can bound the size of the subtyping constraints C_i in σ_i as a function of the size of the underlying simple binding-time type κ_i. Let n be the size of τ. Then $|\kappa_i| = O(n)$, and $|C_i| = O(n^2)$. Since $\sigma_i \sqsubseteq \sigma_{i+1}$, it must be that the Kleene-Mycroft sequence we construct for fix f. e is finite and of length $O(n^2)$; that is, there is $k = O(n^2)$ such that $\sigma_{k+1} \sqsubseteq \sigma_k$. This argument shows that the partial order induced by the

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{W}(A, e) = \text{case } e \text{ of}} \\
\quad x: & \text{if A(x)} = \forall\vec{\alpha}.C \Rightarrow \kappa \\
 & \text{then } (C[\vec{\beta}/\vec{\alpha}], \kappa[\vec{\beta}/\vec{\alpha}]) \\
 & \text{else } \mathit{fail} \\
\quad \lambda x.\ e': & \text{let } \kappa \text{ be new linear binding-time type} \\
 & \text{let } (C, \kappa') = \mathcal{W}((A, x : \kappa), e') \\
 & \text{in } (C \cup \mathit{constraints}(\kappa \text{ wft}), \kappa \xrightarrow{\text{S}} \kappa') \\
\quad e'@e'': & \text{let } (C', \kappa \xrightarrow{b} \kappa') = \mathcal{W}(A, e') \\
 & \text{let } (C'', \kappa'') = \mathcal{W}(A, e'') \\
 & \text{in } (C' \cup C'' \cup \mathit{constraints}(\kappa'' \leq_s \kappa), \kappa') \\
\quad \text{true(false)}: & (\emptyset, S) \\
\quad \text{if e}' \text{ then e}'' \text{ else e}''': & \text{let } (C', b) = \mathcal{W}(A, e') \\
 & \text{let } (C'', \kappa'') = \mathcal{W}(A, e'') \\
 & \text{let } (C''', \kappa''') = \mathcal{W}(A, e''') \\
 & \text{let } \kappa \text{ be new linear binding-time type} \\
 & \text{in } (C' \cup C'' \cup C''' \cup \mathit{constraints}(\kappa \text{ wft}) \\
 & \quad \cup\, \mathit{constraints}(\kappa'' \leq_s \kappa) \cup \mathit{constraints}(\kappa''' \leq_s \kappa) \\
 & \quad \cup\, \mathit{constraints}(b \leq_f \kappa), \kappa) \\
\quad \text{let x} = \text{e}' \text{ in e}'': & \text{let } (C', \kappa') = \mathcal{W}(A, e') \\
 & \text{let } \sigma' = \mathit{close}_A(C' \Rightarrow \kappa') \\
 & \text{in } \mathcal{W}(A, x : \sigma', e'') \\
\quad \text{fix x} : \tau.\ \text{e}': & \text{let } \sigma_0 = \bot_\tau \\
 & \text{let } A_0 = A, x : \sigma_0 \\
 & \text{repeat for } i \geq 0 \\
 & \qquad \text{let } (C_{i+1}, \kappa_{i+1}) = \mathcal{W}(A_i, e') \\
 & \qquad \text{let } C'_{i+1} = \mathit{elim}(FV(A) \cup FV(\kappa_{i+1}), C_{i+1}) \\
 & \qquad \text{let } \sigma_{i+1} = \mathit{close}_A(C_{i+1} \Rightarrow \kappa_{i+1}) \\
 & \qquad \text{let } A_{i+1} = A, x : \sigma_{i+1} \\
 & \text{until } \sigma_{i+1} \sqsubseteq \sigma_i \\
 & \text{in } (C_{i+1}, \sigma_{i+1})
\end{array}
$$

Fig. 4. Algorithm $\mathcal{W}$

preordering $\sqsubseteq$ has a finite ascending chain condition, which, in turn, shows that fix x. e has a finite Kleene-Mycroft sequence and thus a principal polymorphic type.

This argument also works for fix-expressions with free variables and nested fix-expressions: Figure 4 contains an adaptation of Algorithm $\mathcal{W}$ to our binding-time analysis with qualified type schemes. It is syntax-directed and operates on open expressions, with arbitrary nesting depth of fix-expressions. Algorithm $\mathcal{W}$ was originally introduced by Milner [11] for ML type inference and later extended by Mycroft [12] to polymorphic recursion. We use the following auxiliary partial functions in it:

- *constraints*(κ wft) denotes the smallest set of subtyping assumptions C *induced* by κ wft; that is, the smallest set C such that $C \vdash \kappa$ wft and whenever $C' \vdash \kappa$ wft then $C' \vdash C$. Similarly, *constraints*($\kappa \leq_s \kappa'$) and *constraints*($b \leq_f \kappa$) denotes the smallest set C that guarantees $C \vdash \kappa \leq_s \kappa'$ *resp.* $C \vdash b \leq_f \kappa$.
- $close_A(\rho) = \forall\vec{\alpha}.\rho$ where $\vec{\alpha}$ is a sequence of all the binding-time variables free in ρ, but not in A.
- $elim(V, C) = C'$ if C' is obtained from C by eliminating all the binding-time variables $\beta \notin V$ according to Lemma 6.

A binding-time type is *linear* if all its binding-time annotations are pairwise distinct binding-time variables.

Lemma 9. *If* $(C, \kappa) = \mathcal{W}(A, e)$ *then* $C, A \vdash_n e : \kappa$, *and any derivable* $C', A \vdash_n e : \kappa'$ *is a lazy instance of* $C, A \vdash_n e : \kappa$.

This lemma is halfway towards demonstrating that every expression has a principal type. To complete the proof we must show that $\mathcal{W}(A, e)$ always terminates without failure. This is not obvious and indeed not generally the case since *constraints*() is a partial function. We have to make sure that free program variables in e are bound to *reasonable* binding-time types in A. A binding-time type $\forall\vec{\alpha}.C \Rightarrow \kappa$ is reasonable if C is a set of subtyping constraints between binding-time variables only (that is, C must be a set of subtyping assumptions), and binding-time annotations occurring negatively in κ do not contain disjunctions.

Lemma 10. *If A maps every free program variable in e to a reasonable binding-time type, then $\mathcal{W}(A, e)$ terminates without failure.*

Theorem 11. *Let A map all program variables free in e to reasonable binding-time types. Then e has a principal binding-time type under A.*

A final observation: the finite ascending chain condition for qualified type schemes guarantees that we could also apply the rather simple strategy of skipping the variable elimination step

$$\text{let } C'_{i+1} = elim(FV(A) \cup FV(\kappa_{i+1}), C_{i+1})$$

in Figure 4. This is inadvisable, however, if Algorithm $\mathcal{W}$ is to be actually used in an implementation: The subtyping assumptions have a tendency of multiplying rapidly since they are "pumped" up at each iteration step. This is because a principal type from one iteration is used as assumption in the following iteration, with copies of constraints thus being generated at *all* subsequent iterations since they enter into the principal type of the following iteration.

Example 2. We are now able to compute a principal type for the function presented in the introduction:

$$\text{fix f. } \lambda\text{x. } \lambda\text{y. if x} = 0 \text{ then } 2 \text{ else } (\text{f@y})\text{@}(\text{x} - 1)$$

We start with

$$\sigma_0 = \bot_{\mathrm{Int}\to\mathrm{Int}\to\mathrm{Int}} = \forall\beta_1, \beta_2, \beta_x, \beta_y, \beta.\beta_x \xrightarrow{\beta_1} \beta_y \xrightarrow{\beta_2} \beta$$

and compute our first approximation to be

$$\sigma_1 = \forall\beta_x, \beta_y, \beta.\beta_x \leq \beta \Rightarrow \beta_x \xrightarrow{S} \beta_y \xrightarrow{S} \beta$$

The next aproximation will be:

$$\sigma_2 = \forall\beta_x, \beta_y, \beta.\beta_x \leq \beta, \beta_y \leq \beta \Rightarrow \beta_x \xrightarrow{S} \beta_y \xrightarrow{S} \beta$$

which is also a fixed-point. As expected, the principal type indicates that the function depends on *both* of its arguments.

4 A polynomial-time algorithm: Accelerated Kleene-Mycroft Iteration

The total number of iterations in Algorithm $\mathcal{W}$ is, at worst, exponential in the *nesting depth* of fix-expressions since every fix-expression $\mathrm{fix\,x.\,e}$ corresponds to a while-loop in which the type associated with x is initialized to $\bot_\tau$ for *each* execution of the loop. Upon closer inspection we can see that Kleene-Mycroft Iteration contains a lot of *repeated* computation steps. This iterated repetition can, in the worst case, lead to — and is solely accountable for — exponential iteration behavior. A simple, but critical change to Kleene-Mycroft Iteration reduces the worst-case run-time complexity from exponential to polynomial in the size of the underlying explicitly typed program. That change is based on the following property for continuous functions. We write $f_x(y)$ for the unary function that results from fixing the first argument of a binary function f.

Lemma 12. *Let $f : D \times E \to E$ be a continuous function on the product cpo $D \times E$. Let $d \sqsubseteq d' \in D$. Let $g(d) = \bigsqcup_{i\in\omega} f_d^i(\bot)$. Then $g(d') = \bigsqcup_{i\in\omega} f_{d'}^i(\bot) = \bigsqcup_{i\in\omega} f_{d'}^i(g(d))$.*

This lemma lets us recompute a fixed point *incrementally* by starting the iteration at a *previous fixed point* — in the lemma above this is $g(d)$ — instead of starting the Kleene sequence all the way from the bottom element. We can thus *accelerate* Algorithm $\mathcal{W}$ by replacing the initialization

$$\text{let } A_0 = A, x : \sigma_0$$

of A_0 in the code for fix-expressions by

$$\text{let } A_0 = A, x : \text{most recent binding-time type computed for } e = \mathrm{fix\,x.\,e'}$$

This presupposes that we store the binding-time type computed for $\mathrm{fix\,x.\,e}$ with $\mathrm{fix\,x.\,e}$ itself. At the beginning this value is set to $\bot_\tau$. We call the resulting algorithm Algorithm $\mathcal{W}$ with *accelerated Kleene-Mycroft Iteration.*

Theorem 13. *Let m be the size of the typing derivation (in the standard simple type system of the language) for e.*

Algorithm $\mathcal{W}$ with accelerated Kleene-Mycroft Iteration computes a principal type of e in time polynomial in m for any assumptions that map the free variables of e to reasonable binding times.

Proof. (Sketch) It is sufficient to show that the cumulative cost of executing the loops corresponding to fix-expressions is polynomially bounded.

Recall that chains of binding times at a (standard) type of size n are at most of length $O(n^2)$. Since computation for fix x. e restarts at the value that was produced during the most recent execution of the corresponding loop, the *total* number of steps executed by *all* executions of the loop together is bounded by $O(n^2)$.

Let us say that, every time the line

$$\text{let } \sigma_{i+1} = close_A(C_{i+1} \Rightarrow \kappa_{i+1})$$

in Algorithm W is executed there is a *time click*. Note that the test $\sigma_{i+1} \sqsubseteq \sigma_i$ can be implemented in time polynomial in $|\sigma_{i+1}| = |\sigma_i|$, since σ_i and σ_{i+1} are *minimal*. It can be checked to see that there goes no more than time polynomial in m in between two clicks.

Finally, the total number of clicks is bounded by $O(m^2)$ since the clicks associated with a particular individual fix-expression with standard type of size n is bounded by $O(n^2)$.

Thus we obtain an upper bound that is the product of two polynomials in m, which is still a polynomial.

Tofte uses, with good practical performance, the "trick" of restarting at the last computed result of a loop [16] in his inference algorithm for region-based storage management [17]. In the area of transformational programming Cai and Paige describe efficient incremental recomputation of fixed points; our Lemma 12 is a variation on theirs [2]. The "restarting" at the value of a previously computed fixed point does not seem to be widely known, however, even though it has, if applicable, tremendous impact on the theoretical and practical performance of algorithms with nested iteration structures. It is, for example, also applicable to Mycroft's original algorithm.

5 Related Work

This work is based on [7] where a binding-time analysis polymorphic in let and fix was presented and proven correct with respect to standard and specialisation semantics. An important property of this and the analysis in this paper is that the result is invariant under definition unfolding, in the sense that the result cannot be improved by inlining definitions.

Contemporary with this work Consel, Jouvelot and Ørbæk developed a type- and effect-based binding-time analysis similar in spirit to ours [3]. They annotate standard types with binding-time *effects* and allow variables to range over these. Their effect-annotated types correspond more or less directly to our binding-time annotated types. This allows them to analyse modules independently.

Consel *et al.* do not, however, introduce actual polymorphism over binding-time effects, though they state that let-polymorphism can be achieved by considering let $x = e$ in e' as syntactic sugar for $[e/x]e'$. This corresponds to an aggressive form of cloning rather than actual polymorphism, however. An algorithm is sketched but no complexity estimate is given.

6 Conclusion and Further Work

We have presented an algorithm for a realistic program analysis that combines subtyping, polymorphism and polymorphic recursion. The algorithm is an adaption of Mycroft's algorithm for computing principal types for ML extended with polymorphic recursion [12]. We show that, in contrast to polymorphic recursion over standard types (as considered by Mycroft), polymorphic recursion over binding times can be done in polynomial time in the size of the underlying statically typed program. This solves the open problem of devising an algorithm for the polymorphic binding-time analysis presented in [7], although we have slightly changed the presentation of the type system by including lubs.

It may sound surprising at first to hear that a generalization of polymorphic recursion should have an efficient algorithm whereas typability with polymorphic recursion by itself is undecidable. The reason for this is that we have a program *analysis* problem for a program that has already been type checked. The analysis follows the structure of the — given — typing derivation, which it uses as a fixed "skeleton". In this sense type-based program analysis is different from the problem of type checking/inference; that is, deciding whether a certain syntactic expression has a typing derivation in the first place. Intuitively, in the latter problem there is no fixed structure to stick to, and, as is the case for ML typing with polymorphic recursion, types may blow up in size without a priori bound or structure.

We believe that the general development of Sections 3 and 4 transcends the immediate context of the particular program analysis we have studied. The algorithm in Section 4 can be further improved by (rather complicated) constraint solving techniques. This needs to be further and carefully investigated, especially since we hope to arrive at a practically very efficient algorithm as the basis for an implementation effort that is to put the results of this paper to work.

Let us add a final note on the interrelation of type checking (is this syntactic construct a type correct expression?) and program analysis (which relevant properties does this [statically type correct] expression have?). Note that the "polymorphism in the analysis" is not tied to the polymorphism in the type system that defines the static semantics of the language of discourse: the programming language in this paper is simply typed, yet its binding-time logic has powerful polymorphism and subtyping — though harnessed by the "underlying" standard types. From these remarks it should also be clear that having a very powerful underlying type system neither necessitates a powerful program analysis logic, nor does it entail that the analysis must be algorithmically as hard as the type checking/inference problem for the standard type system. The point is that there is a *phase distinction* between type checking and program analysis. The program analysis operates conceptually not on an unchecked expression, but on (a particular type derivation for) a type checked expression. This distinction is sometimes blurred because type checking and type-based analysis use the same repertoire of conceptual, theoretical and practical tools, and for practical purposes type checking and program analysis may practically be merged into a single computational phase.

References

1. A. Aiken, E. L. Wimmers, and T. Lakshman. Soft typing with conditional types. In *Proc. 21st Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Portland, Oregon*. ACM, ACM Press, Jan. 1994.
2. J. Cai and R. Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11:197–261, 1989.
3. C. Consel, P. Jouvelot, and P. Ørbæk. Seperate polyvariant binding time reconstruction. Technical Report CRI Report A/261, Ecole des Mines, Oct. 1994.
4. L. Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1984. Technical Report CST-33-85 (1985).
5. L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, pages 207–212, Jan. 1982.
6. F. Henglein. Type inference and semi-unification. In *Proc. ACM Conf. on LISP and Functional Programming (LFP), Snowbird, Utah*, pages 184–197. ACM Press, July 1988.
7. F. Henglein and C. Mossin. Polymorphic binding-time analysis. In D. Sanella, editor, *5th European Symposium on Programming*, LNCS 788, pages 287–301. Springer-Verlag, 1994.
8. M. Jones. A theory of qualified types. In *Proc. 4th European Symposium on Programming (ESOP), Rennes, France*, volume 582 of *Lecture Notes in Computer Science*, pages 287–306. Springer-Verlag, Feb. 1992.
9. S. Kaes. Type inference in the presence of overloading, subtyping, and recursive types. In *Proc. ACM Conf. on LISP and Functional Programming (LFP), San Francisco, California*, pages 193–204. ACM Press, June 1992.
10. A. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. Technical Report BUCS-89-010, Boston University, Oct. 1989. also in Proc. of Symp. on Theory of Computing, Baltimore, Maryland, May 1990.
11. R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375, 1978.
12. A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proc. 6th Int. Conf. on Programming, LNCS 167*, 1984.
13. H. Nielson and F. Nielson. Automatic binding time analysis for a typed lambda calculus. *Science of Computer Programming*, 10:139–176, 1988.

14. M. Rittri. Dimension inference under polymorphic recursion. In *Proc. Conf. on Functional Programming Languages and Computer Architecture (FPCA), San Diego, California*. ACM Press, June 1995.
15. G. S. Smith. Polymorphic type inference with overloading and subtyping. In *Proc. Theory and Practice of Software Development (TAPSOFT), Orsay, France*, volume 668 of *Lecture Notes in Computer Science*, pages 671–685. Springer-Verlag, April 1993.
16. M. Tofte. Personal communication, June 1995.
17. M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proc. 21st Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Portland, Oregon (this proceedings)*. ACM, ACM Press, Jan. 1994.