# Dynamic Typing*

Fritz Henglein
University of Copenhagen
Universitetsparken 1
2100 Copenhagen Ø
Denmark
Internet: henglein@diku.dk

December 18, 1991

**Abstract**

We present an extension of a statically typed language with a special type Dynamic and explicit type tagging and checking operations (coercions). Programs in run-time typed languages are viewed as incomplete programs that are to be *completed* to well-typed programs by explicitly inserting coercions into them.

Such completions are generally not unique. If the meaning of an incomplete program is to be the meaning of *any* of its completions and if it is too be unambiguous it is necessary that all its completions are *coherent* (semantically equivalent). We characterize with an equational theory the properties a semantics must satisfy to be coherent.

Since "naive" coercion evaluation does not satisfy all of the coherence equations we exclude certain "unsafe" completions from consideration that can cause avoidable type errors at run-time.

Various *classes* of completions may be used, parameterized by whether or not coercions may only occur at data creation and data use points in a program and whether only primitive coercions or also induced coercions. For each of these classes any term has a *minimal completion* that is optimal in the sense that it contains no coercions that could be avoided by a another coercion in the same class. In particular, minimal completions contain no coercions at all whenever the program is statically typable.

If only primitive type operations are admitted we show that minimal completions can be computed in almost-linear time. If induced coercions are also allowed the minimal completion can be computed in time $O(nm)$ where $n$ is the size of the program and $m$ is the size of the value flow graph of the program, which may be of size $O(n^2)$, but is typically rather sparse.

Finally, we sketch how this explicit dynamic typing discipline can be extended to let-polymorphism by parameterization with respect to coercions.

The resulting language framework leads to a seamless integration of statically typed and dynamically typed languages by relying on type inference for programs that have no type information and no explicit coercions whatsoever.

## 1 Introduction

We present an extension of the (statically) typed $\lambda$-calculus with a special type Dynamic and *explicit coercions* representing run-time tagged values and associated tagging and checking opera-

*In Proc. 4th European Symposium on Programming (ESOP), Feb. '92, Rennes, France, Springer Lecture Notes in Computer Science, Vol. 582, pp. 233-253. This research has been supported by Esprit BRA 3124, Semantique.

tions as they are found in run-time typed (dynamically typed with implicit coercions) languages. A program in a run-time typed language can be embedded into this language without relying on a fixed translation, but instead permitting all possible *completions* of the program with inserted explicit coercions such that the typing rules are satisfied.

Since there are generally many different completions for the same run-time typed program we characterize *coherence* of completions by an equational theory that includes the equality $c^{-1}; c = \iota$ where $c$ is a tagging operation, $c^{-1}$ its corresponding checking operation, $\iota$ denotes the identity ("no-op") coercion, and ";" denotes left-to-right sequential composition. This equality does not hold for "naive" coercion evaluation as the left-hand side may produce a type error (in some context) where the right-hand side does not. Thus we define and restrict ourselves to a class of *safe* completions, all of which are equivalent under naive coercion evaluation.

Making coercions explicit enables reasoning about them in an implementation-independent fashion and bringing efficiency concerns to bear. We prove that certain classes of completions have *minimal* completions that avoid as many coercions as possible within the type system. In particular, a minimal completion of a statically typable program contains guaranteed no coercions, unlike the *canonical completion* used by (unoptimized) implementations of run-time typed languages.

We give efficient algorithms for computing minimal completions. For completions that use only primitive coercions we present an algorithm that computes a minimal completion in almost-linear time, $O(n\alpha(n, n))$, where $\alpha$ is an inverse of Ackermann's function. For completions that may also use induced coercions there is an algorithm that executes in time $O(nm)$ using the fastest known dynamic transitive closure algorithm under edge additions. Here $n$ is the size of the input program and $m$ is the size of its value flow graph; in the worst case the value flow graph is dense, i.e. $m$ is $O(n^2)$, but for well-designed programs it is typically sparse.[1]

Finally, we discuss an extension to a let-polymorphic type discipline, in which let-bound variables can be parameterized by coercions.

The resulting language framework, which we refer to simply as *dynamic typing*, leads to a seamless integration of statically typed and run-time typed languages by connecting implicitly and explicitly dynamically typed programs by automatic type inference (completion). Both static and dynamic language programmers profit from such integration. The static language programmer has a universal interface type for communicating with the environment and may choose to use operations that require run-time checking. The dynamic language programmer has a way of expressing type properties that can be checked *statically* instead of dynamically; i.e., once instead of repeatedly. More importantly, abstract data types can be integrated into a dynamically typed language in a modular and representation-independent fashion. In principle they do not even have to be implemented in the same language. The type system together with the coercions make sure that no undetected representation-dependent effects slip through.

An immediate application of the minimal completion algorithms is in tag optimization of run-time typed languages such as Scheme, Common LISP or SETL. The completion algorithms extended to let-polymorphism may also be applicable in ML-like languages for improved type error identification and recovery since they keep track of the creation and use points of values.

[1] This algorithm is not presented for space reasons.

## 2 Dynamically typed lambda calculus

In this section we introduce the *dynamically typed λ-calculus (dynamic λ-calculus)*. It is an extension of the (statically) typed λ-calculus with a distinguished type constant Dynamic and special embedding and projection functions we call *coercions*.

We can think of elements of type Dynamic as "(type) tagged" values; that is, as tag-value pairs where the tag indicates the type of the value component. Coercions represent a special class of functions that embed values into the "universal" type Dynamic and project them back from Dynamic. In general, for every type constructor $TC$ of arity $k$ there is an *embedding* that maps elements of type $TC(\mathsf{Dynamic}, \ldots, \mathsf{Dynamic})$ to Dynamic by pairing them with their type. For example, the coercion Func maps a function $f$ of type Dynamic → Dynamic to Dynamic. Note that since $f$ is required to have domain and codomain type Dynamic it is sufficient to tag $f$ with the type *constructor*, →, alone as all the arguments to → are required to be Dynamic. This is in contrast to the dynamic typing disciplines described in [Myc84,ACPP91,LM91] where values of *any* type may be tagged with their type *expression*.

For every embedding $c$ for type constructor $TC$ there is a corresponding *projection*, denoted by $c^{-1}$, that maps elements of type Dynamic to $TC(\mathsf{Dynamic}, \ldots, \mathsf{Dynamic})$: it checks whether its argument has the tag $TC$; if so, it strips the tag and returns the untagged value; if not, it generates a (run-time) type error. It is possible to include a general typecase form (see [ACPP91]) in the language; this way projections can be *defined* instead of added as language primitives; e.g.,

$$\begin{array}{l}\textbf{typecase } e \textbf{ of}\\ \quad ([\mathsf{Func}]f)\ f\\ \quad \textbf{else } \epsilon_{\mathsf{Dynamic}\to\mathsf{Dynamic}}\\ \textbf{end}\end{array}$$

is the definition of $\mathsf{Func}^{-1}$, where $\epsilon_{\mathsf{Dynamic}\to\mathsf{Dynamic}}$ represents a run-time type error. Indeed, general type dispatching can be described, which is not "directly" possible with projections alone; e.g.,

$$\begin{array}{l}\textbf{typecase } e \textbf{ of}\\ \quad ([\mathsf{Func}]f)\ \textbf{typecase } (f([\mathsf{Bool}]\mathit{true}))\ \textbf{of}\\ \qquad\qquad ([\mathsf{Bool}]b)\ b\\ \qquad\qquad \textbf{else } \epsilon_{\mathsf{Bool}}\\ \qquad \textbf{end}\\ \quad ([\mathsf{Bool}]b)\ b\\ \textbf{end}.\end{array}$$

In this paper we are primarily interested in automatically *inferring* embeddings and projections in programs that use them implicitly, as is the case in run-time typed languages where they correspond to tagging and check-and-untag operations (see Section 3). Consequently we omit the general typecase form for the present purposes.

Coercion type signatures are expressions of the form $\tau \rightsquigarrow \tau'$, where $\tau, \tau'$ are type expressions (see below). Taking embeddings and projections and the identity function, $\iota$, as *primitive coercions* we can build a calculus of coercions as follows:

- coercions can be (functionally) composed to form new coercions as long as their types match up;

- for every type constructor $TC$ of $k > 0$ arguments there is a *coercion constructor* that takes $k$ coercions, one for each argument position, as inputs and combines them to a new coercion.

For example, if $c_1 : \tau_1 \rightsquigarrow \tau_1', c_2 : \tau_2 \rightsquigarrow \tau_2'$ are coercions then $c_1 \to c_2 : (\tau_1' \to \tau_2) \rightsquigarrow (\tau_1 \to \tau_2')$ is an *induced* coercion that operates on functions $f$ of type $\tau_1' \to \tau_2$. It returns a function of type $\tau_1 \to \tau_2'$, which is the composition of (in diagrammatical order) coercion $c_1$, function $f$ and finally coercion $c_2$. Using $\beta$- and $\eta$-equality it is possible to *define* the coercion constructor $\to$ by $c \to c' = \lambda y.\lambda x.[c'](y([c]x))$.[2]

Coercions defined only from embeddings and the identity (no projections) are *positive coercions*; those defined only from projections and the identity (no embeddings) are *negative coercions*. A language with *only* positive coercions corresponds to a "coercion formulation" of a subtyping discipline; c.f. [Tha88,BCGS89,CG90]. If negative coercions are added to a subtyping theory without explicit coercions (e.g.,[FM88]) in a naive fashion this leads to a complete collapse of the type hierarchy — every type is equal to Dynamic. In this sense the presence of negative coercions makes dynamic typing fundamentally different from subtyping.

The pure dynamically typed $\lambda$-calculus with only the type constructor $\to$ is operationally uninteresting since no type errors can occur. In this case the coercions have no operational significance and may be ignored during execution. For this purpose we use as a vehicle for our investigations the dynamically typed $\lambda$-calculus with an additional primitive type, the Booleans. The type expressions in this language are generated by the production

$$\tau ::= \alpha \mid \mathsf{Bool} \mid \tau' \to \tau'' \mid \mathsf{Dynamic}$$

The typing rules for the dynamic $\lambda$-calculus with Booleans are given in Figure 1 in natural deduction style. Throughout this paper we use the following notational conventions: $e, e', \ldots$ denote (dynamically typed or untyped) $\lambda$-terms; $c, c', d, d', \ldots$ denote coercions; $\tau, \tau', \ldots$ denote type expressions; and $\alpha, \beta, \ldots$ are type variables. Introduction of a typing assumption for a variable $x$ hides all other assumptions for $x$ until it is discharged. If $e : \tau$ is derivable from a set of typing assumptions $A$ we write $A \vdash e : \tau$. We say $e$ is a *dynamically typed $\lambda$-term* if $A \vdash e : \tau$ for some typing assumptions $A$ and type expression $\tau$.

The rule (COERCE) is the only rule with which additional typings beyond those of the simply typed $\lambda$-calculus can be inferred. Note that on the one hand coercions cannot *directly* be passed as arguments to or returned from functions. On the other hand, every coercion $c : \tau \rightsquigarrow \tau'$ can be represented by the function and "first-class" value $\lambda x.[c]x : \tau \to \tau'$. Since not *all* ($\lambda$-definable) functions are coercions, however, we keep coercions strictly separately from arbitrary functions.

Induced coercions give us the effect of tagging with full type expressions since for every type $\tau$ there is a coercion $c : \tau \rightsquigarrow \mathsf{Dynamic}$; e.g., $[\mathsf{Bool}^{-1} \to \mathsf{Bool}; \mathsf{Func}](\lambda x : \mathsf{Bool}.\mathbf{if}\ x \mathbf{then}\ \mathit{false} \mathbf{else}\ \mathit{true}$ has type Dynamic. Yet a general typecase form with matching on full type expressions is counter to our desire to treat this dynamically typed $\lambda$-term as equivalent to

$$[\mathsf{Func}](\lambda x : \mathsf{Dynamic}.\mathbf{if}\ [\mathsf{Bool}^{-1}]x\ \mathbf{then}\ [\mathsf{Bool}]\mathit{false}\ \mathbf{else}\ [\mathsf{Bool}]\mathit{true}$$

(see [Tha90] and Section 4 of this paper).

We shall not give a semantics of the dynamically typed $\lambda$-calculus, but leave this question deliberately open at this point. In Section 3 we will treat an *untyped* $\lambda$-term as an *incomplete*

[2] We treat $c \to c'$ separately since we do not rely on $\beta$- and/or *eta*-equality. See Section 4.

$$\text{(ABSTR)}\quad \frac{[x : \tau'] \quad e : \tau}{\lambda x.e : \tau' \to \tau} \qquad \text{(CONST)}\quad \mathit{true}, \mathit{false} : \mathsf{Bool}$$

$$\text{(APPL)}\quad \frac{e : \tau' \to \tau \quad e' : \tau'}{ee' : \tau} \qquad \text{(IF)}\quad \frac{e : \mathsf{Bool} \quad e' : \tau \quad e'' : \tau}{\textbf{if } e \textbf{ then } e' \textbf{ else } e'' : \tau}$$

$$\text{(}\to\text{-EMBED)}\quad \mathsf{Func} : (\mathsf{Dynamic} \to \mathsf{Dynamic}) \rightsquigarrow \mathsf{Dynamic}$$

$$\text{(}\to\text{-PROJ)}\quad \mathsf{Func}^{-1} : \mathsf{Dynamic} \rightsquigarrow (\mathsf{Dynamic} \to \mathsf{Dynamic})$$

$$\text{(}\to\text{-CONSTR)}\quad \frac{c_1 : \tau_1 \rightsquigarrow \tau_1' \quad c_2 : \tau_2 \rightsquigarrow \tau_2'}{c_1 \to c_2 : (\tau_1' \to \tau_2) \rightsquigarrow (\tau_1 \to \tau_2')}$$

$$\text{(BOOL-EMBED)}\quad \mathsf{Bool} : \mathsf{Bool} \rightsquigarrow \mathsf{Dynamic}$$

$$\text{(BOOL-PROJ)}\quad \mathsf{Bool}^{-1} : \mathsf{Dynamic} \rightsquigarrow \mathsf{Bool}$$

$$\text{(NOP)}\quad \iota : \tau \rightsquigarrow \tau$$

$$\text{(COMP)}\quad \frac{c_1 : \tau \rightsquigarrow \tau' \quad c_2 : \tau' \rightsquigarrow \tau''}{c_1; c_2 : \tau \rightsquigarrow \tau''}$$

$$\text{(COERCE)}\quad \frac{e : \tau \quad c : \tau \rightsquigarrow \tau'}{[c]e : \tau'}$$

Figure 1: Typing rules for the dynamically typed $\lambda$-calculus with Booleans

program into which explicit coercions must be *inserted* to form a dynamically typed $\lambda$-term. In Section 4 (resp. 5) we *characterize* the semantic properties that a semantics of the dynamically typed $\lambda$-calculus must satisfy if all possible (resp. safe) *completions* of an untyped $\lambda$-term are to be *coherent*, i.e., denote the same value.

# 3 Completions

In the implementation of programming languages with implicit dynamic type checking, type handling operations are in effect "inserted" into the source code in a canonical fashion: Every variable is assigned type Dynamic; at every program point where a value is *created* (e.g., by a constant or a $\lambda$-abstraction) the corresponding tagging operation (embedding) is inserted; and at every program point where a value is *used* (e.g., by the test in a conditional or by a function application), the appropriate check-and-untag operation (projection) is inserted. In this fashion the resulting "completed" program satisfies the typing rules of Section 2.

The main disadvantage of this scheme is that dynamic type operations are *always* used, even in cases where they could be omitted; in particular, statically well-typed programs are also annotated with type operations, which typically results in slower execution compared to execution without any type operations.[3]

We view a program with implicit run-time checking as an *incompletely typed* program; that is, a program from which coercions (and type declarations of variables) have been omitted. It is the task of the type inferencer to *complete* this program by inserting explicit coercions such that the typing rules are satisfied. This extends the role of conventional type inferencers in that not only type information but also identity and placement of coercions in the source program are inferred.

Formally, the *untyped $\lambda$-terms (with Booleans)* are generated by the production

$$e ::= x \mid \lambda x.e' \mid e'e'' \mid \mathit{true} \mid \mathit{false} \mid \textbf{if } e' \textbf{ then } e'' \textbf{ else } e'''.$$

The *erasure* of a dynamically typed $\lambda$-term $e$ is the untyped $\lambda$-term that arises from "erasing" all coercions from $e$ (including the square brackets, of course). Conversely, a *completion* of an untyped $\lambda$-term $e$ is a dynamically typed $\lambda$-term whose erasure is $e$. Since there is generally more than one completion for the same incomplete program we treat the resulting ambiguity as a problem of *coherence* [BCGS89,CG90] (see Section 4) or *safety* (c.f. [Tha90]; see Section 5) of the semantics of the completions.

A completion models the process of making coercions explicit that are implicit, but nonetheless present, in run-time typed languages. The process of making them explicit opens the opportunity for source-level compile-time optimization.

Note that the "local" translation of untyped $\lambda$-terms to dynamically typed $\lambda$-terms described at the beginning of this section is a completion in this sense; we shall call it the *canonical completion* of an untyped $\lambda$-term. Intuitively, it *maximizes* the use of embeddings and projections.

We illustrate this translation for the familiar fixpoint combinator $Y$ of Church. The $Y$-combinator is defined by

$$Y = \lambda f.(\lambda x.f(xx))(\lambda y.f(yy)).$$

Its canonical translation into the dynamically typed $\lambda$-calculus is

$$Y_c \;=\; [\mathsf{Func}]\lambda f : \mathsf{Dynamic}.$$

[3] We use as a fundamental assumption that operations on untagged data are generally more efficient than the corresponding operations on tagged data, which also have to perform tagging and checking.

$$[\mathsf{Func}^{-1}][\mathsf{Func}](\lambda x : \mathsf{Dynamic}.[\mathsf{Func}^{-1}]f([\mathsf{Func}^{-1}]xx))$$
$$[\mathsf{Func}](\lambda y : \mathsf{Dynamic}.[\mathsf{Func}^{-1}]f([\mathsf{Func}^{-1}]yy)).$$

The canonical translation generates a dynamically typed $\lambda$-term for *every* untyped $\lambda$-term. Thus we have the following proposition.

**Proposition 1** *Every untyped $\lambda$-term has at least one completion.*

Another possible completion for the Y-combinator that actually minimizes the use of coercions is

$$\begin{aligned} Y_m \quad = \quad & \lambda f : \mathsf{Dynamic} \to \mathsf{Dynamic}. \\ & (\lambda x : \mathsf{Dynamic} \to \mathsf{Dynamic}.f(x[\mathsf{Func}]x)) \\ & (\lambda y : \mathsf{Dynamic}.f([\mathsf{Func}^{-1}]yy)), \end{aligned}$$

which is of type $(\mathsf{Dynamic} \to \mathsf{Dynamic}) \to \mathsf{Dynamic}$. $Y_m$ looks, in an intuitive sense, more "efficient" than $Y_c$ because fewer type operations have to be executed during its evaluation.

# 4 Coherence

Completions induce a congruence relation on dynamically typed $\lambda$-terms and coercions: $e' \cong e''$ if $A \vdash e' : \tau$ and $A \vdash e'' : \tau$ for some set of typing assumptions $A$ and type expression $\tau$, and $e', e''$ have the same erasure; $c' \cong c''$ if $x : \tau \vdash [c']x : \tau'$ and $x : \tau \vdash [c'']x : \tau'$ for some $\tau, \tau'$. If any two such congruent $\lambda$-terms, respectively coercions, are semantically equivalent, we can define the meaning of an untyped $\lambda$-term as the meaning of *any arbitrary one* of its completions. This opens the door to *intensional* considerations: finding operationally efficient completions by taking the *global* program structure into account. This is addressed in Section 6. In this section we characterize the properties a semantics of the dynamically typed $\lambda$-calculus must satisfy to be coherent (yield the same meaning) for *all* completions of any untyped $\lambda$-term.

Consider the equational theory given in Figure 2 over dynamically typed $\lambda$-terms and (well-formed) coercions.

**Theorem 1** *(Coherence of completions)*
*The equational axioms and rules of Figure 2 together with the* additional *rule $c^{-1}; c = \iota$ (for every embedding $c$) is an axiomatization of completion congruence; that is, for all dynamically typed $\lambda$-terms $e', e''$ we have $e' \cong e''$ if and only if $e' = e''$ is derivable with the standard equational axioms and inference rules (reflexivity, symmetry, transitivity, congruence under arbitrary contexts). Furthermore, the axiom system is irredundant; i.e., no rule or axiom can be derived from the others.*

We need a lemma for the proof that guarantees that coercions are congruent exactly when they have the same type signature. The *coercion equalities* are the axioms and rules in Figure 2 in which no $\lambda$-terms occur, together with the additional equation $c^{-1}; c$.

**Lemma 2** *(Equality of coercions)*
*Let $c : \tau \rightsquigarrow \sigma, c' : \tau' \rightsquigarrow \sigma'$ be arbitrary coercions. Then $c = c'$ is derivable from the coercion equalities if and only if $\tau = \tau'$ and $\sigma = \sigma'$.*

$$
\begin{aligned}
(c;c');c'' &= c;(c';c'') && (1)\\
c;\iota &= c && (2)\\
\iota;c &= c && (3)\\
c;c^{-1} &= \iota && (4)
\end{aligned}
$$

$$
\begin{aligned}
\iota \to \iota &= \iota && (5)\\
(c \to c');(d \to d') &= (d;c) \to (c';d') && (6)\\
[\iota]e &= e && (7)\\
[c'][c]e &= [c;c']e && (8)\\
[c \to d]\lambda x.e &= \lambda x.[d](e\{x \mapsto [c]x\}) && (9)\\
([c \to d]e)e' &= [d](e([c]e')) && (10)
\end{aligned}
$$

$$
[c]\textbf{if } e \textbf{ then } e' \textbf{ else } e'' = \textbf{if } e \textbf{ then } [c]e' \textbf{ else } [c]e'' \quad (11)
$$

Figure 2: Conversions for dynamically typed $\lambda$-terms with Boolean truth values

It is easy to see that for every coercion $c : \tau \rightsquigarrow \tau'$, primitive or induced, there is an "inverse" coercion $c' : \tau' \rightsquigarrow \tau$ such that $c;c' = \iota$ and $c';c = \iota$. We reserve the notation $c^{-1}$ for projections corresponding to (primitive) embeddings, however, as we have no need for a general inverse operation on coercions.

**Proof:** (Proof of theorem) Let $e' = e''$ be derivable. By inspection of the axioms and rules it can be verified that $e'$ and $e''$ have the same erasure. Similarly it can be checked that $A \vdash e' : \tau$ if and only if $A \vdash e'' : \tau$. It follows that they are congruent completions; i.e., $e' \cong e''$.

For the converse, we call a dynamically typed $\lambda$-term *head coercion free* (c.f. [CG90]) if it is *not* of the form $[c]e'$. W.l.o.g. we may assume that coercions are only applied to head coercion free $\lambda$-terms and every head coercion free subterm has exactly one coercion applied to it. This follows from $[c_k]\ldots[c_1]e = [c_1;\ldots;c_k]e$ for $k \geq 2$ and $e = [\iota]e$. We prove $e' \cong e'' \Rightarrow e' = e''$ by induction on the erasure $e$ of $e'$ and $e''$.

(Basis, I) If $e = x$ then $e' = [c']x, e'' = [c'']x$. This implies $c, c' : A(x) \rightsquigarrow \tau$ and thus $c = c'$ by Lemma 2.

(Basis, II) If $e = \mathit{true}$ or $e = \mathit{false}$ then similar as above.

(Inductive step, I) If $e = \lambda x.f$ then $e' = [c']\lambda x : \tau'.f'$ and $e'' = [c'']\lambda x : \tau''.f''$. Since there is a coercion from any type to any other type there are coercions $d : \tau'' \leq \tau', d' : \upsilon' \leq \upsilon''$ such that $[d \to d']\lambda x : \tau'.f' = \lambda x : \tau''.[d']f'\{x \mapsto [d]x\}$. That is, we have for $A\{x : \tau''\}$ the completions $[d']f'\{x \mapsto [d]x\}$ and $f''$ of type $\upsilon''$, and by inductive hypothesis, $[d']f'\{x \mapsto [d]x\} = f''$. Consequently we have $[c''][d \to d']\lambda x : \tau'.f' = [c'']\lambda x : \tau''.f''$. Since $(d \to d';c''), c' : \tau' \leq \tau''$ we know $(d \to d';c'') = c'$ and the result follows.

(Inductive step, II) If $e = fg$, then $e' = [c'](f'g')$ and $e'' = [c''](f''g'')$ where $f' : \sigma' \to \upsilon', f'' : \sigma'' \to \upsilon''$. There are coercions $d : \sigma'' \to \sigma', d^{-1} : \sigma' \leq \sigma'', d' : \upsilon' \to \upsilon''$. We have $[d \to d']f' = f''$ and $[d^{-1}]g' = g''$ by induction hypothesis, and thus $[c'']([d \to d']f'[d^{-1}]g') = [c'']([f''g'')$. We get $[c''][d'](f'g') = [c''](f''g'')$. Because of uniqueness of coercions it follows that $(d';c'') = c'$ and the

result follows.

It is easy to construct for every axiom and rule a pair of congruent completions such that they cannot be proved congruent without it.

This shows that, *independent of $\beta-$ and $\eta$-equality, all* congruent completions of an untyped $\lambda$-term have the same behavior if and only if their meanings satisfy the equations in Figure 2.

# 5 Safety

In the characterization of coherence of completions (Theorem 1) we have used the equality $c^{-1}; c = \iota$. Accordingly, we have the equality $[\mathsf{Bool}; \mathsf{Func}^{-1}; \mathsf{Func}; \mathsf{Bool}^{-1}]\mathit{true} = \mathit{true}$ since $\mathsf{Bool}; \mathsf{Func}^{-1}; \mathsf{Func}; \mathsf{Bool}^{-1} = \mathsf{Bool}; \iota; \mathsf{Bool}^{-1} = \mathsf{Bool}; \mathsf{Bool}^{-1} = \iota$ and $[\iota]\mathit{true} = \mathit{true}$. With naive evaluation of coercions, however, this equality does *not* hold: $[\mathsf{Bool}; \mathsf{Func}^{-1}; \mathsf{Func}; \mathsf{Bool}^{-1}]\mathit{true}$ is evaluated by first applying the tagging operation $\mathsf{Bool}$ to *true*, then the check-and-untag operation $\mathsf{Func}^{-1}$ and finally $\mathsf{Func}$ and $\mathsf{Bool}^{-1}$. Since the tag of the value after applying $\mathsf{Bool}$ is "$\mathsf{Bool}$", however, the second operation, $\mathsf{Func}^{-1}$ generates a type error. In contrast, evaluating *true* by itself yields no type error. So $c^{-1}; c = \iota$ does *not* hold with naive evaluation of coercions.

In view of Theorem 1 we have three possibilities to address this problem:

1. Allow arbitrary completions, retain naive evaluation of coercions, but give up on coherence of completions.

2. Allow arbitrary completions and devise a different evaluation strategy for coercions to retain coherence.

3. Retain naive evaluation of coercions, but restrict the class of admissible completions to retain coherence.

Since we envisage the process of completing an untyped program to be automatic, Option 1 is least attractive since it puts the task of deciding the *meaning* of a program into the hands of the completion process, over which a programmer has no control.[4]

We can accomplish Option 2 if coercions are not evaluated until a value is used (as a function in an application or in the test of a conditional). In this way every type operation just adds itself as a tag (even check-and-untag operations!) to a value and at the point of use the resulting sequence of tags is *simplified* by rewriting until an untagged value of the correct type is reached or a type error is generated (see [Tha90]). This form of "simplificational" coercion evaluation has two disadvantages: it is inefficient since it requires complex, long-living tagging and symbolic rewriting, and it gives delayed error messages.

Since naive coercion evaluation is more conventional, generally more efficient, and reports type errors earlier we adopt Option 3. Notice that with naive coercion evaluation $C[c^{-1}; c]$ generates a type error or yields the same value as $C[\iota]$ for *any* context $C$; never a different (proper) value. We replace the equalities of the form $c^{-1}; c = \iota$ by *inequalities* $\iota \sqsubseteq c^{-1}; c$ for all embeddings $c$ and extend them to other coercions, $d \sqsubseteq d'$, and dynamically typed $\lambda$-terms, $e' \sqsubseteq e''$, by combining them with the equalities of Figure 2 and closing them under reflexivity, transitivity and arbitrary context. Here an equality $e' = e''$ is interpreted as the inequalities $e' \sqsubseteq e''$ and $e'' \sqsubseteq e'$. An inequality $e' \sqsubseteq e''$ expresses that, in any context, if $e'$ generates a

[4]This is a fundamental difference from the dynamic typing disciplines of [ACPP89] and [LM91] since in those type systems the programmer is expected to control coercions completely.

type error then so does $e''$ in the same context. These inequalities are a syntactic analogue to Thatte's semantic "wrongness" relation in a fixed denotational interpretation [Tha90].

We say that a completion $e'$ of $e$ is *safe* if for every congruent completion $e''$ we have $e' \sqsubseteq e''$. Intuitively, this guarantees that $e'$ generates as few type errors as possible at run-time; i.e., it does not generate *avoidable* type errors. More importantly, it can be shown that for safe completions naive and simplificational coercion evaluation behave equivalently. So by restricting ourselves to safe completions we can reap the benefits of combining the efficiency and simplicity of naive coercion evaluation with unambiguous semantics and still retain a great degree of freedom of choosing amongst different *safe* completions.

Analogous to the proof of Theorem 1 we can show that every untyped $\lambda$-term has a safe completion. In fact, the canonical completion is safe.

**Proposition 3** *(Safety of canonical completions)*
*Every untyped $\lambda$-term has at least one safe completion.*

Note that for two congruent safe completions $e', e''$ of an untyped $\lambda$-term $e$ we can derive $e' = e''$ from the equational axiom system in Figure 2 alone, *without* the equation $c^{-1}; c = \iota$.

# 6 Minimal completions

As we have seen, the canonical completion of an untyped $\lambda$-term is safe. But it is also inefficient. In this section we define a general syntactic criterion for discussing which completion is operationally "better" than another. This criterion is robust in the sense that no particular concrete operational semantics, implementation technology, *etc.*, is assumed, but only that execution of a tagging operation (embedding) and then its corresponding check-and-untag operation (projection) is less efficient than executing nothing at all. For various classes of safe completions we report efficient algorithms for computing *minimal* (optimal) completions w.r.t. to that syntactic criterion. In particular, we describe a theoretically and practically very efficient algorithm for a class of completions that has possible applications in the optimization of run-time typed languages such as Scheme, Common LISP, SETL and others.

Consider the coercion equality $(c; c^{-1}) = \iota$ (Figure 2). With naive coercion evaluation the left-hand side and the right-hand side are equivalent since first tagging a value and then untagging it again has the same effect as doing nothing at all to the value. Clearly, however, literally *executing* the left-hand side is wasteful and unnecessary. Based on this observation we define a preorder on *safe* completions by replacing the equality $c; c^{-1} = \iota$ with the inequality

$$\iota \leq c; c^{-1}.$$

We extend $\leq$ to arbitrary coercions and dynamically typed $\lambda$-terms by adding the *remaining* equalities of Figure 2 (*without* the equality $(c; c^{-1}) = \iota$, of course) and closing it under reflexivity, transitivity and arbitrary contexts ($e = e'$ is interpreted as $e \leq e'$ and $e' \leq e$).

Intuitively, if we have $e \leq e'$ then $e$ and $e'$ are observably equivalent (i.e., $e = e'$ can be proved from the equational theory of Figure 2), but $e$ has no more coercions than $e'$. This expresses itself by $e'$ having *syntactically* fewer coercions than $e$, but it also executes fewer coercions at run-time for any reasonable operational semantics.

A completion $e'$ is *minimal* in a class of safe congruent completions $C$ if it is in $C$ and for every $e''$ in $C$ we have $e' \leq e''$. In this sense a minimal completion is an operationally optimal completion in a class w.r.t. to ordering $\leq$. Note, however, that minimal completions need not

be unique as there may be distinct safe congruent completions $e', e''$ such that both $e' \leq e''$ and $e'' \leq e'$. For any untyped $\lambda$-term $e$, type assumptions $A$ and type expression $\tau$ we define four different classes of safe completions $e'$ of $e$ such that $A \vdash e' : \tau$: completions that use only primitive coercions (embeddings and projections) and place them at data creation and data use points only; completions that use only primitive coercions (and place them anywhere); completions that use arbitrary coercions, but place them at data creation and data use points only; and arbitrary completions (using arbitrary coercions placed anywhere). We denote these four completion classes by $C_{pf}^{A,\tau}(e), C_{p*}^{A,\tau}(e), C_{*f}^{A,\tau}, C_{**}^{A,\tau}(e)$, respectively.

Henceforth let $A, e, \tau$ be fixed, but arbitrary. We shall simply write $C_{pf}$, $C_{p*}$, $C_{*f}$ and $C_{**}$, respectively, for the four classes above. Let $C$ be any one of these.

**Theorem 2** *$C$ has a minimal completion.*

Let $e$ be of size $n$. We denote by $m$ the size of the *value flow graph* of $e$. This is essentially the higher-order extension of the call graph of a program; its construction is also called closure analysis [Ses89]. In the worst case $m$ is $O(n^2)$, but for well-designed programs the value flow graph is typically sparse, i.e., $m = O(n)$.

**Theorem 3** *A minimal completion of $C$ can be computed in the complexity given in the following chart.*[5]

| *completions* | *only primitive coercions* | *arbitrary coercions* |
|---|---|---|
| *only at fixed places* | $O(n\alpha(n,n))$ | $O(nm)$ |
| *at arbitrary places* | $O(n\alpha(n,n))$ | $O(nm)$ |

These results follow from the constraint system characterization and normalization that is the heart of our (minimal) completion algorithms. The algorithms are variants on two basic algorithms, one for completions with only primitive coercions, the other for completions with arbitrary coercions. The first of these two can be viewed as an instrumented unification closure algorithm with some additional postprocessing and has been used for efficient binding-time analysis [Hen91]. At the core of the second algorithm is an efficient dynamic transitive closure algorithm (e.g., La Poutré and van Leeuwen [LPvL87] and Yellin [Yel88]) for computing value flow graphs (closure analysis); it has been used for the efficient solution of a specialized semi-unification problem [Hen90]. We only describe the minimal completion algorithm for $C_{pf}$; that is, for completions that use only primitive coercions, which are placed at creation and use points only. We restrict ourselves to closed $\lambda$-terms. We shall not present algorithms or proofs for the other cases as this would substantially lengthen this paper.

Our type inference algorithm for $C_{pf}$ consists of the following steps

1. For given $\lambda$-term $e$ construct a type constraint system $C$;
2. normalize $C$ to $C'$ with respect to a set of constraint transformation rules;
3. construct a "minimal" solution from $C'$;
4. translate the minimal solution into a (minimal) completion of $e$.

The advantage of "distilling" the essence of type inference into constraint systems is that it frees the type inference problem from the syntactic structure of programs and permits solution strategies that are *not* strictly syntax-directed.

[5] $\alpha$ is an inverse of Ackermann's function, which may be considered a small constant for all practical purposes [Tar83].

For a $\lambda$-term $e$ associate a type variable $\alpha_x$ with every $\lambda$-bound and free variable $x$ (w.l.o.g. we assume they are distinct) and with every subterm occurrence $e'$ of $e$ associate a type variable $\alpha_{e'}$ with $e'$. Define $C(e)$ as follows.

1. If $e = \lambda x.e'$ then $C(e) = \{\alpha_x \to \alpha_{e'} \stackrel{?}{\leq} \alpha_e\} \cup C(e')$;
2. if $e = e'@e''$ then $C(e) = \{\alpha_{e''} \to \alpha_e \stackrel{?}{\leq} \alpha_{e'}\} \cup C(e') \cup C(e'')$;
3. if $e = \textbf{if } e' \textbf{ then } e'' \textbf{ else } e'''$ then $C(e) = \{\mathsf{Bool} \stackrel{?}{\leq} \alpha_{e'}, \alpha_{e''} = \alpha_{e'''}, \alpha_e = \alpha''_e\} \cup C(e') \cup C(e'') \cup C(e''')$;
4. if $e = c$ then $C(e) = \{\mathsf{Bool} \stackrel{?}{\leq} \alpha_e\}$ where $c = \mathit{true}$ or $c = \mathit{false}$;
5. if $e = x$ ($x$ variable) then $C(e) = \{\alpha_e \stackrel{?}{=} \alpha_x\}$.

Figure 3: Extracting typing constraints

### 6.1 Basic constraint system extraction

A type constraint system is a multiset of constraints of the following forms.

$$\begin{array}{rcl} \alpha \to \alpha' & \stackrel{?}{\leq} & \beta \\ \mathsf{Bool} & \stackrel{?}{\leq} & \beta \\ \alpha & \stackrel{?}{=} & \alpha' \end{array}$$

where $\alpha, \alpha', \beta$ denote type variables or the type constant $\mathsf{Dynamic}$. A *solution* of a constraint system $C$ is a substitution $S$ of type expressions for type variables such that

- for $\alpha \to \alpha' \stackrel{?}{\leq} \beta$ in $C$ there exists a primitive coercion[6] $c$ such that $c : S(\alpha) \to S(\alpha') \leq S(\beta)$;
- for $\mathsf{Bool} \stackrel{?}{\leq} \beta$ in $C$ there exists a primitive coercion $c$ such that $c : \mathsf{Bool} \leq S(\beta)$;
- for $\alpha \stackrel{?}{=} \alpha'$ we have $S(\alpha) = S(\alpha')$; and
- $S(\alpha) = \alpha$ for all $\alpha$ *not* occurring in $C$.

Note that in general we can develop our treatment of constraints over an arbitrary term algebra representing type expressions. This is useful in a realistic setting where we have a multitude of predefined basic types and type constructors (and possibly also user-defined types and type constructors).

First we define the constraint "extraction" function $C$. It is given in Figure 3. The solutions of $C(e)$ and the completions of $e$ are in a very close relation, as expressed in the following theorem.

**Theorem 4** *(Soundness and completeness of constraint characterization)*
*The completions of $e$ and the solutions of $C(e)$ are in a one-to-one correspondence.*

[6]Recall that $\iota$ is a primitive coercion.

**Proof:** Analogous to Theorem 1 in [Hen91].

Let us write $e_1 = \lambda x.f(xx)$ and $e_2 = \lambda y.f(yy)$. Church's fixed point combinator is $Y = \lambda f.e_1e_2$. Then the constraints $C(Y)$ for $Y$ are:

$$\begin{array}{rcl}
\alpha_x \to \alpha_{xx} & \stackrel{?}{\leq} & \alpha_x \\
\alpha_{xx} \to \alpha_{f(xx)} & \stackrel{?}{\leq} & \alpha_f \\
\alpha_x \to \alpha_{f(xx)} & \stackrel{?}{\leq} & \alpha_{e_1} \\
\alpha_y \to \alpha_{yy} & \stackrel{?}{\leq} & \alpha_y \\
\alpha_{yy} \to \alpha_{f(yy)} & \stackrel{?}{\leq} & \alpha_f \\
\alpha_y \to \alpha_{f(yy)} & \stackrel{?}{\leq} & \alpha_{e_2} \\
\alpha_{e_2} \to \alpha_{e_1e_2} & \stackrel{?}{\leq} & \alpha_{e_1} \\
\alpha_f \to \alpha_{e_1e_2} & \stackrel{?}{\leq} & \alpha_Y .
\end{array}$$

### 6.2 Constraint system normalization

Constraint system normalization transforms $C(e)$ into an equivalent constraint system $C'(e)$ over the same class of constraints. It preserves the set of solutions, but it generates normal forms that make it easy to construct concrete solutions since it eliminates *all* constraints that *cannot* be solved equationally.

Let $G(C)$ be the graph defined on type variables occurring in $C$ such that there is a directed edge from $\alpha$ to $\beta$ whenever there is a constraint $\alpha \to \alpha' \stackrel{?}{\leq} \beta$ or $\alpha' \to \alpha \stackrel{?}{\leq} \beta$ in $C$. We say $C$ is *cyclic* if $G(C)$ is; and *acyclic* otherwise. The transformation rules for normalizing $C(e)$ are given in Figure 4.

The normalization of $C(e)$ results in a substitution $S$ and a normal form constraint system $C'(e)$ with the following properties:

1. For all inequality constraints $\ldots \stackrel{?}{\leq} \alpha$ in $C'(e)$ the right-hand side, $\alpha$, is a (type) variable.
2. There is at most one constraint of the form $\ldots \stackrel{?}{\leq} \alpha$ in $C'(e)$ for every $\alpha$.
3. $C$ is acyclic.
4. No constraints of the form $\alpha \stackrel{?}{=} \alpha'$ are in $C'(e)$.

**Theorem 5** *(Correctness of constraint normalization)*
*If $C \stackrel{S}{\Rightarrow} C'$ and $\mathcal{U}$ is the set of solutions of $C'$ then $\{U \circ S | U \in \mathcal{U}\}$ is the set of solutions of $C$.*

Normalization of $C(Y)$ results in the substitution

$$S_Y = \{\alpha_x, \alpha_{xx}, \alpha_{f(xx)}, \alpha_y, \alpha_{yy}, \alpha_{f(yy)}, \alpha_{e_2}, \alpha_{e_1e_2} \mapsto \mathsf{Dynamic}\}$$

Let $C$ be a constraint system with constraints of the form

- $\alpha \to \alpha' \overset{?}{\leq} \alpha''$,
- $\mathsf{Bool} \overset{?}{\leq} \alpha$, and
- $\alpha \overset{?}{=} \alpha'$

where $\alpha, \alpha', \alpha'' \in V \cup \{\mathsf{Dynamic}\}$. The transformation rules are:

1. (inequality constraint rules)
   (a) $C \cup \{\alpha \to \alpha' \overset{?}{\leq} \gamma, \beta \to \beta' \overset{?}{\leq} \gamma\} \Rightarrow C \cup \{\alpha \to \alpha' \overset{?}{\leq} \gamma, \alpha \overset{?}{=} \beta, \alpha' \overset{?}{=} \beta'\}$;
   (b) $C \cup \{\alpha \to \alpha' \overset{?}{\leq} \gamma, \mathsf{Bool} \overset{?}{\leq} \gamma\} \Rightarrow C \cup \{\alpha \to \alpha' \overset{?}{\leq} \gamma, \mathsf{Bool} \overset{?}{\leq} \gamma, \gamma \overset{?}{=} \mathsf{Dynamic}\}$;
   (c) $C \cup \{\alpha \to \alpha' \overset{?}{\leq} \mathsf{Dynamic}\} \Rightarrow C \cup \{\alpha \overset{?}{=} \mathsf{Dynamic}, \alpha' \overset{?}{=} \mathsf{Dynamic}\}$;
   (d) $C \cup \{\mathsf{Bool} \overset{?}{\leq} \mathsf{Dynamic}\} \Rightarrow C$;
2. (equational constraint rules)
   (a) $C \cup \{\mathsf{Dynamic} \overset{?}{=} \alpha\} \Rightarrow C \cup \{\alpha \overset{?}{=} \mathsf{Dynamic}\}$ if $\alpha$ is a type variable;
   (b) $C \cup \{\mathsf{Dynamic} \overset{?}{=} \mathsf{Dynamic}\} \Rightarrow C$;
   (c) $C \cup \{\alpha \overset{?}{=} \alpha'\} \overset{S}{\Rightarrow} S(C)$ if $\alpha$ is a type variable and $S = \{\alpha \mapsto \alpha'\}$;
3. (occurs check rule)
   (a) $C \Rightarrow C \cup \{\alpha \overset{?}{=} \mathsf{Dynamic}\}$ if $C$ is cyclic and $\alpha$ is on a cycle in $G(C)$.

Figure 4: Normalizing constraints

and the normalized constraint system $C'(Y)$ containing

$$\begin{aligned} \mathsf{Dynamic} \to \mathsf{Dynamic} &\stackrel{?}{\leq} \alpha_f \\ \mathsf{Dynamic} \to \mathsf{Dynamic} &\stackrel{?}{\leq} \alpha_{e_1} \\ \alpha_f \to \mathsf{Dynamic} &\stackrel{?}{\leq} \alpha_Y \end{aligned}$$

### 6.3 Solution construction

We can construct a "canonical" solution from normal form constraint system $C'(e)$ by simply unifying all inequalities. The properties of a normal form constraint system guarantee that all the constraints in $C'(e)$ can be satisfied *equationally*. We shall call this solution the *minimal solution* of $C'(e)$ and, by extension, of $C(e)$. Because of Theorem 4 this solution corresponds to a unique completion $e'$ of $e$ that has some type $\tau$. It can be shown by induction on $e$ that for any other completion $e''$ in $C_{pf}$ of type $\tau$ there exists a coercion $c : \tau \rightsquigarrow \tau$ such that $e'' = [c]e'$ is derivable in the equational system of Figure 2 *without* using equality $c; c^{-1} = \iota$. Since $\iota \leq c$ this implies $e' \leq e''$, which shows that $e'$ is minimal in $C_{pf}$.

**Theorem 6** *(Complexity of computing minimal completions)*
*The minimal completion of an untyped $\lambda$-term of size $n$ can be computed in time $O(n\alpha(n,n))$ and space $O(n)$ where $\alpha$ is an inverse of Ackermann's function [Tar83].*

**Proof:** (Sketch) It is easily seen that constraint extraction and minimal solution construction construction can be implemented in linear time.[7] In [Hen91] an amortization argument is given that shows that constraint normalization is implementable in time $O(n\alpha(n,n))$ (and space $O(n)$) using the union/find data structure with ranked union and path compression [GI91].

The minimal solution of $C'(Y)$ is the substitution

$$U_Y = \{\alpha_f, \alpha_{e_1} \mapsto \mathsf{Dynamic} \to \mathsf{Dynamic}, \alpha_Y \mapsto (\mathsf{Dynamic} \to \mathsf{Dynamic}) \to \mathsf{Dynamic}\}$$

The resulting minimal completion in $C_{pf}(Y)$ is

$$\begin{aligned} Y_{mf} \quad = \quad & \lambda f : \mathsf{Dynamic} \to \mathsf{Dynamic}. \\ & (\lambda x : \mathsf{Dynamic}. f([\mathsf{Func}^{-1}]xx)) \\ & [\mathsf{Func}](\lambda y : \mathsf{Dynamic}. f([\mathsf{Func}^{-1}]yy)). \end{aligned}$$

Comparing it to the completion $Y_m$ in Section 3 note that $Y_m < Y_{mf}$; i.e., $Y_m \leq Y_{mf}$, but not $Y_{mf} \leq Y_m$. So $Y_m$ is "better" than $Y_{mf}$, but $Y_m$ is *not* in $C_{pf}(Y)$, but $Y_m$ is the minimal completion in $C_{p*}(Y)$.

## 7 Polymorphism

In this section we sketch an extension of dynamic typing to a type discipline with let-polymorphism [Mil78]. A thorough and satisfactory treatment of polymorphic dynamic typing will have to be deferred to future work.

[7] We assume type expressions may be represented with sharing to avoid the well-known exponential blow-up of string representations of the solutions of unification problems.

A minimal completion of $e$ for one context $C$ cannot generally be extended to a safe completion for $C'[e]$ where $C'$ is another context for $e$. Consider the following example from [Tha90], in which we assume we have also lists and integers in our language: $e = \lambda x.cons\ 1\, x$. Its minimal completion in the context $C[] = car([](cons\, 1\ nil))+1$ is $\lambda x.cons\, 1\, x : \mathsf{List}(integer) \to \mathsf{List}(integer)$. However, in the context $C'[] = [](cons\ true\, nil)$ its minimal completion is $\lambda x.cons\ ([integer]1)\, x : \mathsf{List}\ (\mathsf{Dynamic}) \to \mathsf{List}\ (\mathsf{Dynamic})$. Note that the first completion of $e$ in the context $C$ cannot be extended to a *safe* completion for $C'[e]$[8], and that there is *no* completion of $e$ with the apparent polymorphic generalization $\forall\alpha.\mathsf{List}\ (\alpha) \to \mathsf{List}\ (\alpha)$ of these two completions. This has the implication, as observed by Thatte [Tha88], that instantiating an "unknown" coercion in a completion of $e$ must be delayed and instantiated separately in every context in which $e$ is used. In a polymorphic type discipline this necessitates that the language provide for passing coercions as arguments to let-bound variables. Through such formal coercion parameters the different contexts of the applied occurrences of a let-bound variable $x$ can pass the concrete coercions to $x$ that are necessary to safely evaluate the body bound to $x$.

These considerations motivate an extension of the dynamically typed $\lambda$-calculus to let-polymorphic programs where let-bindings may be *parameterized* by formal coercion parameters. In this way we can give a completion of $\lambda x.cons\ 1\, x$ that fits both contexts $C[]$ and $C'[]$ above:

$$\begin{array}{l}\textbf{let}\ f[c] = \lambda x.cons\ ([c]1)\, x\ \textbf{in}\\ car\,((f[\iota])(cons\ 1\ nil)) + 1\\ \ldots\\ (f[integer])(cons\ \ true\, nil)\end{array}$$

Unfortunately formal coercion parameters are easily used where they are not even necessary. Consider for example $\textbf{let}\ f = \lambda y.y\ \textbf{in}\ f\ true$. One completion of type $\mathsf{Bool}$ is $\textbf{let}\ f[c] = [c]\lambda y.y\textbf{in}(f[\iota])true$. Clearly an operationally preferable completion is simply $\textbf{let}f = \lambda y.y\textbf{in}f\ true$, also of type $\mathsf{Bool}$.

It remains to extend the notions of minimality to dynamically typed $\lambda$-terms with let-expressions in such a fashion that minimal principal completions coincide with static principal typings for statically polymorphically typable $\lambda$-terms.

We do not anticipate any problems with integrating other language features into dynamic typing with polymorphism such as exceptions, side-effects, continuations and pointers beyond those already observed for statically typed polymorphic languages.

# 8 Related work

Dynamic typing in a static language can be found in several programming languages. For a survey and historical perspective we refer the reader to [ACPP91].

The main motivation behind the work of Abadi, Cardelli, Pierce and Plotkin [ACPP89, ACPP91] and Leroy and Mauny [LM91] is in using type Dynamic as a universal interface to a changing environment that may contain persistent objects, concurrently executing programs or generally elements not under complete control of a single program. As a consequence these languages have very powerful explicit constructs for tagging and checking values that are both conceptually complex and expensive to implement. This is not an attractive model in a language in which tagging and checking values may be *inferred* since different completions may have very

[8] If simplificational coercion evaluation is used then $(c^{-1}; c = \iota$ is valid and minimal completions may be used as principal completions.

different and unexpected behavior (c.f. remarks by Thatte [Tha90]). By relying on a fixed number of tags — one for each type constructor — dynamic typing is conceptually easier and less expressive than full type tagging; the corresponding typecase form needs to match only type constructors, not complete type expressions and can thus be implemented efficiently using switches (indirect jumps).

In the absence of negative coercions dynamic typing turns into a subtyping discipline with Dynamic functioning as the "top" type. Thatte [Tha88] has investigated such a language with induced coercions where coercions are inserted at fixed places (function applications). He characterized the typability problem as a problem of solving subtyping constraints, but left its decidability open.[9] This problem has recently been shown to be decidable by O'Keefe and Wand [OW91]. The notion of coherence arises in coercion interpretations of subtyping. Breazu-Tannen, Cardelli, Coquand, Gunter and Scedrov [BCGS91] use coherent translations from a language with subtyping into one without, to provide models for a language integrating subtyping (inheritance), parametric polymorphism and recursive types. Similarly, Curien and Ghelli [CG90] give an axiomatization of coherence in $F_{\leq}$ using explicit coercions and use it to show typable $F_{\leq}$ programs have minimal types.[10] Our equational characterization of coherence extends the first-order subset of $F_{\leq}$ with negative coercions and a rule relating $\lambda$-terms to each other that have different types bound to the same variable.

Thatte introduced negative coercions in [Tha90]. In his type system the distinction between positive and negative coercions is carried over to induced coercions. Positive coercions may be placed anywhere, but negative coercions can only be placed at use points. Programs are required to have explicit type declarations for every variable; they are completed with explicit coercions such that the resulting program is a convergent completion with explicit coercions. (Thatte's semantically defined notion of convergence has motivated the syntactic notion of safety in this paper.) The denotational semantics is similar to Abadi et al.'s [ACPP91], and the operational semantics uses a form of simplificational evaluation of coercions in which values are tagged with sequences of full type expressions. Note that our completion problem is more general in that programs do not require type declarations for variables and that may insert arbitrary coercions any place.

Gomard [Gom90] inspired our approach to dynamic typing by type inference. He describes type inference for implicitly typed programs with no required type information at all. In dynamic typing terms his algorithm produces a completion with primitive coercions in which positive coercions may only occur at creation points ($\lambda$-abstractions, constants). Negative coercions for checking functions may occur at application points, but no negative coercions for base types are permitted; instead tagged versions of base operations are used. As a consequence tagging may "spread" to every point reachable from a single tagging operation. His type inference algorithm is a backtracking adaptation of Algorithm W [Mil78] that executes $\Theta(n^2)$ calls to a unification procedure and thus runs in time $\Theta(n^3)$ with an optimal unification algorithm. Our completion algorithm improves this bound to almost-linear time and "isolates" tagging operations better.

Cartwright and Fagan [CF91] present a very ambitious extension of ML's type inference system with regular recursive types, union types and implicit subtyping based on extension of unions. Dynamic type checking operations are not included in the type system, but they are added during type inference as a consequence of unification failure. All (non-type-variable) types

[9] Note that $\lambda$-bound variables have no type declarations in this type inference problem, which sets it apart from the (easier) type checking problem for the first-order fragment of $F_{\leq}$.

[10] An error was later discovered in their proof of decidability of typability in $F_{\leq}$; recently Pierce [Pie91] has announced that type checking $F_{\leq}$ is undecidable.

are represented as union types, which are encoded using a type representation scheme pioneered by Remy [Rem89] for record-based inheritance. There are several problems, however, both with the type system and with the "Remy encoding".[11] A typing rule for induced containments of union types is missing (e.g., $\tau_1 \subseteq \tau_3, \tau_2 \subseteq \tau_4 \vdash \tau_1 + \tau_2 \subseteq \tau_3 + \tau_4$) and the subtyping rule for recursive types in the stated form $t \subseteq u \Rightarrow T \subseteq U \vdash \mu t.T \subseteq \mu u.U$ is unsound. The Remy encoding results in restricting subtyping steps to language primitives; yet, on the other hand, it permits encoding of polymorphic types that are not expressible in the original type system. As a consequence the encoding has typing power incomparable to the original type system. For example, for primitive types $\mathsf{Bool}$, *integer* and $f0 : (\mathsf{Bool} + \mathit{integer}) \to \mathsf{Bool}$ the expression

$$\begin{array}{l}\textbf{let } \mathit{twice} = \lambda f.\lambda x.f(fx) \textbf{ in}\\ \textbf{let } f1 = \mathit{twice} f0 \textbf{ in}\\ \textbf{if } (f1(\textbf{if } \mathit{false} \textbf{ then } \mathit{true} \textbf{ else } 6)) \textbf{ then } 0 \textbf{ else } 1\end{array}$$

is typable in the original type system without negative coercions, but not in the Remy encoded system. On the other hand,

$$\begin{array}{l}\textbf{let } \mathit{cons1} = \lambda x.\mathit{cons}\, 1\, x \textbf{ in}\\ \mathit{car}(\mathit{cons1}(\mathit{cons}\ 1\ \mathit{nil})) + 1\\ \ldots\\ \mathit{cons1}(\mathit{cons}\ \mathit{true}\, \mathit{nil})\end{array}$$

(adapted from [Tha88]) is not typable in the original type system, but it is in the Remy encoded system. Furthermore, counter to a claim in their paper currently there appears to be no known linear-time algorithm for circular unification (unification closure) [KR90].

# 9 Conclusion

Dynamic typing promises to integrate the advantages of compile-time and run-time type checked programming languages without inheriting their disadvantages. In particular, inferring minimal completions of implicitly dynamically typed programs makes it possible to "only pay for the amount of dynamic typing that is unavoidable" in the underlying static type system.

To estimate the practicality of the minimal completion algorithms we have presented we plan on implementing them for Scheme. Since we do not believe that completions with induced coercions lead to better results in most cases than with only primitive coercions we expect the almost-linear time minimal completion algorithms to be of particular practical value, both to programmers and to optimizers. For a practical adaptation of dynamic typing to a polymorphic type discipline the problem of minimizing the number of coercion parameters to let-bound variables needs to be addressed. It is an intriguing prospect that a polymorphic minimal completion algorithm may lead to novel implementation techniques and optimizations for conventional run-time typed languages.

**Acknowledgements**

This paper would not have reached the form it has without Satish Thatte's insights, comments and corrections. I am especially greatful for his inquisitive questions that led to the definition of safety. I am also grateful for helpful discussions with members of the TOPPS group at DIKU. Finally, I would like to thank one anonymous referee for some corrections and helpful suggestions for improved exposition. Needless to say, any remaining mistakes and expository deficiencies are entirely my own fault.

[11] A revision of this paper, in which these problems are addressed and rectified, is currently underway.

# References


[ACPP89] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages*, pages 213–227, ACM, Jan. 1989.

[ACPP91] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):237–268, April 1991. Presented at POPL '89.

[BCGS89] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance and explicit coercion. In *Proc. Logic in Computer Science (LICS)*, pages 112–129, 1989.

[BCGS91] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–221, July 1991. Presented at LICS '89.

[CF91] R. Cartwright and M. Fagan. Soft typing. In *Proc. ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation, Toronto, Ontario*, pages 278–292, ACM, ACM Press, June 1991.

[CG90] P. Curien and G. Ghelli. Coherence of subsumption. In A. Arnold, editor, *Proc. 15th Coll. on Trees in Algebra and Programming, Copenhagen, Denmark*, pages 132–146, Springer, May 1990.

[FM88] Y. Fuh and P. Mishra. Type inference with subtypes. In *Proc. 2nd European Symp. on Programming*, pages 94–114, Springer-Verlag, 1988. Lecture Notes in Computer Science 300.

[GI91] Z. Galil and G. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, 23(3):319–344, Sept. 1991.

[Gom90] C. Gomard. Partial type inference for untyped functional programs (extended abstract). In *Proc. LISP and Functional Programming (LFP), Nice, France*, July 1990.

[Hen90] F. Henglein. Fast left-linear semi-unification. In *Proc. Int'l. Conf. on Computing and Information*, Springer, May 1990. Lecture Notes of Computer Science, Vol. 468.

[Hen91] F. Henglein. Efficient type inference for higher-order binding-time analysis. In *Proc. Conf. on Functional Programming Languages and Computer Architecture (FPCA), Cambridge, Massachusetts*, pages 448–472, Springer, Aug. 1991. Lecture Notes in Computer Science, Vol. 523.

[KR90] P. Kanellakis and P. Revesz. *On the Relationship of Congruence Closure and Unification*, chapter 2, pages 23–41. *Frontier Series*, Addison-Wesley, ACM Press, 1990.

[LM91] X. Leroy and M. Mauny. Dynamics in ML. In *Proc. Conf. on Functional Programming Languages and Computer Architecture (FPCA), Cambridge, Massachusetts*, pages 406–426, Springer, Aug. 1991. Lecture Notes in Computer Science, Vol. 523.

[LPvL87] J. La Poutré and J. van Leeuwen. Maintenance of transitive closures and transitive reductions of graphs. In *Proc. Int'l Workshop on Graph-Theoretic Concepts in Computer Science*, pages 106–120, Springer-Verlag, June 1987. Lecture Notes in Computer Science, Vol. 314.

[Mil78] R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375, 1978.

[Myc84] A. Mycroft. Dynamic types in statically typed languages. Aug. 1984. Unpublished manuscript, 2nd draft version.

[OW91] P. O'Keefe and M. Wand. Type inference for partial types is decidable. Sept. 1991. Submitted to ESOP '92.

[Pie91] B. Pierce. *Bounded Quantification is Undecidable.* Technical Report CMU-CS-91-161, Carnegie Mellon University, July 1991. To be presented at POPL '92.

[Rem89] D. Remy. Typechecking records and variants in a natural extension of ML. In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages*, pages 77–88, ACM, Jan. 1989.

[Ses89] P. Sestoft. Replacing function parameters by global variables. In *Proc. Functional Programming Languages and Computer Architecture (FPCA), London, England*, pages 39–53, ACM Press, Sept. 1989.

[Tar83] R. Tarjan. *Data Structures and Network Flow Algorithms.* Volume CMBS 44 of *Regional Conference Series in Applied Mathematics*, SIAM, 1983.

[Tha88] S. Thatte. Type inference with partial types. In *Proc. Int'l Coll. on Automata, Languages and Programming (ICALP)*, pages 615–629, 1988.

[Tha90] S. Thatte. Quasi-static typing. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 367–381, ACM, Jan. 1990.

[Yel88] D. Yellin. *A Dynamic Transitive Closure Algorithm.* Technical Report RC 13535, IBM T.J. Watson Research Ctr., June 1988.