Generalized Boxings, Congruences and Partial Inlining

Jean Goubault

Bull Corporate Research Center, rue Jean Jaurès, Les Clayes sous Bois, France
Email: `Jean.Goubault@frcl.bull.fr` Tel: (33 1) 30 80 69 28

Abstract. We present a new technique for optimizing programs, based on data-flow analysis. The initial goal was to provide a simple way to improve on Leroy and Peyton-Jones' techniques for optimizing boxings (allocations) and unboxings (field selection) away in strongly-typed functional programs. Our techniques achieve this goal, while not needing types any more (so it applies to Lisp as well as to ML), and providing a finer analysis; moreover, our techniques also improve on classical common subexpression elimination. The methods are twofold: intraprocedurally, we use a data-flow analysis technique that propagates congruences on program variables and terms built on these variables; interprocedurally, we help the intraprocedural analysis by inlining, at least partially, all functions whose definitions are known at call sites.

1 Introduction

One of the source of inefficiencies of compiled functional language implementations is the great number of boxings (allocating tuples on the heap) and unboxings (reading off components of tuples) that are to be done at run-time by the compiled code. Unboxing a value that has just been boxed is costly because it may force a useless read to get data that was already in a register. Boxing a value is costly in allocation time, but also in garbage collection time: the more data are boxed, the more time the garbage collector spends to detect and free dead data. Interesting cases to detect at compile-time are when boxings are *redundant* (the same tuple is allocated twice), or *useless* (a tuple that is never used is allocated), or *annihilated* by a subsequent unboxing operation (a tuple is allocated, only to be unboxed).

In general, boxings and unboxings are not limited to tuples. Floating-point values are usually boxed in functional language implementations, thereby strongly penalizing all numerical computations: on modern architectures, floating-point operations take just a few cycles and run in parallel with integer and addressing instructions; boxing then incurs a loss of parallelism and a high overhead. In dynamically-typed languages like Lisp, we might also want to box machine integers to make tag tests faster on a system-wide basis (this is unnecessary in languages like ML). Moreover, allocating strings on the heap, allocating closures and run-time environments, allocating references (in ML) are also special cases of boxings. We can generalize the notion, and consider, for example, that integer addition is a boxing operation, and subtraction is an unboxing; or in a set language, that building a set is a boxing operation, and that testing for set membership is an unboxing.

We propose a static analysis framework for detecting redundant boxings, useless boxings, and annihilating boxing/unboxing pairs simultaneously. It propagates finite sets of equalities between program variables and expressions, in a way reminiscent of congruence closure. This analysis decorates the control-flow graph for the program, and this decoration is in turn used for simplifying the program.

To achieve gains across procedure calls, we introduce the notion of *partial inlining.* Whereas inlining transforms interprocedural analyses into intraprocedural ones, it is not always wise or possible to inline systematically all procedures. For our purposes, it will be enough to inline only parts of functions that unbox input arguments and box the results to achieve significant improvements.

The plan of the paper is as follows. In Section 2 we present related work in the field of optimizations related to boxing and unboxing, and in data-flow analysis. As this is a compilation problem, we present in Section 3 a suitable abstract syntax on which to apply our optimizations. We explain the intraprocedural techniques in Section 4, and deal with higher-order functions in Section 5. We conclude in Section 6.

2 Related Work

In the framework of functional programming languages, a simple and elegant system for converting a ML program with uniform representation (all values boxed) to mixed representation (some values are unboxed) was proposed in 1992 by Leroy [17]. The technique consists in translating source ML code to a target language with explicit boxing and unboxing operations. The translation is guided by the structure of type derivations. The target language is essentially ML again, except that types are enriched by types of the form $[\tau]$ (the type of all boxed data of type τ; τ itself represents the type of all unboxed data of type τ), and there are two new *coercion* operations, `wrap`(τ) to box values of type τ, and `unwrap`(τ) to unbox values of type $[\tau]$. This language can then be optimized, using for instance techniques by Peyton-Jones [20], which are expressed as source-to-source transformations on the target language.

This system is simple, elegant, and deals naturally with higher-order functions; however, as Leroy notices, it has a few drawbacks, and notably that all coercions are strict (not lazy), so that this may introduce spurious annihilating boxing and unboxing operations. Peyton-Jones' techniques may be used to eliminate the latter, but any naive implementation of these would require many passes over the whole code, and it is not clear how we could implement these in a clever way. Moreover, special care should be taken to extend the system to call-by-value and to include side-effects on mutable data, as Peyton-Jones heavily relies on the validity of β-reduction to prove his transformations correct.

Henglein and Jørgensen [13] present a complicated calculus to optimize annihilating boxing/unboxing pairs, again in a typed and side-effect free setting.

Redundant and useless boxings are not dealt with. Moreover, some doubts have recently been expressed as to the validity of their results.

Instead of producing lazy coercions directly, our idea is to produce strict coercions again, and to eliminate them by data-flow analysis techniques. This is already needed in Leroy's system, which produces annihilating boxing/unboxing pairs. By using a different abstract syntax (presented in Section 3), we shall actually dispense with coercions at all, and get a system that does not depend on typing to work. So our system will be applicable even to languages that are untyped, impure (with side-effects) and with any kind of semantics for calls (by value, by need, by name, for example). In these respects, we shall subsume both Leroy's and Peyton-Jones' techniques.

Data-flow analysis techniques have been thoroughly developed for imperative-style programming languages [1], and already solve some of our problems: redundant (generalized) boxings are also called *common subexpressions* [4, 9], and can be detected by *congruence closure* [9, 18, 10] and then eliminated. The first algorithms to do this were limited to basic blocks, but they can be extended to full procedures [21] by using *static single assignment form* (SSA) for programs [6] (see Section 3).

If all we want is to detect redundant boxings, then very fast techniques are available [2] which run in time $O(E \log E)$, where E is the number of edges in the control-flow graph. To detect useless boxings, a *dependence flow graph* (DFG) [14] can be built that links definitions to uses (this is the converse of a use-def chains graph [15]); as only possibly useful definitions are needed for building the DFG, useless ones will remain outside the DFG, and needn't give rise to any actual code. Apart from Henglein and Jørgensen, annihilating boxing/unboxing pairs have been dealt with by Peterson [19]. But his technique is based on the assumption that data live in exactly one state at any time (boxed or unboxed, but not both), which can limit the benefits of the claimed optimality of his optimization algorithm. We consider boxed data as different from their unboxed counterparts, so that this restriction (which can be partly lifted in Peterson's case) has no equivalent in our framework. The only drawback of doing so is that we may force additional register spills because we now need more registers to hold more different representations of the same data: our point is that these spills are usually less costly than the boxings and unboxings we save.

Finally, it is a common assumption that these techniques, which are all intraprocedural, generalize to interprocedural analyses through procedure inlining. But, even though full inlining is possible by translating procedure calls to gotos and managing a call stack by hand, this might lead to impractical analyses for large programs. Moreover, the assumption that we can always inline breaks when separate compilation is needed, unless we leave the job of optimizing programs to the linker. We solve the problem by inlining functions *partially*, as we show in Section 5. This can be seen as an efficient way of doing deforestation [26]. This will, by the way, enable us to correct a defect in Leroy's treatment of higher-order functions. Our technique is quite close to *call forwarding* [7]. We use a simpler scheme in that instructions are not reordered: we feel this should be enough for

ML-like languages, although for dynamically-typed languages, we agree with the authors that reordering is certainly needed. In contrast, whereas call forwarding moves actions done on entry to procedures to their call sites, we also move exit actions from exits of procedures to just after their call sites (by analogy, we might call this "return backwarding").

3 An Abstract Syntax

Leroy presents his technique on a mini-ML-like language, i.e. a call-by-value λ-calculus with constants and the `let` construct, where the only data structures are pairs. To get a more realistic language, we add mutable ML references to be able to do side-effects, and conditional expressions for convenience.

A good way of compiling any language, including functional ones, is to translate the source language to an intermediate representation, which should reflect closely enough the operational semantics of the target machine. We use a *static single-assignment form* representation [6]. A code fragment for representing a function is a graph whose nodes are statements. Apart from function headers, statements are return instructions, tests, joins (merging two control flow paths together) and assignments. Most assignments will have the form $x := f(x_1, \ldots, x_n)$, where x, x_1, …, x_n are variables, and f is a function symbol (we shall say a tag) representing a basic primitive of the target machine. Moreover, distinct assignments define distinct variables: there is only one site where any variable x can be defined, hence the name "static single assignment".

Figure 1 describes such a minimal language. We have not explicitly included booleans or integers, as they are unboxed anyway in most implementations. Statements can be: conditionals $\mathsf{if}(x, s, s')$ (test x, and continue execution at statement s if true, at s' if false); return statements $\mathsf{return}(x)$ (returning the value of x); or assignments $x := e; s$ which evaluate the expression e, define x as being the value, and resume execution at statement s. Each statement has zero, one or more continuations, which are pointers to statements, and which we have denoted by objects of type $\uparrow$stmt: this can be seen as describing a control flow graph. It is not quite in continuation-passing style [22, 3], as continuations are constant and cannot be passed to functions. Moreover, should we want to eliminate control artifacts that both control-flow graphs and continuation-passing style programs introduce, better but more complicated representations should be used [27]. We stick to our minimal language for simplicity.

Join nodes are assignments of the form $x := \mathsf{join}(x_1, \ldots, x_n)$. They have exactly n predecessors, and have the effect of assigning x to x_i on arrival from the ith predecessor: they are the same as the ϕ function of [2]. They also allow us to group all return statements into one join node followed by a unique return statement: we shall therefore assume that the return statement is unique.

Function headers are lists of variables, decorated with storage information, either boxed (any boxed value), or unboxed types (floating-point values, code addresses; we could also have added integers, booleans, etc.) Notice that the storage class of the return value of functions (or void if nothing is returned) can

```
code    ::= header ↑stmt        header ::= decl*

decl    ::= var : storage       storage ::= boxed | unboxed-float | unboxed-code

stmt    ::= if(var,↑stmt,↑stmt)
         |  return(var)
         |  var := expr; ↑stmt

expr    ::= const | var
         |  box-float(var) | unbox-float(var)
         |  box-pair(var,var) | fst(var) | snd(var)
         |  box-ref(var,var) | unbox-ref(var)
         |  box-clos(var,var) | unbox-fn(var) | unbox-env(var)
         |  apply(var,var*)
         |  set-ref(var,var)
         |  join(var*)
         |  ...

const   ::= nil | 0.0 | 1.0 | -1.0 | 0.5 | -2.0 | ... | code(code)
```

Fig. 1. A minimal static single-assignment language

be statically determined by looking at the last tag of an expression building a return value. When translating from Lisp or ML, these storage classes will all be boxed initially. The transformations we shall present in Section 5 will preserve the fact that the storage class is unique and does not depend on the particular chosen execution path through the code.

Expressions may be variables, constants, or simple instructions, identified by tags (box-float, etc.) Constants are all assumed to denote unboxed values, so there are instructions to box and unbox each type of values, as in [13]: reals (2nd row of the expr definition), couples or pairs (3rd row; fst selects the first component, snd the second component, as in Lisp), references (4th row), closures (5th row; they are pairs of a code address, i.e. a constant built with code, and an environment, that we take as a list of variable/value pairs, built with nil and box-cons to make the exposition simpler). Functions (or rather, code addresses) can be called on a list of arguments, so that apply$(f, x_1, \ldots, x_n)$ calls the code at address f with arguments x_1, ..., x_n (of which one in general is a closure environment). References can be modified by using set-ref (7th row), and join encode ϕ functions. Finally, the ellipsis (...) in the definition of expressions stands for any extra primitive instructions.

We chose these primitives because we wanted to clearly distinguish boxed and unboxed objects. The difference with Leroy's or Peterson's representation is small, technically: whereas they consider objects x in either boxed or unboxed states, we consider these states as different objects, with a boxing operation to build the boxed one from the unboxed one, and an unboxing operation to build the unboxed one from the boxed one. This is why we don't need unboxed pairs, notably: here, an unboxed pair (x, y) just consists in x and y separately.

This representation transparently encodes use-def chains [15], which are of great help in data-flow analysis. Indeed, variables x are either parameters (put in the header of the current code) or defined in exactly one assignment $x := e$, so we can identify variables with assignments and parameters, i.e. definition sites. So, in an expression e', the set of variables appearing in e' is precisely the set of definition sites corresponding to the use e'. In the implementation, we code an assignment $x := e$ as a mere new reference to e, which we see ambiguously as both the assignment statement and the variable x. Hence the value graph of [2] is already a subgraph of our control-flow graph.

We don't give the semantics of operators, as they should be clear from their informal description. In fact, since we rely on data-flow analysis, we only need to know what equations are generated and which equations are killed at each statement. Let us just say for the sake of illustration that box-ref is generative, and creates a new reference each time it is called (by allocating a new reference cell from the store, say), that unbox-ref gets back the current contents of this reference; and that, as in Standard ML, references are the only mutable data. In particular, pairs are not mutable. Finally, code is not generative: it returns the address of the compiled code for the function in argument.

Translating a piece of Lisp or ML code to this abstract syntax is straightforward: it basically consists in simulating the evaluation of this code by an interpreter on an SECD machine [16]. Care should however be taken in the way we translate constants. Because our language has fine-grain boxing operations, translating the evaluation of a constant like 1 means producing the sequence of assignments $x_1 := 1$; $x_2 :=$box-float(x_1). Likewise, assuming for instance that we have an additional tag add for adding two floating-point values, adding two boxed reals stored in x and y respectively should give rise to the sequence $x_1 :=$unbox-float(x); $x_2 :=$unbox-float(y); $x_3 :=$add(x_1, x_2); $x_4 :=$box-float(x_3);. To sum up, all boxing and unboxing operations are made explicit.

We do not need types to produce such a translation, so that we are not tied to ML, and can do this on Lisp for instance. However, this introduces many spurious boxing/unboxing pairs: this is why we shall be even more interested in optimizing them away. We describe our technique in Section 4. Moreover, when translating function calls, we assume that all functions take boxed values and return boxed values. So, this translation does not (yet) deal with the higher-order part of Leroy's method. But the partial inlining technique (described in Section 5), a modified kind of call forwarding, does precisely that.

4 Intraprocedural Analysis with Congruences

One of the primary goals we want to achieve is to eliminate redundant boxing/unboxing pairs. For example, assume we have the following piece of code:

s; $x_1 :=$box-float(x_0); $\ldots$; $x_n :=$unbox-float(x_1); s'

We would like to replace the unbox-float instruction by the faster $x_n := x_0$, and also to replace all subsequent uses of x_n by uses of x_0. Then, as x_n is not used any more, we might as well not produce any code for the useless definition $x_n := x_0$ when translating this to, say, assembler.

Detecting useless statements is easy: the only really needed statements are headers, return statements, statements with side-effects (just set-ref assignments in our language), and definitions of values used by needed statements. Detecting needed statements is then done by marking statements, following use-def chains. All other statements are useless, and no code needs to be generated for them. Note that we don't need to eliminate useless statements from the control-flow graph, except to speed up the analysis.

It remains to detect the semantical property that allows us to replace x_n :=unbox-float(x_1) by $x_n := x_0$. This is: unbox-float(box-float(x)) $= x$ for all x, instantiated to the case where x is x_0. Logically speaking, the program variable x_0 is a fixed object, that is, an uninterpreted constant, so this is a ground equation. In usual common subexpression elimination, these ground equations are further restricted to be equations between variables of the program (or, logically speaking, equations between constants only). Here we need more general equations like unbox-float(box-float(x_0)) $= x_0$ at the definition site for x_1, where no variable has yet been encountered that could describe the left-hand side.

4.1 Data-flow Analysis of Congruences

The general theoretical framework is the usual data-flow analysis one [1]. We need a set V of values to be propagated, a way of computing, for each statement s, a set $gen(s)$ of values generated by s, and a set $kill(s)$ of values killed by s; and a binary meet ($\wedge$) operation on values to be applied at join nodes. $(V, \wedge)$ should then be a lower semi-lattice, and the analysis computes a greatest fixed point of equations $out(s) = (in(s) \setminus kill(s)) \cup gen(s)$ (for non-join nodes), $out(x) = \bigcap_{i=1}^{n} out(x_i)$ at join nodes x :=join($x_1, \ldots, x_n$) (remember we identify variables with their defining statements), and $in(s') = out(s)$ if s' is a continuation of s. The set $out(s)$ is then a set of valid values after statement s, and is computed from the set $in(s)$ of valid values before s. The analysis terminates if the semi-lattice is well-founded; otherwise, we need narrowing operators (see [5], where data-flow analysis is shown to be a special case of the general abstract interpretation technique).

In our case, V should be the set of all finitely-generated congruences between ground terms built on program variables (playing the rôle of logical constants) with pure — side-effect free — instruction tags (playing the rôle of logical function symbols). These terms enjoy all the properties of classical ground terms in first-order logic [10]. A *congruence* is a binary relation $\cong$ that is reflexive, symmetric, transitive and hereditary: if $t_1 \cong t'_1$, ..., $t_m \cong t'_m$, then $f(t_1, \ldots, t_m) \cong f(t'_1, \ldots, t'_m)$. Such a congruence is finitely generated if it is exactly the set of equational consequences of a finite set of equations $t_1 \cong t'_i$, $1 \leq i \leq n$, i.e, the set of equations deductible from these and the rules for congruences.

Whether a given equation is an equational consequence of a finite set of ground equations is decidable in almost linear time [18, 9]. However, the natural order on V is $(\cong_1) \leq (\cong_2)$, defined as $\forall t, t' \cdot t \cong_1 t' \Rightarrow t \cong_2 t'$, i.e. as a set inclusion ordering, $\{(t,t') \mid t \cong_1 t'\} \subseteq \{(t,t') \mid t \cong_2 t'\}$. This indeed yields a lower semi-lattice, but it is not well-founded, as for instance the sequence of congruences generated by the single equation $x_0 = f^{n!}(x_0)$, for $0 \leq n < +\infty$, is infinite and strictly decreasing.

Restricting equations, rather severely, to act on program variables alleviates the problem, and yields the usual common subexpression problem semi-lattice. Note that the Union-Find structure used by the congruence closure algorithm is encoded inside the value graph itself (which is part of the control-flow graph in our representation). Equations $x = y$, once oriented as $x \to y$ with the definition of y dominating x, build a Union-Find tree, i.e. a tree where all edges are oriented links that point towards the roots (there are no cycles, as they could only be generated by join nodes, but join tags hide the uses of variables there).

4.2 Extending the Common Subexpression Framework

To get a decidable procedure, the limitation of only using equations between program variables is not necessary, and it is enough to limit the height of terms to some finite constant: the set of allowed terms is then finite, hence the set of congruences on them, too. Although speaking of program variables only is not enough for us, dealing with equations of the form $t = x$, where x is a program variable and t is either a variable or a term $f(x_1, \ldots, x_m)$, where the x_i's are variables, will suffice.

This could be achieved by adding assignments of all possible such terms to new, unused variables, and then applying standard algorithms as in [21]. Intuitively, we could cache the unboxed values in unboxed temporaries. But we should then also cache boxed values: on defining x, we should forecast that we might use any couple (x, y) or (y, x) in the future, with y any arbitrary other variable. This is clearly impractical.

Among the useful equations in our language, we find the following (by convention, we write finite sets of equations instead of the corresponding finitely-generated congruences). On encountering $x :=$box-float(y), define $gen(x) = \{$box-float$(y) = x,$unbox-float$(x) = y\}$, $kill(x) = \emptyset$ (recall that assignments are identified with variables); symmetrically, for $x :=$unbox-float(y), $gen(x) = \{$unbox-float$(y) = x,$box-float$(x) = y\}$, $kill(x) = \emptyset$. For $x :=$cons(y, z), $gen(x) = \{$cons$(y, z) = x,$fst$(x) = y,$snd$(x) = z\}$, $kill(x) = \emptyset$ (our cons-cells are immutable); and for $x :=$fst(y), $gen(x) = \{$fst$(y) = x\}$, $kill(x) = \emptyset$ (and similarly with snd). For $x :=$box-ref(y), $gen(x) = \{$unbox-ref$(x) = y\}$ (no equation on box-ref, as it has a side-effect: it must produce a *new* reference cell), $kill(x) = \emptyset$. For $x :=$set-ref(y, z), $gen(x) = \{$unbox-ref$(y) = z\}$, and $kill(x)$ is the set of all equations unbox-ref$(y') = z'$ in $in(x)$ such that y' may be aliased to y (to make things simple, we can take all equations of this form; more sophisticated alias analyses may be used [8, 23]). For $x :=$apply$(x_1, \ldots, x_m)$, $gen(x) = \emptyset$, $kill(x)$ is the set of all equations of the form unbox-ref$(y') = z'$ in $in(x)$ (again, this is an alias problem; for a simple implementation, recognizing that x_1 is a function without side-effects is enough to produce $gen(x) = \{\mathsf{apply}(x_1, \ldots, x_m) = x\}$, $kill(x) = \emptyset$). The equations for the remaining tags follow the same principles.

In the fst and snd cases, we don't generate any equations of the form $\mathsf{cons}(x, z) = y$ where we know that $z = \mathsf{snd}(y)$, for example. This would not only be complex, but useless as well: either y was built by cons, and we already had all needed equations, or y was an input parameter to the current code. In the latter case, we can recover the needed equations by inserting just after the header the sequence $x_1 := \mathsf{fst}(y)$; $x_2 := \mathsf{snd}(y)$; $x_3 := \mathsf{cons}(x_1, x_2)$; $x_4 := \mathsf{assert}(y, x_3)$, where assert is a tag having no run-time effect, but asserting that its two arguments are equal (its second argument should never be marked as needed, since we don't want to output any code for the corresponding sequence of statements; this also forces us *not* to eliminate useless code at analysis time). Such an assertion can be deduced from the most general type of the input parameter y in ML. In other languages, like Lisp, such an equation cannot be inferred, but it would not hold anyway, as two calls to cons are then required to produce two different cells, even if they have the same contents (in contrast with Standard ML, where equality is determined by contents).

4.3 Implementation

We represent these restricted congruences as finite sets of oriented equations, represented as the disjoint union of a set of rewrite rules $x \to y$ between program variables, and a set of rewrite rules $f(x_1, \ldots, x_m) \to y$ rewriting expressions to variables. This does not yield a canonical representation for congruences, but it would probably too costly to maintain one in general by systematically cross-rewriting each rule by all the other rules. An exception is when the control-flow graph is reducible; then, if we traverse the control-flow graph in topological order, we need only rewrite the generated equations and the killed equations, never the equations in $in(s)$. But other optimizations might destroy the reducibility of the control-flow graph: in ML, for example, replacing raise expressions inside the scope of exception handlers by gotos in general produces irreducible graphs.

Replacing the semi-lattice of congruences by the semi-lattice of such restricted finite sets of equations, ordered by set inclusion, provides a lower, hence safe, approximation. Indeed, if E and E' are two finite sets of equations, the congruence generated by $E \cap E'$ is lower than or equal to the meet of the congruences generated by E and E' respectively. We therefore need only be able to compute set intersections, unions and differences quickly. Bit-vectors cannot be used easily here, and hash-tables are hardly usable for these operations. But binary hash-tries [24], as used in the HimML system [11, 12], provide a nice representation: hash-cons all terms (i.e, share all equal terms by using a global hash-table), so that terms have a unique address; a hash-trie is then an acyclic minimal deterministic finite automaton that recognizes a finite set of addresses, seen as binary numbers, or as sequences of bits. If n and n' are the cardinal of the two input sets,with $n \leq n'$, all these set operations take average time $O(1)$ if $n = 0$, or otherwise $O(n + \log(n'/n))$ (in practice, this means it takes time almost linear in the *smallest* of the two cardinals). Moreover, the standard deviation is negligible [25], and these times do not depend on the size of the elements. Finally, these operations are non-destructive: their arguments are not modified during the computation.

Assume that, at each statement s, $kill(s)$ and $gen(s)$ contain at most k equations, there are n nodes in the graph, and $k \leq n$. Then $in(s)$ and $out(s)$ will contain at most kn equations, and the average time for computing $out(s)$ as a function of $in(s)$ at a non-join node is $O(k + \log n)$, and $O(nkp)$ at p-ary join nodes: with the right data structures, analyzing a node in this approach can be done quite efficiently. To analyze a whole code fragment, we propagate systems of equations forward, decorating nodes with the resulting $out(s)$ systems of equations, until a fixed-point is reached, i.e. until all decorations are stationary. All standard techniques for speeding-up data-flow analyses are usable, notably the use of expression ranks as in [21].

Once the control-flow graph has been decorated with such systems of equations, it only remains to simplify all expressions in the program by following the arrows in the computed rewrite rules. Then, mark all needed expressions, and produce assembler code only for needed expressions.

4.4 Discussion

This optimization technique uses much more space than that of [2] (roughly quadratic space in n instead of linear). The reason is not our using a more expressive language of equations, but the fact that we have side-effects on general pointers, as simulated by operations on references. In [2], there are no pointers that called procedures might tread on. So all kill sets are indeed empty, giving rise to the simplification that only the set of equations at the return node is needed (in SSA form) to simplify the whole current code.

To see what is achieved by our technique, consider the arithmetic expression $a + b * b$, working on floating-point values computed by expressions a and b. Assuming we have all needed tags with the obvious interpretation, and that b is side-effect free, this would be translated to:

$x_1 := a$; $x_2 := b$; $x_3 := b$; $x_4 :=$unbox-float(x_2); $x_5 :=$unbox-float(x_3);
$x_6 :=$mult(x_4, x_5); $x_7 :=$box-float(x_6); $x_8 :=$unbox-float(x_1);
$x_9 :=$unbox-float(x_7); $x_{10} :=$add(x_8, x_9); $x_{11} :=$box-float(x_{10}); return x_{11}

which would be optimized to:

$x_1 := a$; $x_2 := b$; $x_3 := x_2$; $x_4 :=$unbox-float(x_2); $x_5 := x_4$
$x_6 :=$mult(x_4, x_4); $x_7 :=$box-float(x_6); $x_8 :=$unbox-float(x_1);
$x_9 := x_6$; $x_{10} :=$add(x_8, x_6); $x_{11} :=$box-float(x_{10}); return x_{11}

where, if we consider only needed statements, is equivalent to:

$x_1 := a$; $x_2 := b$; $x_4 :=$unbox-float(x_2); $x_6 :=$mult(x_4, x_4);
$x_8 :=$unbox-float(x_1); $x_{10} :=$add(x_8, x_6); $x_{11} :=$box-float(x_{10}); return x_{11}

The computation unboxes the values for a and b, computes the arithmetical expression entirely in registers (unboxed floats usually lie in floating-point registers at run-time), and boxes the result. This is important, as unboxing a into x_8 can be done on modern architectures in parallel with multiplication, since integer and floating-point ALUs run in parallel. Finally, notice that the computation of the common subexpression b has been factorized in x_4.

This optimization can also be applied to operations that are not boxings or unboxings in the strictest sense. The only things we have to produce are new ground equations in the required format. Therefore, we can express equations like $x+0=0$ or $(x+y)-y=0$, simplifications on high-level data-structures like lists or even on sets in a set-based language, for instance. We call these *generalized* boxings, and although the gains may not be big on arithmetic expressions, on all other data-structures they usually are; but this would lead us too far astray.

Our optimization is flow-insensitive, but we can make it partially flow-sensitive by interpreting ifs so that they generate new equations on each of their two respective branches. However, to get full flow-sensitivity we would have to change the basic semi-lattice V rather deeply.

5 Higher-Order Functions and Partial Inlining

Some boxings and unboxings remain in the optimized code of the example of Section 4.4; we should be able to get rid of them if they are not needed. The only case where they are needed, in the example, is when a and b come from components of data structures, and the result is to be put again in a data structure. As Leroy notes [17], it is not only hard but undesirable to flatten representations of data structures by putting unboxed values inside boxed structures; so we shall consider, as he does, that all boxed structures contain only boxed data.

The only other case where we cannot dispense (yet) with boxings and unboxings are when a and b are input parameters, and when the result is returned by the current piece of code. Leroy's technique can be thought of as solving this very problem. It converts all ML functions (originally taking one boxed argument and returning one boxed result) into *unwrapped* functions, which take unboxed arguments and return unboxed results. (Unboxed tuples are just a bunch of unboxed values, so unwrapped functions must return multiple results in general.)

On the other hand, we may use functionals, i.e. functions F that take functions f in argument. If F is polymorphic, then f may have different input/output specifications, but we should still be able to call F on f. Following Leroy, we therefore also need *stub functions* around unwrapped functions f. Stubs have a uniform interface (take one boxed argument, return one boxed argument); they first unbox their argument, then call f and box the result, which they return.

Leroy's scheme has a few defects, however. First, because of ML's explicit type annotations (not provided in Leroy's expository language): `fn x => (x,x)` is compiled as is, but annotating `x` with the specialized type τ `*` τ' would force Leroy to convert this into a function `fn (x1,x2) => ((x1,x2),(x1,x2))`, with a stub around it extracting `x1` and `x2` from `x`. Although this may seem a minor defect, this is pathological: type annotations should improve, not degrade the quality of the code. Another problem stems from the use of definitions by pattern matching in Standard ML: the function `fn (x,[]) => x | (x,[y]) => x+y`, for example, has type `num * num list -> num`, so Leroy would convert this to a function taking two arguments `x` and `z`, and computing `case z of [] => x | [y] => x+y`. However, if we do not inline the function, the test of `z` against `[]` hides the unboxing of `z` into `y` if `z` $\neq$ `[]`; these would be valuable information to increase the accuracy of our data-flow analyses. Our system is more versatile than Leroy's in this respect.

Be aware that we do not claim to get optimal solutions in any sense, but reasonably efficient compiled code using reasonably efficient compilation algorithms. Leroy's solution is optimal for monomorphic programs, we are not. But the gain seems small, as higher-order functions like `map` are useful, most of the time, precisely because they are polymorphic. The only serious flaw of our scheme is that, contrarily to Leroy, we cannot optimize across modules, as we cannot optimize calls unless we know the code of the called function; Leroy only needs their specification (their type), and so can do better in this case.

5.1 Partial Inlining

Our technique is similar to call forwarding [7]: copy entry actions in a procedure f to just before each of its call sites, and modify f so as to skip these actions. Additionally, we do the converse ("return backwarding"): copy exit actions of f to just after its call sites, and modify f so as to skip these exit actions.

To gain enough significant information to be re-injected into the intraprocedural data-flow optimizer, we define the entry (resp. exit) actions of interest as all possible unboxing (resp. boxing) operations at entry (resp. exit). After using call-forwarding and return-backwarding on a function f, we get its *unwrapped* representation $\tilde{f}$. And because f may be passed as an argument to a functional which may then call it, we leave a stub for f that just calls $\tilde{f}$ after doing the call-forwarded unboxings, and then does the return-backwarded boxings.

With call forwarding, it is usually necessary to reorder entry (and exit) actions so that a most favorable sequence can be extracted from the body of f. This is NP-complete in general because it needs to find an optimal sequence for tests (if nodes); considering only unboxings and boxings is simpler, as we now show. Initially, replace the code for f by code that just calls $\tilde{f}$, where $\tilde{f}$ is a new function whose body is the former body of f (now, $\tilde{f}$ has only one call site, i.e. the one in the body of f). Then proceed to call-forward and return-backward: as long as there is an argument x to $\tilde{f}$ that is only used by unboxing operations (we can detect this on a DFG [14]) in $\tilde{f}$, say y :=fst(x) and z :=snd(x), generate the same unboxings just before the call site of $\tilde{f}$ in f, replace the boxed argument x by the unboxed ones (here, y and z), both in the header of $\tilde{f}$ and at its call site, and replace all uses of the unboxed variables in the body of $\tilde{f}$ by references to the new formals. Symmetrically, as long as the result x of f is created by the same boxing operation on all paths (we detect this on use-def chains) in $\tilde{f}$, say x :=box-ref(y), generate the same boxing just after the call site of $\tilde{f}$ in f, replace the return(x) instruction at the end of $\tilde{f}$ by an instruction returning all unboxed components (here, y). Note that in general, we need a n-ary return instruction; if we compile to C, it is wise to restrict ourselves to unary boxings.

The code for f now serves as a stub for $\tilde{f}$, and also as a template that we inline systematically; we don't inline $\tilde{f}$, unless it is small enough (in this case, we are just doing classical inlining). This way, call sites to f become call sites to $\tilde{f}$, in the middle of call-forwarded and return-backwarded instructions. The whole interprocedural optimization then triggers a new intraprocedural pass to simplify codes containing calls that have just been inlined, and then stops. (We might also go on seeking for new inlinable functions after this pass, and continue until no progress can be made, but this is probably not worth it.)

If we had a call to some function f in the original program, what our procedure does depends on what we know about f. If f is a function whose code is unknown (an argument of the current function, or a function in another module), then f will necessarily be the wrapped version, and plays the rôle of Leroy's stub functions. We lose some efficiency w.r.t. Leroy's approach only when f was a function in another module. Otherwise, we know the code for f, and either it is small enough for us to inline it, and all annihilating boxing/unboxing pairs that appear are eliminated by the data-flow technique of Section 4; or it is still too big, and replacing f by some unboxings, a call to the unwrapped function $\tilde{f}$, and some new boxings gives good opportunities for new data-flow optimizations; actually, as many as in Leroy's approach, but without needing any type information, so this approach is equally well-suited to Lisp, for example.

Our approach can be extended to call-forward tests as well (as in the usual call forwarding mechanism). This involves return-backwarding corresponding join nodes as well. Even without reordering sequences of if nodes, this might prove useful for compiling ML patterns (if reordering is necessary, the algorithm of [7] should be used). A ML function `fn` pat_1 `=>` $expr_1$ `| ... |` pat_n `=>` $expr_n$ (where $expr_i$ is executed if the argument given to the function matches pat_i but none of the pat_j's, $1 \leq j < i$) is translated to code where, in addition to usual unboxing operations, conditionals are used to decide which pattern is the first to match. In this case, we also extract the conditionals that select on values of arguments from $\tilde{f}$, and put them in f; we then need to split $\tilde{f}$ in several $\tilde{f}_i$'s corresponding to each of the branches of the conditional; this is straightforward.

For instance, `fn (x,[]) => x | (x,[y]) => x+y` would be translated to a function f taking a couple, decomposed as `(x,z)`, and computing `case z of [] =>` $\tilde{f}_1$`(x) | [y] =>` $\tilde{f}_2$`(x,y)`, with $\tilde{f}_1$ `=fn x => x`, $\tilde{f}_2$ `=fn x,y => x+y` (where we have not explicitly shown the various numerical boxings and unboxings; also, in such a simple case, f_1 and f_2 should be inlined since they are so small, but this is only an example). Then, inlining f at its call sites gives our intraprocedural analysis an opportunity to optimize boxings and unboxings depending on the branch taken, calling either $\tilde{f}_1$ or $\tilde{f}_2$.

Finally, notice that our technique usually optimizes allocation of run-time environments away in lexically-scoped languages. When no closures are returned, we only need to build environments when currified functions are used. However, currified functions like `fn x => fn y => x` are typically small functions which return functions (here, `fn y => x`). The latter are then in turn partially inlined, eventually producing unwrapped functions that do not need to build or read values from heap-allocated environments any more.

6 Conclusion

We have shown that a combination of a simple intraprocedural data-flow analysis technique propagating systems of oriented equations between variables and terms of height at most 1 over program variables, and of a refinement of inlining that we call partial inlining, provided a more versatile and more powerful analysis technique than Leroy and Peyton-Jones' for detecting and eliminating redundant boxings, useless computations and annihilating boxing/unboxing pairs, at least inside a common module. Leroy's technique is superior when optimizing across modules, because it uses types, and we don't. We don't feel this is a serious problem in practice; but in ML, types should be used to give this level of code quality. We leave the problem of integrating this as future research.

Our technique is also beneficial to dynamically-typed languages like Lisp. And whatever the type discipline of the language it is applied to, it is not limited to boxings and unboxings, and generalizes the detection and elimination of redundant computations to that of annihilating computations; the cost does not appear to be higher than classical data-flow analyses.

This technique is being implemented in the HimML compiler [12], a compiler for a variant of Standard ML with fast sets and maps, for which it is in particular of paramount importance in simplifying computations on sets (building a set can be seen as a boxing operation, and testing for set membership as the corresponding unboxing operation). Although we don't have yet any practical measurements that would corroborate or invalidate our claims, we feel the techniques we have presented are promising.

Acknowledgements

I am grateful to Régis Cridlig for many fruitful discussions, to Patrick Cousot for pertinent advice, and finally to the anonymous referees for their comments.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.
2. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *15th PoPL*, pages 1–11, 1988.
3. A. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.
4. J. Cocke and J. Schwartz. Programming languages and their compilers: preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York, 1970. second, revised version.
5. P. Cousot and R. Cousot. A constructive version of Tarski's fixpoint theorems. *Pacific J. Math.*, 82(1):43–57, 1979.

6. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method for computing static single assignment form. In *16th PoPL*, pages 25–35, 1989.
7. K. De Bosschere, S. Debray, D. Gudeman, and S. Kannan. Call forwarding: A simple interprocedural optimization technique for dynamically typed languages. In *21st PoPL*, pages 409–420, 1994.
8. A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *ICCL'92*, 1992.
9. P. K. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, 1980.
10. J. H. Gallier. *Logic for Computer Science — Foundations of Automatic Theorem Proving.* John Wiley and Sons, 1987.
11. J. Goubault. Une implémentation efficace de structures de données ensemblistes, fondée sur le hash-consing. In *JFLA'93*, 1993.
12. J. Goubault. HimML: Standard ML with fast sets and maps. In *ACM Workshop on ML*, 1994.
13. F. Henglein and J. Jørgensen. Formally optimal boxing. In *21st PoPL*, pages 213–226, 1994.
14. R. Johnson and K. Pingali. Dependence-based program analysis. In *PLDI'93*, pages 78–89, 1993.
15. K. W. Kennedy. Node listings applied to data flow analysis. In *3rd PoPL*, pages 10–21, 1976.
16. P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.
17. X. Leroy. Unboxed objects and polymorphic typing. In *19th PoPL*, pages 177–188, 1992.
18. G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, April 1980.
19. J. Peterson. Untagged data in tagged environments: Choosing optimal representations at compile time. In *FPCA'89*, pages 89–99, 1989.
20. S. L. Peyton-Jones. Unboxed values as first-class citizens. In *FPLCA'91.* LNCS 523, Springer-Verlag, 1991.
21. B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *15th PoPL*, pages 12–27, 1988.
22. G. L. Steele. Rabbit: A compiler for Scheme. Technical Report MIT AI TR 474, MIT, May 1978.
23. J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *LICS'92*, 1992.
24. L. Trabb Pardo. Set representation and set intersection. Technical report, Stanford University, 1978. PhD thesis.
25. J. Vitter and P. Flajolet. Average-case analysis of algorithms and data structures. In J. v. Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 9. Elsevier Science Publishers b.v., 1990.
26. P. Wadler. Deforestation: Transforming programs to eliminate trees. *TCS*, 73:231–248, 1990.
27. D. Weise, R. F. Crew, M. Ernst, and B. Steensgard. Value dependence graphs: Representation without taxation. In *21th PoPL*, pages 297–310, 1994.