Dynamic Typing: Syntax and Proof Theory*

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

July 31, 1992; revised March 10, 1993

Abstract

We present the *dynamically typed* λ-calculus, an extension of the statically typed λ-calculus with a special type Dyn and explicit *dynamic type coercions* corresponding to run-time type tagging and type check-and-untag operations. Programs in run-time typed languages can be interpreted in the dynamically typed λ-calculus via a nondeterministic *completion process* that inserts explicit coercions and type declarations such that a well-typed term results.

We characterize when two different completions of the same run-time typed program are *coherent* with an equational theory that is independent of an underlying λ-theory. This theory is refined by orienting some equations to define *safety* and *minimality* of completions. Intuitively, a safe completion is one that does not produce an error at run-time which another completion would have avoided, and a minimal completion is a safe completion that executes fewest tagging and check-and-untag operations amongst all safe completions.

We show that every every untyped λ-term has a safe completion at any type and that it is unique modulo a suitable congruence relation. Furthermore, we present a rewriting system for generating minimal completions. Assuming strong normalization of this rewriting system we show that every λI-term has a minimal completion at any type, which is furthermore unique modulo equality in the dynamically typed λ-calculus.

1 Introduction

We present an extension of the statically typed λ-calculus with a special type Dyn and *dynamic type coercions*. These represent run-time tagged values and associated tagging and check-and-untag operations as they are found in run-time typed languages that use these dynamic type coercions implicitly. A program with implicit dynamic type coercions 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.

We can think of elements of type Dyn as "(type) tagged" values; that is, as tag-value pairs where the tag indicates the type constructor or primitive type of the value component. Dynamic

*Published in Science of Computer Programming special issue on ESOP '92 (to appear). This research has been partially supported by Esprit BRA 3124, Semantique.

type coercions represent a special class of functions that embed values into the "universal" type Dyn and project them back from Dyn.

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. It connects implicitly and explicitly dynamically typed programs by automatic type inference that, when possible, generates so-called *minimal completions*. Minimal completions conservatively extend statically typed λ-terms in the sense that the minimal completion of any untyped λ-term e at a type τ contains no dynamic type coercions (in fact no coercions at all) if e is statically typed at τ. The practical significance of minimal completions is that one only "pays" for the amount of dynamic typing really *needed* as opposed to paying for it *always*. 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.

For every type constructor tc of arity k there is a *tagging operation* $tc!$ that maps elements of type $tc(\mathsf{Dyn},\ldots,\mathsf{Dyn})$ to Dyn by pairing them with their type. For example, the coercion Func! maps a function f of type $\mathsf{Dyn}\to\mathsf{Dyn}$ to Dyn. For every tagging operation $tc!$ there is a corresponding *check-and-untag operation* $tc?$ that maps elements of type Dyn to $tc(\mathsf{Dyn},\ldots,\mathsf{Dyn})$: 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. Starting with these *dynamic type coercions* in Section 2 we build a calculus of coercions by adding identity coercions, coercion composition and a coercion constructor corresponding to each type constructor. 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 result of composing (in diagrammatical order) coercion c_1, function f and finally coercion c_2.

In Section 3 we extend the equational and reduction theory of dynamic type coercions to the dynamically typed λ-calculus, which is the simply typed λ-calculus extended with coercion application.

Every dynamically typed program (λ-term) is a completion of the underlying run-time typed program. There are generally many different completions for the same run-time typed program, however. In Section 5 we characterize *coherence* of completions by an equational theory. This equational theory is independent of an underlying λ-theory (*i.e.*, it does not include α-, β- or η-conversion) and may thus be understood as an equational theory of coercions embedded in a higher-order language.

The equations characteristic of coherence in this dynamic typing discipline (and not present in the coercion-theoretic treatment of other typed λ-calculi, such as subtyping) are the rules specific to dynamic type coercions:

$$\begin{array}{rcll} tc!; tc? & = & \iota & (\phi) \\ tc?; tc! & = & \iota & (\psi) \end{array}$$

where $tc!$ is a tagging operation, $tc?$ its corresponding check-and-untag operation, ι the "no-op" (do nothing) coercion, and ; is diagrammatic composition of coercions. The asymmetry in these

equations give rise to a reduction-theoretic treatment — left-hand side reduces to right-hand side — of the dynamically typed λ-terms:

- the ψ-equation is not satisfied by ordinary run-time type checking since first checking a tagged value for a specific tag and then tagging it again may generate a type error if the tagged value has a different tag; the right hand side is *safer*;
- the ϕ-equation is semantically satisfied by ordinary run-time type checking as first tagging a value (of the appropriate type) with a tag and then checking for that tag and untagging it is equivalent to returning the original untagged value; however, *operationally* tagging and then checking is more inefficient than a "no-op"; the right-hand side is more *efficient*.

Extending the reduction $tc?;tc! >_\psi \iota$ to the full dynamically typed λ-calculus in Sectin 6 we define a notion of *safe completions*, which do not contain avoidable type errors generated by the left-hand side of a ψ-rule. We show that the *canonical completion* corresponding to the interpretation of an untyped λ-term with implicit run-time type operations is safe in this sense.

Extending the reduction $tc!;tc? >_\phi \iota$ to dynamically typed λ-terms in Section 7 we define a completion to be *minimal* if it is safe and $>_\phi$-minimal with respect to all other safe completions (at the same type). By distinguishing between *positive, negative* and *neutral* coercions we specialize the equations for dynamically typed λ-terms and orient them according to the polarity of coercions involved. Assuming strong normalization of the resulting rewriting system we use techniques from the theory of reduction systems and (term) rewriting to prove that every λI-term has a unique minimal completion at every type .

We shall draw heavily on reduction and term rewriting theory. Working knowledge of general terminology, methods and results as covered by Huet in [Hue80] is desirable for an understanding and reconstruction of the proofs presented here.[1] For space reasons many proofs are omitted. These are contained in a full version of this paper [Hen93].

2 Coercion calculus

In this section we describe syntax and proof theory of the *coercion calculus* for dynamic typing. Our coercion calculus is parameterized over a given set of type constructors and primitive types. The pure dynamically typed λ-calculus with only functions 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 λ-calculus with an additional primitive type, the Booleans. The type expressions in this language are generated by the production

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

Our results extend to other types and their coercions in a straightforward manner, as will be indicated where appropriate.

Primitive coercions and constructions on coercions are defined by inference systems (sets of rule[2] schemes) over judgements of the form $c : \tau \rightsquigarrow \tau'$. Equality is defined axiomatically by equations between well-formed coercions with the same type signature.

[1] We use the terms "reducing" and "rewriting" interchangeably. Note that we use "reduction modulo equivalence" differently from Huet: in our case this refers to reduction on the congruence classes of terms factored by the equivalence, which is specified by a set of equations.

[2] We treat axioms as rules with no antecedents.

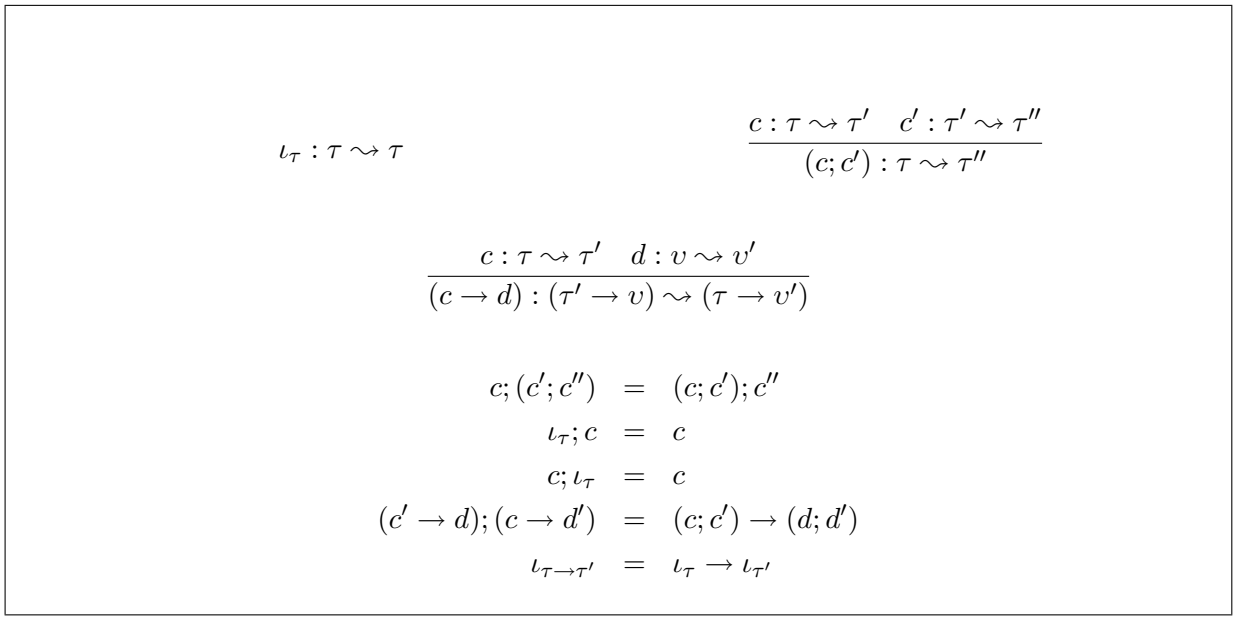

Figure 1: Core coercion formation rules and equations

2.1 Coercion constructions and equality

The core of the coercion calculus is captured by the inference rules and associated equations in Figure 1.

Two coercions c, c' can be *composed*, written $c;c'$ in diagrammatic order, if their type signatures match. There is a special *identity coercion* ι_τ for every type τ. The corresponding equations state that composition is associative with the identity coercions being left- and right-identities. Altogether these equations guarantee that coercions form a category under composition.

For every type constructor tc there is a *coercion constructor*, also denoted by tc. In our case we have only $\to$. The corresponding equations for $\to$ express that $\to$ is a bifunctor on coercions that is contravariant in its first argument and covariant in its second. For other type constructors tc such as Pair or List we would add the corresponding coercion constructor and equations to make tc a (covariant) (multi)functor on coercions.

Definition 1 *(Coercion, equality)*

A coercion (from τ to τ') *is an expression c such that $c : \tau \rightsquigarrow \tau'$ is derivable from any given primitive coercions and the formation rules in Figure 1.*

Coercions c, c' are equal*, written $\vdash c = c'$ or simply $c = c'$ if $c = c'$ can be derived from the core coercion equations in Figure 1 together with reflexivity, symmetry, transitivity and compatibility of the equality relation.*

The formation rules and the coercion equations in Figure 1 constitute the *core* part of any coercion calculus as they specify the expected coercion constructions and properties for coercions without actually defining any nontrivial coercions. Indeed we have that every $c : \tau \rightsquigarrow \tau'$ formed from the identity coercions, composition and the coercion constructor(s) alone is equal to an identity coercion. A *proper* coercion is a coercion not equal to an identity coercion.

Note that we call two coercions equal even though they may be different as expression trees or strings. If coercions c, c' are literally identical then we write $c \equiv c'$. Thus $c \equiv c'$ implies $c = c'$

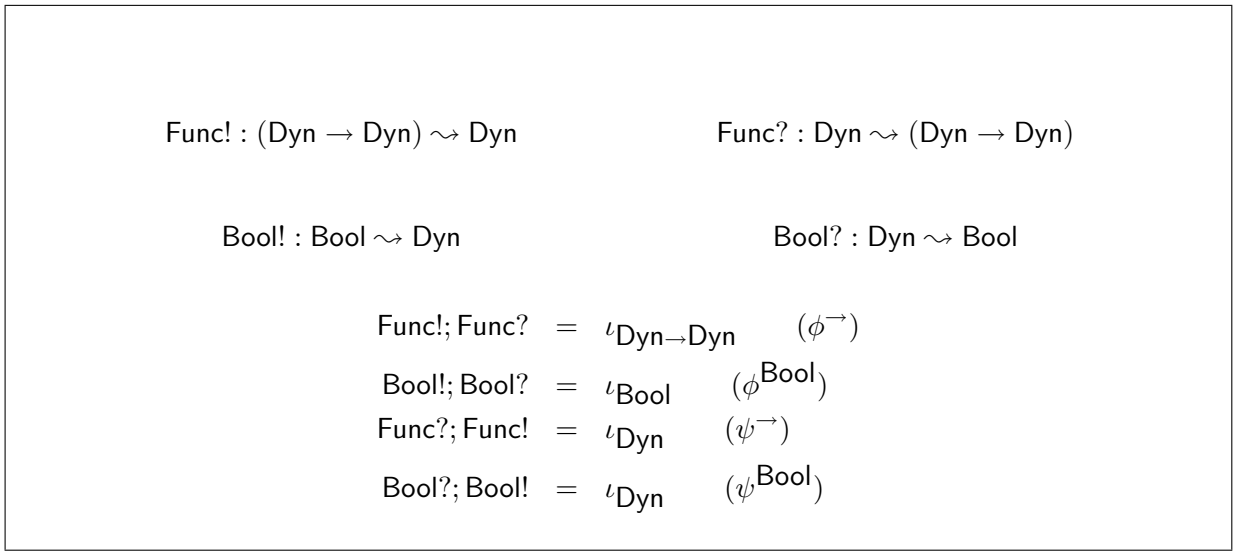

Figure 2: Formation rules and equations for dynamic type coercions

as long as c is a legal coercion expression, but $c = c'$ does not generally imply $c \equiv c'$.

2.2 Dynamic type coercions

The essence of dynamically typed λ-calculus are the special type Dyn and *dynamic type coercions*. For every type constructor and primitive type tc there is an *tagging coercion* $tc! : tc(\mathsf{Dyn}, \ldots, \mathsf{Dyn}) \rightsquigarrow \mathsf{Dyn}$ and a corresponding *untagging coercion* $tc? : \mathsf{Dyn} \rightsquigarrow tc(\mathsf{Dyn}, \ldots, \mathsf{Dyn})$. The operational intuition is that $tc!$ tags its input with the type constructor or primitive type tc, and $tc?$ checks to see if the input has tag tc and then returns the untagged value. If $tc?$ finds another tag than tc generates a *(dynamic) type error*. The type Dyn contains all tagged values created by tagging operations. Since a value is only tagged with a type constructor, *not* a whole type, we must be sure a value has fixed known type before it is tagged by $tc!$. This explains why $tc!$ expects an input of type $tc(\mathsf{Dyn}, \ldots, \mathsf{Dyn})$. Note, however, that $tc!$ can be composed with induced coercions to give a coercion from $tc(\tau_1, \ldots, \tau_k)$ to Dyn for arbitrary $\tau_1, \ldots, \tau_k$.

In our type language we have type constructor $\rightarrow$ and primitive type Bool. The corresponding dynamic type coercions are denoted by $\mathsf{Func!}, \mathsf{Func?}, \mathsf{Bool!}$ and $\mathsf{Bool?}$. See the formation rules in Figure 2.

2.3 Conversion

The interesting aspect of dynamic typing is its *conversion* theory.

Definition 2 *(E-conversion)*
*Let E be a set of coercion equations. We say coercions c and c' are E-*convertible *if $c = c'$ is derivable from E and coercion equality; i.e., by adding E to the core equations. This is written $E \vdash c = c'$ or $c =_E c'$.*

The equations $\phi^{\rightarrow}$ and ϕ^{Bool} in Figure 2 express that first tagging a value with a type constructor/primitive type and then checking for the same tag and untagging is equivalent to doing nothing at all. These rules together are denoted by ϕ.

The equations $\psi^{\rightarrow}$ and ψ^{Bool} in Figure 2 state that first checking for a certain type constructor and then tagging with the same type constructor is equivalent to doing nothing at all. We refer to these rules collectively by ψ.

2.4 Reduction

Notice that there are semantic and operational reasons *not* to treat ϕ and ψ as equations.

1. The ψ-equations will generally not be valid in a conventional semantics for coercions. In particular, the right-hand side is *safer* than the left-hand side as, *e.g.*, $\mathsf{Func?};\mathsf{Func!}$ may generate a type error when applied to a tagged boolean value whereas the right-hand side, being an identity coercion that is operationally a "no-op", will not.

2. The ϕ-equations will be valid, but their right-hand sides are operationally more *efficient* than their left-hand sides as, *e.g.*, $\mathsf{Func!};\mathsf{Func?}$ first tags a function just to untag it immediately afterwards whereas the right-hand side is a simple "no-op".

To address the semantic and operational asymmetry in the ϕ- and ψ-rules we introduce reductions on (congruence classes of) coercions.

Definition 3 *(R-reduction)*
Let R, E be sets of equations. Consider R as a term rewriting system on coercions by interpreting the equations in it as left-to-right rewriting rules.

Coercion c R-reduces in one step to c' under E-conversion, written $E \vdash c >_R c'$, if $E \vdash c = \bar{c}, c' = \bar{c}'$ for some $\bar{c}, \bar{c}'$ and $\bar{c}$ reduces to $\bar{c}'$ under R.

We say coercion c R-reduces to c' under E-conversion, $E \vdash c >_R^* c'$, if $E \vdash c = c'$ or $E \vdash c >_R c_1 >_R \ldots >_R c_n >_R c'$ for some $c_1, \ldots, c_n$ where $n \geq 0$.

Putting it differently, a coercion c R-reduces in one step under E to c' if, viewing the equations in R as left-to-right rewriting rules on the E-congruence classes, c is an element of an E-congruence class that can be rewritten in a single step to an E-congruence class containing c'. If E is empty we may omit E and the turnstile.

The following lemmas state some basic reduction-theoretic properties of ϕ-, ψ-, and $\phi\psi$-reduction.

Lemma 1 *(Commutativity of ϕ and ψ)*
ϕ- and ψ-reduction commute: for all $c, c', c'' : \tau \rightsquigarrow \tau'$ if $c >_\phi c'$ and $c >_\psi c''$ then $c' = c''$ or there exists $c''' : \tau \rightsquigarrow \tau'$ such that $c' >_\psi c'''$ and $c'' >_\phi c'''$.

Proof: Viewing all the core equations but the associativity rule for composition as left-to-right reduction rules it is easy to verify that they commute with ϕ- and ψ-reductions. Consequently it is sufficient to prove the result for the case where no left-to-right oriented core equation is applied to c.

If c has both a ϕ- and a ψ-redex then these are either nonoverlapping, and the result follows trivially because the nonreduced reduct is preserved, or the redex has the form $tc!;tc?;tc!$ or $tc?;tc!;tc?$ where tc is either Func or Bool. In the first case a ψ-reduction step yields $tc!;\iota_{\mathsf{Dyn}}$, and a ϕ-reduction step yields $\iota_{tc(\mathsf{Dyn},\ldots,\mathsf{Dyn})};tc!$, both coercions being equal to $tc!$. Similar for the second case. ■

Lemma 2 *(Strong normalization of ϕ, ψ and $\phi\psi$)*
Let R be ϕ, ψ or $\phi\psi$. Then R-reduction is strongly normalizing; that is, there is no infinite reduction sequence $c >_R c' >_R c'' \ldots$.

Proof: We define an integer interpretation $\chi(c)$ of coercions by mapping $\rightarrow$ and ; to addition, ι_τ, Func! and Bool! to 0, and Func?, Bool? to 1. It can be verified that $\chi(c) = \chi(c')$ if $c = c'$ and $\chi(c) > \chi(c')$ if $c >_R c'$. Since $\chi(c) \geq 0$ for all c it follows that $\phi\psi$-reduction (and thus trivially ϕ- and ψ-reduction) is strongly normalizing. ■

From the previous two lemmas and confluence of ϕ-reduction and ψ-reduction we get:

Lemma 3 *(Confluence of ϕ, ψ, and $\phi\psi$)*
Let R be ϕ, ψ or $\phi\psi$. Then R is confluent: for every coercion $c : \tau \rightsquigarrow \tau'$ with $c >_R^* c'$ and $c >_R^* c''$ there exists $c''' : \tau \rightsquigarrow \tau'$ such that $c' >_R^* c'''$ and $c'' >_R^* c'''$.

Proof: Since ϕ-redexes do not overlap with each other ϕ-reduction has the diamond property: for $d >_\phi d'$ and $d >_\phi d''$ there is d''' such that $d' >_\phi d'''$ and $d'' >_\phi d'''$. Similarly, ψ-reduction has the diamond property. Note that the diamond property implies confluence. The result now follows from the confluence of ϕ-reduction and of ψ-reduction and the commutativity of ϕ- and ψ-reduction (Lemma 1). ■

Confluence together with strong normalization implies that every coercion has a normal form coercion and any two normal form coercions are equal.

Theorem 1 *(Church-Rosser, uniqueness of $\phi\psi$-normal forms)*
For every pair of coercions c, c' such that $c =_{\phi\psi} c'$ there is a $\phi\psi$-normal form coercion c'' such that $c >_{\phi\psi}^* c''$ and $c' >_{\phi\psi}^* c''$. Furthermore, c'' is unique up to coercion equality.

In the next subsection we give an independent characterization of a canonical class of $\phi\psi$-normal form coercions.

2.5 Canonical coercions

Since there may be different coercions with the same type signature and since there are generally many different ways of writing equal coercions we give an inductive construction for picking out a unique coercion for every pair of types τ, τ'.

Definition 4 *(Canonical coercions)*
If coercion $c : \tau \rightsquigarrow \tau'$ is derivable using solely the inference system in Figure 3 we say c is a canonical coercion (from τ to τ') *and write $\vdash_c c : \tau \rightsquigarrow \tau'$.*

The following proposition can be checked easily.

Proposition 4 *1. For every type signature $\tau \rightsquigarrow \tau'$ there is a canonical coercion $c : \tau \rightsquigarrow \tau'$.*

2. Canonical coercions are unique as expressions*; i.e., if $\vdash_c c : \tau \rightsquigarrow \tau'$ and $\vdash_c c' : \tau \rightsquigarrow \tau'$ then $c \equiv c'$.*

Canonical coercions interact well with $\phi\psi$-reduction:

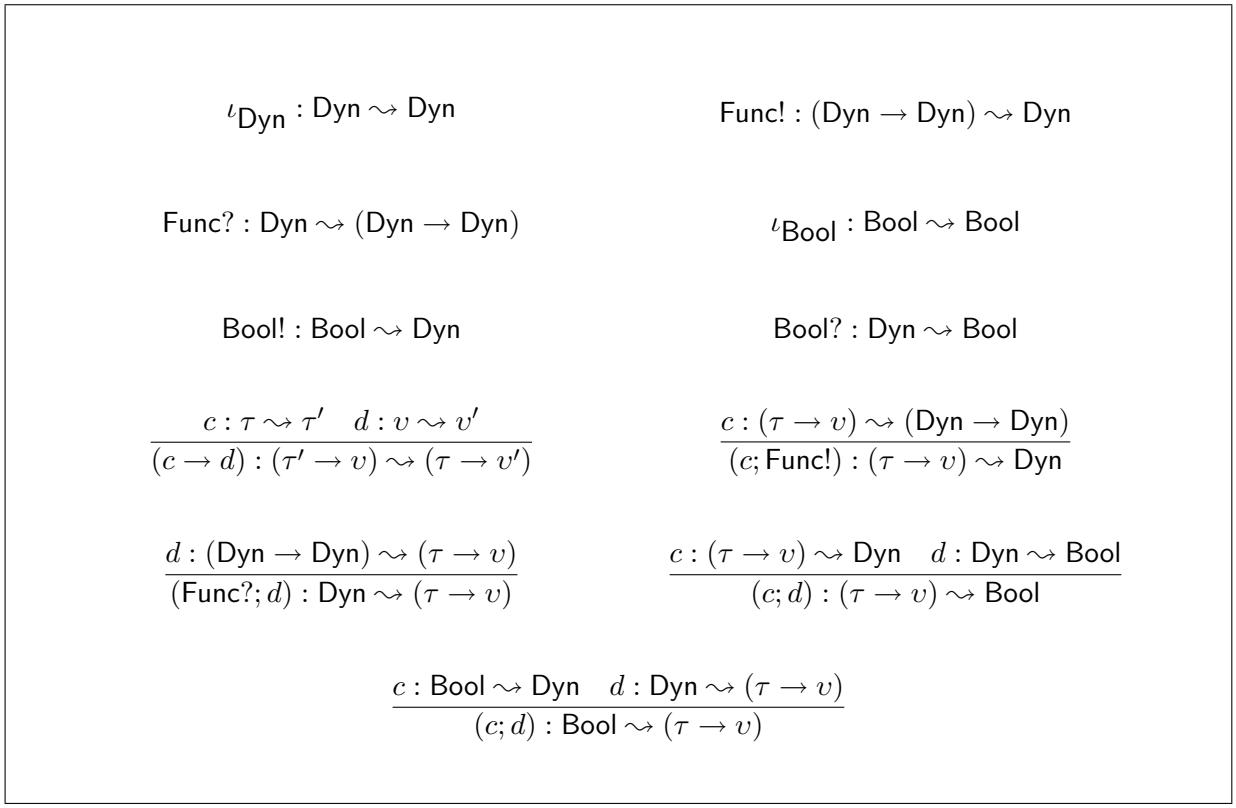

Figure 3: Canonical coercions

Lemma 5 *(Canonicity of canonical coercions)*

1. *Every canonical coercion is a $\phi\psi$-normal form.*
2. *Every $\phi\psi$-normal form is equal to a canonical coercion.*

Proof:

1. It can be checked that no canonical coercion has a ϕ- or ψ-redex. Since the core equations commute with the ϕ- and ψ-rules it follows that no coercion equal to a canonical coercion has a ϕ- or ψ-redex.

2. By induction on formation of coercions it can be shown that for all $c : \tau \rightsquigarrow \tau'$ there exists a canonical coercion $c' : \tau \rightsquigarrow \tau'$ such that $c =_{\phi\psi} c'$. If c is a $\phi\psi$-normal form Corollary 1 implies that $c = c'$ as any two $\phi\psi$-convertible $\phi\psi$-normal forms must be equal.

■

Thus every set of pairwise equal $\phi\psi$-normal forms contains exactly one canonical coercion. Since any coercion normalizes to a canonical coercion and since there is exactly one canonical coercion for every type signature it follows that any two coercions with the same type signature normalize to the same canonical coercion.

Theorem 2 *(Uniqueness of coercions under $\phi\psi$-conversion)*
Coercions c and c' have the same type signature $\tau \rightsquigarrow \tau'$ if and only if they are $\phi\psi$-convertible; i.e., $c : \tau \rightsquigarrow \tau'$ and $c' : \tau \rightsquigarrow \tau'$ if and only if $\phi\psi \vdash c = c'$.

As a consequence we obtain that *all* types are isomorphic under $\phi\psi$-conversion. This is due to the ability inherent in the ψ-equations to "undo" an arbitrary number of check-and-untag operations. Since this would require costly tracing and storing of the check-and-untag operations performed on a value this is not satisfied by a conventional semantics. (See however the simplificational semantics of Thatte [Tha90], where any two coercions with the same type signature are equal.)

2.6 Safe coercions

As noted before a coercion semantics can be expected to satisfy ϕ, but not ψ. More precisely, certain coercions may generate type errors where others, with the same signature, do not. We define safety by treating the ψ-rules as inequations with the right-hand sides being *safer* than (or rather at least as safe as) the corresponding left-hand sides. A safe coercion is one that is at least as safe as *any* other coercion with the same type signature.

Definition 5 *(Safe coercions)*
We say a coercion $c : \tau \rightsquigarrow \tau'$ is safe *if for all $c' : \tau \rightsquigarrow \tau'$ we have $\phi \vdash c' >^*_\psi c$.*

This is a stronger definition than requiring a coercion not to be reducible to a properly safer coercion. It turns out, however, that the same coercions are safe by either definition.

Note that ψ-reduction is not normalizing under ϕ-conversion, and that a coercion may very well be safe even though it contains a ψ-redex; *e.g.*, the coercion $\mathsf{Func!}; \mathsf{Func?}; \mathsf{Func!}$ is safe since it is ϕ-convertible to $\mathsf{Func!}$, which is a canonical coercion.

Prinicipally it is conceivable that the congruence induced by ψ-reduction under ϕ is not conservative over ϕ-conversion; *i.e.*, that there are two coercions that can be reduced to each other by ψ-steps under ϕ-conversion without being directly ϕ-convertible. This is not so, however, as the following lemma states.

Lemma 6 *(Conservative extension of ϕ-conversion by ψ-reduction)*
If $\phi \vdash c >^*_\psi c'$ and $\phi \vdash c' >^*_\psi c$ then $\phi \vdash c = c'$.

Proof: Let c, c' be such that $\phi \vdash c >^*_\psi c'$ and $\phi \vdash c' >^*_\psi c$. Let $\bar{c}, \bar{c'}$ be ϕ-normal forms for c, c' respectively, which exist by Lemma 2. Since ϕ-reduction is confluent (Theorem 3) and commutes with ψ-reduction (Lemma 1) it follows that:

$$\begin{array}{lll} \bar{c} & >^*_{\phi\psi} & \bar{c'} \\ \bar{c'} & >^*_{\phi\psi} & \bar{c} \end{array}$$

Since $\phi\psi$-reduction is strongly normalizing (Lemma 2) the last two reductions can only hold if $\bar{c} = \bar{c'}$. Since $c =_\phi \bar{c}$ and $c' =_\phi \bar{c'}$ by the first two reductions it follows that $c =_\phi c'$. ■

Lemma 7 *(Safety of canonical coercions)*
Every canonical coercion is safe.

Proof: Let $c : \tau \rightsquigarrow \tau'$ be a canonical coercion, and let $c' : \tau \rightsquigarrow \tau'$ be any coercion with the same type signature. We need to show $\phi \vdash c' >^*_\psi c$. By Theorem 2 we know that $c' =_{\phi\psi} c$. Theorem 1 implies that there exists a $\phi\psi$-normal form c'' such that $c' >^*_{\phi\psi} c''$ and $c^* >_{\phi\psi} c''$. By Lemma 5 every $\phi\psi$-normal form is equal to a canonical coercion and since c is the only canonical coercion with type signature $\tau \rightsquigarrow \tau'$ it must be that $c'' = c$ and so $c' >^*_{\phi\psi} c$. This implies immediately that $\phi \vdash c' >^*_\psi c$. ■

By Proposition 4, part 1, there is a canonical coercion for every type signature. And by Lemma 5 any two coercions that can be ψ-reduced to each other under ϕ-conversion are already ϕ-convertible. This yields the following theorem for safe coercions.

Theorem 3 *(Existence and uniqueness modulo ϕ-conversion of safe coercions)*

1. *For every type signature $\tau \rightsquigarrow \tau'$ there is a safe coercion; in particular the canonical coercion at that type is safe.*
2. *All safe coercions with the same type signature are unique modulo ϕ-conversion.*

2.7 Minimal coercions

A coercion should ideally be both safe and, amongst safe completions, operationally the best; that is, it should have no "avoidable" dynamic type coercions. Note that the left-hand sides in ϕ have dynamic type coercions that are eliminated on the corresponding right-hand sides. Treating ϕ as left-to-right reduction rules with the intention that the right-hand-sides are operationally better than the left-hand sides we are led to the following definition of *minimal coercions*.

Definition 6 *(Minimal coercions)*
A coercion $c : \tau \rightsquigarrow \tau'$ is minimal *if*

1. *it is safe;*
2. *for every safe coercion* $c' : \tau \rightsquigarrow \tau'$ *we have* $c' >^*_\phi c$.

A safe coercion is thus minimal if it is at least as efficient (in the sense of ϕ-reduction) as any other safe completion with the same type signature. A somewhat weaker definition that considers a coercion minimal if there is no properly more efficient safe coercion leads to the same notion of minimality.

Minimal coercions exist and are unique. In fact it is not difficult to verify that canonical coercions are minimal.

Lemma 8 *(Minimality of canonical completions)*
Every canonical coercion is minimal.

Proof: By Lemma 7 we know that every canonical coercion c is safe. By Theorem 3, part 2, any other safe coercion c' with the same type signature must be ϕ-convertible to c. Note that ϕ-reduction is confluent (Lemma 3) and thus Church-Rosser. Thus there exists a c'' such that $c >^*_\phi c''$ and $c' >^*_\phi c''$. By Lemma 5, part 1, we know that c, being canonical, is a $\phi\psi$-normal form. So $c >^*_\phi c''$ can only be the case if $c = c''$, which implies $c' >^*_\phi c$ as desired. ■

Analogous to Theorem 3 we obtain that minimal completions are unique and exist for every type signature.

Theorem 4 *(Existence and uniqueness of minimal coercions)*

1. *For every pair of types* τ, τ' *there is a minimal coercion.*
2. *Minimal coercions are unique; that is, if* $c : \tau \rightsquigarrow \tau'$ *and* $c' : \tau \rightsquigarrow \tau'$ *are both minimal coercions then* $c = c'$.

Proof:

1. Follows from the fact that every type signature has a canonical coercion (Proposition 4, part 1) and canonical coercions are minimal (Lemma 8).
2. Let c, c' be minimal coercions with the same type signature. By definition we have $c >^*_\phi c'$ and $c' >^*_\phi c$. Since ϕ-reduction is strongly normalizing (Lemma 2) this is only possible if $c = c'$.

■

2.8 Positive, negative and neutral coercions

We categorize coercions by polarity. This will be useful in Sections 6 and 7.

Definition 7 *(Positive, negative and neutral coercions)*
Define positive, negative *and* neutral coercions *by induction:*

- *The tagging operations* Func! *and* Bool! *are positive. The untag-and-check operations* Func? *and* Bool? *are negative. The coercions* Func!; Bool? *and* Bool!; Func? *are neutral. The identity coercions are positive, negative and neutral (but not* properly*).*

- *Let* c_1^+, c_2^+ *be positive,* c_1^-, c_2^- *negative, and* c_1^*, c_2^* *neutral. Then*
  - $c_1^+; c_2^+$ *(if well-formed) and* $c_1^- \to c_2^+$ *are positive;*
  - $c_1^-; c_2^-$ *(if well-formed) and* $c_1^+ \to c_2^-$ *are negative;*
  - $(c_1^*; c_2^*), (c_1^*; c_2^-), (c_1^+; c_2^*)$ *(if well-formed) and* $c_1^* \to c_2^*$ *are neutral.*

Proposition 9 *(Basic properties of positive and negative coercions)*

1. *Let* c, c' *both be positive or negative, respectively. If* $\phi\psi \vdash c = c'$ *then* $\vdash c = c'$.
2. *If* $c^+ : \tau \rightsquigarrow \tau'$ *is positive and* $c^- : \tau' \rightsquigarrow \tau$ *negative then* $\phi \vdash c^+; c^- = \iota_\tau$ *and* $\psi \vdash c^-; c^+ = \iota_{\tau'}$.
3. *If* $c^+ : \tau \rightsquigarrow \tau'$ *is positive and* $c^- : \tau' \rightsquigarrow \tau''$ *is negative then* $c^+; c^-$ *is safe.*

Under ϕ-conversion tagging operations embed their domain type in the range type as any tagged element can be projected back by the corresponding check-and-untag operation. Positive coercions extend the embedding by tagging to other types. Let us first give an independent definition of a partial order between types.

Definition 8 *(Subtype relation)*
Define the subtype *relation* $\tau \leq \tau'$ *on types by induction:*

- $\mathsf{Bool} \leq \mathsf{Dyn}$ *and* $\mathsf{Dyn} \to \mathsf{Dyn} \leq \mathsf{Dyn}$*;*
- $\tau \leq \tau$ *for all types* τ*;*
- *if* $\tau_1 \leq \tau_2$ *and* $\tau_2 \leq \tau_3$ *then* $\tau_1 \leq \tau_3$*;*
- *if* $\tau_1 \leq \tau_1'$ *and* $\tau_2 \leq \tau_2'$ *then* $\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$.

Note that the type constructor $\to$ is *co*variant *w.r.t.* $\leq$ in its first as well as its second argument! We write $\tau < \tau'$ if $\tau \leq \tau'$ but not $\tau = \tau'$.

Proposition 10 *(Basic properties of subtype relation)*

1. *There are no infinite ascending chains* $\tau_1 < \tau_2 < \ldots$.
2. *Every set of types has a least upper bound.*
3. *If a set of types has a lower bound then it has a greatest lower bound.*

The subtype relation is an independent characterization of the embedding relation defined by positive coercions:

Proposition 11 *(Characterization of subtype relation)*
The following statements are equivalent.

1. $\tau \leq \tau'$.
2. *There exists a positive coercion* $c : \tau \rightsquigarrow \tau'$.
3. *There exists a negative coercion* $c : \tau' \rightsquigarrow \tau$.

Instead of reducing to a minimal coercion we shall wish to *factor* (*w.r.t.* composition ;) coercions into positive and negative coercions in Sections 6 and 7.

Let all subscripts superscripted with $^+$ be *proper* positive coercions and those superscripted with $^-$ *proper* negative coercions.

$(c; c'); c''$	$=$	$c; (c'; c'')$	(ASSOC)
$c; \iota_\tau$	$\Rightarrow$	c	
$\iota_\tau; c$	$\Rightarrow$	c	
$\iota_\tau \to \iota_{\tau'}$	$\Rightarrow$	$\iota_{\tau \to \tau'}$	
$(c \to c'); (d \to d')$	$\Rightarrow$	$(d; c) \to (c'; d')$	$(F1)$
$c \to (d; d^+)$	$\Rightarrow$	$(c \to d); (\iota \to d^+)$	$(F2_1^+)$
$(c^-; c) \to d$	$\Rightarrow$	$(c \to d); (c^- \to \iota)$	$(F2_2^+)$
$c \to (d^-; d)$	$\Rightarrow$	$(\iota \to d^-); (c \to d)$	$(F2_1^-)$
$(c; c^+) \to d$	$\Rightarrow$	$(c^+ \to \iota); (c \to d)$	$(F2_2^-)$
$\mathsf{Func!}; \mathsf{Func?}$	$\Rightarrow$	ι_{Dyn}	$(\phi^{\to})$
$\mathsf{Bool!}; \mathsf{Bool?}$	$\Rightarrow$	ι_{Dyn}	(ϕ^{Bool})

Side condition for rule $F1$: d not properly negative, c not properly positive, c' not properly negative, and d' not properly positive.

Figure 4: Coercion rewriting system for $-/+$-factoring

Proposition 12 *(+/−-factoring of safe coercions)*
For every safe coercion $c : \tau \rightsquigarrow \tau'$ there is $c^+ : \tau \rightsquigarrow \mathsf{Dyn}$ and $c^- : \mathsf{Dyn} \rightsquigarrow \tau'$ such that $\phi \vdash c = c^+; c^-$.

Proof: Clearly $\tau \leq \mathsf{Dyn}$ and $\tau' \leq \mathsf{Dyn}$. Thus, by Proposition 11 there is positive $c^+ : \tau \rightsquigarrow \mathsf{Dyn}$ and negative $c^- : \mathsf{Dyn} \rightsquigarrow \tau'$. By Proposition 9 $c^+; c^-$ is safe and has the same type signature as c. By Theorem 3 two safe coercions with the same type signature are ϕ-convertible. Thus we get $\phi \vdash c = c^+; c^-$. ■

Since the subtype relation has no bottom type there is no corresponding $-/+$-factoring, even under $\phi\psi$-conversion; just consider $\mathsf{Func!}; \mathsf{Bool?}$. Nonetheless we can factor, under ϕ-conversion, any coercion into a triple with a neutral coercion stuck in between a negative and a positive coercion. In fact the following stronger proposition holds.

Proposition 13 *(−/+-factoring of coercions)*
For every coercion c there is a negative c^-, a neutral c^* and a positive c^+ such that $c >_\phi^* c^-; c^*; c^+$.

Proof: Consider the rewriting system in Figure 4 for coercions, which is modulo associativity of composition. It is readily seen to be locally confluent, strongly normalizing and Church-Rosser *w.r.t.* to ϕ-conversion. Furthermore, every normal form is either negative or positive or a product $c^-; c^+$, where c^- is negative and c^+ positive, or a product $c^-; c; c^+$, where c^- (negative) and/or c^+ (positive) may be missing. In the latter case it must be that $c \equiv c'; c''$ or $c''' \to c''''$. Note that c cannot have a proper negative left factor nor a proper positive right factor. We can check by case analysis that c must be a neutral coercion. ■

The rewriting system for $-/+$-factoring actually accomplishes a factoring of a coercion into a maximal negative left factor and simultaneously a maximal positive right factor. This yields the following lemma.

Lemma 14 *(Common factors of pairs of coercions)*
Let $c : \tau \rightsquigarrow \tau_1, d : \tau_1 \rightsquigarrow \tau, c' : \tau \rightsquigarrow \tau_2, d : \tau_2 \rightsquigarrow \tau$ such that $\phi \vdash c; d = c'; d'$.

1. *If d and d' are positive then there exist positive $d^+ : \bar{\tau}$ $conv\tau_1, d'^+ : \bar{\tau} \rightsquigarrow \tau_2$ and coercion $\bar{c} : \tau \rightsquigarrow \bar{\tau}$ for some type $\bar{\tau}$ such that:*

$$\begin{array}{lcl} \phi \vdash c & = & \bar{c}; d^+ \\ \phi \vdash c' & = & \bar{c}; d'^+ \end{array}$$

   Furthermore, τ_1 and τ_2 have a greatest lower bound τ_{12} and the above holds for $\bar{\tau} = \tau_{12}$.

2. *If c and c' are negative then there exist negative $c^- : \tau_1 \rightsquigarrow \bar{\tau}', c'^- : \tau_2 \rightsquigarrow \bar{\tau}'$ and coercion $\bar{d} : \bar{\tau}' \rightsquigarrow \tau'$ for some $\bar{\tau}'$ such that:*

$$\begin{array}{lcl} \phi \vdash d & = & c^-; \bar{d} \\ \phi \vdash d' & = & c'^-; \bar{d} \end{array}$$

   Furthermore, τ_1 and τ_2 have a greatest lower bound τ_{12}, and the above holds for $\bar{\tau}' = \tau_{12}$.

Proof: Since $\phi \vdash c; d = c'; d'$ both sides of the equality have a common reduct $c^-; c^*; c^+$ under the rewriting system of Figure 4. Since the reduction rules leave positive coercions at the right alone it follows that both d and d' must be right factors of c^+ if they are positive. Furthermore since the domain type of c^+ must be a lower bound of both τ_1 and τ_2 the unique positive coercion from the greatest lower bound to τ_1 and τ_2 to τ' is also a right factor of both $c; d$ and $c'; d'$. Similarly, c and c' must be left factors of c^- if they are negative. ■

The following theorem follows from this lemma directly.

Theorem 5 *(Factoring of sets of coercions)*

1. *Let C be a* nonempty *set of coercions with the same range type. If C has the common positive right factors c_1^+, c_2^+ then it has a common positive right factor c^+ such that $c^+ = d_1^+; c_1^+$ and $c^+ = d_2^+; c_2^+$ for some positive d_1^+, d_2^+.*
2. *Let C be a* nonempty *set of coercions with the same domain type. If C has the common negative left factors c_1^-, c_2^- then it has a common negative left factor c^- such that $c^- = c_1^-; d_1^-$ and $c^- = c_2^-; d_2^-$ for some negative d_1^-, d_2^-.*

This theorem is used in Section 7. Note that it does *not* hold for empty sets of coercions. Both Func? and Bool? are negative left factors (with the same range type) of the empty set of coercions; yet, there are no coercions c, d (negative or otherwise) such that $\mathsf{Func?}; c = \mathsf{Bool?}; d$, even under ϕ-conversion.

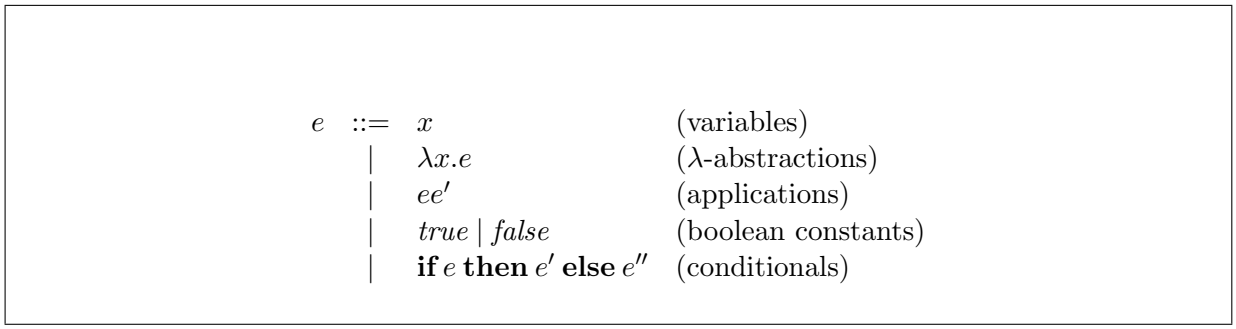

Figure 5: Grammar for implicit dynamically typed λ-calculus

3 Dynamically typed lambda calculus

In this section we introduce the *dynamically typed λ-calculus (dynamic λ-calculus)*. It consists of two languages, an implicit and an explicit language, and a notion of translation from the implicit to the explicit language. This translation is called *completion* since it "completes" an expression in the implicit language by inserting explicit types and coercions into it without changing its syntactic structure. The implicit and explicit languages are presented in this section. The translation between them, in particular what constitutes a "valid" translation, is treated in Section 4.

3.1 Implicit language

The syntax of the implicit language is the untyped λ-calculus with additional forms and constants for other types, in our case for the Boolean truth values. It is given by the grammar in Figure 5. We presuppose a given domain of *variables* and the standard syntactic notions of free and bound variables, open and closed expressions. The generated expressions are those of an applied (untyped) λ-calculus. The purpose of calling them (implicitly) dynamically typed is that we will consider certain expressions as generating *type errors*; *e.g.*, $true(\lambda x.x)$. In following tradition, however, we shall call the expressions of the implicit language *untyped λ-terms*.

Definition 9 *(Untyped λ-terms, equality)*
A (closed) untyped λ-term *is a closed expression derived from e in the grammar of Figure 5. Untyped λ-terms e, e' are* equal, *$e = e'$, if they are identical as strings (or expression trees).*

Note that no conversions, not even α-conversion, are taken into account in the definition of equality of untyped λ-terms.

3.2 Explicit language

The explicit language is an extension of the (statically) typed λ-calculus with the type Dyn and the coercions introduced in Section 2. We call it $\lambda^{\rightarrow}\Delta[\mathsf{Bool}]$, the *(explicit) dynamically typed λ-calculus (with Booleans)*.

The formation rules and equations for $\lambda^{\rightarrow}\Delta[\mathsf{Bool}]$ are given in Figure 6.

Definition 10 *(Dynamically typed λ-term, equality)*

$$\frac{e : \tau \quad [x : \tau']}{\lambda x : \tau'.e : \tau' \to \tau} \qquad \frac{e : \tau' \to \tau \quad e' : \tau'}{ee' : \tau}$$

$$true : \mathsf{Bool} \qquad false : \mathsf{Bool}$$

$$\frac{e : \mathsf{Bool} \quad e' : \tau \quad e'' : \tau}{\mathbf{if}\, e\, \mathbf{then}\, e'\, \mathbf{else}\, e'' : \tau} \qquad \frac{\begin{array}{c} e : \tau \\ c : \tau \rightsquigarrow \tau' \end{array}}{[c]e : \tau'}$$

$$\begin{array}{rcl} [\iota_\tau]e & = & e \\ [c'][c]e & = & [c; c']e \\ [c \to d]\lambda x.e & = & \lambda x.[d]e\{x \mapsto [c]x\} \\ ([c \to d]e)e' & = & [d](e([c]e')) \\ [c]\mathbf{if}\, e\, \mathbf{then}\, e'\, \mathbf{else}\, e'' & = & \mathbf{if}\, e\, \mathbf{then}\, [c]e'\, \mathbf{else}\, [c]e'' \end{array}$$

Figure 6: Formation rules and equations for $\lambda^{\to}\Delta[\mathsf{Bool}]$

$$\begin{array}{lcll} (c; c'); c'' & = & c; (c'; c'') & \\ c; \iota & = & c & \\ \iota; c & = & c & \\ \iota_{\tau \to \tau'} & = & \iota_\tau \to \iota_{\tau'} & \\ (c \to c'); (d \to d') & = & (d; c) \to (c'; d') & \\ [\iota]e & = & e & \\ [c'][c]e & = & [c; c']e & \\ [c \to d]\lambda x.e & = & \lambda x.[d](e\{x \mapsto [c]x\}) & \\ ([c \to d]e)e' & = & [d](e([c]e')) & \\ [c]\mathbf{if}\, e\, \mathbf{then}\, e'\, \mathbf{else}\, e'' & = & \mathbf{if}\, e\, \mathbf{then}\, [c]e'\, \mathbf{else}\, [c]e'' & \\ \mathsf{Func!}; \mathsf{Func?} & = & \iota_{\mathsf{Dyn} \to \mathsf{Dyn}} & (\phi^{\to}) \\ \mathsf{Bool!}; \mathsf{Bool?} & = & \iota_{\mathsf{Bool}} & (\phi^{\mathsf{Bool}}) \\ \mathsf{Func?}; \mathsf{Func!} & = & \iota_{\mathsf{Dyn}} & (\psi^{\to}) \\ \mathsf{Bool?}; \mathsf{Bool!} & = & \iota_{\mathsf{Dyn}} & (\psi^{\mathsf{Bool}}) \end{array}$$

Figure 7: Summary of all coercion and $\lambda\Delta$-term equations

A dynamically typed (closed) λ-term ($\lambda\Delta$-term) *is an expression e such that $e : \tau$ is derivable using the formation rules in Figure 6 and those for coercions in Figures 1 and 2.*

The $\lambda\Delta$-terms $e : \tau, e' : \tau$ are equal*, written $\vdash e = e'$ or simply $e = e'$, if $e = e'$ is derivable using the core coercion equations of Figure 1 and the $\lambda\Delta$-term equations of Figure 6 (but* not *the ϕ- and ψ-equations!).*

As for coercions we assume that the $\lambda\Delta$-terms on both sides of an equation are well-formed and have the same type. For convenience we may omit the type of λ-bound variables in $\lambda\Delta$-terms as well as the type subscripts of identity coercions.

The formation rules for $\lambda\Delta$-terms are those of the simply typed λ-calculus with Booleans plus an additional rule for coercion application, denoted with the special syntax $[c]e$ where c is a coercion applied to $\lambda\Delta$-term e. Coercions are those of Section 2. The equations for coercion application express that coercion composition associates with coercion application and that the identity coercions are indeed identities when applied to a $\lambda\Delta$-term. The significance of the remaining equations becomes apparent in Section 5. For convenience all the equations introduced so far for coercions and $\lambda\Delta$-terms, including the core coercion equations as well as ϕ- and ψ-conversion, are reproduced in Figure 7.

Syntactically coercions are not first-class values: they cannot be passed as arguments to or returned from functions, for example. On the other hand, every coercion $c : \tau \rightsquigarrow \tau'$ can be represented canonically by the function and first-class value $\lambda x : \tau.[c]x : \tau \to \tau'$. In this sense coercions with type signature $\tau \rightsquigarrow \tau'$ are contained in the domain of functions of type $\tau \to \tau'$. Since not *all* (λ-definable) functions are coercions in this sense, however, this containment is proper.

Note that β-, η-, or even α-conversion are *not* part of the dynamically typed λ-calculus, neither in the implicit nor the explicit language. Our intention is to study the coercion-theoretic properties and their implications independently of the (other) properties of the language constructs. This will also make the results independent of the specific semantics of the λ-language; for example, both call-by-name and call-by-value with nonstrict error propagation satisfy ϕ-conversion.

3.3 Conversion and reduction

We extend the notions of conversion and reduction from coercions to $\lambda\Delta$-terms.

Definition 11 *(Conversion and reduction between $\lambda\Delta$-terms)*

Let R, E be sets of coercion equations such as ϕ, ψ, or $\phi\psi$. Consider R as a term rewriting system on coercions by interpreting the equations in it as left-to-right rewriting rules.

1. *We say $\lambda\Delta$-terms e and e' are E-*convertible*, written $E \vdash e = e'$ or $e =_E e'$, if $e = e'$ is derivable by adding E to the equations defining equality of $\lambda\Delta$-terms.*
2. *$\lambda\Delta$-term e R-*reduces in one step to e' under E-conversion*, written $E \vdash e >_R e'$ or $e >_R e'$ if E is empty, if $E \vdash e = \bar{e}, e' = \bar{e}'$ for some $\bar{e}, \bar{e}'$ and $\bar{e}$ reduces to $\bar{e}'$ under R.*
   *$\lambda\Delta$-term e R-*reduces to e' under E-conversion*, $E \vdash e >_R^* e'$, if $E \vdash e = e'$ or $E \vdash e >_R e_1 > \ldots > e_n >_R e'$ for some $e_1, \ldots, e_n$ where $n \geq 0$.*

We would like to extend the results on safety and minimality for coercions to $\lambda\Delta$-terms. There are three crucial difficulties, however:

1. There is no way of simply orienting the two equations

$$[c \to d]\lambda x.e \;=\; \lambda x.[d](e\{x \mapsto [c]x\}) \quad (1)$$
$$([c \to d]e)e' \;=\; [d](e([c]e')) \quad (2)$$

   to obtain a confluent rewrite system, even for the case where the equations are left- and right-linear; that is, if e in Equation 1 contains exactly one occurrence of x.

2. The right-hand sides of Equation 1 and

$$[c]\mathbf{if}\, e\, \mathbf{then}\, e'\, \mathbf{else}\, e'' = \mathbf{if}\, e\, \mathbf{then}\, [c]e'\, \mathbf{else}\, [c]e''$$

   may duplicate a coercion on the left-hand side; *i.e.*, they may be nonlinear.

3. The right-hand side of Equation 1 may "throw away" the coercion c on the left-hand side if e does not contain an occurrence of x.

Note that the type of bound variable x (omitted for reasons of convenience only, but present in all $\lambda\Delta$-terms) may be different on the left- and right-hand sides of Equation 1. This is in contrast to Thatte's quasi-static type system [Tha90] and (the first-order part of) Curien and Ghelli's coercion formulation for System $F_{\leq}$ [CG90]. This entails that, in contrast to their systems, the two rules cannot be simply oriented so as to give a confluent reduction system, even in the absence of ϕ- and ψ-reduction.

The second complication alone leads to a break-down of commutativity of ϕ-reduction and ψ-reduction for $\lambda\Delta$-terms (compare Lemma 1 in Section 2). Consider a $\lambda\Delta$-term e of the form

$$(\lambda x : \mathsf{Dyn}. \ldots [\mathsf{Func?}]x \ldots [\mathsf{Bool?}]x \ldots)(\mathbf{if}\ \ldots\ \mathbf{then}\, [F?; F!]y\, \mathbf{else}\, [F!]z)$$

with exactly two applied occurrences of x and with assumptions $y : \mathsf{Dyn}, z : \mathsf{Dyn} \to \mathsf{Dyn}$. Now, by ψ-reduction e can be reduced to

$$e_1 = (\lambda x : \mathsf{Dyn}. \ldots [\mathsf{Func?}]x \ldots [\mathsf{Bool?}]x \ldots)(\mathbf{if}\ \ldots\ \mathbf{then}\, [\iota_{\mathsf{Dyn}}]y\, \mathbf{else}\, [F!]z),$$

and by ϕ-reduction to

$$e_2 = (\lambda x : \mathsf{Dyn} \to \mathsf{Dyn}. \ldots [\iota_{\mathsf{Dyn}\to\mathsf{Dyn}}]x \ldots [\mathsf{Func!}; \mathsf{Bool?}]x \ldots)(\mathbf{if}\ \ldots\ \mathbf{then}\, [F?]y\, \mathbf{else}\, z).$$

But e_1 and e_2 have no common reduct under $\phi\psi$-reduction. We have $\phi \vdash e_2 >^*_\psi e_1$, but not $\phi \vdash e_1 >^*_\psi e_2$, and thus e_1 is *safer* than e_2. To overcome this problem we shall, preferring safety to efficiency, only ϕ-reduce inside *safe* $\lambda\Delta$-terms, which we obtain by *first* ψ-reducing under ϕ-conversion.

The third complication alone makes ϕ-reduction nonnormalizing if we admit rewriting from the right-hand side to the left-hand side in rule 1.

The second and third complications together break the confluence of ϕ-reduction. Consider for example the three $\lambda\Delta$-terms below.

$$\begin{aligned}
e_0 &\equiv (\lambda x : \mathsf{Dyn}.\mathit{true})(\mathbf{if}\, \mathit{false}\, \mathbf{then}\, [\mathsf{Func!}](\lambda y : \mathsf{Dyn}.y)\, \mathbf{else}\, [\mathsf{Bool!}]\mathit{true} : \mathsf{Bool} \\
e_1 &\equiv (\lambda x : \mathsf{Dyn} \to \mathsf{Dyn}.\mathit{true})(\mathbf{if}\, \mathit{false}\, \mathbf{then}\, (\lambda y : \mathsf{Dyn}.y)\, \mathbf{else}\, [\mathsf{Bool!}; \mathsf{Func?}]\mathit{true} : \mathsf{Bool} \\
e_2 &\equiv (\lambda x : \mathsf{Bool}.\mathit{true})(\mathbf{if}\, \mathit{false}\, \mathbf{then}\, [\mathsf{Func!}; \mathsf{Bool?}](\lambda y : \mathsf{Dyn}.y)\, \mathbf{else}\, \mathit{true} : \mathsf{Bool}
\end{aligned}$$

We have $e_0 >_\phi e_1$ and $e_0 >_\phi e_2$, but there is no common $\lambda\Delta$-term to which both e_1 and e_2 can be ϕ-reduced.[3]

[3]The example above demonstrates that Theorem 2 in [Hen92a] is not correct; it appears to hold for λI-terms, however. See Section 7.

4 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 Dyn; at every program point where a value is *created* (e.g., by a constant or a λ-abstraction) the corresponding tagging operation is inserted; and at every program point where a value is *used* (e.g., by the test in a conditional or when a function is applied), the appropriate check-and-untag operation is inserted. In this fashion the resulting "completed" program satisfies the typing rules of Section 3; *i.e.*, it is a well-formed dynamically typed λ-term.

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 generally results in loss of information (type Dyn gives no information) and specifically in slower execution speed compared to execution without any type operations.[4]

We view a program in the implicit language 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 coercions and their placement in the source program are inferred. 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 reasoning with them and, specifically, performing source-level optimization.

Definition 12 *(Erasure, completion)*
The erasure *of a dynamically typed λ-term e is the untyped λ-term that arises from "erasing" all occurrences of coercions and types from e (including square brackets and colons, of course).*
Conversely, a completion *of an untyped λ-term e is a dynamically typed λ-term whose erasure is e.*

We write $e \longrightarrow e' : \tau$ if e' is a completion of e at τ, and completions of untyped λI-terms are referred to as *$\lambda I\Delta$-terms*.

Since there is generally more than one completion for the same incomplete program we treat the resulting ambiguity as a problem of *coherence* [BTCGS89,CG90] (see Section 5) or *safety* (c.f. [Tha90]; see Section 6) of the set of all the completions.

Note that the "local" translation of untyped λ-terms to dynamically typed λ-terms described at the beginning of this section is a completion in this sense; we shall call it the *canonical completion* of an untyped λ-term and extend it from a translation to type Dyn to a translation to arbitrary types.

Definition 13 *(Canonical completion)*
If $e \longrightarrow_c \overline{e} : \tau$ is derivable in the inference system of Figure 8 then $\overline{e}$ is the canonical completion *of e at type τ.*

It is easy to check that every untyped λ-term has a unique canonical completion at every type.

[4] This presupposes that programs equated by our equality theory execute roughly the same amount of tagging and check-and-untag operations. Depending on the particular semantics and implementation this may or may not be defensible; consider, for example, Equation 1.

$$\frac{e \longrightarrow_c \overline{e} : \mathsf{Dyn} \quad [x \longrightarrow_c x : \mathsf{Dyn}]}{\lambda x.e \longrightarrow_c [\mathsf{Func!}]\lambda x : \mathsf{Dyn}.\overline{e} : \mathsf{Dyn}} \qquad \frac{e \longrightarrow_c \overline{e} : \mathsf{Dyn} \quad e' \longrightarrow_c \overline{e'} : \mathsf{Dyn}}{ee' \longrightarrow_c ([\mathsf{Func?}]\overline{e})\overline{e'} : \mathsf{Dyn}}$$

$$true \longrightarrow_c [\mathsf{Bool!}]true : \mathsf{Dyn} \qquad false \longrightarrow_c [\mathsf{Bool!}]false : \mathsf{Dyn}$$

$$\frac{e \longrightarrow_c \overline{e} : \mathsf{Dyn} \quad e' \longrightarrow_c \overline{e'} : \mathsf{Dyn} \quad e'' \longrightarrow_c \overline{e''} : \mathsf{Dyn}}{\mathbf{if}\, e\, \mathbf{then}\, e'\, \mathbf{else}\, e'' \longrightarrow_c \mathbf{if}\, [\mathsf{Bool?}]\overline{e}\, \mathbf{then}\, \overline{e'}\, \mathbf{else}\, \overline{e''} : \mathsf{Dyn}}$$

$$\frac{e \longrightarrow_c \overline{e} : \mathsf{Dyn}}{e \longrightarrow_c [c]e : \tau} \quad (\text{if } \tau \neq \mathsf{Dyn} \text{ and } c : \mathsf{Dyn} \rightsquigarrow \tau \text{ canonical})$$

Figure 8: Canonical completions of untyped λ-terms

Proposition 15 *(Uniqueness and existence of canonical completions)*

1. *If $e \longrightarrow_c \overline{e'}$ and $e \longrightarrow_c \overline{e''}$ then $\overline{e'} \equiv \overline{e''}$; i.e., $\overline{e'}$ and $\overline{e''}$ are identical.*
2. *For any type τ every untyped λ-term has a canonical completion at τ.*

Proof:

1. It is immediate that every untyped λ-term has a unique canonical completion at Dyn. Uniqueness at any other type follows from uniqueness of canonical coercions (Proposition 4, part 1).
2. Follows from the last rule and the fact that there is a canonical coercion between any two types (Proposition 4, part 2).

■

Intuitively and in a sense made precise in Section 6 the canonical completions *maximize* the use of dynamic type coercions; *i.e.*, they are the most *in*efficient. Note that by Proposition 15 there is a completion for *every* untyped λ-term. Thus *no* untyped λ-term is "rejected" by the dynamic typing discipline.

We illustrate the completion process of untyped λ-terms for the familiar fixpoint combinator Y of Church. The (untyped) Y-combinator is defined by

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

Its canonical translation into the dynamically typed λ-calculus, *i.e.* its canonical completion at Dyn, is

$$\begin{array}{rcl} Y_c & = & [\mathsf{Func!}]\lambda f : \mathsf{Dyn}. \\ & & [\mathsf{Func?}][\mathsf{Func!}](\lambda x : \mathsf{Dyn}.[\mathsf{Func?}]f([\mathsf{Func?}]xx)) \\ & & [\mathsf{Func!}](\lambda y : \mathsf{Dyn}.[\mathsf{Func?}]f([\mathsf{Func?}]yy)). \end{array}$$

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

$$\begin{aligned} Y_m \;=\; & \lambda f : \mathsf{Dyn} \to \mathsf{Dyn}. \\ & (\lambda x : \mathsf{Dyn} \to \mathsf{Dyn}. f(x[\mathsf{Func!}]x)) \\ & (\lambda y : \mathsf{Dyn}. f([\mathsf{Func?}]yy)), \end{aligned}$$

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

5 Coherence

The notion of completion induces a congruence relation on dynamically typed λ-terms and coercions.

Definition 14 *(Completion congruence)*
Dynamically typed λ-terms e', e'' are completion congruent*, written $e' \cong e''$, if they are completions of the same untyped λ-term at the same type; i.e., $e \longrightarrow e' : \tau$ and $e \longrightarrow e'' : \tau$ for some untyped λ-term e and type τ.*

If any two such congruent λ-terms, respectively coercions, are semantically equivalent, we can define the meaning of an untyped λ-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 7. In this section we characterize the equational properties dynamic type coercions must satisfy to yield *coherent* completions; *i.e.*, such that *any* two completions of an untyped λ-term at a given type are *provably* equal.

Theorem 6 *(Coherence of completions, characterization of completion congruence)*
Dynamically typed λ-terms e', e'' are completion congruent if and only if $e =_{\phi\psi} e'$. Furthermore, this equational characterization is irredundant; i.e., it fails for any proper subset of the equations in Figure 7.

Proof: (If) Assume $e' =_{\phi\psi} e''$. By definition of $\phi\psi$-conversion this can only be the case if e' and e'' are well-formed dynamically typed λ-terms having the same type. By inspection of the equations in Figure 7 it can be verified that e' and e'' must have the same erasure. It follows that they are congruent completions; i.e., $e' \cong e''$.

(Only if) We call a dynamically typed λ-term *head coercion free* (c.f. [CG90]) if it is *not* of the form $[c]e'$. Without loss of generality we may assume that coercions are only applied to head coercion free λ-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 $e = [\iota]e$ for $k = 0$. We prove $e' \cong e'' \Rightarrow e' = e''$ by induction on the erasure e of e' and e'' under an arbitrary set of assumptions $x : \tau$.

Basis, I: Let $e' \equiv [c']x : \tau'$ and $e'' \equiv [c'']x : \tau'$ be completions of x under an assumption set containing $x : \tau$ (and no other assumption for x). Then c' and c'' must both have type signature $\tau \rightsquigarrow \tau'$. By Theorem 2 it follows that $c' =_{\phi\psi} c''$ and thus $[c']x =_{\phi\psi} [c'']x$.

Basis, II: If $e \equiv \mathit{true}$ or $e \equiv \mathit{false}$ then similar as above.

Inductive step, I: If $e \equiv \lambda x.f$ then $e' \equiv [c']\lambda x : \tau'.f'$ and $e'' \equiv [c'']\lambda x : \tau''.f''$.

Since there is a coercion from any type to any other type there are coercions $d : \tau'' \rightsquigarrow \tau'$ and $d' : \upsilon' \rightsquigarrow \upsilon''$ such that

$$[d \to d']\lambda x : \tau'.f' = \lambda x : \tau''.[d']f'\{x \mapsto [d]x\}.$$

That is, we have, under the additional assumption $x : \tau''$ the completions $[d']f'\{x \mapsto [d]x\}$ and f'' of f, both of type υ''. By inductive hypothesis we thus get

$$[d']f'\{x \mapsto [d]x\} =_{\phi\psi} f''.$$

Consequently we have

$$[c'']][d \to d']\lambda x : \tau'.f' =_{\phi\psi} [c'']\lambda x : \tau''.f''.$$

Since $(d \to d'; c'')$ and c' both have type signature $\tau' \rightsquigarrow \tau''$ we know by application of Theorem 2 that $(d \to d'; c'') =_{\phi\psi} c'$, and the result follows.

Inductive step, II: If $e \equiv fg$, then $e' \equiv [c'](f'g') : \tau$ and $e'' \equiv [c'']](f''g'') : \tau$ where $f' : \tau' \to \upsilon'$, $f'' : \tau'' \to \upsilon''$.

There exist coercions $d : \tau'' \rightsquigarrow \tau', d^{-1} : \tau' \rightsquigarrow \tau''$ and $d' : \upsilon' \rightsquigarrow \upsilon''$. Since $[d \to d']f'$ and f'' both have type $\tau'' \to \upsilon''$ and are completions of f (under the same set of variable assumptions) we get by induction hypothesis that

$$[d \to d']f' =_{\phi\psi} f''$$

and similarly

$$[d^{-1}]g' =_{\phi\psi} g''.$$

Consequently we have

$$([d \to d']f')([d^{-1}]g') =_{\phi\psi} f''g''.$$

Now,

$$([d \to d']f')([d^{-1}]g') = [d'](f'[d^{-1}; d]g')$$

by rule 2, the second function rule, and

$$[d'](f'[d^{-1}; d]g') =_{\phi\psi} [d'](f'g')$$

because $d^{-1}; d$ has type signature $\tau' \rightsquigarrow \tau'$ and must, by Theorem 2, be $\phi\psi$-convertible to $\iota_{\tau'}$. Putting the last three equations together we have

$$[d'](f'g') =_{\phi\psi} f''g''.$$

Since $d'; c'$ and c'' both have type signature $\tau'' \rightsquigarrow \tau$ they are $\phi\psi$-convertible, which yields

$$[c'][d'](f'g') =_{\phi\psi} [c'']](f''g'')$$

and thus the result.

Induction step, III: If $e \equiv \textbf{if } f \textbf{ then } g \textbf{ else } h$, then $e' \equiv [c']\textbf{if } f' \textbf{ then } g' \textbf{ else } h' : \tau$ and $e'' \equiv [c'']\textbf{if } f'' \textbf{ then } g'' \textbf{ else } h'' : \tau$ with $c' : \tau' \rightsquigarrow \tau$ and $c'' : \tau'' \rightsquigarrow \tau$.

Let d be a coercion with type signature $\tau' \rightsquigarrow \tau''$. Note that $[d]g'$ and g'' are completions of g at τ''; $[d]h'$ and h'' are completions of h also at τ''; and both f', f'' are completions of f at Bool. By induction hypothesis we obtain

$$\begin{array}{rcl} f' & =_{\phi\psi} & f'' \\ [d]g' & =_{\phi\psi} & g'' \\ [d]h' & =_{\phi\psi} & h'' \end{array}$$

and thus

$$\textbf{if } f' \textbf{ then } [d]g' \textbf{ else } [d]h' =_{\phi\psi} \textbf{if } f'' \textbf{ then } g'' \textbf{ else } h''.$$

By the equation for conditionals we have

$$\textbf{if } f' \textbf{ then } [d]g' \textbf{ else } [d]h' = [d]\textbf{if } f' \textbf{ then } g' \textbf{ else } h'.$$

Since $d; c'$ and c'' have the same type signature $\tau'' \rightsquigarrow \tau$ we have $d; c' =_{\phi\psi} c''$ by Theorem 2 and thus

$$[c'][d]\textbf{if } f' \textbf{ then } g' \textbf{ else } h' =_{\phi\psi} [c'']\textbf{if } f'' \textbf{ then } g'' \textbf{ else } h'',$$

which yields the desired result.

For every equation $e = e'$ in Figure 7 it is easy to construct two completion congruent $\lambda\Delta$-terms that cannot be proved equal without $e = e'$. ■

This shows that, *independent of $\beta-$ and η-equality, all* completions of an untyped λ-term have the same behavior if and only if their meanings satisfy *all* the equations in Figure 7, in particular also the ψ-equations.

6 Safety

In the characterization of coherence of completions (Theorem 6) we have used the ψ-equations

$$\begin{array}{rcll} \mathsf{Func?}; \mathsf{Func!} & = & \iota_{\mathsf{Dyn}} & (\psi^{\rightarrow}) \\ \mathsf{Bool?}; \mathsf{Bool!} & = & \iota_{\mathsf{Dyn}} & (\psi^{\mathsf{Bool}}). \end{array}$$

Accordingly, we have

$$[\mathsf{Bool!}; \mathsf{Func?}; \mathsf{Func!}; \mathsf{Bool?}]\mathit{true} =_{\phi\psi} \mathit{true}$$

since

$$\begin{array}{rcl} \mathsf{Bool!}; \mathsf{Func?}; \mathsf{Func!}; \mathsf{Bool?} & =_{\psi} & \mathsf{Bool!}; \iota_{\mathsf{Dyn}}; \mathsf{Bool?} \\ & = & \mathsf{Bool!}; \mathsf{Bool?} \\ & =_{\phi} & \iota_{\mathsf{Bool}} \end{array}$$

and

$$[\iota_{\mathsf{Bool}}]\mathit{true} = \mathit{true}.$$

With ordinary evaluation of coercions, however, this equation is *not* satisfied:

$$[\mathsf{Bool!}; \mathsf{Func?}; \mathsf{Func!}; \mathsf{Bool?}]\mathit{true}$$

is evaluated by first applying the tagging operation Bool! to *true*, then the check-and-untag operation Func? and finally Func! and Bool?. Since the tag of the value after applying Bool! is "Bool", however, the second operation, Func?, generates a type error. In contrast, evaluating *true* by itself yields no type error. So the ψ-equations are not satisfied by ordinary evaluation of coercion application. The ϕ-equations, however, are satisfied: [Func?][Func!]f is evaluated by tagging the value of f (which must be a function) with the type constructor $\rightarrow$, and Func? will check for the presence of $\rightarrow$, find it and return the value of f; so the net effect is the same as returning the value of f.

When completing untyped λ-terms we have three possibilities:

1. Allow arbitrary completions, retain ordinary 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 ordinary 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. 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; *i.e.*, completion coherence is a nonissue since there is no notion of implicit language or completion in the first place.

We can accomplish Option 2 if coercions are not evaluated until a value is used (as a function in an application or as a Boolean in the test of a conditional). In this way every dynamic type coercion 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 ordinary coercion evaluation is standard, more efficient, and reports type errors earlier we adopt Option 3. Notice that with ordinary coercion evaluation $C[tc?;tc!]$ generates a type error or yields the same value as $C[\iota_{\mathsf{Dyn}}]$ for *any* context C; never a different (proper) value. Intuitively, an inequation $\phi \vdash e' >^*_\psi e''$ expresses that, in any context, if e' returns a proper value (not a type error) then e'' returns the same value, but e'' may return a proper value (or loop) when e' generates a type error. In this sense ψ-reduction under ϕ-conversion is a proof-theoretic analogue to Thatte's semantic "wrongness" relation in a fixed denotational interpretation [Tha90].

On this background we extend the notion of safety introduced for coercions in Section 2 to $\lambda\Delta$-terms. Recall the notions of conversion and reduction for λ-terms introduced in Section 3.

Definition 15 *(Safe completions)*
A completion e' of e at type τ is safe *if for every completion e'' of e at τ we have $\phi \vdash e'' >^*_\psi e'$.*

Intuitively, a safe completion generates at most as many type errors as any other completion at the same type; *i.e.*, it does not generate any *avoidable* type errors. This does *not* mean,

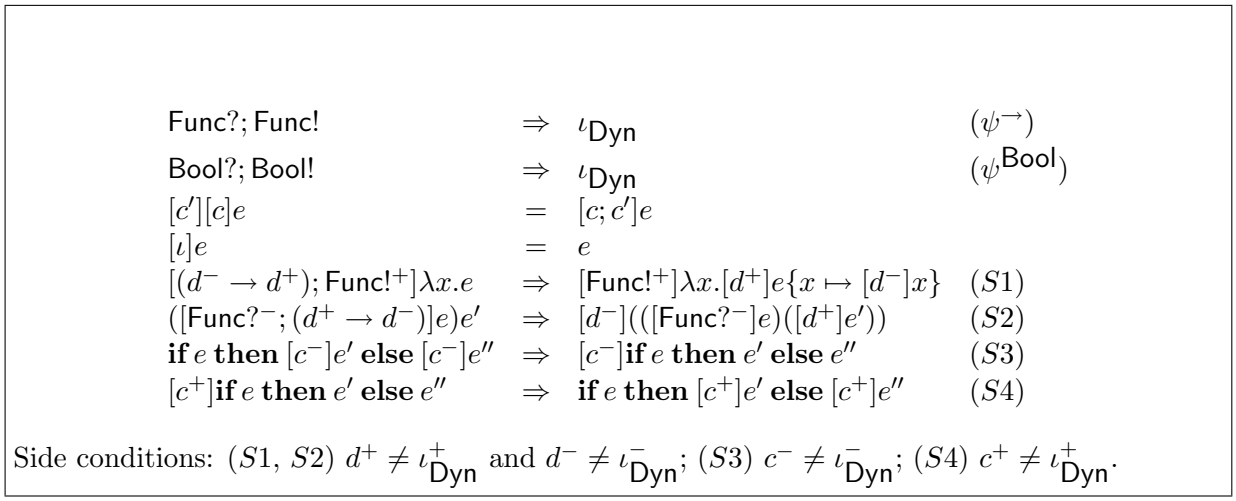

Figure 9: Safety rewriting system

however, that it does not generate any type errors whatsoever. Most importantly, for safe completions ordinary and simplificational coercion evaluation behave equivalently. So by restricting ourselves to safe completions we reap the benefits of combining the efficiency and simplicity of ordinary coercion evaluation with unambiguous semantics and still retain a great degree of freedom of choosing amongst different *safe* completions. We shall exploit this degree of freedom in Section 7.

In analogy to Lemma 7, which states that canonical coercions are safe, we can show that canonical completions, as defined in Figure 8, are safe.

Lemma 16 *(Safety of canonical completions)*
Every untyped λ-term has a safe completion.

Proof: By induction on untyped structure of λ-term e. (This follows the proof of Theorem 6: e' is assumed to be the canonical completion at τ, e'' any other completion at τ. It must be verified that whenever two coercions c', c'', one for e', the other for e'', have the same type signature then $c'' >^*_\psi c'$. Details left to the reader.) ■

Furthermore, Lemma 6, which states that any two coercions that are ψ-reducible to each other under ϕ-conversion are already ϕ-convertible, can also be extended to $\lambda\Delta$-terms, although with great technical complications due to the first two problems we mentioned at the end of Section 3.

Consider the rewriting system in Figure 9. Coercions superscripted with $+$ are positive with type signature $\tau \rightsquigarrow \mathsf{Dyn}$ and those superscripted with $-$ are negative with type signature $\mathsf{Dyn} \rightsquigarrow \tau'$ for some τ, τ'. The rewriting system is modulo ϕ-conversion on coercions and the equations for coercion application.

Lemma 17 *(Safety rewriting system properties)*
The safety rewriting system of Figure 9 extends the oriented ψ-reductions $\mathsf{Func?};\mathsf{Func!} \Rightarrow \iota_{\mathsf{Dyn}}$ *and* $\mathsf{Bool?};\mathsf{Bool!} \Rightarrow \iota_{\mathsf{Dyn}}$ *such that:*

1. *its reflexive, symmetric, transitive and compatible closure is $\phi\psi$-conversion;*

2. *it is locally confluent (weakly Church-Rosser);*
3. *the ψ-reductions commute with the other reduction rules; and*
4. *it is Noetherian (strongly normalizing).*

Proof:

1. All the rewrite rules are valid equalities *w.r.t.* $\phi\psi$-conversion. Furthermore, for every coercion $c : \tau \rightsquigarrow \tau'$ we have $c \Rightarrow c^+; c^-$ for some positive coercion $c^+ : \tau \rightsquigarrow \mathsf{Dyn}$ and negative coercion $c^- : \mathsf{Dyn} \rightsquigarrow \tau'$ by Proposition 12 in Section 2. Thus all equations of Figure 7 are contained in the symmetric-transitive closure of the rewriting rules.

2. Consider a coercion application $[c]e$. Without loss of generality we may assume that c is the only coercion at this "point". Note that a non-ψ-rule can rewrite c only if it is of the form $c^+; c^-$ where c^+ is positive and has range type Dyn, and c^- is negative. But then, by Proposition 12, no ψ-rule is applicable in c. Thus no critical pair arises from a ψ and a non-ψ-rule. Furthermore, ψ-reduction on coercions is locally confluent modulo ϕ-conversion (follows from Theorem 3). Finally it is easy to see that the non-ψ-rules by themselves have no critical pair. Thus the safety rewriting system is locally confluent (modulo the coercion and coercion application equations).

3. Since there are no critical pairs involving a ψ-rule and a non-ψ-rule these commute with the remaining rules.

4. Let $[c]e'$ be the coercion applied to a subexpression in the underlying untyped λ-term with $c : \tau \rightsquigarrow \tau'$. Without loss of generality we may assume that this is the only coercion applied to this subexpression. The coercion rewritings due to the ψ-rules terminate (Lemma 2), and every λ-term rewriting rule properly increases τ or τ' *w.r.t.* to the subtyping order $\leq$. Since $\leq$ is finitely ascending the λ-term rewriting steps can only be applied a finite number of times.

■

This lemma enables application of the proof method used for Lemma 6 to show that ψ-reduction is conservative over ϕ-conversion also for λ-terms.

Theorem 7 *(Conservative extension of ϕ-conversion for λ-terms)*
Let e, e' be dynamically typed λ-terms. Then $\phi \vdash e >^*_\psi e'$ and $\phi \vdash e' >^*_\psi e$ if and only if $e =_\phi e'$.

Proof: Analogous to the proof of Lemma 6 using the properties stated in Lemma 17. (Extending the technique of Lemma 6 doesn't seem to pan out. Requires the technique used for minimal completions, even more complicated. Don't see how to do this without throwing complicated graph theory at it. Too complicated.) Let us denote the nonlinear rules

$$\begin{array}{lcl} [c \to d]\lambda x.e & = & \lambda x.[d](e\{x \mapsto [c]x\}) \\ [c]\mathbf{if}\, e\, \mathbf{then}\, e'\, \mathbf{else}\, e'' & = & \mathbf{if}\, e\, \mathbf{then}\, [c]e'\, \mathbf{else}\, [c]e'' \end{array}$$

by NL and the linear rule

$$([c \to d]e)e' = [d](e([c]e'))$$

by L. Below shall consider R-reduction always under L-conversion where R is NL, ϕ or ψ.
The result follows by:

1. R_1-reduction commutes with R_2-reduction under L-conversion where R_1, R_2 are NL, ϕ, or ψ. (This is not true without L-conversion.)

2. $NL\phi\psi$-reduction is strongly normalizing under L-conversion.

3. Diagram chasing as in the proof of Lemma 6.

■

This implies immediately that safe completions at the same type are ϕ-convertible.

Corollary 18 *(Uniqueness of safe completions up to ϕ-conversion)*
If both e', e'' are safe completions of an untyped λ-term at the same type then $e' =_\phi e''$.

7 Minimal completions

As we have seen, the canonical completion of an untyped λ-term is safe. But it is also inefficient. In this section we define a general reduction-theoretic criterion for discussing which completion is operationally "better" than another by extending ϕ-reduction to λ-terms. Intuitively, if we have $e' >^*_\phi e$ then e and e' are semantically equivalent, *i.e.* $e =_\phi e'$, but e has at most as many dynamic type coercions as e' and possibly fewer.

Definition 16 *(Minimal completion)*
A completion e' of untyped λ-term e at type τ is minimal *if*

1. *it is safe;*
2. *for every safe completion e'' of e at τ we have $e'' >^*_\phi e'$.*

We have already seen in Section 3 that arbitrary untyped λ-terms do not generally have minimal completions. Below we shall see that this appears to be entirely due to λ-abstraction where the bound variable does not occur in its body. Within restricted classes of completions such as C_{pf} — in which only primitive coercions are permitted, tagging operations may only be applied at data creation points and check-and-untag operations at data use points — minimal completions exist for all untyped λ-terms [Hen92a].[5] There are efficient algorithms for computing such restricted minimal completions [Hen92a] that have applications in the optimization of run-time typed languages such as Scheme, Common LISP, SETL and others [Hen92b].

In Section 6 we were able to devise a rewriting system with only harmless critical pairs by orienting and restricting the $\lambda\Delta$-term equations according to polarity of coercions without losing "completeness" of the resulting rewriting system *w.r.t.* $\phi\psi$-conversion on $\lambda\Delta$-terms. We shall apply an analogous method to prove that all λI-terms have minimal completions that are unique modulo equality (of $\lambda\Delta$-terms) assuming the rewriting system we devise is strongly normalizing.

Consider the rewriting rules in Figure 10 for λ-terms. These are modulo coercion equality, the equations $[\iota]e = e$ and $[c'][c]e = [c; c']e$ for coercion application, and the $\lambda\Delta$-equations, but restricted to neutral coercions.

Lemma 19 *(Characterization of ϕ-conversion)*
The reflexive, symmetric, transitive and compatible closure of the minimization rewriting system (Figure 10) gives ϕ-conversion.

[5] Note that [Hen92a] claims that *all* untyped λ-terms have minimal (general) completions, which is not correct!

$\mathsf{Func!};\mathsf{Func?}$	$\Rightarrow$	$\iota_{\mathsf{Dyn}\to\mathsf{Dyn}}$	$(\phi^{\to})$
$\mathsf{Bool!};\mathsf{Bool?}$	$\Rightarrow$	ι_{Bool}	
$[c'][c]e$	$=$	$[c;c']e$	
$[\iota]e$	$=$	e	
$[d^+ \to d^-]\lambda x.e$	$\Rightarrow$	$\lambda x.[d^-]e\{x \mapsto [d^+]x\}$	$(M1^-)$
$\lambda x.[d^+]e\{x \mapsto [d^-]x\}$	$\Rightarrow$	$[d^- \to d^+]\lambda x.e$	$(M1^+)$
$\lambda x.[d_2^*]e\{x \mapsto [d_1^*]x\}$	$=$	$[d_1^* \to d_2^*]\lambda x.e$	$(M1^*)$
$([d^- \to d^+]e)e'$	$\Rightarrow$	$[d^+](e([d^-]e'))$	$(M2^+)$
$[d^-](e([d^+]e'))$	$\Rightarrow$	$([d^+ \to d^-]e)e'$	$(M2^-)$
$([d_1^* \to d_2^*]e)e'$	$=$	$[d_2^*](e([d_1^*]e'))$	$(M2^*)$
$\mathbf{if}\, e\, \mathbf{then}\, [c^+]e'\, \mathbf{else}\, [c^+]e''$	$\Rightarrow$	$[c^+]\mathbf{if}\, e\, \mathbf{then}\, e'\, \mathbf{else}\, e''$	$(M3^+)$
$[c^-]\mathbf{if}\, e\, \mathbf{then}\, e'\, \mathbf{else}\, e''$	$\Rightarrow$	$\mathbf{if}\, e\, \mathbf{then}\, [c^-]e'\, \mathbf{else}\, [c^-]e''$	$(M3^-)$
$\mathbf{if}\, e\, \mathbf{then}\, [c^*]e'\, \mathbf{else}\, [c^*]e''$	$=$	$[c^*]\mathbf{if}\, e\, \mathbf{then}\, e'\, \mathbf{else}\, e''$	$(M3^*)$

Side conditions: $(M1^-,\ M2^-)$ $d^+ \to d^-$ properly negative; $(M1^+,\ M2^+)$ $d^- \to d^+$ properly positive; $(M3^+)$ c^+ properly positive; $(M3^-)$ c^- properly negative.

Figure 10: Minimization rewriting system

Proof: Every coercion can be factored into $c^-; c^*; c+$ (Proposition 13); in particular, a coercion $c \to d$ can be factored into $(c_1^+ \to c_2^-); (c_1^* \to c_2^*); (c_1^- \to c_2^+)$. Thus every $\lambda\Delta$-term equation is in the symmetric, transitive closure of the three corresponding rewriting rules in Figure 10. ■

Lemma 20 *(Local confluence of minimization rewriting system)*
The minimization rewriting system in Figure 10 is locally confluent (modulo the equations) for $\lambda I\Delta$-terms.

Proof: Consider a coercion application $[c]e$. Without loss of generality we may assume that c is the only coercion at this "point". A non-ϕ rewriting rule applies only at $[c]e$ if $c = c^-; c'$ or $c = c''; c^+$ where c^+ is a proper positive coercion and c^- a proper negative coercion. If a ϕ-rule is applicable in c then it can only be applicable in c', respectively c''. Thus a ϕ-rule and a non-ϕ-rule cannot give rise to a critical pair.

However, the same non-ϕ-rule may be applicable, only with two different negative left, respectively positive right factors. If there is at least *one* coercion being rewritten in this fashion then Theorem 5 guarantees that the critical pair has a common reduct. All rules but $M1^+$ satisfy this property for all $\lambda\Delta$-terms, and $M1^+$ satisfies it for $\lambda I\Delta$-terms since every bound variable must have at least one applied occurrence. (Note, however, that rule $M1^+$ applied to a λ-expression $\lambda x.e$ with no occurrence of x in e may generate a nonconfluent critical pair; *c.f.*, the example at the end of Section 3.)

Furthermore, ϕ-reduction is confluent modulo coercion equality (follows from Lemma 3).

Finally, there are only two critical pairs arising from applying two different rewriting rules to:

1. $[d^+ \to d^-]\lambda x.[c^+]e\{x \mapsto [c{-}]x\}$ n be rewritten to

$$e_1 \equiv \lambda x.[d^-][c^+]e\{x \mapsto [c^-][d^+]x\}$$

using rule $M1^-$ and to

$$e_2 \equiv [d^+ \to d^-][c^- \to c^+]\lambda x.e$$

using rule $M1^+$. Now, $[d^-][c^+]e\{\ldots\} = [c^+; d^-]e\{\ldots\}$ and, by Proposition 13, $c^+; d^-$ can be ϕ-reduced to $c_1^-; c_1^*; c_1^+$ with the indicated polarities. Similarly, $[c^-][d^+]x = [d^+; c^-]x$ with $d^+; c^- >_\phi^* c_2^-; c_2^*; c_2^+$. Thus we have

$$e_1 >_\phi^* \lambda x.[c_1^-; c_1^*; c_1^+]e\{x \mapsto [c_2^-; c_2^*; c_2^+]x\} = \lambda x.[c_1^+][c_1^*][c_1^-]e\{x \mapsto [c_2^+][c_2^*][c_2^-]x\}$$

which in turn reduces to

$$\bar{e}_1 \equiv [c_2^- \to c_1^+]\lambda x.[c_1^*][c_1^-]e\{x \mapsto [c_2^+][c_2^*]x\}$$

using rule $M1^+$. For e_2 we have

$$e_2 >_\phi^* [(c_2^-; c_2^*; c_2^+) \to (c_1^-; c_1^*; c_1^+)]\lambda x.e = [c_2^- \to c_1^+][c_2^* \to c_1^*][c_2^+ \to c_1^-]\lambda x.e$$

which reduces by rule $M1^-$ to

$$\bar{e}_2 \equiv [c_2^- \to c_1^+][c_2^* \to c_1^*]\lambda x.[c_1^-]e\{x \mapsto [c_2^+]x\}.$$

Finally $\bar{e}_1 = \bar{e}_2$ by virtue of equation $M1^*$ and thus e_1 and e_2 have a common reduct.

2. $[d^-]([c^- \to c^+]e[d^+]e')$ can be rewritten using rules $M2^+$ and $M2^-$. By argument analogous to the above they have a common reduct, too.

Using Proposition 13 the resulting critical pairs can be reduced to a common reduct. ■

The same rule that caused problems with local confluence, $M1^+$, applied to non-$\lambda I\Delta$-terms is also the cause that the rewriting system is not normalizing. This is due to the fact that the subtype relation has infinite descending chains. Thus starting with $\lambda x : \mathsf{Dyn}.e$ where x does not occur in e we can create an infinite reduction path. It has the form $[c_1 \to \iota]\lambda x : \tau_1.e \Rightarrow \ldots \Rightarrow [c_i \to \iota]\lambda x : \tau_i.e \Rightarrow \ldots$ where $\mathsf{Dyn} = \tau_1 > \tau_2 > \ldots > \tau_i > \ldots$. For $\lambda I\Delta$-terms, however, the rewriting system appears to be strongly normalizing. Unfortunately we have been unable to construct a convincing argument so far. Nonetheless we conjecture:

Conjecture 1 *(Strong normalization of minimization rewriting system)*
There is no infinite reduction path $e_1 \Rightarrow e_2 \Rightarrow \ldots e_i \ldots$ in the rewriting system of Figure 10 if e_1 is a $\lambda I\Delta$-term.

Together with local confluence strong normalization implies the existence of minimal completions for all λI-terms:

Theorem 8 *(Existence of minimal completions)*
Every untyped λI-erm has a minimal completion at every type if the minimization rewriting system in Figure 10 is strongly normalizing.

Using the method of proving Lemma 6 we obtain that, for $\lambda I\Delta$-terms, ϕ-reduction is conservative over equality.

Theorem 9 *(Conservative extension of equality)*
Let e, e' be dynamically typed λI-terms and assume that the minimization rewriting system in Figure 10 is strongly normalizing. Then $e >^*_\phi e'$ and $e' >^*_\phi e$ if and only if $e = e'$.

Thus we have in particular:

Corollary 21 *(Uniqueness of minimal completions)*
If e', e'' are safe completions of an untyped λ-term e at the same type then $e' = e''$ (assuming strong normalization of the rewriting system in Figure 10).

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 Mycroft [Myc83,Myc84], Abadi, Cardelli, Pierce, Plotkin and Rémy [ACPP89,ACPP91,ACPR92], 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 different and unexpected behavior (*c.f.* remarks by Thatte [Tha90]). Furthermore, 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 Dyn functioning as the "top" type. On the surface this is similar to the partial typing discipline introduced by Thatte [Tha88]. In our case the resulting subtype theory is *co*variant in the first argument of the type constructor $\rightarrow$, however, whereas in it is contravariant partial typing . If we define positive coercions to be coercions containing only tagging operations and no check-and-untag operations (as in [Hen92a]) the resulting subtype system is weaker than partial typing. In this case $\tau \rightarrow \mathsf{Dyn} \leq \mathsf{Dyn}$ only holds for $\tau \equiv \mathsf{Dyn}$ whereas it holds for all τ in partial typing. Thatte [Tha88] originally investigated type checking for partial typing where applications of the subtype rule are only allowed at function applications. He characterized the typability problem as a problem of solving subtyping constraints, but left its decidability open. (Note that λ-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 $F_{\leq}$.) This problem has been shown to be decidable by O'Keefe and Wand [OW91] and to be in polynomial time by [KPS92]. (By the lower bound method presented in [HM91] it is easily seen to be hard for P.)

Thatte introduced the notion of explicit positive and negative coercions in his quasi-static type discipline [Tha90]. The positive (negative) coercions have type signature $\tau \rightsquigarrow \tau'$ ($\tau' \rightsquigarrow \tau$) where $\tau \leq \tau'$ in the partial type hierarchy (see above). Positive coercions may be placed anywhere, but negative coercions can only be placed at data 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 so-called convergent internal expression 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 the type inference problem for partial typing and our completion problem is more general than quasi-static typing in that programs do not require type declarations for variables and that arbitrary coercions may be inserted at any place.

The notion of coherence arises in coercion interpretations of subtyping. Breazu-Tannen, Cardelli, Coquand, Gunter and Scedrov [BTCGS91] 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. Our equational characterization of coherence extends the first-order subset of $F_{\leq}$ with negative coercions and a rule relating λ-terms to each other that have different types bound to the same variable.

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 only (no induced coercions) in which positive coercions may only occur at creation points (λ-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 and check-and-untag operations may "spread" to every point reachable from a single tagging operation.

Cartwright and Fagan [CF91,CF92] present a very ambitious "ideal" extension of ML's type inference system with regular recursive types, union types and implicit subtyping based on extension of unions as well as a "practical" type system, which is a workable, simpler variant of the ideal type system. 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 are represented as union types, which are encoded using a type representation scheme pioneered by Remy [Rem89] for record-based inheritance. The type inference algorithm is based on Milner's Algorithm W operating on Remy-encoded types.

The "pure" dynamically typed λ-terms may be readily seen to be the internal language used in the construction of categorical models for the pure untyped λ-calculus. This is based on the observation that what we have termed the canonical completion of an untyped λ-term at type Dyn is a faithful translation of β-equality ($\beta\eta$-equality) in the untyped λ-calculus to $\beta\phi$-conversion ($\beta\eta\phi\psi$-conversion) in the dynamically typed λ-calculus [Han88]. Our results imply that, for $\beta\eta$-equality, we may choose *any* completion instead of the canonical one, and for β-equality we may choose *any safe* completion.

9 Conclusion and future work

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.

This paper treats the proof-theoretic aspects (equality and reduction theory of dynamic type coercions) of the (first order) dynamically typed λ-calculus. In companion papers we shall:

1. study the model theory of dynamically typed λ-calculus, including different denotational and operational semantics;

2. the algorithmic and complexity-theoretic aspects of computing minimal completions, including algorithms for various restricted, but practically relevant classes of safe completions;

3. applications of dynamic type inference; in particular, to global tagging optimization in realistic run-time typed languages, and to intelligent type error recovery in type inference for ML-like languages.

The most pressing open problem is a proof of strong normalization of Conjecture 1.

Neutral coercions are just "indirect" representations of coercions that always generate type errors. We could have added a type *Error* with negative coercions from any type and positive coercions to any type. This would make the subtype hierarchy a complete lattice. The operational interpretation of any coercion to *Error* is "generate an error". Equating $\mathsf{Func!};\mathsf{Bool?}$ with a composition of first a negative coercion to **Error** and then the positive "injection" of this error element in the Booleans would enable a simplification of the proofs in Section 7 and generalization of Theorem 8 to completions of all λ-terms, not just λI-terms.

Furthermore, the extension of dynamic typing to an ML-polymorphic and a second order polymorphic type discipline should be investigated. Automatically inferred polymorphic minimal completions may lead to novel implementation techniques and optimizations for conventional run-time typed languages. An extension to ML-polymorphism requires a let-construct that can be parameterized with coercions. For a practical adaptation of dynamic typing to such a polymorphic type discipline, however, the problem of minimizing the number of coercion parameters needs to be addressed.

To estimate the practicality of dynamic typing we have implemented a simple, but very efficient completion algorithm for a substantial subset of IEEE Scheme [Hen92b]. It computes minimal completions in the completion class C_{pf} (see [Hen92a]). Even in such a simple completion class more than half of the tagging and check-and-untag operations that would naively be executed can be eliminated in Scheme programs. A Scheme translator/compiler based on dynamic type inference and type-specific implementation of Scheme primitives is currently under way at DIKU.

Acknowledgements

This paper would not have reached the form it has without the intense discussions I have had with Jesper Jørgensen on coercion calculi. Satish Thatte's questions, insights, comments and corrections have been valuable in detecting several mistakes in various stages of my work on dynamic typing. I am especially greatful for his inquisitive questions that led to the definition of safety. I am also grateful for helpful discussions with Corky Cartwright, Mike Fagan, Matthias Felleisen, Benjamin Pierce, Andrew Wright and members of the TOPPS group at DIKU.

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.

[ACPR92] M. Abadi, L. Cardelli, B. Pierce, and D. Rémy. Dynamic typing in polymorphic languages. In *Proc. ACM SIGPLAN Workshop on ML and its Applications (ML), San Francisco, California*, pages 92–103, June 1992.

[BTCGS89] 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.

[BTCGS91] 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.

[CF92] R. Cartwright and M. Fagan. Soft typing. Draft. Corrected revision of SIGPLAN '91 paper, Dec. 1992.

[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.

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

[Han88] J. Hannan. Embedding the untyped λ-calculus in a simply typed λ-calculus. Unpublished manuscript, Oct. 1988.

[Hen92a] F. Henglein. Dynamic typing. In *Proc. European Symp. on Programming (ESOP), Rennes, France*, pages 233–253. Springer, Feb. 1992. Lecture Notes in Computer Science, Vol. 582.

[Hen92b] F. Henglein. Global tagging optimization by type inference. In *Proc. LISP and Functional Programming (LFP), San Francisco, California*, June 1992. To appear.

[Hen93] Fritz Henglein. Dynamic typing: Syntax and proof theory. TOPPS Report D-163, DIKU, University of Copenhagen, March 1993.

[HM91] F. Henglein and H. Mairson. The complexity of type inference for higher-order typed lambda calculi. In *Proc. 18th ACM Symp. on Principles of Programming Languages (POPL), Orlando, Florida*, pages 119–130. ACM Press, Jan. 1991.

[Hue80] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. Assoc. Comput. Mach.*, 27(4):797–821, Oct. 1980.

[KPS92] Dexter Kozen, Jens Palsberg, and Michael Schwartzbach. Efficient inference of partial types. Technical Report DAIMI PB-394, DAIMI, Aarhus University, April 1992.

[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.

[Myc83] A. Mycroft. Dynamic types in statically typed languages. Unpublished manuscript, preliminary draft, 1983.

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

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

[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.

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

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