Formally Optimal Boxing

Fritz Henglein & Jesper Jørgensen
DIKU, Department of Computer Science
University of Copenhagen, Universitetsparken 1
DK-2100 Copenhagen Ø
Denmark
e-mail: henglein@diku.dk & knud@diku.dk

Abstract

An important implementation decision in polymorphically typed functional programming languages is whether to represent data in boxed or unboxed form and when to transform them from one representation to the other. Using a language with explicit representation types and boxing/unboxing operations we axiomatize equationally the set of all explicitly boxed versions, called *completions*, of a given source program. In a two-stage process we give some of the equations a rewriting interpretation that captures eliminating boxing/unboxing operations without relying on a specific implementation or even semantics of the underlying language. The resulting reduction systems operate on congruence classes of completions defined by the remaining equations E, which can be understood as moving boxing/unboxing operations along data flow paths in the source program. We call a completion e_{opt} *formally optimal* if every other completion for the same program (and at the same representation type) reduces to e_{opt} under this two-stage reduction.

We show that every source program has formally optimal completions, which are unique modulo E. This is accomplished by first "polarizing" the equations in E and orienting them to obtain two canonical (confluent and strongly normalizing) rewriting systems. The completions produced by Leroy's and Poulsen's algorithms are generally not formally optimal in our sense.

The rewriting systems have been implemented and applied to some simple Standard ML programs. Our results show that the amount of boxing and unboxing operations is also in practice substantially reduced in comparison to Leroy's completions. This analysis is intended to be integrated into Tofte's region-based implementation of Standard ML currently underway at DIKU.

Keywords

Representation analysis, polymorphism, type inference.

1 Introduction

1.1 Representation analysis

Representation analysis seeks to optimize the run-time representation of elements of data types in high-level programming languages. A problem specific to polymorphically typed languages such as Standard ML or Haskell is how to represent the actual arguments to polymorphic functions. The polymorphic (generic) parts of arguments to a polymorphic function can be of any type and will usually be called with actual arguments of different types. There are several possible ways of implementing such polymorphic functions. The predominant one is to ensure that actual arguments are represented *uniformly*, independent of their actual type, using *boxed representations*.

A *boxed* representation of a data structure is a pointer to some area in memory where the actual contents of the data structure reside.[1] The point of this representation is that it has the same "size" for all types of data structures. By passing only arguments in boxed representation a (truly) polymorphic function can be correctly implemented by a single piece of code since it is guaranteed to never actually inspect the data structure itself. Other operations, however, such as integer addition or a conditional checking the value of a Boolean expression require access to the contents of the data and are penalized by the additional level of indirection incurred by boxing, as they first have to *unbox* the representation; i.e., dereference the pointer. Furthermore, boxed representations require more space than unboxed representations thus increasing the space demand and garbage collection costs. Parameter passing, on the other hand, is generally more efficient for boxed than unboxed data representations. Thus there are competing demands on the representation of data in a program. A boxed representation can, of course, be transformed to an unboxed representation at run-time, and vice versa. These conversions can contribute substantially to the run-time cost of a program, however, both in terms of time and space.

Boxing analysis is a special representation analysis that seeks to minimize the need for run-time conversions whilst satisfying the representation demands on all data in a program. Boxing analysis can be facilitated by making representation choices and boxing/unboxing operations in a program explicit. This amounts to a translation to a language

[1] The elements of "small" data structures such as pointer-sized integer representations may be considered simultaneously boxed and unboxed. In the following we shall think of these as two separate representations with associated trivial boxing and unboxing operations.

with explicit boxed and unboxed types and new operations denoting boxing and unboxing operations without, however, changing the "underlying" program. We shall call these explicit boxing and unboxing operations *(representation) coercions*. There are, in principle, many different possible translations for the same program corresponding to different representation choices for the data structures in the program and different needs for representation coercions. We shall refer to any one of these translations as a *completion* of the underlying program. The question then is: which completion should be chosen for a given program?

In a naive translation every expression is translated to (a computation of) its boxed representation where operations that need to inspect the contents of such a representation use explicit unboxing operations. The rationale for making boxing explicit is that some boxing/unboxing operations can be eliminated in the later transformational stages of such a compiler [PJL91], as for example in the Glasgow Haskell Compiler. Other translations may elide some of these boxing and unboxing operations directly; e.g., the type inference based translations of Leroy [Ler92] and Poulsen [Pou93].

1.2 A coercion calculus for boxing

Beyond offering yet another translation we seek to formulate and answer the more fundamental questions that underlie the very purpose of boxing analysis and, more generally, similar static analyses: Given two completions for the same program, which of them is better? What does it mean for one completion to be "better" than another completion in the first place? Which programs, if any, have "optimal" completions; i.e., completions that are better than any other for the same program? Can such optimal completions be computed, and how? Of course, it doesn't make much sense to compare the quality of completions on the basis of their actual run-time cost on a specific computer assuming a specific language implementation. In any scenario where we take the actual semantics of the language fully into account the answer to the last two questions would be "no" on recursion-theoretic grounds anyway (assuming the language is universal, of course).

If we can pick any one of a collection of completions for a given program it is a fundamental assumption that all completions must be *coherent*; i.e., they have the same observational behavior. Our approach is to assume that we know *nothing else* about the programming language than that any two completions of the same program are coherent. For a paradigmatic functional language we show that coherence can be axiomatized by an equational theory; i.e., a theory of equations of the form $e'{=}e''$ where e', e'' are completions of the same program (for a given result representation type). This axiomatization contains the equations `box;unbox` $= \iota$ and `unbox;box` $= \iota$, which express that first boxing and then immediately unboxing (or the other way round) a value (boxed value) is observationally indistinguishable from doing nothing at all to the value. We interpret these equations as left-to-right rewriting rules in accordance with our expectation that performing a pair of coercions is operationally more expensive than doing nothing at all. This gives us a rewriting system modulo the remaining equational axioms. These remaining equations intuitively simply "push" coercions back and forth — e.g., from actual argument to formal parameter in a function application — but they do not eliminate them.

The rewriting system gives us a relatively simple — and coarse — notion of quality: if $e' \Longrightarrow^* e''$ then e'' is better than e', and if $e' \Longrightarrow^* e''$ for *all* completions e' of a given program then e'' is an "optimal" completion (modulo the remaining equational axioms mentioned above). Unfortunately the resulting notion of reduction is not Church-Rosser; i.e., there are two coherent completions that have no common reduct and are thus "locally" optimal. This is due to the fact that a `box;unbox`-redex may only be eliminated at the expense of *introducing* a `unbox;box`-redex, and vice versa.

By prioritizing elimination of `unbox;box`-redexes over `box;unbox` or the other way round, however, we arrive at two *formal optimality* criteria for completions. We show that every program has a formally optimal completion at any given representation type under each of the two prioritizations. This is accomplished by orienting the equations E as left-to-right or right-to-left rewriting rules depending on the *polarity* of the coercions involved. (Any simple-minded orientation of E leads to nonconfluence and nontermination of Knuth-Bendix completion.) The resulting two rewriting systems can be used to compute specific optimal completions.

Formulating boxing analysis in the framework of a formal coercion calculus has the advantage that the results we obtain are extremely general and robust:

1. They apply to any interpretation whatsoever of the underlying programming language; e.g., to a call-by-value, call-by-name, or lazy interpretation of our functional language.

2. They can be combined with other optimizations unrelated to boxing as the calculus makes few assumptions about the underlying language or its implementation technology.

3. They admit talking about optimality relative to an explicit, formally specified criterion.

4. They leave a great degree of freedom as optimality is accomplished up to a well-defined congruence relation on completions; for example, the notion of optimality is not overcommitted by insisting on syntactic uniqueness.

1.3 New results

The contributions in this paper are:

- A general framework and robust criterion for the quality of boxing completions, which accounts for the costs of boxing/unboxing operations, but abstracts from other language properties and implementation concerns.

- Proof of the existence of *formally optimal (boxing) completions* and their uniqueness modulo an equational theory for moving boxing and unboxing operations along data flow paths. Our notion of formal optimality is independent of any specific properties of the underlying programming language.

- A rewriting-based algorithm for computing formally optimal completions, which are uniformly better than those described in [PJL91, Ler92, Pou93] in our (formal) sense.

- An experimental implementation of the algorithm and test results for a call-by-value language that support empirically that our completions are also better in practice than those reported in the literature previously.

The boxing algorithm and the quality of its output is apparently the most immediate and practically most relevant contribution of our work. It could certainly have been presented, together with the empirical results, independently of the coercion calculus and its formal optimality criteria. But this would have been unsatisfactory in several respects:

With a proliferation of different algorithms for the same problem there is a clear need for a systematic comparison between them. Using exclusively empirical data is unsatisfactory for this purpose as they can only report on system performance where the interaction of boxing with other system properties changes frequently and is difficult to quantify. Our optimality criterion is simple, natural and facilitates a completely formal comparison of boxing completions; furthermore, it makes the basis of comparison explicit and thus, if nothing else, facilitates a substantive criticism of its rationale.

Our boxing algorithm has been developed from a systematic analysis of the coercion calculus and its optimality criterion. Without the general framework it would doubtlessly appear *ad hoc*. It would also be impossible to say anything about its "robustness" and global properties; for example, the algorithm produces the *same* completion when given either one of the completions of [PJL91, Ler92, Pou93] as its initial input. This follows from the coherence of all completions and the Church-Rosser and strong normalization properties of the rewriting systems.

1.4 Notation and terminology

Since most of the notation in this paper is fairly standard, we will only describe the notation that is not. The notation $[x^i \mapsto t_i]t$ means "substitute t_i for the i'th occurrence of x in t", for some fixed ordering of occurrences of $x$in t. We will also use the term $[x^i \mapsto t_i]t$ as a pattern. If a term t' matches this pattern then the part of t' that matches the i'th occurrence of x in t will have to match t_i. Ordinary substitution $[x \mapsto t']t$ will also be used as a pattern in a similar way, except that then all the occurrences of x will have to match the same expression.

We use the notation $\overline{t}$ for tuples, and $\overline{t}$!i selects the i'th element of $\overline{t}$. If $\overline{x}$ is a tuple of variables and $\overline{t'}$ a tuple of terms of the same length then $[\overline{x} \mapsto \overline{t'}]t$ is parallel substitution of the variables in $\overline{x}$ for the corresponding elements of $\overline{t'}$.

Free indices are always assumed universally quantified, i.e. if we write $x_i{=}t_i$ this means for all i $x_i{=}t_i$ and the range of i is assumed given by the context.

We use $\equiv$ for syntactic equality to distinguish it from provable equality $=$.

We write $A \vdash e \implies_R e'$ to indicate that e rewrites to e'modulo the equational theory axiomatized by A. The rewriting rules are given by R. Often R will be a set of equations E oriented uniformly from left to right or from right to left, in which case we shall write $E^{\rightarrow}$ or $E^{\leftarrow}$, respectively.

2 A functional language: Core-XML

Our setting is a polymorphically typed higher-order functional language. We shall restrict ourselves to a small core language with no primitive types, called Core-XML in [HM93], to develop our theory. In Section 6 it is shown how to extend our results to arbitrary type constructors such as integers, Booleans, pairs and lists and to (monomorphically or polymorphically typed) constants such as a fixed point operator and primitive operations for other kinds of type constructors.

$e \in \mathit{Expression}$; $x \in \mathit{Variable}$; $\tau \in \mathit{Type}$;
$\sigma \in \mathit{TypeScheme}$

$e ::= x \mid \lambda x{:}\tau.e \mid e\ e \mid \texttt{let}\ x{:}\sigma\ \texttt{=}\ e\ \texttt{in}\ e \mid \Lambda\alpha.e \mid e\{\tau\}$
$\tau ::= \alpha \mid \tau \rightarrow \tau$
$\sigma ::= \tau \mid \forall\alpha.\sigma$

Figure 1: Syntactic categories of Core-XML

The syntactic constructs for *expressions, types* and *type schemes* in Core-XML are given in Figure 1. We call an expression e *well-formed* (or simply a *Core-XML expression*) under type assumption Γ if $\Gamma \vdash e : \sigma$ is derivable from the inference rules in Figure 2 for some type scheme σ. It is easy to see that there is at most one typing derivation for e, which also determines σ.

A *type normalized* Core-XML expression is one that satisfies the following two conditions. Type abstractions occur only as the bound expression of `let`-expressions; i.e., $\texttt{let}\ x\ \texttt{=}\ \Lambda\alpha_1.\Lambda\alpha_1....\Lambda\alpha_n.e_1\ \texttt{in}\ e_2$. We shall abbreviate this to $\texttt{let}\ x\ \texttt{=}\ \Lambda\alpha_1...\alpha_n.e_1\ \texttt{in}\ e_2$. Type applications are only allowed for variables; i.e., $x\{\tau_1\}\{\tau_2\}...\{\tau_n\}$. This will be written as $x\{\tau_1...\tau_n\}$.

Proviso: Henceforth all Core-XML expressions will be assumed to be type normalized.

Usually ML is presented as an implicitly typed language [Mil78, DM82]. By *erasing* all occurrences of types and type schemes in expressions (including curly brackets, colons, periods and Λ's) in the typing rules for Core-XML, we arrive at the implicitly typed language of *Core-ML expressions* [HM93]. In contrast to Core-XML an implicitly typed Core-ML expression e may have many different typing derivations. Every one of its typing derivations, however, corresponds to a unique explicitly typed Core-XML expression whose erasure is e, and vice versa.

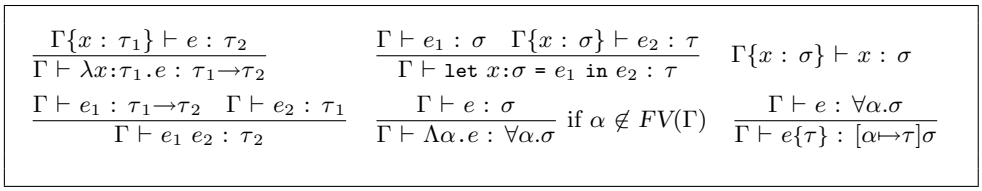
Our point of departure for boxing analysis is that we are given an explicitly typed Core-XML expression or, equivalently, a Core-ML expression and a specific typing derivation for it. Even though the specific nature of typing derivations is irrelevant for typability of implicitly typed expressions the quality of boxing analysis is very much dependent on which typing derivation is chosen for a Core-ML expression; i.e., one derivation may result in less boxing than another for the same Core-ML expression. More on this in Section 7.

3 Explicitly boxed Core-XML

Explicitly boxed Core-XML is a refinement of Core-XML in which representation types (boxed/unboxed types) and conversions between these are made explicit.

3.1 Representation types

Representation types ρ are just the standard types of Core-XML, together with one additional type constructor, [.].



$$\frac{\Gamma\{x:\tau_1\}\vdash e:\tau_2}{\Gamma\vdash \lambda x{:}\tau_1.e:\tau_1{\to}\tau_2} \qquad \frac{\Gamma\vdash e_1:\sigma \quad \Gamma\{x:\sigma\}\vdash e_2:\tau}{\Gamma\vdash \texttt{let } x{:}\sigma \texttt{ = } e_1 \texttt{ in } e_2:\tau} \qquad \Gamma\{x:\sigma\}\vdash x:\sigma$$

$$\frac{\Gamma\vdash e_1:\tau_1{\to}\tau_2 \quad \Gamma\vdash e_2:\tau_1}{\Gamma\vdash e_1\ e_2:\tau_2} \qquad \frac{\Gamma\vdash e:\sigma}{\Gamma\vdash \Lambda\alpha.e:\forall\alpha.\sigma}\ \text{if } \alpha\notin FV(\Gamma) \qquad \frac{\Gamma\vdash e:\forall\alpha.\sigma}{\Gamma\vdash e\{\tau\}:[\alpha{\mapsto}\tau]\sigma}$$

Figure 2: Typing rules for Core-XML

Types of the form $[\rho]$ are *boxed types*; they describe elements that have been boxed. Types with any other top-level type constructor (in our case only $\to$) are *unboxed types*; their elements are not boxed. Since doubly boxed representations are useless we prohibit boxed types of the form $[[\rho]]$. Boxed types may otherwise, however, occur inside both boxed and unboxed types. We add type variables denoted by metavariables α, β for unboxed and boxed types, respectively. *Type schemes* are now prenex-quantified representation types where the quantification is over boxed type variables *only*. Abstract syntax definitions of representation types, boxed and unboxed types, and type schemes are given in Figure 3.

$\rho \in RepType$; $\upsilon \in UnboxedTyped$; $\pi \in BoxedType$;
$\sigma \in PolyType$

$$\begin{array}{lcl} \rho & ::= & \upsilon \mid \pi \\ \upsilon & ::= & \alpha \mid \rho{\to}\rho \\ \pi & ::= & \beta \mid [\upsilon] \\ \sigma & ::= & \rho \mid \forall\beta.\sigma \end{array}$$

Figure 3: Representation types

Definition 1 The *(underlying) standard type* (or *type erause*) $|\rho|$ of representation type ρ is the type arrived at by erasing all occurrences of [.] in ρ and treating both boxed and unboxed type variables as standard type variables. We say that ρ *represents* $|\rho|$. We say ρ is *valid* for (closed) Core-XML expression e if e has type $|\rho|$. □

3.2 Coercions

Representation coercions (or simply *coercions*) are operations that coerce an element from one representation to another. The primitive coercions are

$$\texttt{box}_\upsilon : \upsilon \rightsquigarrow [\upsilon]$$
$$\texttt{unbox}_\upsilon : [\upsilon] \rightsquigarrow \upsilon$$

where $\texttt{box}_\upsilon$ coerces an unboxed element of type υ to a boxed representation, and $\texttt{unbox}_\upsilon$ takes such a boxed representation and coerces it back to the unboxed representation.

Beyond these primitive coercions we add the identity coercion ι_τ (at every type τ), composition of coercions c, c′ written in diagrammatical order c;c′ and coercions *induced* by the type constructors. In our case these are coercions of the form c→c′ and [c]. A coercion of the form c→c′ applies to functions f by "wrapping" them with the input coercion c and the output coercion c′. The result is an unboxed function where c is applied to the input before it is passed to f and c′ is applied to the result of f before it is returned as the output. A coercion of the form [c] applies c to the underlying unboxed value of a boxed representation and returns a boxed representation for the result. We will sometimes omit subscripts on coercions when these are not important for the presentation.

The rules for forming coercions are displayed in Figure 4. In the following we will use c, c′, d, etc. to denote arbitrary coercions.

The formation rules for coercions are sufficient to construct coercions that can transform a value from any one of its representations to any other representation:

Proposition 2 Let ρ_1, ρ_2 be arbitrary representation types. Then

$$|\rho_1| = |\rho_2| \Leftrightarrow (\exists \mathrm{c}) \vdash \mathrm{c} : \rho_1 \rightsquigarrow \rho_2.$$

□

Indeed this is possible even without the [.]-coercion constructor. We have added it solely to facilitate coercion factoring and simplification "underneath" boxed representations for $\psi^{\rightarrow}E_p^{\leftarrow}$-reduction modulo ϕ (see Section 5), where we may introduce box;unbox-redexes in order to eliminate unbox;box-redexes.

3.3 Type inference rules

The type inference system for explicitly boxed Core-XML is almost identical to the standard type system. See Figure 5. There are two noteworthy differences, however.

1. Quantification in type schemes is only over boxed type variables. The fact that these type variables indeed range over boxed types *only* is captured in the rule for type application: a polymorphically typed expression can only be applied to a boxed type, not an unboxed type.
2. There is an additional rule for applying coercions to expressions.

Definition 3 (Erasure, completion)
The *erasure* (or *underlying Core-XML expression*) $|e|$ of an explicitly boxed Core-XML expression e is the (standard) Core-XML expression arising from e by erasing all occurrences of coercions (including angled brackets) and replacing all representation type occurrences ρ by $|\rho|$. We say e is a *(boxing) completion* of $|e|$ *at type* ρ if e has type ρ. □

4 Axiomatization of completion congruence

A given Core-XML expression e can have many completions. Without going into the semantics of Core-XML we assume that all completions of e at a specific representation type

$$\vdash \iota_\rho : \rho \rightsquigarrow \rho \qquad \frac{\vdash \mathrm{c} : \rho_1 \rightsquigarrow \rho_1' \quad \vdash \mathrm{c}' : \rho_2 \rightsquigarrow \rho_2'}{\vdash \mathrm{c}{\rightarrow}\mathrm{c}' : \rho_1'{\rightarrow}\rho_2 \rightsquigarrow \rho_1{\rightarrow}\rho_2'} \qquad \frac{\vdash \mathrm{c} : \rho \rightsquigarrow \rho' \quad \vdash \mathrm{c}' : \rho' \rightsquigarrow \rho''}{\vdash \mathrm{c};\mathrm{c}' : \rho \rightsquigarrow \rho''}$$

$$\vdash \mathtt{box}_\upsilon : \upsilon \rightsquigarrow [\upsilon] \quad \vdash \mathtt{unbox}_\upsilon : [\upsilon] \rightsquigarrow \upsilon \qquad \frac{\vdash \mathrm{c} : \upsilon \rightsquigarrow \upsilon'}{\vdash [\mathrm{c}] : [\upsilon] \rightsquigarrow [\upsilon']}$$

Figure 4: Coercion formation rules

$$\frac{\Gamma\{x : \rho_1\} \vdash e : \rho_2}{\Gamma \vdash \lambda x{:}\rho_1 . e : \rho_1{\rightarrow}\rho_2} \qquad \frac{\Gamma \vdash e_1 : \rho_1{\rightarrow}\rho_2 \quad \Gamma \vdash e_2 : \rho_1}{\Gamma \vdash e_1\ e_2 : \rho_2}$$

$$\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma\{x : \sigma\} \vdash e_2 : \rho}{\Gamma \vdash \mathtt{let}\ x{:}\sigma = e_1\ \mathtt{in}\ e_2 : \rho} \qquad \Gamma\{x : \sigma\} \vdash x : \sigma$$

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \Lambda\beta . e : \forall\beta.\sigma}\ \text{if } \beta \notin FV(\Gamma) \qquad \frac{\Gamma \vdash e : \forall\beta.\sigma}{\Gamma \vdash e\{\pi\} : [\beta{\mapsto}\pi]\sigma}$$

$$\frac{\Gamma \vdash e : \tau \quad \vdash \mathrm{c} : \rho \rightsquigarrow \rho'}{\Gamma \vdash \langle \mathrm{c} \rangle e : \rho'}$$

Figure 5: Typing rules for the explicitly boxed Core-ML language

have the same observational (input/output) behavior, but possibly different performance. In fact we shall assume *nothing else* about the semantics of explicitly boxed Core-XML, and in fact nothing at all about the semantics of standard Core-XML.

Definition 4 (Completion congruence) Representation coercions c and c′ are *congruent*, written $\mathrm{c} \cong \mathrm{c}'$, if they have the same *type signature* $\rho \rightsquigarrow \rho'$. Explicitly boxed expressions e and e' are *congruent*, written $e \cong e'$, if they have the same erasure and type (under the same type assumptions); that is, they are completions of the same expression at the same representation type. □

In this section we shall give an axiomatization of completion congruence by a typed equality theory. The point of this is that the axioms can be grouped into a "pure" equality theory E that moves coercions along data flow paths without eliminating them, and a small group consisting of two axioms that express that boxing composed with unboxing in either order is equal to the identity coercion. In Section 5 we interpret the second group as rewriting rules modulo E to capture the intuition that boxing and unboxing coercions are more expensive than the identity, but that the effects of moving along data flow paths are *not* taken into account. (Note that for any specific semantics moving coercions *does* make quite a difference — this is the most important part of a semantics that is disregarded in our treatment!)

$$\begin{aligned}
(\mathrm{c};\mathrm{c}');\mathrm{c}'' &= \mathrm{c};(\mathrm{c}';\mathrm{c}'') && (1)\\
\mathrm{c};\iota &= \mathrm{c} && (2)\\
\iota;\mathrm{c} &= \mathrm{c} && (3)\\
(\mathrm{c}_1 \rightarrow \mathrm{c}_2);(\mathrm{c}_1' \rightarrow \mathrm{c}_2') &= (\mathrm{c}_1';\mathrm{c}_1) \rightarrow (\mathrm{c}_2;\mathrm{c}_2') && (4)\\
\iota \rightarrow \iota &= \iota && (5)\\
[\mathrm{c}_1];[\mathrm{c}_2] &= [\mathrm{c}_1;\mathrm{c}_2] && (6)\\
[\iota] &= \iota && (7)\\
\mathrm{c};\mathtt{box}_{\upsilon'} &= \mathtt{box}_\upsilon;[\mathrm{c}] && (8)\\
\mathtt{unbox}_\upsilon;\mathrm{c} &= [\mathrm{c}];\mathtt{unbox}_{\upsilon'} && (9)
\end{aligned}$$

Figure 6: Equality rules for coercions

$$\begin{aligned}
\mathtt{box}_\upsilon;\mathtt{unbox}_\upsilon &= \iota \quad (\phi)\\
\mathtt{unbox}_\upsilon;\mathtt{box}_\upsilon &= \iota \quad (\psi)
\end{aligned}$$

Figure 7: The ϕ and ψ rules for coercions

4.1 Coercion coherence

Consider the equality axioms in Figures 6 and 7 for coercions. We assume that the coercions on both sides of an equality are well-formed and have the same type signature. We denote the single equality axiom $\mathtt{box};\mathtt{unbox} = \iota$ by ϕ, and $\mathtt{unbox};\mathtt{box} = \iota$ by ψ.

Definition 5 (Coercion equality)
We say c and c' are A-*equal*, written $A \vdash \mathrm{c} = \mathrm{c}'$, if $\mathrm{c} = \mathrm{c}'$ is derivable from the equality axioms A and the equations in Figure 6 together with reflexivity, symmetry, transitivity and compatibility of $=$ with arbitrary contexts. □

If A in the definition is empty we say c and c′ are *equal* and write $\vdash \mathrm{c} = \mathrm{c}'$. Note that equality is *not* just syntactic identity; e.g., we have $\vdash \iota_\alpha \rightarrow \iota_\alpha = \iota_{\alpha\rightarrow\alpha}$.

Lemma 6 (Coherence of coercions)
Coercions c and c′ are congruent if and only if they are $\phi\psi$-equal; i.e., $\mathrm{c} \cong \mathrm{c}'$ **iff** $\phi\psi \vdash \mathrm{c} = \mathrm{c}'$ □

4.2 Expression coherence

Let us extend the equality axioms for coercions with the equality axioms for explicitly boxed Core-XML expressions in Figures 8 and 9. The expressions on both sides of an equality are assumed to be well-formed and to have the same type in a single type environment. In other words, the equations in Figures 8 and 9 should be understood as abbreviations for more complex rules for typed equality. For example, rule 10 is an abbreviation for:

$$\frac{\Gamma \vdash e : \rho \quad \Gamma \vdash \langle \iota_\rho \rangle e : \rho}{\Gamma \vdash \langle \iota_\rho \rangle e = e : \rho}$$

DEFINITION 7 (Completion equality)

We say $e$and $e'$are A-equal, $A \vdash e = e'$, if $e = e'$ is derivable from the axioms A together with equations in Figures 6 and 8 and rules for reflexivity, symmetry, transitivity and compatibility of $=$. □

THEOREM 8 (Coherence of completions) Explicitly boxed expressions e' and e'' are congruent if and only if they are $E\phi\psi$-equal; i.e.,

$$e' \cong e'' \textbf{ iff } E\phi\psi \vdash e' = e''$$

PROOF: (If) Assume $E\phi\psi \vdash e' = e''$. By inspection of E we can verify that e' and e'' have the same erasure. Since both e' and e'' are completions at the same type they are congruent, i.e., $e' \cong e''$.

(Only if) We will prove this by induction on the structure of the common erasure of e' and e''. We call an explicitly boxed Core-XML expression *head coercion free* (c.f. [CG90]) if it is not of the form $\langle \mathrm{c} \rangle e$. Without loss of generality we may assume that coercions are only applied to head coercion free expressions and that every head coercion free subexpression has exactly one coercion applied to it. This follows from $\langle \mathrm{c} \rangle \langle \mathrm{c}' \rangle e = \langle \mathrm{c}' ; \mathrm{c} \rangle e$, and $e = \langle \iota \rangle e$ (see Figure 8).

Now assume that we have $\Gamma \vdash e' : \rho$ and $\Gamma \vdash e'' : \rho$:

Base case 1: $|e'| \equiv x\{\overline{\tau}\}$. Let $e' \equiv \langle \mathrm{c}' \rangle x\{\overline{\pi'}\}$ and $e'' \equiv \langle \mathrm{c}'' \rangle x\{\overline{\pi''}\}$. Let τ'be the type of $x\{\overline{\pi'}\}$ and τ'' the type of $x\{\overline{\pi''}\}$. Note that $\tau'$and τ'' have the same type erasure $\tau = \tau'$. By Proposition 2 there exists a coercion cwith type signature $\tau' \rightsquigarrow \tau''$. Since $\mathrm{c};\mathrm{c}'$ and c'' both have the same type signature we obtain:

$$\begin{array}{lcl} \langle \mathrm{c}' \rangle x\{\overline{\pi'}\} & = & \text{Lemma 6} \\ \langle \mathrm{c};\mathrm{c}'' \rangle x\{\overline{\pi'}\} & = & \text{Equation (11), Figure 8} \\ \langle \mathrm{c}'' \rangle \langle \mathrm{c} \rangle x\{\overline{\pi'}\} & = & \text{Equation (15), Figure 9} \\ \langle \mathrm{c}'' \rangle x\{\overline{\pi''}\} & & \end{array}$$

Base case 2: $|e'| \equiv x$ follows from Lemma 6.

Inductive step 1: $|e'| \equiv \lambda x{:}\tau_{\mathrm{x}}.e_1$. Assume that $e' \equiv \langle \mathrm{c}' \rangle \lambda x{:}\rho'.e_1{}'$ and $e'' \equiv \langle \mathrm{c}'' \rangle \lambda x{:}\rho''.e_1{}''$. Since c' has type signature $\rho' \rightarrow \rho_1{}' \rightsquigarrow \rho$ and c'' has type signature $\rho'' \rightarrow \rho_1{}'' \rightsquigarrow \rho$ for some types $\rho_1{}'$ and $\rho_1{}''$ by Lemma 6 there exists a coercion d such that $\phi\psi \vdash \mathrm{c}' = \mathrm{c}_1{}' \rightarrow \mathrm{c}_2{}';\mathrm{d}$ and $\phi\psi \vdash \mathrm{c}'' = \mathrm{c}_1{}'' \rightarrow \mathrm{c}_2{}'';\mathrm{d}$. In fact, without loss of generality $\mathrm{d} = \iota_\rho$ or $\mathrm{d} = \mathtt{box}_\upsilon$ where $\rho = [\upsilon]$. So in either cases d is completely determined by ρ.

This means that, under $E\phi\psi$-equality, we have

$$e' = \langle \mathrm{d} \rangle \lambda x{:}\rho_{\mathrm{d}}.\langle \mathrm{c}_2{}' \rangle [x \mapsto \langle \mathrm{c}_1{}' \rangle x] e_1{}'$$

and

$$e'' = \langle \mathrm{d} \rangle \lambda x{:}\rho_{\mathrm{d}}.\langle \mathrm{c}_2{}'' \rangle [x \mapsto \langle \mathrm{c}_1{}'' \rangle x] e_1{}''$$

for some type ρ_{d} and by induction

$$\langle \mathrm{c}_2{}' \rangle [x \mapsto \langle \mathrm{c}_1{}' \rangle x] e_1{}' = \langle \mathrm{c}_2{}'' \rangle [x \mapsto \langle \mathrm{c}_1{}'' \rangle x] e_1{}''$$

which proves that $e' = e''$.

Inductive step 2: $|e'| \equiv e_1\ e_2$. Let $e' \equiv \langle \mathrm{c}' \rangle (e_1{}'\ e_2{}')$ and $e'' \equiv \langle \mathrm{c}'' \rangle (e_1{}''\ e_2{}'')$. We know that there exists a coercion d such that $\langle \mathrm{d} \rangle e_2{}'$ and $e_2{}''$ have the same type. Let the inverse d^{-1} of a coercion d be a coercion such that $\phi\psi \vdash \mathrm{d}^{-1};\mathrm{d} = \iota$ (such a coercion always exists according to Proposition 2). We then have, under $E\phi\psi$-equality:

$$\begin{array}{ll} \langle \mathrm{c}' \rangle (e_1{}'\ e_2{}') & = \\ \langle \iota \rightarrow \mathrm{c}' \rangle e_1{}'\ e_2{}' & = \\ \langle (\mathrm{d}^{-1};\mathrm{d}) \rightarrow \mathrm{c}' \rangle e_1{}'\ e_2{}' & = \\ \langle \mathrm{d} \rightarrow \iota \rangle \langle \mathrm{d}^{-1} \rightarrow \mathrm{c}' \rangle e_1{}'\ e_2{}' & = \\ \langle \mathrm{d}^{-1} \rightarrow \mathrm{c}' \rangle e_1{}'\ \langle \mathrm{d} \rangle e_2{}' & = \text{(induction)} \\ \langle \mathrm{d}^{-1} \rightarrow \mathrm{c}' \rangle e_1{}'\ e_2{}'' & = \text{(induction)} \\ \langle \iota \rightarrow \mathrm{c}'' \rangle e_1{}''\ e_2{}'' & = \\ \langle \mathrm{c}'' \rangle (e_1{}''\ e_2{}'') & \end{array}$$

Inductive step 3: $|e'| \equiv \mathtt{let}\ x = \Lambda\overline{\alpha}.e_1\ \mathtt{in}\ e_2$. We will only show this in the case where the tuple $\overline{\alpha}$ has length one. The other cases are similar. So the case we prove is $|e'| \equiv \mathtt{let}\ x = \Lambda\alpha.e_1\ \mathtt{in}\ e_2$. Let $e' \equiv \langle \mathrm{d}' \rangle (\mathtt{let}\ x = \Lambda\beta.e_1{}'\ \mathtt{in}\ e_2{}')$ and $e'' \equiv \langle \mathrm{d}'' \rangle (\mathtt{let}\ x = \Lambda\beta.e_1{}''\ \mathtt{in}\ e_2{}'')$. Since $e_1{}'$ and $e_1{}''$ have the same type erasure there exists a coercion $\mathrm{c}(\beta)$ such that $\langle \mathrm{c}(\beta) \rangle e_1{}'$ and $e_1{}''$ have the same type. We then have:

$$\begin{array}{ll} \langle \mathrm{d}' \rangle (\mathtt{let}\ x = \Lambda\beta.e_1{}'\ \mathtt{in}\ e_2{}') & = \\ \mathtt{let}\ x = \Lambda\beta.e_1{}'\ \mathtt{in}\ \langle \mathrm{d}' \rangle e_2{}' & = \\ \mathtt{let}\ x = \Lambda\beta.\langle \mathrm{c}(\beta)^{-1} \rangle \langle \mathrm{c}(\beta) \rangle e_1{}'\ \mathtt{in}\ \langle \mathrm{d}' \rangle e_2{}' & = \\ \mathtt{let}\ x = \Lambda\beta.\langle \mathrm{c}(\beta) \rangle e_1{}'\ \mathtt{in} & \\ \quad \langle \mathrm{d}' \rangle ([x\{[\upsilon_{\mathrm{i}}]\} \mapsto \langle \mathrm{c}(\upsilon_{\mathrm{i}})^{-1} \rangle (x\{[\upsilon_{\mathrm{i}}]\})] e_2{}') & = \text{(induction)} \\ \mathtt{let}\ x = \Lambda\beta.e_1{}''\ \mathtt{in} & \\ \quad \langle \mathrm{d}' \rangle ([x\{[\upsilon_{\mathrm{i}}]\} \mapsto \langle \mathrm{c}(\upsilon_{\mathrm{i}})^{-1} \rangle (x\{[\upsilon_{\mathrm{i}}]\})] e_2{}') & = \text{(induction)} \\ \mathtt{let}\ x = \Lambda\beta.e_1{}''\ \mathtt{in}\ \langle \mathrm{d}'' \rangle e_2{}'' & = \\ \langle \mathrm{d}'' \rangle (\mathtt{let}\ x = \Lambda\beta.e_1{}''\ \mathtt{in}\ e_2{}'') & \end{array}$$

□

This shows that if any two congruent completions of a Core-XML expression are observationally indistinguishable — a prerequisite for our assumption that we are allowed to pick any completion of a program at all — then the observational congruence of explicitly boxed Core-XML must satisfy $E\phi\psi$-equality, and vice versa. Otherwise one could find two congruent completions with different observable behavior.

4.3 Positive and negative coercions

It is difficult to reason directly about reduction systems on congruence classes defined by an equational theory. What we would actually like to do is to characterize E-equality by a canonical term rewriting system that commutes with ψ-reduction and ϕ-reduction. Finding a confluent rewriting system for E-equality is not straightforward, however. In particular, the E-equations cannot simply be oriented in one direction or the other since they will inevitably lead to critical pairs without common reducts. Consider for example the rules of Figure 9 oriented from left to right. In the expression

$$(\langle \mathrm{c} \rightarrow \mathrm{c}' \rangle \lambda x.e)\ e'$$

both rules 12 and 13 are applicable, and Knuth-Bendix completion appears not to terminate. Note that by following one reduction path we might fail to eliminate a box/unbox pair using ϕ or ψ that could be eliminated by following the other path.

The main idea behind the reduction system for $E\phi\psi$-equality we are about to describe is that coercions may be split up into two kinds of coercions that interact differently with the E-equations. The two kinds of coercions are called *positive* and *negative coercions*.

DEFINITION 9 (Positive and negative coercions)

$$\langle\iota\rangle e = e \quad (10)$$
$$\langle \mathrm{c}\rangle\langle \mathrm{d}\rangle e = \langle \mathrm{d};\mathrm{c}\rangle e \quad (11)$$

Figure 8: Equality rules for coercion application

$$\langle \mathrm{c}\to\mathrm{d}\rangle \lambda x.e = \lambda x.\langle \mathrm{d}\rangle([x\mapsto\langle \mathrm{c}\rangle x]e) \quad (12)$$
$$(\langle \mathrm{c}\to\mathrm{d}\rangle e)\ e' = \langle \mathrm{d}\rangle(e\ (\langle \mathrm{c}\rangle e')) \quad (13)$$
$$\langle \mathrm{d}\rangle \texttt{let } x = \Lambda\overline{\beta}.\langle \mathrm{c}\rangle e \texttt{ in } [x^i\mapsto x\{\pi\}]e' = \texttt{let } x = \Lambda\overline{\beta}.e \texttt{ in } \langle \mathrm{d}\rangle([x^i\mapsto\langle \mathrm{c}\rangle x\{\pi\}]e') \quad (14)$$
$$\langle \mathrm{c}\rangle x\{\overline{\pi}\} = x\{\overline{\pi}'\} \quad (15)$$

Figure 9: Equality rules for explicitly boxed expressions

A coercion c is *positive* if c:+ is derivable from the rules in Figure 10 and *negative* if c:− is derivable. □

A coercion may be neither positive nor negative. By adding a superscript + or − to a coercion we indicate that a coercion is in fact positive or negative. (These annotations can be regarded as side conditions that have to hold before a rule may be applied. So in the equations of Figure 11 a superscript + on a coercion means that the coercion has to be positive for the rule to be applicable.) However, it can be shown that it is always possible to factor a coercion c into a positive coercion ${\mathrm{c}_1}^+$ and a negative coercion ${\mathrm{c}_2}^-$ such that $\phi\psi \vdash \mathrm{c} = {\mathrm{c}_1}^+;{\mathrm{c}_2}^-$, and into a negative coercion ${\mathrm{c}_1}'^-$ and a positive coercion ${\mathrm{c}_2}'^+$ such that $\phi\psi \vdash \mathrm{c} = {\mathrm{c}_1}'^-;{\mathrm{c}_2}'^+$.

$$\iota_\tau : + \quad \texttt{box}_\upsilon : + \quad \frac{\mathrm{c}:- \quad \mathrm{d}:+}{\mathrm{c}\to\mathrm{d}:+} \quad \frac{\mathrm{c}:+ \quad \mathrm{d}:+}{\mathrm{c};\mathrm{d}:+} \quad \frac{\mathrm{c}:+}{[\mathrm{c}]:+}$$
$$\iota_\tau : - \quad \texttt{unbox}_\upsilon : - \quad \frac{\mathrm{c}:+ \quad \mathrm{d}:-}{\mathrm{c}\to\mathrm{d}:-} \quad \frac{\mathrm{c}:- \quad \mathrm{d}:-}{\mathrm{c};\mathrm{d}:-} \quad \frac{\mathrm{c}:-}{[\mathrm{c}]:-}$$

Figure 10: Positive and negative coercions

The positive coercions alone define a subtype hierarchy on representation types.

DEFINITION 10 We define $\rho \le \rho'$ if there exists a positive coercion c^+ such that $\vdash \mathrm{c}^+ : \rho \rightsquigarrow \rho'$. □

PROPOSITION 11 The representation types of any (standard) type τ (i.e., representation types whose type erasure is τ) form a finite lattice under $\le$. □

4.4 A Polarized Axiomatization of congruence completions

To define our reduction system we first define a new axiomatization of completion congruence which takes the polarity of coercions into account. Most importantly, the new equations can be easily oriented in one or the other direction to yield a confluent rewriting system together with ϕ respectively ψ-reduction.

First we need to define notation used in our new axiomatization.

DEFINITION 12 For every representation type $\rho(\overline{\beta})$, i.e. with type variables $\overline{\beta}$, we define a coercion $\widehat{\rho}(\overline{\mathrm{c}},\overline{\mathrm{d}})$ parameterized over the tuples of coercions $\overline{\mathrm{c}}$ and $\overline{\mathrm{d}}$ in the following way:

$$\widehat{\beta_\mathrm{i}}(\overline{\mathrm{c}},\overline{\mathrm{d}}) = \mathrm{c}!\mathrm{i}$$
$$\widehat{\rho_1\to\rho_2}(\overline{\mathrm{c}},\overline{\mathrm{d}}) = \widehat{\rho_1}(\overline{\mathrm{d}},\overline{\mathrm{c}})\to\widehat{\rho_2}(\overline{\mathrm{c}},\overline{\mathrm{d}})$$
$$\widehat{[\upsilon]}(\overline{\mathrm{c}},\overline{\mathrm{d}}) = [\![\widehat{\upsilon}(\overline{\mathrm{c}},\overline{\mathrm{d}})]\!]$$

□

The equations of the new axiomatization, shown in Figure 11, are those of Figure 9 where all but the last equation have been split into two polarized equations with side conditions on the *polarity* (+ or −) of the coercions occurring in them. The only exception is the last equation in Figure 9. If we had chosen to treat it analogously to the other equations we would have obtained the following rules:

$$\langle \mathrm{c}^-\rangle x\{\overline{\pi}\} = x\{\overline{\pi}'\} \quad (15^{-\prime})$$
$$\langle \mathrm{c}^+\rangle x\{\overline{\pi}\} = x\{\overline{\pi}'\} \quad (15^{+\prime})$$

This would, however, not be strong enough to characterize $E\phi\psi$-equality.

If one examines rule $15^{-\prime}$ more closely one will notice that c must have signature $[\beta\mapsto\pi]\rho \rightsquigarrow [\beta\mapsto\pi']\rho$ where the type of x is $\forall\beta.\rho$ (we assume without loss of generality that the type of x is only quantified over one type variable). Furthermore, one can show $\phi\psi \vdash \mathrm{c} = \widehat{\rho}(\mathrm{d},\mathrm{d}^{-1})$, where $\vdash \mathrm{d} : \pi \rightsquigarrow \pi'$ and $\vdash \mathrm{d}^{-1} : \pi' \rightsquigarrow \pi$. From this one can see, if we regard rules $15^{-\prime}$ and $15^{+\prime}$ as a left-to-right rewrite rule, that rule $15^{-\prime}$ is more restrictive than rule 15^- since it requires more of the structure of the coercion involved.

The following lemma will be used in proving Theorem 14:

LEMMA 13 Let $\overline{\mathrm{c_1}}$, $\overline{\mathrm{c_2}}$, $\overline{\mathrm{d_1}}$, etc. be tuples of coercions. Then the following results hold:

1. $\phi\psi \vdash \widehat{\rho}(\overline{\mathrm{c_1};\mathrm{c_2}},\overline{\mathrm{d_1};\mathrm{d_2}}) = \widehat{\rho}(\overline{\mathrm{c_1}},\overline{\mathrm{d_2}});\widehat{\rho}(\overline{\mathrm{c_2}},\overline{\mathrm{d_1}})$
2. $\phi\psi \vdash \widehat{\rho}(\overline{\mathrm{c}},\overline{\iota});\widehat{\rho}(\overline{\iota},\overline{\mathrm{d}}) = \widehat{\rho}(\overline{\iota},\overline{\mathrm{d}});\widehat{\rho}(\overline{\mathrm{c}},\overline{\iota})$ □

Let E_p be the set of equations in Figure 11. We have the following theorem:

THEOREM 14 For all completions e and e':

$$E\phi\psi \vdash e = e' \quad \textbf{iff} \quad E_p\phi\psi \vdash e = e'$$

PROOF: (Only if) This is trivial for all equations except equations 15^+ and 15^-. We will therefore cover one of these cases here. Equation 15^- is shown by the following:

$$\langle\widehat{\rho}(\mathrm{c}^-,\iota)\rangle x\{\pi\} =$$
$$\langle\widehat{\rho}(\mathrm{c}^-,(\mathrm{c}^-)^{-1};\mathrm{c}^-)\rangle x\{\pi\} = \text{(Lemma 13)}$$
$$\langle\widehat{\rho}(\iota,\mathrm{c}^-)\rangle\langle\widehat{\rho}(\mathrm{c}^-,(\mathrm{c}^-)^{-1})\rangle x\{\pi\} = \text{(13)}$$
$$\langle\widehat{\rho}(\iota,\mathrm{c}^-)\rangle x\{\pi'\}$$

$$
\begin{array}{rcll}
\langle(\mathrm{c}^+ \to \mathrm{d}^-)\rangle\lambda x.e & = & \lambda x.\langle \mathrm{d}^-\rangle([x{\mapsto}\langle \mathrm{c}^+\rangle x]e) & (12^-)\\
\lambda x.\langle \mathrm{d}^+\rangle([x{\mapsto}\langle \mathrm{c}^-\rangle x]e) & = & \langle(\mathrm{c}^- \to \mathrm{d}^+)\rangle\lambda x.e & (12^+)\\
\langle \mathrm{d}^-\rangle(e\ (\langle \mathrm{c}^+\rangle e')) & = & (\langle \mathrm{c}^+ \to \mathrm{d}^-\rangle e)\ e' & (13^-)\\
(\langle(\mathrm{c}^- \to \mathrm{d}^+)\rangle e)\ e' & = & \langle \mathrm{d}^+\rangle(e\ (\langle \mathrm{c}^-\rangle e')) & (13^+)\\
\langle \mathrm{d}^-\rangle\texttt{let}\ x = \Lambda\overline{\beta}.e\ \texttt{in}\ [x^i{\mapsto}\langle \mathrm{c}^-\rangle x\{\pi\}]e' & = & \texttt{let}\ x = \Lambda\overline{\beta}.\langle \mathrm{c}^-\rangle e\ \texttt{in}\ \langle \mathrm{d}^-\rangle([x^i{\mapsto}x\{\pi'\}]e') & (14^-)\\
\texttt{let}\ x = \Lambda\overline{\beta}.\langle \mathrm{c}^+\rangle e\ \texttt{in}\ \langle \mathrm{d}^+\rangle([x^i{\mapsto}x\{\pi\}]e') & = & \langle \mathrm{d}^+\rangle\texttt{let}\ x = \Lambda\overline{\beta}.e\ \texttt{in}\ [x^i{\mapsto}\langle \mathrm{c}^+\rangle x\{\pi'\}]e' & (14^+)\\
\langle\widehat{\rho}(\overline{\mathrm{c}}^-,\overline{\iota})\rangle x\{\overline{\pi}\} & = & \langle\widehat{\rho}(\overline{\iota},\overline{\mathrm{c}}^-)\rangle x\{\overline{\pi}'\} & (15^-)\\
\langle\widehat{\rho}(\overline{\iota},\overline{\mathrm{c}}^+)\rangle x\{\overline{\pi}\} & = & \langle\widehat{\rho}(\overline{\mathrm{c}}^+,\overline{\iota})\rangle x\{\overline{\pi}'\} & (15^+)
\end{array}
$$

Figure 11: Polarized Equality equations for explicitly boxed expressions

Equation 15^- is shown similarly.

(if) All cases except equation 15 are fairly straightforward. We will cover equations 13 and 15. First equation 13:

$$
\begin{array}{ll}
(\langle \mathrm{c}{\to}\mathrm{d}\rangle e)\ e' & = \text{(factoring)}\\
(\langle \mathrm{c}^+{\to}\mathrm{d}^-\rangle\langle \mathrm{c}^-{\to}\mathrm{d}^+\rangle e)\ e' & = (13^-)\\
\langle \mathrm{d}^-\rangle((\langle \mathrm{c}^-{\to}\mathrm{d}^+\rangle e)\ \langle \mathrm{c}^+\rangle e') & = (13^+)\\
\langle \mathrm{d}^-\rangle\langle \mathrm{d}^+\rangle(e\ (\langle \mathrm{c}^-\rangle\langle \mathrm{c}^+\rangle e')) & =\\
\langle \mathrm{d}\rangle(e\ (\langle \mathrm{c}\rangle e'))
\end{array}
$$

Then on to equation 15. We will only proof this for the case where the type of x have the form $\forall\beta.\rho$ (the proof for $\forall\overline{\beta}.\rho$ is similar). The signature of c in equation 15 is then $[\beta{\mapsto}\pi]\rho \rightsquigarrow [\beta{\mapsto}\pi']\rho$ and we may therefore prove $\phi\psi \vdash \mathrm{c} = \widehat{\rho}(\mathrm{d},\mathrm{d}^{-1})$ where $\vdash \mathrm{d} : \pi \rightsquigarrow \pi'$ and $\vdash \mathrm{d}^{-1} : \pi' \rightsquigarrow \pi$. We now have:

$$
\begin{array}{ll}
\langle \mathrm{c}\rangle x\{\pi\} & =\\
\langle\widehat{\rho}(\mathrm{d},\mathrm{d}^{-1})\rangle x\{\pi\} & = \text{(Lemma 13)}\\
\langle\widehat{\rho}(\iota,\mathrm{d}^{-1})\rangle\langle\widehat{\rho}(\mathrm{d},\iota)\rangle x\{\pi\} & = \text{(factoring)}\\
\langle\widehat{\rho}(\iota,\mathrm{d}^{-1})\rangle\langle\widehat{\rho}(\mathrm{d}^-,\iota)\rangle\langle\widehat{\rho}(\mathrm{d}^+,\iota)\rangle x\{\pi\} & = (15^+)\\
\langle\widehat{\rho}(\iota,\mathrm{d}^{-1})\rangle\langle\widehat{\rho}(\mathrm{d}^-,\iota)\rangle\langle\widehat{\rho}(\iota,\mathrm{d}^+)\rangle x\{\pi''\} & = \text{(Lemma 13)}\\
\langle\widehat{\rho}(\iota,\mathrm{d}^{-1})\rangle\langle\widehat{\rho}(\iota,\mathrm{d}^+)\rangle\langle\widehat{\rho}(\mathrm{d}^-,\iota)\rangle x\{\pi''\} & = (15^-)\\
\langle\widehat{\rho}(\iota,\mathrm{d}^{-1})\rangle\langle\widehat{\rho}(\iota,\mathrm{d}^+)\rangle\langle\widehat{\rho}(\iota,\mathrm{d}^-)\rangle x\{\pi'\} & = \text{(Lemma 13)}\\
\langle\widehat{\rho}(\iota,\mathrm{d}^{-1})\rangle\langle\widehat{\rho}(\iota,\mathrm{d})\rangle x\{\pi'\} & = \text{(Lemma 13)}\\
x\{\pi'\}
\end{array}
$$

□

5 Reduction of completions

The axiomatization of completion congruence by $E_p\phi\psi$-equality gives a "local" characterization of congruence of completions: Any completion of a Core-XML expression e can be transformed to any other completion of e at the same type by substituting equals for equals; i.e., by replacing any subexpression that matches one side of an equation by the other side. In this section we treat ϕ and ψ as rewriting rules corresponding to "improvements" of a completion, but taken modulo all the remaining equations.

5.1 Optimal coercions

Our aim is to find completions with a minimum of boxing und unboxing operations without, however, taking the actual operational semantics of explicitly boxed Core-XML into account beyond the fact that it must satisfy $E_p\phi\psi$-equality. Let us take a look at the equations for coercions then. Clearly, ϕ and ψ eliminate primitive coercions when applied as left-to-right rewriting rules whereas the remaining-coercion equations just express differences in the presentation of a coercion.

$$
\begin{array}{rcll}
(\mathrm{c};\mathrm{c}');\mathrm{c}'' & = & \mathrm{c};(\mathrm{c}';\mathrm{c}'') & \\
\mathrm{c};\iota & \Longrightarrow & \mathrm{c} & \\
\iota;\mathrm{c} & \Longrightarrow & \mathrm{c} & \\
\iota \to \iota & \Longrightarrow & \iota & \\
(\mathrm{c} \to \mathrm{d});(\mathrm{c}' \to \mathrm{d}') & \Longrightarrow & (\mathrm{c}';\mathrm{c}) \to (\mathrm{d};\mathrm{c}') & \\
\texttt{[}\iota\texttt{]} & \Longrightarrow & \iota & \\
\texttt{[c];[c']} & \Longrightarrow & \texttt{[c;c']} & \\
\texttt{box;[c]} & \Longrightarrow & \mathrm{c};\texttt{box} & \\
\texttt{[c];unbox} & \Longrightarrow & \texttt{unbox;c} & \\
 & & & \\
\texttt{box;unbox} & \Longrightarrow & \iota & (\phi^{\rightarrow})\\
\texttt{unbox;box} & \Longrightarrow & \iota & (\psi^{\rightarrow})\\
\texttt{unbox;c;box} & \Longrightarrow & \texttt{[c]} & (\psi^{\rightarrow})
\end{array}
$$

Figure 12: Coercion reduction

Definition 15 (Formally optimal coercions)
A coercion c is *(formally) optimal* if all congruent coercions $\mathrm{c}' = \mathrm{c}$ can be reduced to c by $\phi\psi$-reduction. □

Clearly, every coercion equal to an optimal coercion is also optimal. We shall see that, for every coercion type signature $\tau \rightsquigarrow \tau'$ with $|\tau| = |\tau'|$, optimal coercions exist and are unique modulo coercion equality.

Consider the coercion reduction rules in Figure 12. We write $\mathrm{c} \Longrightarrow^*_R \mathrm{c}'$ if c reduces to c' by these rules.

Theorem 16
The coercion reduction system in Figure 12 has the following properties.

1. It is confluent.
2. It is strongly normalizing.
3. If $\mathrm{c} \Longrightarrow^*_R \mathrm{c}'$then $\vdash \mathrm{c} \Longrightarrow^*_{\phi\psi} \mathrm{c}'$.
4. It preserves polarity; that is, if c is positive or negative and $\mathrm{c} \Longrightarrow^*_R \mathrm{c}'$, then c' is also positive, respectively negative.
5. If c is a normal form then:
   - $\mathrm{c} \equiv \iota$,

- $c \equiv \texttt{box}$,
- $c \equiv \texttt{unbox}$,
- $c \equiv c' \to c''$, $c \equiv (c' \to c'')\texttt{;box}$, or $c \equiv \texttt{unbox;}(c' \to c'')$ where c' and c'' are normal forms, or
- $c \equiv \texttt{[}c'\texttt{]}$ where c' is a normal form.

□

This theorem guarantees that optimal coercions exist and are unique:

COROLLARY 17
For all ρ, ρ' with $|\rho| = |\rho'|$ there exists a unique optimal coercion $\vdash c : \rho \rightsquigarrow \rho'$. □

By analysis of R-normal forms we can also guarantee that all optimal coercions can be uniquely $+/-$ and $-/+$ factored:

COROLLARY 18

1. Every optimal coercion c has a unique $+/-$-factoring; that is, there exist unique d_1^+, d_2^- such that $\vdash c = d_1^+ ; d_2^-$.
2. Every optimal coercion c has a unique $-/+$-factoring; that is, there exist unique d_1^-, d_2^+ such that $\vdash c = d_1^- ; d_2^+$.

□

5.2 Optimal completions

We saw that all congruent coercions can be $\phi\psi$-reduced to a unique optimal coercion. For explicitly boxed Core-XML expressions we could interpret ϕ and ψ as left-to-right rewriting rules *modulo* (or *under*) E; that is, on the E-congruence classes of explicitly boxed Core-XML expressions. This expresses that we consider any two E-equal completions as indistinguishable in terms of boxing performance in a formal sense whereas rewriting with ϕ or ψ is an improvement of the boxing performance of a completion.

Unfortunately, $\phi\psi$-reduction on E-congruence classes is not Church-Rosser; that is, there are congruent completions that have no common reduct. Consider, for example, the two completions

$$e_1 \equiv \texttt{let } id{:}\forall\beta.\beta\to\beta = \Lambda\beta.\lambda y{:}\beta.\, y \texttt{ in}$$
$$(\lambda x{:}\texttt{int}.\, x + \langle\texttt{unbox}\rangle(id\{\texttt{[int]}\}\ \langle\texttt{box}\rangle x))$$
$$(\texttt{if ... then 2 else } \langle\texttt{unbox}\rangle(id\{\texttt{[int]}\}\ \langle\texttt{box}\rangle 5))$$

$$e_2 \equiv \texttt{let } id{:}\forall\beta.\beta\to\beta = \Lambda\beta.\lambda y{:}\beta.\, y \texttt{ in}$$
$$(\lambda x{:}\texttt{[int]}.\langle\texttt{unbox}\rangle x + \langle\texttt{unbox}\rangle(id\{\texttt{[int]}\}\ x))$$
$$(\texttt{if ... then } \langle\texttt{box}\rangle 2 \texttt{ else } (id\{\texttt{[int]}\}\ \langle\texttt{box}\rangle 5))$$

Neither one of them is reducible to the other by $\phi\psi$-reduction modulo E. The main difference between e_1 and e_2 is the representation type of x. In e_1 it is unboxed whereas in e_2 it is boxed. By introducing a box;unbox-pair in front of the constant 2 in e_1 we can ψ-reduce (modulo E) e_1 to e_2. Conversely, by introducing an unbox;box-pair in front of $(id\{\texttt{[int]}\}\ \langle\texttt{box}\rangle 5)$ in e_2 we can ϕ-reduce (modulo E) e_2 to e_1. Thus we can trade off a box;unbox-redex for an unbox;box-redex.

By prioritizing elimination of one kind of redex over the other we end up with a formal notion of optimality that entails that, for any given representation type, every source Core-XML expression has an optimal completion that is unique modulo E_p-equality.

5.3 ψ-free and ϕ-free completions

In the example above we saw that by introducing a redex of one kind (say ϕ) we could eliminate a redex of the other kind (ψ). This is an improvement if redexes of the second kind are considered arbitrarily more expensive than redexes of the first kind. But it is not obvious which of the two kinds of redexes should be considered more expensive. Thus we shall pursue two different notions of optimality. In the first we get rid of all ψ-redexes first — even at the cost of introducing additional ϕ-redexes — and then getting rid of all ϕ-redexes without letting ψ-redexes slip back in. In the second we, dually, get rid of all ϕ-redexes first, possibly introducing new ψ-redexes, and then eliminate all ψ-redexes without readmitting ϕ-redexes.

DEFINITION 19 (ψ-free completions, ϕ-free completions)

1. We say a completion e is *ψ-free* if every congruent completion $e' \cong e$ ψ-reduces to e under $E\phi$-equality; i.e, $E\phi \vdash e' \Longrightarrow_\psi^* e$.
2. We say a completion e is *ϕ-free* if every congruent completion $e' \cong e$ ϕ-reduces to e under $E\psi$-equality; i.e., $E\psi \vdash e' \Longrightarrow_\phi^* e$.

□

Because of the strong global requirement that *all* congruent completions must be ψ-reducible modulo $E\phi$ to c for c to be called ψ-free it is not even clear that ψ-free completions (or ϕ-free completions) exist. This can be shown, however, by orienting the E_p-equations from right to left, treating them as rewriting steps modulo ϕ-equality, and combining them with $\psi\phi$-reduction on coercions. We shall refer to the resulting system somewhat loosely as $\psi^{\to}E^{\leftarrow}$-rewriting modulo ϕ, even though ψ-rewriting is *not* modulo ϕ.

LEMMA 20 (Properties of $\psi^{\to}E_p^{\leftarrow}$-reduction modulo ϕ)
$\psi^{\to}E_p^{\leftarrow}$-reduction modulo ϕ is canonical; that is, it is strongly normalizing and confluent.

PROOF: **Strong normalization:** Without loss of generality we may assume that every completion has exactly one coercion applied to each subexpression in the underlying Core-XML expression, since a consecutive coercion application $\langle c\rangle\langle c'\rangle e$ is equal to $\langle c'; c\rangle e$ and a subexpression e without a coercion is equal to $\langle\iota\rangle e$.

Let $c_1, \ldots, c_k$ be the vector of all coercion occurrences in a completion in some particular order such that they are in one-to-one correspondence with the subexpressions of the underlying Core-XML expression. Since completion rewriting does not change the underlying Core-XML expression, a completion rewriting step corresponds to a rewriting step on this vector.

A $\psi^{\to}$-reduction step operates on a single element of the coercion vector above. By Theorem 16 $\phi\psi$-reducing a coercion is strongly normalizing. Thus there can be only finitely many ψ-reduction steps at the beginning of the reduction or after an $E_p^{\leftarrow}$ step is executed.

An $E_p^{\leftarrow}$ step generally operates on several coercions in the coercion vector simultaneously. Consider the type signatures of the coercions in the coercion vector. An $E_p^{\leftarrow}$ step rewrites at least one coercion $\vdash \mathrm{c} : \rho \rightsquigarrow \rho'$ to a new coercion c' that has domain type or range type properly increased in the subtype hierarchy of Definition 10. Since the subtype hierarchy has only finite ascending chains (Proposition 11) it follows that $E_p^{\leftarrow}$ steps can only be applied a finite number of times. Thus every $\psi^{\rightarrow}E_p^{\leftarrow}$-reduction sequence is finite.

Confluence: Since $\psi^{\rightarrow}E_p^{\leftarrow}$-rewriting modulo ϕ is strongly normalizing it is, by Newton's Lemma, sufficient to show local confluence; that is, if e has overlapping redexes and reduces by single rewriting steps to e_1 and e_2 then there exists a common reduct e' to which both e_1 and e_2 reduce, possibly in several steps.

Let us consider such triples e, e_1, e_2. By Theorem 16 $\psi\phi$-reduction on coercions is confluent. Note that ψ-redexes do not overlap with any $E_p^{\leftarrow}$-redex due to the polarization and orientation of the E_p rules. We only have to worry about overlaps of $E_p^{\leftarrow}$-rules modulo ϕ-equality. There are only two kinds of overlaps:

1. Application of the same rule to the same subexpression, only with different coercions; e.g.,

   $$\langle (c_1^- \rightarrow c_2^+); \bar{c}\rangle \lambda x.e \Longrightarrow \langle \bar{c}\rangle \lambda x.\langle c_2^+\rangle([x \mapsto \langle c_1^-\rangle x]e)$$

   and

   $$\langle (d_1^- \rightarrow d_2^+); \bar{d}\rangle \lambda x.e \Longrightarrow \langle \bar{d}\rangle \lambda x.\langle d_2^+\rangle([x \mapsto \langle d_1^-\rangle x]e)$$

   where

   $$\phi \vdash (c_1^- \rightarrow c_2^+); \bar{c} = (d_1^- \rightarrow d_2^+); \bar{d}.$$

   In this case it is sufficient to show that

   $$\phi \vdash \bar{c} \Rightarrow_\psi c_3^+; c_4^-$$
   $$\phi \vdash \bar{d} \Rightarrow_\psi d_3^+; d_4^-$$

   for some positive c_3^+, d_3^+ and negative ${c_4}^-, {d_4}^-$, where c_3^+ and d_3^+ have the same range type ρ, since then we can apply the same rule again to each of the two different reducts to get a common reduct. By choosing the maximal representation type greater than domain types of $\bar{c}$ and $\bar{d}$ for ρ this is easily accomplished.

2. Overlaps due to three pairs of adjacent rules in Figure 11: $12^-/12^+$, $13^-/13^+$, and $14^-/14^+$. Let us consider $12^-/12^+$:

   $$\langle c_1^- \rightarrow d_1^+\rangle \lambda x.\langle d_2^-\rangle([x \mapsto \langle c_2^+\rangle x]e)$$

   can be rewritten to

   $$\langle c_1^- \rightarrow d_1^+\rangle\langle c_2^+ \rightarrow d_2^-\rangle \lambda x.e$$

   and to

   $$\lambda x.\langle d_1^+\rangle\langle d_2^-\rangle([x \mapsto \langle c_2^+\rangle\langle c_1^-\rangle x]e).$$

   For the first reduct we get furthermore

   $$\begin{array}{rll}
   \langle c_1^- \rightarrow d_1^+\rangle\langle c_2^+ \rightarrow d_2^-\rangle \lambda x.e & = & \\
   \langle (c_2^+ \rightarrow d_2^-); (c_1^- \rightarrow d_1^+)\rangle \lambda x.e & = & \\
   \langle (c_1^-; c_2^+) \rightarrow (d_2^-; d_1^+)\rangle \lambda x.e & \Rightarrow_\psi^* & (+/-\text{-fact.}) \\
   \langle (c_3^+; c_4^-) \rightarrow (d_3^+; d_4^-)\rangle \lambda x.e & = & \\
   \langle (c_4^- \rightarrow d_3^+); (c_3^+ \rightarrow d_4^-)\rangle \lambda x.e & = & \\
   \langle c_3^+ \rightarrow d_4^-\rangle\langle c_4^- \rightarrow d_3^+\rangle \lambda x.e & \Rightarrow_{E_p^{\leftarrow}} & \\
   \langle c_3^+ \rightarrow d_4^-\rangle \lambda x.\langle d_3^+\rangle([x \mapsto \langle c_4^-\rangle x]e) & &
   \end{array}$$

   Similarly, we can rewrite the second reduct to the same final completion above.

   $$\begin{array}{rll}
   \lambda x.\langle d_1^+\rangle\langle d_2^-\rangle([x \mapsto \langle c_2^+\rangle\langle c_1^-\rangle x]e) & = & \\
   \lambda x.\langle d_2^-; d_1^+\rangle([x \mapsto \langle c_1^-; c_2^+\rangle x]e) & \Rightarrow_\psi^* & (+/-\text{-fact.}) \\
   \lambda x.\langle d_3^+; d_4^-\rangle([x \mapsto \langle c_3^+; c_4^-\rangle x]e) & = & \\
   \lambda x.\langle d_4^-\rangle\langle d_3^+\rangle([x \mapsto \langle c_4^-\rangle\langle c_3^+\rangle x]e) & \Rightarrow_{E_p^{\leftarrow}} & \\
   \langle c_3^+ \rightarrow d_4^-\rangle \lambda x.\langle d_3^+\rangle([x \mapsto \langle c_4^-\rangle x]e) & &
   \end{array}$$

   In these reductions we used the fact that every coercion can be $\psi\phi$-reduced to a $+/-$-factored coercion by Corollary 18.

   The other two pairs of rules with critical pairs are handled completely analogously.

This completes the proof. □

Having a canonical reduction system for $E_p\phi\psi$-equality and thus for congruence it follows that every congruence class of completions contains both ψ-free and ϕ-free completions.

THEOREM 21 (Existence and uniqueness of ψ-, resp. ϕ-free completions)
Let e be a (closed) Core-XML expression and let ρ be a valid representation type for e.

1. Then e has a ψ-free completion e' at ρ; e' is furthermore uniquely determined up to $E_p\phi$-equality.
2. e has a ϕ-free completion e'' at ρ; e'' is uniquely determined up to $E_p\psi$-equality.

PROOF: We only give a proof of 1. The proof of 2 is similar. (It requires a lemma analogous to Lemma 20.)

Consider all the (congruent) completions of e at ρ. By Theorems 8 and 14 we know that they are all $E_p\phi\psi$-equal. By Lemma 20 $\psi^{\rightarrow}E_p^{\leftarrow}$-rewriting modulo ϕ is canonical. It reduces any two congruent — and thus $E_p\phi\psi$-equal — completions to a normal form e_{nf} that is unique up to ϕ-equality. Thus e_{nf} is a ψ-free completion of e. It can be shown that any ψ-free completion of e at ρ must be $E_p\phi$-equal to e_{nf}. □

Intuitively, a ψ-free completion "prefers" to keep data in a boxed representation and unboxes a representation only when it is sure the unboxed value is required by some operation. This way passing arguments to polymorphic functions and returning their results can be expected to be efficient whereas operations requiring unboxed data such as integer operations may be inefficient due to the cost of unboxing arguments and boxing the results.

Dual to this, a ϕ-free completion prefers to keep data in unboxed representation; it boxes a value only when it is sure to be required due to a call to a polymorphic function. Thus primitive operations will generally be executed fast as no coercions need to be performed for neither the arguments nor the result, but calls to polymorphic functions may be expensive due to the need for boxing (parts of) the arguments and unboxing (parts of) the result.

Since the degree of polymorphism in a program tends to be greatest when higher-order functions are used a ψ-free completion will generally be better for higher-order programs, especially if there is little "ground type" processing

such as arithmetic operations. On the other hand, ϕ-free completions will generally do best where there is little polymorphism and/or lots of operations on ground types.

The rules for $\psi^{\rightarrow}E_p^{\leftarrow}$-rewriting modulo ϕ suggest an explicit construction of ψ-free completions: take an arbitrary completion and execute the ψ-reduction system until a normal form is reached . Analogously for ϕ-free completions. An even simpler method consists of devising syntax-directed translations that produce a ψ-free or ϕ-free completion directly.

The canonical construction of a ψ-free completion consists of keeping all data in their "maximally" boxed representation (i.e., representing a standard type by the maximal type in the representation type hierarchy) and boxing a unboxed value as soon as it is produced by some operation and unboxing it just before it is used by some operation.

The canonical construction of a ϕ-free completion consists of keeping all data in their "minimally" boxed representation where an unboxed value is only boxed just before it is passed to a polymorphic function and the result of a polymorphic function is immediately unboxed. This is actually the construction described by Leroy [Ler92].

5.4 Optimal ψ-free/ϕ-free completions

The two constructions above for a ψ-free and a ϕ-free completion are canonical since they use a universal standard representation (maximally boxed or minimally boxed) for all data independent of their context. They are not "optimal" since they typically contain many ϕ-, respectively ψ-redexes modulo E_p-equality. We shall now set out to construct *optimal* ψ-free and ϕ-free completions, which have no remaining redexes — and are thus $\psi\phi$-normal forms modulo E.

DEFINITION 22 (Optimal ψ-free/ϕ-free completions)

1. A completion e is a *(formally) optimal ψ-free completion* if e is ψ-free and every congruent ψ-free completion e' ϕ-reduces to e modulo E; i.e., $E \vdash e' \Longrightarrow_{\phi}^{*} e$.
2. A completion e is a *(formally) optimal ϕ-free completion* if e is ϕ-free and every congruent ϕ-free completion e' ψ-reduces to e modulo E; i.e., $E \vdash e' \Longrightarrow_{\psi}^{*} e$.

□

We have shown that ψ-free completions are $E_p\phi$-equal, and ϕ-free completions are $E_p\psi$-equal. There are canonical rewriting systems for ϕ-reduction and ψ-reduction modulo E_p. As a result we obtain our main theorem:

THEOREM 23 (Existence of ψ-free and ϕ-free optimal completions)

Let e be a (closed) Core-XML expression and let ρ be a valid representation type for e. Then e has both an optimal ψ-free completion and an optimal ϕ-free completion at ρ. Furthermore, both are unique modulo E_p. □

For the proof we employ again a rewriting system. This time we use $\phi^{\rightarrow}E_p^{\rightarrow}$-rewriting. This rewriting system operates on equality classes defined by the coercion equations in Figure 6 and the application equations in Figure 8. Note that the E_p-equations are oriented from left to right, opposite to the orientation we had chosen for $\psi^{\rightarrow}E_p^{\leftarrow}$-reduction modulo ϕ.

The proof of the theorem is omitted. It is analogous to the proofs of Lemma 20 and Theorem 21.

6 Implementation

We have written a prototype implementation in Haskell of our rewriting systems. The implementation handles the Core-XML language extended with a conditional, pairs, a fixed-point operator and arbitrary polymorphic constants. Polymorphic constants enable us to introduce lists by just adding a list type to the implementation.

The implementation is parameterized in such a way that one can specify from what canonical completion reduction is going to start, and what reduction system is to be used: ($\phi \vdash c' \Longrightarrow_{\psi\rightarrow E_p^{\leftarrow}}^{*} c$, $\vdash c' \Longrightarrow_{\psi\rightarrow E^{\leftarrow}}^{*} c$, $\psi \vdash c' \Longrightarrow_{\phi\rightarrow E_p^{\rightarrow}}^{*} c$, or $\vdash c' \Longrightarrow_{\phi\rightarrow E_p^{\rightarrow}}^{*} c$). (Recall that reduction to an optimal completion involves two phases.)

Running the system will produce a normal form completion in the form of an SML-program in which box and unbox operations behave like the identity function, but also perform profiling operations. That is, besides returning its input the box and unbox coercions count how many times they are called. The final result of running such an SML program is the actual result together with a count of the box and unbox operations executed.

6.1 Adding new type constructors

Adding new type constructors like pairs, list, etc., is quite simple. If we add a new type constructor (e.g., for list types ρ *list*) we also have to add a new constructor (e.g., `map` c) on coercions, and we have to extend Figure 6 with some new equations for this new coercion constructor. For list these rules are

$$\texttt{map}\ \iota_\rho = \iota_{\rho\ list}$$
$$\texttt{map}\ (\mathrm{c};\mathrm{d}) = (\texttt{map}\ \mathrm{c});(\texttt{map}\ \mathrm{d})$$

In terms of category theory type constructors can be seen as functors, and the rules above are simply the two conditions that must hold for a functor.

6.2 Handling of primitives

Extending our work to handle language primitives and polymorphic constants is straightforward and elegant. We will show how one can derive very natural rules for conditionals directly from rules 15^- and 15^+ of Figure 11. The type of the conditional is `if _ then _ else _`: $\forall\alpha.(\mathrm{bool},\alpha,\alpha)\rightarrow\alpha$. Treating the conditional as a free variable we see that equations 15^- and 15^+ provide the necessary and sufficient coherence conditions. Since language primitives are implemented "inline" we can dispense with the requirement that they be applied only to boxed types. Instantiating equation 15^- to `if _ then _ else _` yields

$$\langle \iota_{bool*\rho'*\rho'} \rightarrow \mathrm{c}^- \rangle(\texttt{if \_ then \_ else \_})\{\rho'\} = \langle(\iota_{bool},\mathrm{c}^-,\mathrm{c}^-)\rightarrow\iota_\rho\rangle(\texttt{if \_ then \_ else \_})\{\rho\}$$

for any negative coercion $c^- : \rho' \rightarrow \rho$. Applying both sides of the equation to argument (e_1, e_2, e_3) we obtain the natural equation

$$\langle \mathrm{c}^- \rangle \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 = \texttt{if}\ e_1\ \texttt{then}\ \langle \mathrm{c}^- \rangle e_2\ \texttt{else}\ \langle \mathrm{c}^- \rangle e_3$$

In this way one can develop specific rules for program constructs like conditional, fix-point operator, pairing, and primitive operations. Polymorphic constants can be handled directly by using rules 15^- and 15^+.

We give one more example to show the connection of equations (15), (15^-), and (15^+) to what Wadler has termed "free theorems" (see Reynolds [Rey83] and Wadler [Wad89]). Assume that we have a function r such as reverse with type $\forall\alpha.\alpha\ list{\to}\alpha\ list$. If we instantiate rule 15^- to this we get

$$\langle\iota{\to}\text{map c}^-\rangle\text{r}\{\rho'\} = \langle\text{map c}^-{\to}\iota\rangle\text{r}\{\rho\}$$

but this is essentially, if we disregard the instantiation, the same as

$$\text{map c}^- \circ \text{r} = \text{r} \circ \text{map c}^-$$

which is a well-known free theorem.

6.3 Performance results

Figure 13 gives results of some experiments performed with our implementation. The six programs that were selected for the experiments were: *insert-sort*, a insert sorting program where the insert function is polymorphic; *flip-list*, which "flips" the elements of a list of pairs of integers; *leroy*, which is an almost "pathological" program for which Leroy's benchmark shows a major slow-down compared to a fully boxed implementation; *poulsen*, a program for which Poulsen [Pou93] reports that his algorithm gives very poor results (300/300 box/unbox-operations); *sieve*, which computes the prime numbers between 1 and 100; and *poly*, a constructed example program with a lot of polymorphism. The example programs *insert-sort*, *flip-list*, *leroy*, *poulsen* are all taken from [Pou93].

For all six programs we have generated three completions, the optimal ψ-free normal form, the optimal ϕ-free normal form and Leroy's completion. We have run the three completions and counted the number of box and unbox operations performed. Figure 13 shows the results.

The results indicate that the optimal ψ-free normal form completions found by our system are generally better and often much better than Leroy's completion, especially when a lot of polymorphism is involved, like in *poly*. In one case, *sieve*, Leroy's completion performs fewer unbox-operations than the optimal ψ-free normal form we produce. The reason for this is that the optimal ψ-free completion we produce places unbox operations as "late" as possible thus possibly duplicating unboxing operations.

Program	Completion	box	unbox
insert-sort	opt. ψ-free norm.	17	171
	opt. ϕ-free norm.	289	307
	Leroy	156	307
flip-list	opt. ψ-free norm.	20	20
	opt. ϕ-free norm.	30	35
	Leroy	20	20
leroy	opt. ψ-free norm.	709	709
	opt. ϕ-free norm.	446	446
	Leroy	1219	1219
poulsen	opt. ψ-free norm.	3	3
	opt. ϕ-free norm.	3	3
	Leroy	3	3
sieve	opt. ψ-free norm.	99	847
	opt. ϕ-free norm.	436	748
	Leroy	411	748
poly	opt. ψ-free norm.	100	150
	opt. ϕ-free norm.	100	150
	Leroy	620	670

Figure 13: Performance: benchmarks

7 Related work

7.1 Boxing

The substantial cost of manipulating data in boxed representation, especially for numeric programs, has been observed in both dynamically typed high-level languages like LISP and statically typed polymorphic languages such as Standard ML, and Haskell.

Most of the efforts in LISP implementation have focused on optimizing number representations by keeping them in unboxed form [Ste77, BGS82, KKR$^+$86]. Peterson [Pet89] uses an elaborate execution-frequency based criterion for the cost of representation coercions. In this setting he shows how the optimal placement of coercions can be reduced to a well-known network flow problem. Common to all these efforts is the intent to optimize representations of atomic data, particularly numbers. Indeed in Peterson's framework operations on pairs and lists simply require boxed arguments. Steenkiste and Hennessy, however, have found that in a suite of ten LISP programs up to 80% of the representation coercions are list tagging/untagging operations.

Peyton Jones and Launchbury [PJL91] and Leroy [Ler92] suggested making representation types and boxing and unboxing operations explicit in programs. Even though there are some technical differences, the languages they use are at the core the same: Core-ML with explicit boxing/unboxing coercions.

Peyton Jones and Launchbury do not provide a method of inferring a completion, but concentrate on the semantics of their explicitly boxed language and on optimization of boxing by program transformation. Those optimizations are, for example, a form of common subexpression elimination that cannot be formalized in our framework as the transformations may change the underlying program.

Leroy describes a translation of Core-ML expressions to explicitly boxed Core-ML expressions. This translation is not deterministic as it depends on the specific typing derivation of the underlying Core-ML expression, but every translation of such a source Core-ML expression is a completion in our sense (not the other way round, however). The experimental results of incorporating his boxing analysis in the Gallium compiler for CAML Light show that the resulting performance can be drastically different from the original compiler that uses canonically boxed representations. The results, though, are not uniformly better. The canonically boxed completions are ψ-free whereas Leroy's is ϕ-free in our terminology. The results are in line with our general considerations that indicate that monomorphic programs should generally fare better with a ϕ-free completion whereas highly polymorphic programs are likely to be better off with a ψ-free completion. Our rewriting system for ϕ-free completions will improve the result of Leroy's completion by eliminating all ψ-redexes, and our rewriting system for ψ-free completions will improve the canonically boxed completion by eliminating all ϕ-redexes.

Using Leroy's framework Poulsen [Pou93] presents a more involved translation, but draws on constraint solving to eliminate more boxing/unboxing operations than Leroy's translation in many, but not all cases. The interesting aspect of Poulsen's completions is that they, just like our optimal completions, are not required to have a canonically defined representation type for the types occurring in type applications as in Leroy's work, but determines an appropriate representation type as part of the constraint solving process. On the other hand it appears that some boxing/unboxing operations are built into the constructors of the language and are not accounted for in the question of optimizing the boxing in the program.

Given a Core-ML expression with type τ the result of a boxing analysis depends on the particular typing derivation chosen. Leroy's completion uses implicitly the derivation obtained by Algorithm W [Mil78] since his translation performs type inference and boxing simultaneously where let-bound variables receive the principal type of the bound expression. (Peyton-Jones/Launchbury and Poulsen also appear to assume that type inference is performed in the style of Algorithm W.) The principal type of a function is the "most" polymorphic type and thus imposes the most boxing demands on the arguments to the function. A "more" monomorphic derivation, on the other hand, could still yield the same type for the whole expression, but using more monomorphic types for the local variables. Bjørner gives an algorithm called M for finding a *minimally* polymorphic typing derivation [Bjø92]. Minimally polymorphic derivations do not always exist, but his algorithm generally lowers the local degree of polymorphism in comparison to Algorithm W. Note that our boxing analysis does not presuppose a specific typing derivation for a Core-ML expression, but interfaces with *any* of its type derivations, which is represented by an explicitly typed Core-XML expression.

7.2 Coherence and equivalence

The notion of coherence appeared first in computer science literature in the work of Breazu-Tannen, Coquand, Gunter, Scedrov [BTCGS91, BTGS90] and Curien, Ghelli [CG90, Ghe90]. They use it to give interpretations of subtype based systems, where application of the subtyping rule is interpreted by an (explicit) coercion. Since a given program with subtyping may have many different translations it is integral to prove that all of them are coherent for the semantics (via arbitrary translation and interpretation of the target program) to be well-defined.

Thatte describes a method for inferring very powerful implicit coercions between isomorphic types in a type inference system enriched with coercions between arbitrary isomorphic types. Our application can be viewed as simple variant of this problem as arbitrary representation types of the same standard type — and only those — can be coerced to each other. On the other hand Thatte does not deal with optimizing the coercions required in this fashion.

The notion of completion and its congruence theory is inspired and closely related to the work reported in [Hen93], which explicates type tagging and untagging operations in dynamically typed languages. The purpose of doing so is completely analogous to boxing analysis: to eliminate most statically type tagging and untagging operations and to implement only the remaining ones while still obtaining "safe" program execution; i.e., well-defined program behavior. See also the work by Cartwright, Fagan and Wright on soft typing [CF91, WF92].

8 Conclusion and further work

We have presented a calculus, formal optimality criteria and rewriting-based algorithms for finding good representations of data as boxed or unboxed data in a polymorphically typed programming language. The word "good" here is to be understand in a very general and broad sense. What has been left out is a detailed analysis of specific language properties and implementation considerations that have an effect on the actual performance. This has been done to make the results "universal" and applicable in different settings, even different semantics of the same language.

Judging by experimentation with some short Standard ML programs our "formally" optimal completions also tend to be consistently better in practice than previously described boxing analyses if we count only the number of boxing/unboxing operations executed. Since no implementation decisions are made at the time the boxing analysis is conducted its output should combine well and without much interference with later implementation phases.

The general framework of treating boxing analysis as a translation of a program to a language with explicit boxing and unboxing operations, due to Leroy [Ler92] and implicit also in Peyton Jones and Launchbury [PJL91] encapsulates boxing analysis as a single phase. The representation type of an explicitly boxed program specifies its interface and thus allows separate compilation of program modules.

There are several problems with making full use of boxing-optimized programs:

1. Garbage collection often requires "tagging" of heap-allocated data with explicit type and size information. Thus an unboxed representation may well have to be tagged (= boxed) anyway to accommodate the garbage collector.

2. A boxed representation is the result of an evaluation. In lazy languages often boxed representations are required since the evaluation of an expression is not statically known to terminate or to be advantageous. Thus an expression determined to be best kept in unboxed form by our boxing analysis may still have to be boxed at run-time.

For the future we plan to devise efficient algorithms for computing optimal boxing completions, which also take account of control dependencies and carefully place coercions at points where they get executed with minimum run-time frequency. We expect to use analyses such as Peterson's [Pet89] for this purpose.

Finally, we intend to integrate our boxing analysis after region inference has been performed into the region-based implementation of Standard ML currently underway at DIKU (see [TT94]). The use of region-based memory management also obviates the need for global garbage collection and thus the first of the two restrictions above.

Acknowledgements

We would like to thank Neil Jones for first pointing out the applicability of the framework of dynamic typing to boxing analysis. Special thanks go to Eigil Rosager Poulsen

from whose thesis most of our test examples were taken. We would also like to thank the following people with whom we have had interesting and fruitful discussions on formally optimal boxing: Anders Bondorf, Christian Mossin, Robert Glück, David Sands and Mads Tofte.

References

[BGS82] R. Brooks, R. Gabriel, and G. Steele. An optimizing compiler for lexically scoped LISP. In *Proc. SIGPLAN '82 Symp. on Compiler Construction, Boston, Massachusetts*, pages 261–275, June 1982. SIGPLAN Notices, Vol. 17, No. 6.

[Bjø92] Nikolaj Bjørner. Minimal typing derivations. DIKU Student Report, July 1992.

[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.

[BTGS90] V. Breazu-Tannen, C. Gunter, and A. Scedrov. Computing with coercions. In M. Wand, editor, *Proc. ACM Symp. on Lisp and Functional Programming (LFP), Nice, France*, pages 44–60, 1990.

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

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

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, pages 207–212, Jan. 1982.

[Ghe90] G. Ghelli. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism.* PhD thesis, Universita di Pisa, Dipartimento di Informatica, March 1990.

[Hen93] Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 1993. Special Issue on European Symposium on Programming 1992 (to appear).

[HM93] Robert Harper and John Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):211–252, April 1993. Based on paper presented at POPL '88.

[KKR+86] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. In *Proc. SIGPLAN '86 Symp. on Compiler Construction*, pages 219–233, 1986.

[Ler92] X. Leroy. Unboxed objects and polymorphic typing. In *Proc. 19th Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programmin gLanguages (POPL), Alb uquerque, New Mexico*, pages 177–188. ACM Press, Jan. 1992.

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

[Pet89] J. Peterson. Untagged data in tagged environments: Choosing optimal representations at compile time. In *Proc. Functional Programming Languages and Computer Architecture (FPCA), London, England*, pages 89–99. ACM Press, Sept. 1989.

[PJL91] Simon Peyton Jones and John Launchbury. Unboxed values as first class citizens. In *Proc. Conf. on Functional Programming Languages and Computer Architecture (FPCA), Cambridge, Massachusetts*, pages 636–666. Springer, Aug. 1991. Lecture Notes in Computer Science, Vol. 523.

[Pou93] Eigil Poulsen. Representation analysis for efficient implementation of polymorphism. Master's thesis, DIKU, University of Copenhagen, 1993.

[Rey83] J. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing*, pages 513–523, 1983.

[Ste77] G. Steele. Fast arithmetic in MacLisp. In *Proc. 1977 MACSYMA Users' Conference, NASA Scientific and Technical Information Office, Washington, D.C.*, July 1977.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proc. 21st Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Portland, Oregon (this proceedings).* ACM, ACM Press, Jan. 1994.

[Wad89] P. Wadler. Theorems for free! In *Proc. Functional Programming Languages and Computer Architecture (FPCA), London, England*, pages 347–359. ACM Press, Sept. 1989.

[WF92] A. Wright and M. Fagan. Soft typing and global representation optimization. Manuscript, July 1992.