



ELSEVIER

Science of Computer Programming 22 (1994) 197–230

Science of
Computer
Programming

Dynamic typing: syntax and proof theory^{*}

Fritz Henglein^{**}

University of Copenhagen, Universitetsparken 1, 2100 Copenhagen Ø, Denmark

Received July 1992; revised March 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 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 λ -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

^{*} This research has been partially supported by Esprit BRA 3124, Semantique.

^{**} E-mail: henglein@diku.dk.

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 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 $tc(\text{Dyn}, \dots, \text{Dyn})$ to Dyn by pairing them with their type. For example, the coercion Func! maps a function f of type $\text{Dyn} \rightarrow \text{Dyn}$ to Dyn. For every tagging operation $tc!$ there is a corresponding *check-and-untag operation* $tc?$ that maps elements of type Dyn to $tc(\text{Dyn}, \dots, \text{Dyn})$: 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 $c_1 : \tau_1 \rightsquigarrow \tau'_1$ and $c_2 : \tau_2 \rightsquigarrow \tau'_2$ are coercions then $c_1 \rightarrow c_2 : (\tau'_1 \rightarrow \tau_2) \rightsquigarrow (\tau_1 \rightarrow \tau'_2)$ is an *induced coercion* that operates on functions f of type $\tau'_1 \rightarrow \tau_2$. It returns a function

of type $\tau_1 \rightarrow \tau'_2$, 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:

$$tc!; tc? = \iota \quad (\phi)$$

$$tc?; tc! = \iota \quad (\psi)$$

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

- (1) 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*;
- (2) 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 Section 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 [15] 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 [13].

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

$$\tau ::= \text{Bool} \mid \tau' \rightarrow \tau'' \mid \text{Dyn}$$

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 rule² 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.

2.1. Coercion constructions and equality

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

Two coercions c and 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

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.

$$\begin{array}{c}
l_\tau : \tau \rightsquigarrow \tau \qquad \frac{c : \tau \rightsquigarrow \tau' \quad c' : \tau' \rightsquigarrow \tau''}{(c; c') : \tau \rightsquigarrow \tau''} \\
\\
\frac{c : \tau \rightsquigarrow \tau' \quad d : v \rightsquigarrow v'}{(c \rightarrow d) : (\tau' \rightarrow v) \rightsquigarrow (\tau \rightarrow v')} \\
\\
c; (c'; c'') = (c; c'); c'' \\
l_\tau; c = c \\
c; l_\tau = c \\
(c' \rightarrow d); (c \rightarrow d') = (c; c') \rightarrow (d; d') \\
l_{\tau \rightarrow \tau'} = l_\tau \rightarrow l_{\tau'}
\end{array}$$

Fig. 1. Core coercion formation rules and equations.

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 \rightarrow . The corresponding equations for \rightarrow express that \rightarrow 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 Fig. 1.

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

The formation rules and the coercion equations in Fig. 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 and c' are literally identical then we write $c \equiv c'$. Thus $c \equiv c'$ implies $c = c'$ as long as c is a legal coercion expression, but $c = c'$ does not generally imply $c \equiv c'$.

$\text{Func!} : (\text{Dyn} \rightarrow \text{Dyn}) \rightsquigarrow \text{Dyn}$	$\text{Func?} : \text{Dyn} \rightsquigarrow (\text{Dyn} \rightarrow \text{Dyn})$
$\text{Bool!} : \text{Bool} \rightsquigarrow \text{Dyn}$	$\text{Bool?} : \text{Dyn} \rightsquigarrow \text{Bool}$
$\text{Func!}; \text{Func?} = \iota_{\text{Dyn} \rightarrow \text{Dyn}} \quad (\phi^{\rightarrow})$	
$\text{Bool!}; \text{Bool?} = \iota_{\text{Bool}} \quad (\phi^{\text{Bool}})$	
$\text{Func?}; \text{Func!} = \iota_{\text{Dyn}} \quad (\psi^{\rightarrow})$	
$\text{Bool?}; \text{Bool!} = \iota_{\text{Dyn}} \quad (\psi^{\text{Bool}})$	

Fig. 2. Formation rules and equations for dynamic type coercions.

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* $tc! : tc(\text{Dyn}, \dots, \text{Dyn}) \rightsquigarrow \text{Dyn}$ and a corresponding *untagging coercion* $tc? : \text{Dyn} \rightsquigarrow tc(\text{Dyn}, \dots, \text{Dyn})$. 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 , then it 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 $tc(\text{Dyn}, \dots, \text{Dyn})$. Note, however, that $tc!$ can be composed with induced coercions to give a coercion from $tc(\tau_1, \dots, \tau_k)$ to Dyn for arbitrary τ_1, \dots, τ_k .

In our type language we have type constructor \rightarrow and primitive type Bool . The corresponding dynamic type coercions are denoted by Func! , Func? , Bool! and Bool? . See the formation rules in Fig. 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 ϕ^{\rightarrow} and ϕ^{Bool} in Fig. 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 Fig. 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., $\text{Func?}; \text{Func!}$ 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., $\text{Func!}; \text{Func?}$ 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 and 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} and \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 $E \vdash c >_R c_1 >_R \dots >_R c_n >_R c'$ for some c_1, \dots, 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 4 (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 $c'' >_\phi c'''$.*

Lemma 5 (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 $c >_R c' >_R c'' \dots$.*

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

Lemma 6 (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 $c'' >_R^* c'''$.*

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

Theorem 7 (Church–Rosser, uniqueness of $\phi\psi$ -normal forms). *For every pair of coercions c and 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 and τ' .

$\iota_{\text{Dyn}} : \text{Dyn} \rightsquigarrow \text{Dyn}$	$\text{Func!} : (\text{Dyn} \rightarrow \text{Dyn}) \rightsquigarrow \text{Dyn}$
$\text{Func?} : \text{Dyn} \rightsquigarrow (\text{Dyn} \rightarrow \text{Dyn})$	$\iota_{\text{Bool}} : \text{Bool} \rightsquigarrow \text{Bool}$
$\text{Bool!} : \text{Bool} \rightsquigarrow \text{Dyn}$	$\text{Bool?} : \text{Dyn} \rightsquigarrow \text{Bool}$
$\frac{c : \tau \rightsquigarrow \tau' \quad d : v \rightsquigarrow v'}{(c \rightarrow d) : (\tau \rightarrow v) \rightsquigarrow (\tau \rightarrow v')}$	$\frac{c : (\tau \rightarrow v) \rightsquigarrow (\text{Dyn} \rightarrow \text{Dyn})}{(c; \text{Func!}) : (\tau \rightarrow v) \rightsquigarrow \text{Dyn}}$
$\frac{d : (\text{Dyn} \rightarrow \text{Dyn}) \rightsquigarrow (\tau \rightarrow v)}{(\text{Func?}; d) : \text{Dyn} \rightsquigarrow (\tau \rightarrow v)}$	$\frac{c : (\tau \rightarrow v) \rightsquigarrow \text{Dyn} \quad d : \text{Dyn} \rightsquigarrow \text{Bool}}{(c; d) : (\tau \rightarrow v) \rightsquigarrow \text{Bool}}$
$\frac{c : \text{Bool} \rightsquigarrow \text{Dyn} \quad d : \text{Dyn} \rightsquigarrow (\tau \rightarrow v)}{(c; d) : \text{Bool} \rightsquigarrow (\tau \rightarrow v)}$	

Fig. 3. Canonical coercions.

Definition 8 (*Canonical coercions*). If coercion $c : \tau \rightsquigarrow \tau'$ is derivable using solely the inference system in Fig. 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 9.

- (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:

Lemma 10 (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.

Thus every congruence class of $\phi\psi$ -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 11 (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.

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 12 (*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 $\text{Func!}; \text{Func?}; \text{Func!}$ is safe since it is ϕ -convertible to Func! , which is a canonical coercion.

Principally 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 13 (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 and c' be such that $\phi \vdash c >_{\psi}^* c'$ and $\phi \vdash c' >_{\psi}^* c$. Let \bar{c} and \bar{c}' be ϕ -normal forms for c and c' respectively, which exist by Lemma 5. Since ϕ -reduction is confluent (Lemma 6) and commutes with ψ -reduction (Lemma 4) it follows that:

$$\bar{c} >_{\phi\psi}^* \bar{c}',$$

$$\bar{c}' >_{\phi\psi}^* \bar{c}.$$

Since $\phi\psi$ -reduction is strongly normalizing (Lemma 5) 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 $c =_{\phi} c'$. \square

Lemma 14 (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 11 we know that $c' =_{\phi\psi} c$. Theorem 7 implies that there exists a $\phi\psi$ -normal form c'' such that $c' >_{\phi\psi}^* c''$ and $c >_{\phi\psi}^* c''$. By Lemma 10 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 $\phi \vdash c' >_{\psi}^* c$. \square

By Proposition 9(1), there is a canonical coercion for every type signature. And by Lemma 10 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 15 (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 16 (*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 17 (Minimality of canonical coercions). *Every canonical coercion is minimal.*

Analogous to Theorem 15 we obtain that minimal completions are unique and exist for every type signature.

Theorem 18 (Existence and uniqueness of minimal coercions).

- (1) For every pair of types τ and 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 $c = c'$.

2.8. Positive, negative and neutral coercions

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

Definition 19 (*Positive, negative and neutral coercions*). Define *positive, negative and neutral coercions* by induction:

- (1) The tagging operations Func! and Bool! are positive. The untag-and-check operations Func? and Bool? are negative. The coercions $\text{Func!}; \text{Bool?}$ and $\text{Bool!}; \text{Func?}$ are neutral. The identity coercions are positive, negative and neutral (but not *properly*).
- (2) 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 $c_1^- \rightarrow c_2^+$ are positive;
 - $c_1^-; c_2^-$ (if well-formed) and $c_1^+ \rightarrow c_2^-$ are negative;
 - $(c_1^*; c_2^*), (c_1^*; c_2^-), (c_1^+; c_2^*)$ (if well-formed) and $c_1^* \rightarrow c_2^*$ are neutral.
- (3) $(c; c'); c''$ is neutral if $c; (c'; c'')$ is, and vice versa.

The last clause above is necessary to make sure that the coercion $(\text{Func?} \rightarrow \text{Func!}); \text{Func!}; \text{Bool?}$ associated to the right as $(\text{Func?} \rightarrow \text{Func!}); (\text{Func!}; \text{Bool?})$ and associated to the left as $((\text{Func?} \rightarrow \text{Func!}); \text{Func!}); \text{Bool?}$ is classified as neutral.

Proposition 20 (*Basic properties of positive and negative coercions*).

- (1) If c has polarity p (positive, negative or neutral) and $\vdash c = c'$ then c' has the same polarity.
- (2) Let c and c' both be positive or negative, respectively. If $\phi\psi \vdash c = c'$ then $\vdash c = c'$.
- (3) If $c^+ : \tau \rightsquigarrow \tau'$ is positive and $c^- : \tau' \rightsquigarrow \tau$ negative then $\phi \vdash c^+; c^- = \iota_\tau$ and $\psi \vdash c^-; c^+ = \iota_{\tau'}$.
- (4) If $c^+ : \tau \rightsquigarrow \tau'$ is positive and $c^- : \tau' \rightsquigarrow \tau''$ is negative then $c^+; c^-$ is safe.

Part (1) says that polarity is a property of coercions and not of the particular expression chosen for a coercion; i.e., it is invariant under coercion equality.

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

- (1) $\text{Bool} \leq \text{Dyn}$ and $\text{Dyn} \rightarrow \text{Dyn} \leq \text{Dyn}$;
- (2) $\tau \leq \tau$ for all types τ ;
- (3) if $\tau_1 \leq \tau_2$ and $\tau_2 \leq \tau_3$ then $\tau_1 \leq \tau_3$;
- (4) if $\tau_1 \leq \tau'_1$ and $\tau_2 \leq \tau'_2$ then $\tau_1 \rightarrow \tau_2 \leq \tau'_1 \rightarrow \tau'_2$

Note that the type constructor \rightarrow is *covariant* with respect to \leq in its first as well as its second argument! We write $\tau < \tau'$ if $\tau \leq \tau'$ but not $\tau = \tau'$.

Proposition 22 (Basic properties of subtype relation).

- (1) *There are no infinite ascending chains $\tau_1 < \tau_2 < \dots$.*
- (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 23 (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* (with respect to composition ;) coercions into positive and negative coercions in Sections 6 and 7.

Proposition 24 (+/-factoring of safe coercions). *For every safe coercion $c : \tau \rightsquigarrow \tau'$ there is $c^+ : \tau \rightsquigarrow \text{Dyn}$ and $c^- : \text{Dyn} \rightsquigarrow \tau'$ such that $\phi \vdash c = c^+; c^-$.*

Proof. Clearly $\tau \leq \text{Dyn}$ and $\tau' \leq \text{Dyn}$. Thus, by Proposition 23, there is a positive $c^+ : \tau \rightsquigarrow \text{Dyn}$ and a negative $c^- : \text{Dyn} \rightsquigarrow \tau'$. By Proposition 20, $c^+; c^-$ is safe and has the same type signature as c . By Theorem 15, two safe coercions with the same type signature are ϕ -convertible. Thus we get $\phi \vdash c = c^+; c^-$. \square

Since the subtype relation has no bottom type there is no corresponding $-/+$ -factoring, even under $\phi\psi$ -conversion; just consider $\text{Func!}; \text{Bool?}$. 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 25 ($-/+$ -factoring of coercions). *For every coercion c there is a negative c^- , a neutral c^* and a positive c^+ such that $c >_\phi^* c^-; c^*; c^+$. Furthermore, c^- , c^* , c^+ are unique modulo coercion equality.*

Proof. Consider the rewriting system in Fig. 4 for coercion expressions, which is modulo associativity and neutrality of ι with respect to composition. It is locally confluent, strongly normalizing and Church–Rosser with respect 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 $c''' \rightarrow c''''$. Note that c cannot have a proper negative left factor nor a proper positive right factor. We can check by case analysis

Let all subscripts superscripted with + be *proper* positive coercions and those superscripted with – *proper* negative coercions.

$$(c; c'); c'' = c; (c'; c'') \quad (\text{ASSOC})$$

$$c; \iota_\tau = c$$

$$\iota_\tau; c = c$$

$$\iota_\tau \rightarrow \iota_{\tau'} \Rightarrow \iota_{\tau \rightarrow \tau'}$$

$$(c \rightarrow c'); (d \rightarrow d') \Rightarrow (d; c) \rightarrow (c'; d') \quad (F1)$$

$$c \rightarrow (d; d^+) \Rightarrow (c \rightarrow d); (\iota \rightarrow d^+) \quad (F2_1^+)$$

$$(c^-; c) \rightarrow d \Rightarrow (c \rightarrow d); (c^- \rightarrow \iota) \quad (F2_2^+)$$

$$c \rightarrow (d^-; d) \Rightarrow (\iota \rightarrow d^-); (c \rightarrow d) \quad (F2_1^-)$$

$$(c; c^+) \rightarrow d \Rightarrow (c^+ \rightarrow \iota); (c \rightarrow d) \quad (F2_2^-)$$

$$\text{Func!}; \text{Func?} \Rightarrow \iota_{\text{Dyn}} \quad (\phi^{\rightarrow})$$

$$\text{Bool!}; \text{Bool?} \Rightarrow \iota_{\text{Dyn}} \quad (\phi^{\text{Bool}})$$

Side condition for rule *F1*: The left-hand side must be proper and have a polarity; i.e., it must be properly positive, negative, or neutral.

Side condition for rules *F2*: The left-hand side must *not* have a polarity.

Fig. 4. Coercion rewriting system for –/+ -factoring.

that c must be a neutral coercion. \square

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

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

$$\phi \vdash c = \bar{c}; d^+, \quad \phi \vdash c' = \bar{c}; d'^+.$$

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:*

$$\phi \vdash d = c^-; \bar{d}, \quad \phi \vdash d' = c'^-; \bar{d}.$$

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

The following theorem follows from this lemma directly.

Theorem 27 (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^+ and 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^+ and 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^- and 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^- and 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 and d (negative or otherwise) such that $\text{Func?}; c = \text{Bool?}; d$, even under ϕ -conversion.

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 Fig. 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., $\text{true}(\lambda x.x)$. In following tradition, however, we shall call the expressions of the implicit language *untyped λ -terms*.

Definition 28 (Untyped λ -terms, equality). A (closed) *untyped λ -term* is a closed expression derived from e in the grammar of Fig. 5. Untyped λ -terms e

$e ::= x$	(variables)
$\lambda x.e$	(λ -abstractions)
ee'	(applications)
$true \mid false$	(boolean constants)
if e then e' else e''	(conditionals)

Fig. 5. Grammar for implicit dynamically typed λ -calculus.

$\frac{e : \tau \quad [x : \tau']}{\lambda x : \tau'. e : \tau' \rightarrow \tau}$	$\frac{e : \tau' \rightarrow \tau \quad e' : \tau'}{ee' : \tau}$
$true : \text{Bool}$	$false : \text{Bool}$
$\frac{e : \text{Bool} \quad e' : \tau \quad e'' : \tau}{\text{if } e \text{ then } e' \text{ else } e'' : \tau}$	$\frac{e : \tau \quad c : \tau \rightsquigarrow \tau'}{[c]e : \tau'}$
$[l_\tau]e = e$	
$[c'] [c]e = [c; c']e$	
$[c \rightarrow d] \lambda x.e = \lambda x. [d]e \{x \mapsto [c]x\}$	
$([c \rightarrow d]e)e' = [d](e([c]e'))$	
$[c] \text{if } e \text{ then } e' \text{ else } e'' = \text{if } e \text{ then } [c]e' \text{ else } [c]e''$	

Fig. 6. Formation rules and equations for $\lambda^{\rightarrow A}[\text{Bool}]$.

and 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 $\lambda^{\rightarrow A}[\text{Bool}]$, the (explicit) dynamically typed λ -calculus (with booleans).

The formation rules and equations for $\lambda^{\rightarrow A}[\text{Bool}]$ are given in Fig. 6.

Definition 29 (Dynamically typed λ -term, equality). A dynamically typed (closed) λ -term (λA -term) is an expression e such that $e : \tau$ is derivable

$$\begin{aligned}
(c; c'); c'' &= c; (c'; c'') \\
c; \iota &= c \\
\iota; c &= c \\
\iota_{\tau \rightarrow \tau'} &= \iota_{\tau} \rightarrow \iota_{\tau'} \\
(c \rightarrow c'); (d \rightarrow d') &= (d; c) \rightarrow (c'; d') \\
[\iota]e &= e \\
[c'] [c]e &= [c; c']e \\
[c \rightarrow d] \lambda x. e &= \lambda x. [d] (e \{x \mapsto [c]x\}) \\
([c \rightarrow d]e) e' &= [d] (e ([c]e')) \\
[c] \text{if } e \text{ then } e' \text{ else } e'' &= \text{if } e \text{ then } [c]e' \text{ else } [c]e'' \\
\text{Func!}; \text{Func?} &= \iota_{\text{Dyn} \rightarrow \text{Dyn}} & (\phi^{\rightarrow}) \\
\text{Bool!}; \text{Bool?} &= \iota_{\text{Bool}} & (\phi^{\text{Bool}}) \\
\text{Func?}; \text{Func!} &= \iota_{\text{Dyn}} & (\psi^{\rightarrow}) \\
\text{Bool?}; \text{Bool!} &= \iota_{\text{Dyn}} & (\psi^{\text{Bool}})
\end{aligned}$$

Fig. 7. Summary of all coercion and λA -term equations.

using the formation rules in Fig. 6 and those for coercions in Figs. 1 and 2. The λA -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 Fig. 1 and the λA -term equations of Fig. 6 (but *not* the ϕ - and ψ -equations!).

As for coercions we assume that the λA -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 λA -terms as well as the type subscripts of identity coercions.

The formation rules for λA -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 λA -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 λA -term. The significance of the remaining equations becomes apparent in Section 5. For convenience all the equations introduced so far for coercions and λA -terms, including the core coercion equations as well as ϕ - and ψ -conversion, are reproduced in Fig. 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 $\lambda x : \tau. [c]x : \tau \rightarrow \tau'$. In this sense coercions with type signature

$\tau \rightsquigarrow \tau'$ are contained in the domain of functions of type $\tau \rightarrow \tau'$. 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 30 (*Conversion and reduction between $\lambda\Delta$ -terms*). Let R and 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 $E \vdash e >_R e_1 > \dots > e_n >_R e'$ for some e_1, \dots, 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

$$[c \rightarrow d] \lambda x. e = \lambda x. [d] (e\{x \mapsto [c]x\}), \quad (1)$$

$$([c \rightarrow d]e)e' = [d](e([c]e')), \quad (2)$$

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

- (2) Eq. (1) and

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

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

- (3) The right-hand side of Eq. (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 λA -terms) may be different on the left- and right-hand sides of Eq. (1). This is in contrast to Thatte’s quasi-static type system [23] and (the first-order part of) Curien and Ghelli’s coercion formulation for System F_{\leq} [8]. 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 λA -terms (compare Lemma 4 in Section 2). Consider a λA -term e of the form

$$(\lambda x : \text{Dyn} \dots [\text{Func?}]x \dots [\text{Bool?}]x \dots) (\text{if } \dots \text{ then } [F?; F!]y \text{ else } [F!]z)$$

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

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

and by ϕ -reduction to

$$e_2 = (\lambda x : \text{Dyn} \rightarrow \text{Dyn} \dots [\iota_{\text{Dyn} \rightarrow \text{Dyn}}]x \dots [\text{Func!}; \text{Bool?}]x \dots) \\ (\text{if } \dots \text{ then } [F?]y \text{ else } z).$$

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* λA -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 λA -terms below.

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

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

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 31 (*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 .

³ The example above demonstrates that Theorem 2 in [11] is not correct; it appears to hold for λI -terms, however. See Section 7.

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, Eq. (1).

$\frac{e \longrightarrow_c \bar{e} : \text{Dyn} \quad [x \longrightarrow_c x : \text{Dyn}]}{\lambda x. e \longrightarrow_c [\text{Func!}] \lambda x : \text{Dyn}. \bar{e} : \text{Dyn}}$	$\frac{e \longrightarrow_c \bar{e} : \text{Dyn} \quad e' \longrightarrow_c \bar{e}' : \text{Dyn}}{ee' \longrightarrow_c ([\text{Func?}] \bar{e}) \bar{e}' : \text{Dyn}}$
$\text{true} \longrightarrow_c [\text{Bool!}] \text{true} : \text{Dyn}$	$\text{false} \longrightarrow_c [\text{Bool!}] \text{false} : \text{Dyn}$
$\frac{e \longrightarrow_c \bar{e} : \text{Dyn} \quad e' \longrightarrow_c \bar{e}' : \text{Dyn} \quad e'' \longrightarrow_c \bar{e}'' : \text{Dyn}}{\text{if } e \text{ then } e' \text{ else } e'' \longrightarrow_c [\text{Bool?}] \bar{e} \text{ then } \bar{e}' \text{ else } \bar{e}'' : \text{Dyn}}$	
$\frac{e \longrightarrow_c \bar{e} : \text{Dyn}}{e \longrightarrow_c [c] e : \tau}$	(if $\tau \neq \text{Dyn}$ and $c : \text{Dyn} \rightsquigarrow \tau$ canonical)

Fig. 8. Canonical completions of untyped λ -terms.

We write $e \longrightarrow e' : \tau$ if e' is a completion of e at τ . Completions of untyped λ -terms are referred to as *λ I-terms*.

Since there is generally more than one completion for the same incomplete program we treat the resulting ambiguity as a problem of *coherence* [4,8] (see Section 5) or *safety* (cf. [23]; 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 32 (*Canonical completion*). If $e \longrightarrow_c \bar{e} : \tau$ is derivable in the inference system of Fig. 8 then \bar{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.

Proposition 33 (Uniqueness and existence of canonical completions).

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

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 *inefficient*. Note that by Proposition 33 there is a completion for *every* untyped λ -term. Thus *no* untyped λ -term is “rejected” by the dynamic typing discipline.

5. Coherence

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

Definition 34 (*Completion congruence*). Dynamically typed λ -terms e' and 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 35 (Coherence of completions, characterization of completion congruence). *Dynamically typed λ -terms e' and 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 Fig. 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 Fig. 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* (cf. [8]) 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 $[c_k] \cdots [c_1]e = [c_1; \cdots; c_k]e$ 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 11 it follows that $c' =_{\phi\psi} c''$ and thus $[c']x =_{\phi\psi} [c'']x$.

Basis II. If $e \equiv \text{true}$ or $e \equiv \text{false}$ 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 $d' : v' \rightsquigarrow v''$ such that

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

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 v'' . By inductive hypothesis we

thus get

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

Consequently we have

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

Since $(d \rightarrow d'; c'')$ and c' both have type signature $\tau' \rightsquigarrow \tau''$ we know by application of Theorem 11 that $(d \rightarrow d'; c'') =_{\phi\psi} c'$, and the result follows.

The inductive cases for applications and **if**-expressions are analogous.

For every equation $e = e'$ in Fig. 7 it is easy to construct two completion congruent $\lambda\mathcal{A}$ -terms that cannot be proved equal without $e = e'$. \square

This shows that, *independent of β - 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 Fig. 7, in particular also the ψ -equations.*

6. Safety

In the characterization of coherence of completions (Theorem 35) we have used the ψ -equations

$$\begin{aligned} \text{Func?}; \text{Func!} &= \iota_{\text{Dyn}} (\psi^{\neg}), \\ \text{Bool?}; \text{Bool!} &= \iota_{\text{Dyn}} (\psi^{\text{Bool}}). \end{aligned}$$

Accordingly, we have

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

since

$$\begin{aligned} \text{Bool!}; \text{Func?}; \text{Func!}; \text{Bool?} &=_{\psi} \text{Bool!}; \iota_{\text{Dyn}}; \text{Bool?} \\ &= \text{Bool!}; \text{Bool?} \\ &=_{\phi} \iota_{\text{Bool}} \end{aligned}$$

and

$$[\iota_{\text{Bool}}] \text{true} = \text{true}.$$

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

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

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: $[\text{Func?}][\text{Func!}]f$ 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 [1] and [17] 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. 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 $C[\iota_{\text{Dyn}}]$ 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 [23].

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

Definition 36 (*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, 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 14, which states that canonical coercions are safe, we can show that canonical completions, as defined in Fig. 8, are safe.

Lemma 37 (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 35: e' is assumed to be the canonical completion at τ , and e'' any other completion at τ . It must be verified that whenever two coercions c' and c'' , one for e' , the other for e'' , have the same type signature then $\phi \vdash c'' >_{\psi}^* c'$. Details are left to the reader.) \square

Furthermore, Lemma 13, 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 λA -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 Fig. 9. Coercions superscripted with $+$ are positive with type signature $\tau \rightsquigarrow \text{Dyn}$ and those superscripted with $-$ are negative with type signature $\text{Dyn} \rightsquigarrow \tau'$ for some τ and τ' . Applicability of the rewriting rules S1–S4 is modulo ϕ -conversion and the equations for coercion application. For example, S4 is applicable to $[c]\text{if } e \text{ then } e' \text{ else } e''$ if $\phi \vdash c = c^+$, even though c may not ϕ -reduce to c^+ .

Lemma 38 (Safety rewriting system properties). *The safety rewriting system of Fig. 9 extends $\phi\psi$ -reduction 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).*

$\text{Func!}; \text{Func?} \Rightarrow \iota_{\text{Dyn} \rightarrow \text{Dyn}}$	(ϕ^-)
$\text{Bool!}; \text{Bool?} \Rightarrow \iota_{\text{Bool}}$	(ϕ^{Bool})
$\text{Func?}; \text{Func!} \Rightarrow \iota_{\text{Dyn}}$	(ψ^-)
$\text{Bool?}; \text{Bool!} \Rightarrow \iota_{\text{Dyn}}$	(ψ^{Bool})
$[c'] [c] e = [c; c'] e$	
$[\iota] e = e$	
$[(d^- \rightarrow d^+); \text{Func!}^+] \lambda x. e \Rightarrow [\text{Func!}^+] \lambda x. [d^+] e \{x \mapsto [d^-] x\}$	(S1)
$([\text{Func?}^-; (d^+ \rightarrow d^-)] e) e' \Rightarrow [d^-] (([\text{Func?}^-] e) ([d^+] e'))$	(S2)
if e then $[c^-] e'$ else $[c^-] e'' \Rightarrow [c^-] \text{if } e \text{ then } e' \text{ else } e''$	(S3)
$[c^+] \text{if } e \text{ then } e' \text{ else } e'' \Rightarrow \text{if } e \text{ then } [c^+] e' \text{ else } [c^+] e''$	(S4)

Side conditions: (S1, S2) $d^+ \neq \iota_{\text{Dyn}}^+$ or $d^- \neq \iota_{\text{Dyn}}^-$;
 (S3) $c^- \neq \iota_{\text{Dyn}}^-$; (S4) $c^+ \neq \iota_{\text{Dyn}}^+$.

Fig. 9. Safety rewriting system.

Proof. (1) All the rewrite rules are valid equalities with respect to $\phi\psi$ -conversion. Furthermore, for every safe coercion $c : \tau \rightsquigarrow \tau'$ we have $\phi \vdash c = c^+; c^-$ for some positive coercion $c^+ : \tau \rightsquigarrow \text{Dyn}$ and negative coercion $c^- : \text{Dyn} \rightsquigarrow \tau'$ by Proposition 24. Thus all equations of Fig. 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 24, 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 6). 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 the ψ -rules 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 5), and every λ -term rewriting rule properly increases τ or τ' with respect to the subtyping order \leq . Since \leq is finitely ascending the λ -term rewriting steps can only be applied a finite number of times. \square

This lemma enables application of the proof method used for Lemma 13 to show that ψ -reduction is conservative over ϕ -conversion also for λ -terms.

Theorem 39 (Conservative extension of ϕ -conversion for λ -terms). *Let e and e' be dynamically typed λ -terms. Then $\phi \vdash e >_{\psi}^* e'$ and $\phi \vdash e' >_{\psi}^* e$ if and only if $e =_{\phi} e'$.*

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

Corollary 40 (Uniqueness of safe completions up to ϕ -conversion). *If both e' and 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 41 (*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 [11].⁵ There are efficient algorithms for computing such restricted minimal completions [11] that have applications in the optimization of run-time typed languages such as Scheme, Common LISP, SETL and others [12].

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

⁵ Note that [11] claims that *all* untyped λ -terms have minimal (general) completions, which is not correct!

$\text{Func!}; \text{Func?} \Rightarrow \iota_{\text{Dyn} \rightarrow \text{Dyn}}$	(ϕ^-)
$\text{Bool!}; \text{Bool?} \Rightarrow \iota_{\text{Bool}}$	(ϕ^{Bool})
$[c'] [c] e = [c; c'] e$	
$[\iota] e = e$	
$[d^+ \rightarrow d^-] \lambda x. e \Rightarrow \lambda x. [d^-] e \{x \mapsto [d^+] x\}$	$(M1^-)$
$\lambda x. [d^+] e \{x \mapsto [d^-] x\} \Rightarrow [d^- \rightarrow d^+] \lambda x. e$	$(M1^+)$
$\lambda x. [d_2^*] e \{x \mapsto [d_1^*] x\} = [d_1^* \rightarrow d_2^*] \lambda x. e$	$(M1^*)$
$([d^- \rightarrow d^+] e) e' \Rightarrow [d^+] (e([d^-] e'))$	$(M2^+)$
$[d^-] (e([d^+] e')) \Rightarrow ([d^+ \rightarrow d^-] e) e'$	$(M2^-)$
$([d_1^* \rightarrow d_2^*] e) e' = [d_2^*] (e([d_1^*] e'))$	$(M2^*)$
if e then $[c^+] e'$ else $[c^+] e'' \Rightarrow [c^+] \text{if } e \text{ then } e' \text{ else } e''$	$(M3^+)$
$[c^-] \text{if } e \text{ then } e' \text{ else } e'' \Rightarrow \text{if } e \text{ then } [c^-] e' \text{ else } [c^-] e''$	$(M3^-)$
if e then $[c^*] e'$ else $[c^*] e'' = [c^*] \text{if } e \text{ then } e' \text{ else } e''$	$(M3^*)$
Side conditions: $(M1^-, M2^-)$ $d^+ \rightarrow d^-$ properly negative;	
$(M1^+, M2^+)$ $d^- \rightarrow d^+$ properly positive;	
$(M3^+)$ c^+ properly positive; $(M3^-)$ c^- properly negative.	

Fig. 10. Minimization rewriting system.

system with respect to $\phi\psi$ -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 Fig. 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 42 (Characterization of ϕ -conversion). *The reflexive, symmetric, transitive and compatible closure of the minimization rewriting system (Fig. 10) gives ϕ -conversion.*

Proof. Every coercion can be factored into $c^-; c^*; c^+$ (Proposition 25); in particular, a coercion $c \rightarrow d$ can be factored into $(c_1^+ \rightarrow c_2^-); (c_1^* \rightarrow c_2^*); (c_1^- \rightarrow c_2^+)$. Thus every $\lambda\Delta$ -term equation is in the symmetric, transitive closure of the three corresponding rewriting rules in Fig. 10. \square

Lemma 43 (Local confluence of minimization rewriting system). *The minimization rewriting system in Fig. 10 is locally confluent for $\lambda I\Delta$ -terms (modulo*

the equations).

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 27 guarantees that the critical pair has a common reduct. All rules but $M1^+$ satisfy this property for all λA -terms, and $M1^+$ satisfies it for λIA -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; cf., the example at the end of Section 3.)

Furthermore, ϕ -reduction is confluent modulo coercion equality (follows from Lemma 6).

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

$$[d^+ \rightarrow d^-] \lambda x. [c^+] e \{x \mapsto [c^-] x\} \quad \text{and} \quad [d^-] ([c^- \rightarrow c^+] e [d^+] e').$$

Using Proposition 25 the resulting critical pairs can be reduced to a common reduct. \square

The same rule that caused problems with local confluence, $M1^+$, applied to non- λIA -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 $\lambda x : \text{Dyn}.e$ where x does not occur in e we can create an infinite reduction path. It has the form

$$[c_1 \rightarrow \iota] \lambda x : \tau_1. e \Rightarrow \cdots \Rightarrow [c_i \rightarrow \iota] \lambda x : \tau_i. e \Rightarrow \cdots$$

where $\text{Dyn} = \tau_1 > \tau_2 > \cdots > \tau_i > \cdots$. For λIA -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 44 (Strong normalization of minimization rewriting system). *There is no infinite reduction path $e_1 \Rightarrow e_2 \Rightarrow \cdots \Rightarrow e_i \Rightarrow \cdots$ in the rewriting system of Fig. 10 if e_1 is a λIA -term.*

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

Theorem 45 (Existence of minimal completions). *Every untyped λI -term has a minimal completion at every type if the minimization rewriting system in Fig. 10 is strongly normalizing.*

Using the method of proving Lemma 13 we obtain that, for $\lambda I\Delta$ -terms, ϕ -reduction is conservative over equality.

Theorem 46 (Conservative extension of equality). *Let e and e' be dynamically typed λI -terms and assume that the minimization rewriting system in Fig. 10 is strongly normalizing. Then $e >_{\phi}^* e'$ and $e' >_{\phi}^* e$ if and only if $e = e'$.*

Thus we have in particular:

Corollary 47 (Uniqueness of minimal completions). *If e' and e'' are minimal completions of an untyped λ -term e at the same type then $e' = e''$ (assuming strong normalization of the rewriting system in Fig. 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 [2].

The main motivation behind the work of Mycroft [18,19], Abadi, Cardelli, Pierce, Plotkin and Rémy [1–3], and Leroy and Mauny [17] 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 (cf. remarks by Thatte [23]). 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 type check construct 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 [22]. In our case the resulting subtype theory is *covariant* 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 [11]), the resulting subtype system is

weaker than partial typing. In this case $\tau \rightarrow \text{Dyn} \leq \text{Dyn}$ only holds for $\tau \equiv \text{Dyn}$ whereas it holds for all τ in partial typing. Thatte [22] 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 [20] and to be in polynomial time by Kozen, Palsberg and Schwartzbach [16]. (By the lower bound method presented in [14] 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 [23]. 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 [2], 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 [5] 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 [8] 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 [9] 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 [6,7] 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 Rémy [21] for record-based inheritance. The type inference algorithm is based on Milner’s Algorithm W operating on Rémy-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 [10]. 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 44.

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 $\text{Func!}; \text{Bool?}$ 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 45 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 [12]. It computes minimal completions in the completion class C_{pf} (see [11]). 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 and Jakob Rehof 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 grateful 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

- [1] M. Abadi, L. Cardelli, B. Pierce and G. Plotkin. Dynamic typing in a statically-typed language, in: *Proceedings 16th Annual ACM Symposium on Principles of Programming Languages*, Austin, TX (1989) 213–227.
- [2] M. Abadi, L. Cardelli, B. Pierce and G. Plotkin, Dynamic typing in a statically typed language, *ACM Trans. Programming Languages Syst.* **13** (2) (1991) 237–268.

- [3] M. Abadi, L. Cardelli, B. Pierce and D. Rémy, Dynamic typing in polymorphic languages, in: *Proceedings ACM SIGPLAN Workshop on ML and Its Applications (ML'92)*, San Francisco, CA (1992) 92–103.
- [4] V. Breazu-Tannen, T. Coquand, C. Gunter and A. Scedrov, Inheritance and explicit coercion, in: *Proceedings Fourth IEEE Symposium on Logic in Computer Science (LICS'89)*, Asilomar, CA (1989) 112–129.
- [5] V. Breazu-Tannen, T. Coquand, C. Gunter and A. Scedrov, Inheritance as implicit coercion, *Inform. Comput.* **93** (1) (1991) 172–221; presented at LICS'89.
- [6] R. Cartwright and M. Fagan, Soft typing, in: *Proceedings ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, Toronto, Ont. (1991) 278–292.
- [7] R. Cartwright and M. Fagan, Soft typing. Draft, Corrected revision of SIGPLAN'91 paper (1992).
- [8] P. Curien and G. Ghelli, Coherence of subsumption, in: A. Arnold, ed., *Proceedings 15th Colloquium on Trees in Algebra and Programming*, Copenhagen, Denmark (Springer, Berlin, 1990) 132–146.
- [9] C. Gomard, Partial type inference for untyped functional programs (extended abstract), in: *Proceedings LISP and Functional Programming (LFP'90)*, Nice, France (1990).
- [10] J. Hannan, Embedding the untyped λ -calculus in a simply typed λ -calculus, Unpublished manuscript (1988).
- [11] F. Henglein, Dynamic typing, in: *Proceedings European Symposium on Programming (ESOP)*, Rennes, France, Lecture Notes in Computer Science **582** (Springer, Berlin, 1992) 233–253.
- [12] F. Henglein, Global tagging optimization by type inference, in: *Proceedings LISP and Functional Programming (LFP'92)*, San Francisco, CA (1992).
- [13] F. Henglein, Dynamic typing: syntax and proof theory, TOPPS Report D-163, DIKU, University of Copenhagen (1993).
- [14] F. Henglein and H. Mairson, The complexity of type inference for higher-order typed lambda calculi, in: *Proceedings 18th ACM Symposium on Principles of Programming Languages (POPL'91)*, Orlando, FL (1991) 119–130.
- [15] G. Huet, Confluent reductions: Abstract properties and applications to term rewriting systems, *J. Assoc. Comput. Mach.* **27** (4) (1980) 797–821.
- [16] D. Kozen, J. Palsberg and M. Schwartzbach, Efficient inference of partial types, Tech. Report DAIMI PB-394, DAIMI, Aarhus University, Aarhus, Denmark (1992).
- [17] X. Leroy and M. Mauny, Dynamics in ML, in: *Proceedings Conference on Functional Programming Languages and Computer Architecture (FPCA'91)*, Cambridge, MA, Lecture Notes in Computer Science **523** (Springer, Berlin, 1991) 406–426.
- [18] A. Mycroft, Dynamic types in statically typed languages, Unpublished manuscript, preliminary draft (1983).
- [19] A. Mycroft, Dynamic types in statically typed languages, Unpublished manuscript, 2nd draft version (1984).
- [20] P. O'Keefe and M. Wand, Type inference for partial types is decidable, in: B. Krieg-Brückner, ed., *Proceedings 4th European Symposium on Programming (ESOP'92)*, Rennes, France (Springer, Berlin, 1992) 408–417.
- [21] D. Rémy, Typechecking records and variants in a natural extension of ML, in: *Proceedings 16th Annual ACM Symposium on Principles of Programming Languages*, Austin, TX (1989) 77–88.
- [22] S. Thatte, Type inference with partial types, in: *Proceedings International Colloquium on Automata, Languages and Programming (ICALP'88)*, Lecture Notes in Computer Science (Springer, Berlin, 1988) 615–629.
- [23] S. Thatte, Quasi-static typing, in: *Proceedings 17th Annual ACM Symposium on Principles of Programming Languages*, San Francisco, CA (1990) 367–381.