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

July 29, 1993

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. We give the equality axioms a rewriting interpretation that captures eliminating boxing/unboxing operations without relying on a specific implementation or even semantics of the underlying language. The rewriting system is shown to be canonical (Church-Rosser and strong normalizing) in the sense that it produces a unique "optimal" completion for a given source program within classes of completions. The rewriting system has been implemented and applied to simple Standard ML programs, and first results show that the amount of boxing and unboxing operations is also in practice substantially reduced in comparison to Leroy's completions.

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 the actual arguments have a *uniform representation*, 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

*Submitted to POPL '94

[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.

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 as it is guaranteed that it never actually needs to 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 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; e.g., the explicification of boxing in the Glasgow Haskell Compiler based on Peyton Jones and Launchbury [PJL91]. The rationale for making boxing explicit is that some boxing/unboxing operations can be eliminated in the later transformational stages of such a 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 underly the very purpose of boxing analysis and, more generally, similar static analyses: Given two completions for the same program, which of them is better? Is there an "optimal" completion for it? What does it mean for one completion to be "better" than another completion? Does every program have an "optimal" completion? If so, can it 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 underlying 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 our desired notion of optimality: 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).

We show that this rewriting system is Church-Rosser and strongly normalizing; this guarantees, in particular, that every program has an optimal completion. This is accomplished by orienting the remaining equations as left-to-right or right-to-left rewriting rules depending on the *polarity* of the coercions involved. (Any simple-minded orientation of the equations leads to nonconfluence and nontermination of Knuth-Bendix completion.) This rewriting system can furthermore be used to compute an optimal completion in which, intuitively, any remaining coercions are executed as late as possible.

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. Specifically, they do not

2. They can be combined with other optimizations unrelated to boxing as the calculus does not take any properties of the underlying language or its implementation technology into account.

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; thus the notion of optimality is not overcommitted by insisting on uniqueness.

1.3 New results

Specifically, our contributions in this paper are as follows:

- A general framework and robust criterion for the quality of boxing completions, which accounts for the costs of boxing/unboxing operations, but is otherwise completely independent of other language properties and implementation decisions.

- Proof of the existence of *optimal (boxing) completions* and their uniqueness modulo an equational theory that assumes nothing about the properties of boxing and unboxing operations or the underlying programming language.

- A rewriting-based algorithm for computing optimal completions, which are shown to be uniformly better than those described in [PJL91, Ler92, Pou93].
- An experimental implementation of the algorithm and test results that support empirically that our completions are better 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 criterion. But this would have been unsatisfactory in several respects:

1. 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 insufficient for this purpose in the long run as they can only report on system performance where the interaction of boxing with other system properties 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.

2. 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 coercion calculus.

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 **??**, to develop our theory. In the full length paper 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 type constructors.

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

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

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 A if $A \vdash e : \sigma$ 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.

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

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

Figure 2: Typing rules for Core-XML

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., `let` x = $\Lambda\alpha_1.\Lambda\alpha_1....\Lambda\alpha_n.e_1$ `in` e_2 We shall abbreviate this to `let` x = $\Lambda\alpha_1...\alpha_n.e_1$ `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 Λs), also 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. Interestingly, the derivations produced by Algorithm W [Mil78] style algorithms, which produce a principal type for every subexpression, are in a way the worst possible for boxing analysis as they maximize the polymorphic degree and thus impose requirements for boxed representation. Boxing can thus be optimized by producing a typing derivation with "minimal" polymorphic degree and then applying the algorithm presented in this paper. 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 are just the standard types of Core-XML, however with one additional type constructor, [...]. Types of the form $[\tau]$ 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*

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

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

Figure 3: Representation types for explicitly boxed Core-XML

$$\vdash \iota_\tau : \tau \leadsto \tau \qquad \frac{\vdash \mathrm{c} : \tau_1 \leadsto {\tau_1}' \quad \vdash \mathrm{c}' : \tau_2 \leadsto {\tau_2}'}{\vdash \mathrm{c} {\to} \mathrm{c}' : {\tau_1}' \to \tau_2 \leadsto \tau_1 \to {\tau_2}'} \qquad \frac{\vdash \mathrm{c} : \tau \leadsto \tau' \quad \vdash \mathrm{c}' : \tau' \leadsto \tau''}{\vdash \mathrm{c};\mathrm{c}' : \tau \leadsto \tau''}$$

$$\vdash \mathtt{box}_\upsilon : \upsilon \leadsto [\upsilon] \qquad \vdash \mathtt{unbox}_\upsilon : [\upsilon] \leadsto \upsilon$$

Figure 4: Coercion formation rules

types; their elements are not boxed. Since doubly boxed representations are useless we prohibit boxed types of the form $[[\tau]]$. 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.

DEFINITION 1 The *(underlying) standard type* $|\rho|$ of representation type ρ is the type arrived at by erasing all occurrences of $[\ldots]$ in ρ and treating both boxed and unboxed type variables as standard type variables. We say that ρ represents $|\rho|$. $\square$

$$\frac{A\{x : \tau_1\} \vdash e : \tau_2}{A \vdash \lambda x{:}\tau_1 . e : \tau_1 \to \tau_2} \qquad \frac{A \vdash e_1 : \tau_1 \to \tau_2 \quad A \vdash e_2 : \tau_1}{A \vdash e_1\ e_2 : \tau_2}$$

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

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

$$\frac{A \vdash e : \tau \quad \vdash \mathrm{c} : \tau \leadsto \tau'}{A \vdash \mathtt{[c]}e : \tau'}$$

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

3.2 Coercions

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

$$\begin{aligned}\texttt{box}_\upsilon &: \upsilon \rightsquigarrow [\upsilon]\\ \texttt{unbox}_\upsilon &: [\upsilon] \rightsquigarrow \upsilon\end{aligned}$$

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 (at every type), composition of coercions c, c' written in diagrammatical order $\mathrm{c};\mathrm{c}'$ and coercions *induced* by the type constructors. In our case these are coercions of the form $\mathrm{c}\to\mathrm{d}$, which apply to functions f by "wrapping" them with the input coercion c and the output coercion d. The result is an unboxed function where c is applied to the input before it is passed to f and dis applied to the result of f before it is returned as the output. The rules for forming coercions are displayed in Figure 4.

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

PROPOSITION 2 et ρ_1, ρ_2 be arbitrary representation types. Then

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

□

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 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* $|e|$ of an explicitly typed Core-XML expression e is the (standard) Core-XML expression arising from e by erasing all occurrences of coercions (including square 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 congruence completions

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 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) Explicitly boxed expressions e' and e'' are *completion congruent*, written $e' \cong e''$, if they have the same type and are completions of the same standard expression. □

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 treats the primitive coercions as uninterpreted operations, and a small group of two axioms that express that unboxing after boxing and the other way round is equal to the identity coercion. In Section refsec-reduction 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.

$$
\begin{array}{rcll}
(\mathrm{c};\mathrm{c}');\mathrm{c}'' & = & \mathrm{c};(\mathrm{c}';\mathrm{c}'') & (1) \\
\mathrm{c};\iota & = & \mathrm{c} & (2) \\
\iota;\mathrm{c} & = & \mathrm{c} & (3) \\
(\mathrm{c}_1 \to \mathrm{c}_2);(\mathrm{c}_1{}' \to \mathrm{c}_2{}') & = & (\mathrm{c}_1{}';\mathrm{c}_1) \to (\mathrm{c}_2;\mathrm{c}_2{}') & (4) \\
\iota \to \iota & = & \iota & (5) \\
\mathtt{box}_\upsilon;\mathtt{unbox}_\upsilon & = & \iota & (\mathrm{phi}) \\
\mathtt{unbox}_\upsilon;\mathtt{box}_\upsilon & = & \iota & (\mathrm{psi})
\end{array}
$$

Figure 6: Equality rules for coercions

4.1 Coercion coherence

Consider the equality axioms in Figure 6 for coercions. We assume that the coercions on both sides of an equality are well-formed and have the same type signature and write $c = c'$ if $\vdash c = c'$ is derivable from the axioms together with reflexivity, symmetry, transitivity and compatibility of $=$ with arbitrary contexts.

LEMMA 5 Uniqueness of coercions Let c, c' be coercions with the same type signature. Then $c = c'$. □

The lemma expresses that there is at most one coercion between any two types, which is necessary if we wish to prove the congruent completions $[c]x, [c']x$ equal.

4.2 Expression coherence

Let us extend the equality axioms for coercions with the equality axioms for explicitly boxed Core-XML expressions in Figure 7. 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. We write $e = e'$ if $e = e'$ is derivable from the axioms in Figures 6 and 7 together with rules for reflexivity, symmetry, transitivity and compatibility of $=$.

THEOREM 6 (Coherence of completions) Explicitly boxed expressions e' and e'' are completion congruent if and only if they are equal, i.e:

$e' \cong e''$ **iff** $e' = e''$

$$
\begin{array}{rcll}
[\iota]e & = & e & (8)\\
[\mathrm{c}][\mathrm{d}]e & = & [\mathrm{d};\mathrm{c}]e & (9)\\
[\mathrm{c}\rightarrow\mathrm{d}]\lambda x.e & = & \lambda x.[\mathrm{d}]([x\mapsto[\mathrm{c}]x]e) & (10)\\
([\mathrm{c}\rightarrow\mathrm{d}]e)\;e' & = & [\mathrm{d}](e\;([\mathrm{c}]e')) & (11)\\
\begin{array}{r}[\mathrm{d}]\mathtt{let}\;x = \Lambda\overline{\beta}.[\mathrm{c}(\overline{\beta})]e\;\mathtt{in}\\ {[x^i\mapsto x\{\overline{\pi_\mathrm{i}}\}]e'}\end{array} & = & \begin{array}{l}\mathtt{let}\;x = \Lambda\overline{\beta}.e\;\mathtt{in}\\ \quad[\mathrm{d}]([x^i\mapsto[\mathrm{c}(\overline{\pi_\mathrm{i}})](x\{\overline{\pi_\mathrm{i}}\})]e')\end{array} & (12)\\
[\mathrm{c}]x\{\overline{\pi_\mathrm{i}}\} & = & [\mathrm{c}']x\{\overline{\pi_\mathrm{i}}'\} & (13)
\end{array}
$$

Figure 7: Equality rules for explicitly boxed expressions

PROOF: (If) Assume $e'{=}e''$. Clearly e' and e'' must have the same erasure. Since both e' and e'', by definition of equality, are completions with the same type they must be completion congruent, i.e. $e'{\cong}e''$.

(Only if) We will prove that $e'{\cong}e'' \Rightarrow e'{=}e''$ by induction on the structure of the erasure of e and e'. We can without loss of generality assume that every subterm in e and e', that is not an application of a coercion, has exactly one coercion applied to it. We can do this since any term of the form $[\mathrm{c}][\mathrm{c}']e$ is equal to $[\mathrm{c}';\mathrm{c}]e$, and e is equal to $[\iota]e$. We may also assume without loss of generality that type abstractions only occurs as the topmost expression in a let-binding and that the expressions in type application are variables (as in the Milner/Tofte type system for ML).

Now assume that we have $A \vdash e' : \tau$ and $A \vdash e'' : \tau$:

Base case 1: $|e'| \equiv x\{\overline{\tau}\}$. Let $e' \equiv [\mathrm{c}']x\{\overline{\pi'}\}$ and $e'' \equiv [\overline{\mathrm{c}''}]x\{\pi''\}$. Then it is a trivial consequence of rule 13 in Figure 7 and Lemma 5 that these are equal.

Inductive step 1: $|e'| \equiv \lambda x{:}\tau_\mathrm{x}.e_1$. Assume that $e' \equiv [\mathrm{c}']\lambda x{:}\tau'.e_1{}'$ and $e'' \equiv [\mathrm{c}'']\lambda x'{:}\tau''.e_1{}''$. Since c' has type signature $\tau'{\rightarrow}\tau_1{}' \rightsquigarrow \tau$ and c'' has type signature $\tau''{\rightarrow}\tau_1{}'' \rightsquigarrow \tau$ for some types $\tau_1{}'$ and $\tau_1{}''$ there must exist a canonical coercion d such that $\vdash \mathrm{c}' = \mathrm{c}_1{}'{\rightarrow}\mathrm{c}_2{}';\mathrm{d}$ and $\vdash \mathrm{c}'' = \mathrm{c}_1{}''{\rightarrow}\mathrm{c}_2{}'';\mathrm{d}$. In fact d can only be either ι_τ or $\mathtt{box}_\upsilon$ where $\tau{=}[\upsilon]$ and d is completely determined by τ.

This means that we have

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

and

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

for some type τ_d and by induction

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

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

Inductive step 2: $|e'| \equiv e_1\;e_2$. Let $e' \equiv [\mathrm{c}'](e_1{}'\;e_2{}')$ and $e'' \equiv [\mathrm{c}''](e_1{}''\;e_2{}'')$. We know that there exists a coercion d such that $[\mathrm{d}]e_2{}'$ and $e_2{}''$ have the same type. We then have:

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

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

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

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

□

This shows that if we are allowed to pick *any* completion whatsoever of a core-XML expression then the observational congruence of (any semantics for) explicitly boxed core-XML automatically satisfies the equality theory above, and vice versa. Otherwise one could choose two completions (at the same type) of some program with different observable behavior.

5 Reduction of completions

The equalities in Figures 6 and 7 express which expressions (with coercions) are equal in the sense that they are legal completions of the same original Core-XML expression. So given a completion that matches one side of an equality axiom we may replace it by its other side. We can look at this as a reduction step, but the problem is that we do not know in which direction to use the equalities and what to aim at.

What we aim at is simply to reduce the number of box and unbox operations that have to be performed at run-time. Let us denote the last two equalities of Figure 6 by ϕ and ψ, respectively, and all the other equalities together with all the equalities of Figure 7 by E. Clearly, ϕ and ψ eliminate primitive coercions when applied as left-to-right rewriting rules whereas the equalities in E just "shuffle" the coercions around in an expression without eliminating any primitive coercions. Thus we shall 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.

One attempt might be to orient the E-equalities in some fashion. To see that there is a problem with let us look at an example. Assume that we were to direct also the last equalities in Figure refexp-cong-axioms from left to right and consider the expression:

$([\mathrm{c}{\rightarrow}\mathrm{c}']\,\lambda x.e)\ e'$

To this both rule 10 and 11 are applicable and there seems to be no obvious way to bring the two reducts together again for all cases. 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 of the reduction system we are about to describe is that coercions may be split up into two kinds of coercions that interact differently with the reductions rules. The intuition is that we want to move coercions that box towards as late a position in the evaluation of expressions as possible and to move unbox coercions to early positions. To give an idea of what we mean by "later", we think of a conditional expression being evaluated later than both its branches. The reason for wanting to do this is that this guarantees that an eliminatable box coercions will eventually collide with a corresponding unbox coercion and get eliminated by application of the ϕ-rule. Running the rewrite system *backwards* will then bring any remaining unbox/box pairs together and eliminate these using the ψ-rule.

5.1 Positive and negative coercions

The two kinds of coercions are called *positive* and *negative coercions*. The ones that move outwards in expressions, e.g. from the branches of an if-expression to the if-expression itself, are the positive coercions. Those that move the other way are the negative ones.

An arbitrary coercion will not in general be either positive (equal to its own positive component) or negative (equal to its own negative component) and by adding a superscript + or + to a coercion we indicate that the coercion is in fact positive or negative. (These annotations can be regarded as side conditions that have to hold before the corresponding rule may be applied. So in the rules in Figure 8 a superscript + on a coercion means that the coercion has to be positive for the rule to be applicable.)

Definition 7 (Expressions reduction) Let the term rewriting system R be the R-rules of Figure 8 and the set of equations be the E-rules of Figure 8. We the define expression reduction, written $\vdash e \Rightarrow e'$ and $\vdash e \Rightarrow^* e'$ to be R-reduction under E-equality with these definitions. □

Definition 8 We define $\Longrightarrow_{R\backslash\psi}$ to be the rewrite system of Figure 8 in which the rule ψ has been remove from the coercion equality system. Likewise, we define $\Longleftarrow_{E\backslash\phi}$ to be the rewrite system where the rules of Figure 8 is used in the reverse direction (also reversing the positive and negative conditions on coercions) and where the rule ϕ has been remove from the coercion equality system. □

Definition 9 We define an partial ordering on representation type as the smallest partial order for which $\rho \sqsubseteq [\rho]$ holds. □

Theorem 10 (Strong normalization) The two reduction systems $\Longrightarrow^*_{R\backslash\psi}$ and $\Longleftarrow^*_{E\backslash\phi}$ are strongly normalizing, i.e. it terminates for all completions.

Proof: We consider $\Longrightarrow^*_{R\backslash\psi}$ first. If we examine the rules of Figure 8 then every time a rules is applied there is a subsexpression for which the type gets smaller w.r.t. the partial ordering defined above. Since there are no descending chains in the domain of representation types reduction must terminate. Applying the ϕ rule only makes things better since it remove some of the box/unbox coercions.

The proof for $\Longleftarrow^*_{E\backslash\phi}$ is similar since there are no ascending chains in the domain of representation types either. □

Theorem 11 (Weak confluence of expression reduction)

$$
\begin{array}{rcll}
(\mathrm{c};\mathrm{c}');\mathrm{c}'' & = & \mathrm{c};(\mathrm{c}';\mathrm{c}'') & (E1)\\
[\iota]e & = & e & (E2)\\
[\mathrm{c}][\mathrm{d}]e & = & [\mathrm{d};\mathrm{c}]e & (E3)\\
[\mathrm{c}]e & \Longrightarrow & [\mathrm{c}\downarrow_+][\mathrm{c}\downarrow_-]e & (R1)\\
 & & \text{if c not positive or negative} & \\
[(\mathrm{c}\rightarrow\mathrm{d})^-]\lambda x.e & \Longrightarrow & \lambda x.[\mathrm{d}]([x\mapsto[\mathrm{c}]x]e) & (R2-)\\
\lambda x.[\mathrm{d}^+][x^i\mapsto[\mathrm{c_i}^-]x]e & \Longrightarrow & [\mathrm{c}\rightarrow\mathrm{d}]\lambda x.[x^i\mapsto[\mathrm{c_i}']x]e & (R2+)\\
 & & \text{where } \mathrm{c_i} = \mathrm{c};\mathrm{c_i}' \text{ for all i} & \\
[\mathrm{d}^-](e\ [\mathrm{c}^+]e') & \Longrightarrow & ([\mathrm{c}\rightarrow\mathrm{d}]e)\ e' & (R3-)\\
([(\mathrm{c}\rightarrow\mathrm{d})^+]e)\ e' & \Longrightarrow & [\mathrm{d}](e\ ([\mathrm{c}]e')) & (R3+)\\
\begin{array}{l}[\mathrm{d}^-]\mathtt{let}\ x = \Lambda\overline{\beta}.e\ \mathtt{in}\\ \quad [x^i\mapsto[\mathrm{c_i}(\overline{\pi_\mathrm{i}})^-](x\{\overline{\pi_\mathrm{i}}\})]e'\end{array} & \Longrightarrow & \begin{array}{l}\mathtt{let}\ x = \Lambda\overline{\beta}.[\mathrm{c}(\overline{\beta})]e\ \mathtt{in}\\ \quad [\mathrm{d}][x^i\mapsto[\mathrm{c_i}'(\overline{\pi_\mathrm{i}})](x\{\overline{\pi_\mathrm{i}}\})]e'\end{array} & (R4-)\\
 & & \text{where } \mathrm{c_i}(\overline{\beta}) = \mathrm{c}(\overline{\beta});\mathrm{c_i}'(\overline{\beta}) \text{ for all i} & \\
\begin{array}{l}\mathtt{let}\ x = \Lambda\overline{\beta}.[\mathrm{c}(\overline{\beta})^+]e\ \mathtt{in}\\ \quad [\mathrm{d}^+][x^i\mapsto x\{\overline{\pi_\mathrm{i}}\}]e'\end{array} & \Longrightarrow & \begin{array}{l}[\mathrm{d}]\mathtt{let}\ x = \Lambda\overline{\beta}.e\ \mathtt{in}\\ \quad [x^i\mapsto[\mathrm{c}(\overline{\pi_\mathrm{i}})](x\{\overline{\pi_\mathrm{i}}\})]e'\end{array} & (R4+)
\end{array}
$$

Further side conditions on the reduction rules:

At least one coercions on the left hand sides has to be proper.

In rules $R9+$ and $R11-$ c has to be proper.

Substitution patterns has to be necessary, i.e. there has to be at least one occurrence of the variable in the expression, otherwise use $\mathrm{c}(\overline{\beta}) = \iota$.

Figure 8: Reduction system for explicitly boxed expressions

The two reduction systems $\Longrightarrow^*_{R\backslash\psi}$ and $\Longleftarrow^*_{E\backslash\phi}$ are weakly Church-Rosser.

PROOF: We consider $\Longrightarrow^*_{R\backslash\psi}$ first. If $[\mathtt{unbox}_\rho;\mathtt{box}_\rho]e$ is a subexpression which is part of a redex in a reduction sequence. Then this can be rewritten to $[\mathtt{box}_\rho][\mathtt{unbox}_\rho]e$ where $\mathtt{box}_\rho$ is positive and $\mathtt{unbox}_\rho$ is negative. This means that one of the rule in Figure 8 may be applicable and either $\mathtt{box}_\rho$ or $\mathtt{unbox}_\rho$ may take part in this. If ψ was included in the reduction system $\Longrightarrow^*_{R\backslash\psi}$ then we could also rewrite $[\mathtt{unbox}_\rho;\mathtt{box}_\rho]e$ to $[\iota_\rho]e$, and we would have a critical pair. This is not the case when ψ is removed. By inspecting the rest of the rules one can check that there are not any critical pairs between these.

The prove for $\Longleftarrow^*_{E\backslash\phi}$ is similar. □

COROLLARY 12 (Church-Rosser) The two reduction systems $\Longrightarrow^*_{R\backslash\psi}$ and $\Longleftarrow^*_{E\backslash\phi}$ are Church-Rosser.

PROOF: Follows from Theorem 10, Theorem 11 and Newman's Lemma. □

THEOREM 13 (Characterization of =) Let the operation $\circ$ be the composition of relations, $\Longleftrightarrow$

the symmetric closure of $\Longrightarrow$, and $\_^*$ the reflexive, transitive closure on relations. Then the following holds:

$$e\ (\Longleftrightarrow^*_{R\backslash\psi}\ \circ\ \Longleftrightarrow^*_{E\backslash\phi})^*\ e'\ \textbf{iff}\ \vdash e = e'$$

Proof: (Sketch)

Only if: It is easy to see that the every rewrite rule holds equationally.

If: Let $\vdash e = e'$. Then there is a chain of elementary equalities connecting $e$and e'. By Church-Rosser and strong normalization property of $\Longrightarrow^*_{R\backslash\psi}$ and $\Longleftarrow^*_{E\backslash\phi}$, commutativity of ϕ (ψ) with respect to the other rewriting rules (reverse of the other rewriting rules), and the fact that every coercion without ϕ- or ψ-redex can be $-/+$-factored and $+/-$ factored, it follows that there must be a e'' such that both e and e' reduce to e'' under $(\Longrightarrow^*_{R\backslash\psi}\ \circ\ \Longleftarrow^*_{E\backslash\phi})^*$.

□ This shows that no matter what completion we choose for an

explicit typed core-ML expression reduction will give us a unique one. Is this the best one and in what sense? At least it holds that if any two coercions $\texttt{box}_\upsilon$ and $\texttt{unbox}_\upsilon$ can ever meet and cancel out, then the will by reduction, so in that sense the normal forms are the best ones. But rule $R9-$ may duplicate coercions so work may be duplicated and the normal form expressions may therefore perform more operations than some no normal form expressions. It is however outside this work to decide whether work will be duplicated in this way, since, beside it being undecideable, it would require an evaluation order analysis and would also depend on the given semantics (strict or non-strict).

6 Implementation

We have written a prototype implementation (in Haskell) of the rewrite system shown in Figure 8. The implementation handles the Core-XML languages extended with a conditional, pairs, a fix-point operator and polymorphic constants. The latter makes it possible to introduce list by just adding a list type to the implementation.

The result of running the rewrite system on one of these extended Core-XML programs is a SML program in which box and unbox coercions behave like the identity function, but also performs profiling operations. That is, besides returning its input the box or unbox coercions counts how many times they are called and the final result of running such an SML program is, beside the actual result, also a count of how many box operation and how many unbox operation that were performed while running the program. We believe that by counting the number of operation that are performed instead of actual run-time one gets a accurate measure of the quality of the produced programs which is independent of the actual run-time system.

We have run some preliminary test with the implementation where we compared our results with those obtained by using the completion method of Leroy [Ler92]. The tests run so far consists only of relatively simple programs testing various parts of the reduction system. Test does, however, shows that we obtain considerably better results by our method than by that of Leroy.

The full length paper will contain performance results for more elaborate examples.

7 Related work

7.1 Boxing

The substantial cost of manipulating data in boxed representation, especially for numeric programs, has been observed in high-level languages like Standard ML and Haskell. 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 at the core the languages they use for this purpose are 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 without any boxing optimization. The results, though, are not uniformly better, which raises the question of how much of the differences can be attributed to boxing *per se* and how much to the interaction of boxing with the remaining parts of compiler. Our rewriting system will improve the result of Leroy's target expression and in fact any completion with the same representation types at type applications as Leroy's to a unique, "optimal" completion.

Using Leroy's framework Poulsen [Pou93] presents a more involved translation that, but draws on constraint solving to eliminate more boxing/unboxing operations than Leroy's translation. The interesting aspect of this work is that it does not stipulate a canoncically defined representation type for types occurring in type applications, as in Leroy's and our 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 not accounted for in the question of optimizing the boxing in the program.

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, Gelli [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 arbitary isomorphic types. Our application can be viewed as simple variant of this problem as arbitary 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 by 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.

Acknowledgements

References

[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.

[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.

[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.

[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.* Springer, Aug. 1991.

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