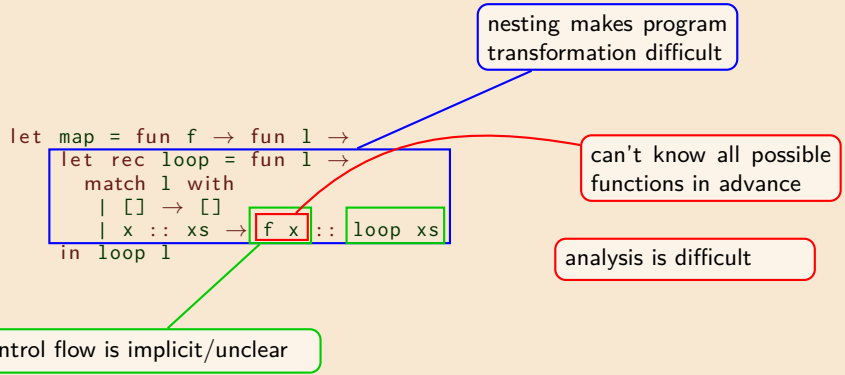
Taming functions

Jeremy Yallop

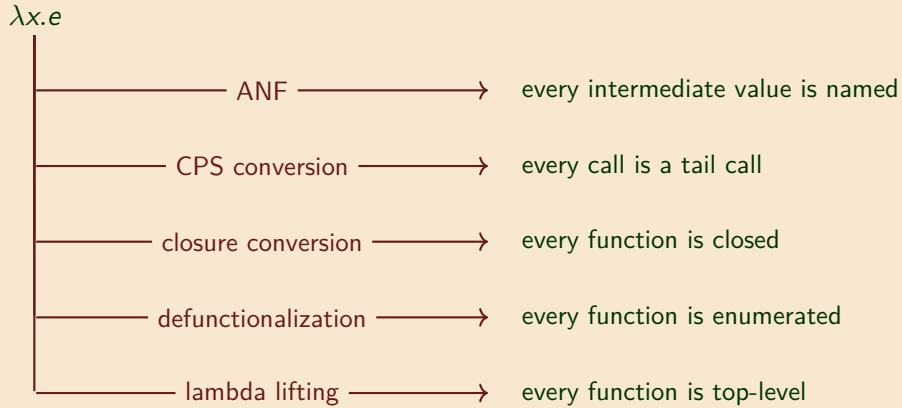
jeremy.yallop@cl.cam.ac.uk

Translations

Functions are hard to handle



We'll consider *total* translations that introduce syntactic invariants:



How should we view these translations?

Translations are *not optimizations*

Translations don't make programs faster (and may well make them slower)

Translations don't use heuristics, and aren't sensitive to structure

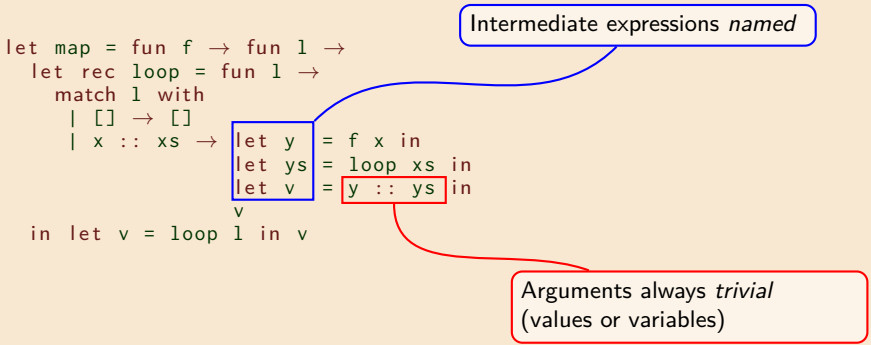
Translations are *representation changes*

The translation targets are *inconvenient as programming languages*

Changing representation makes analysis and further translation easier

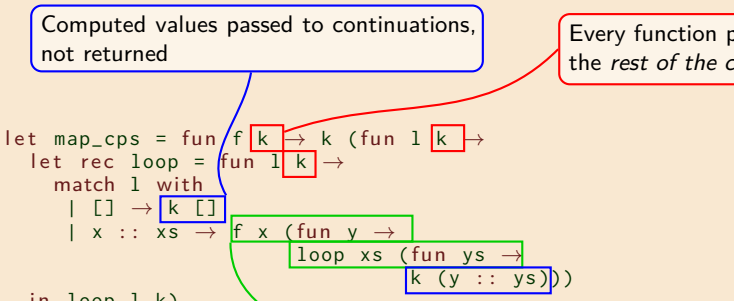
Translations by example

Translation to A-normal form



Full details: *The Essence of Compiling with Continuations* (PLDI 1993)
Cormac Flanagan, Amr Sabry, Bruce F. Duba, Matthias Felleisen

Translation to continuating-passing style



Closure conversion

All functions closed (i.e. C-style function pointers)

Free-variable environments bundled with functions

```
type ('a, 'b) clo =
  Clo : ('a → 'env → 'b) * 'env → ('a, 'b) clo

let app (Clo (f, env)) x = f x env

let fp3 = fun l env → match l with
  | [] → []
  | x :: xs → app env.f x :: app env.loop xs

let fp2 = fun l env →
  let rec loop = Clo (fp3, {f=env.g; loop}) in
  app loop l

let fp1 = fun f env → Clo (fp2, {g=f})

let map = Clo (fp1, ())
```

All source functions enumerated as constructors

```
type ('a, 'b) fn =
  | Map1 : (('a, 'b) fn, ('a list, 'b list) fn) fn
  | Map2 : ('a, 'b) fn → ('a list, 'b list) fn
  | Loop : ('a, 'b) fn → ('a list, 'b list) fn

let rec apply : type a b. (a, b) fn → a → b =
  fun fn x → match fn, x with
  | Map1, f → Map2 f
  | Map2 f, l → apply (Loop f) l
  | Loop f, l →
    match l with
      | [] → []
      | x :: xs → apply f x :: apply (Loop f) xs

let map = Map1
```

Free-variable environments attached to constructors

Free variables passed as additional parameters

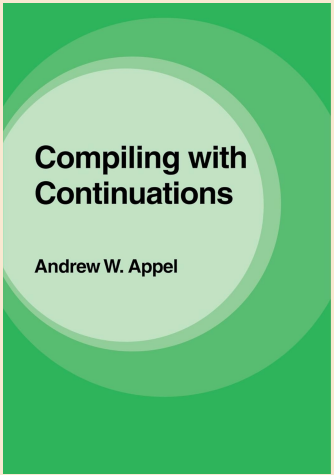
```
let map = fun f → fun l →
  let rec loop = fun l f →
    match l with
     | [] → []
     | x :: xs → f x :: loop xs f
  in loop l f
```

Functions moved to the top level

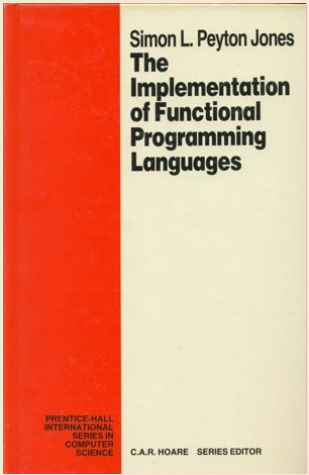
```
let rec loop = fun l f →
  match l with
  | [] → []
  | x :: xs → f x :: loop xs f

let map = fun f → fun l → loop l f
```

Reading

Background reading

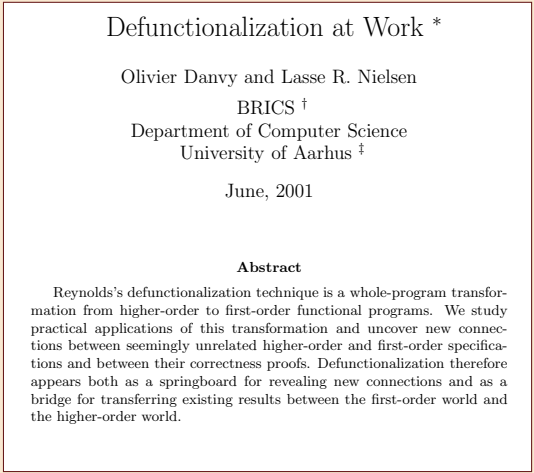


(for closure conversion)



(for lambda lifting)

Paper 1: *Defunctionalization at Work*



Defunctionalization at Work *

Olivier Danvy and Lasse R. Nielsen

BRICS †
Department of Computer Science
University of Aarhus ‡

June, 2001

Abstract

Reynolds's defunctionalization technique is a whole-program transformation from higher-order to first-order functional programs. We study practical applications of this transformation and uncover new connections between seemingly unrelated higher-order and first-order specifications and between their correctness proofs. Defunctionalization therefore appears both as a springboard for revealing new connections and as a bridge for transferring existing results between the first-order world and the higher-order world.

"Defunctionalization is thus *a whole-program transformation where function types are replaced by an enumeration of the function abstractions in this program.*

[...]

Compared to closure conversion and to combinator conversion, defunctionalization has been used very little.

[...]

...one can use several apply functions, e.g., grouped by types, ...One can also defunctionalize a program selectively, e.g., only its continuations ...One can even envision a lightweight defunctionalization"

Paper 2: *Typed closure conversion*

Functions

Translations

Reading

Typed Closure Conversion*

Yasuhiko Minamide †
Research Institute for Mathematical Sciences
Kyoto University
Kyoto 606–01, Japan
nan@kurims.kyoto-u.ac.jp

Greg Morrisett
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213–3891
jgmorris@cs.cmu.edu

Robert Harper
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213–3891
rwh@cs.cmu.edu

Abstract

Closure conversion is a program transformation used by compilers to separate code from data. Previous accounts of closure conversion use only *untyped* target languages. Recent studies show that translating to *typed* target languages is a useful methodology for building compilers, because a compiler can use the types to implement efficient data representations, calling conventions, and tag-free garbage collection. Furthermore, type-based translations facilitate security and debugging through automatic type checking, as well as correctness arguments through the method of logical relations.

We present closure conversion as a type-directed, and type-preserving translation for both the simply-typed and the polymorphic λ-calculus. Our translations are based on a simple "closures as objects" principle: higher-order functions are viewed as objects consisting of a single method (the code) and a single instance variable (the environment). In the simply-typed case, the Pierce-Turner model of object typing where objects are packages of existential type suffices. In the polymorphic case, more careful tracking of type sharing is required. We exploit a variant of the Harper-Lillibridge "translucent type" formalism to characterize the types of polymorphic closures.

1 Introduction

Closure conversion [29, 34, 5, 17, 16, 1, 39, 9] is a program transformation that achieves a separation between code and data. Functions with free variables are replaced by code abstracted on an extra environment parameter. Free variables in the body of the function are replaced by references to the environment. The abstracted code is partially applied to an explicitly constructed environment providing the bindings for these variables. This partial application of the code to its environment is in fact suspended until the function is actually applied to its argument; the suspended application, called a *closure*, is a data structure consisting of a piece of pure code and a representation of its environment.

A critical decision in closure conversion is the choice of representation for the environment — whether to use a flat FAM-like [4], linked CAM-like [5], or hybrid representation [30]. The choice of representation is influenced by a desire to minimize closure creation time, the space consumed by the environment, and the time to access a variable in the environment [39, 30]. An important property of closure conversion is that the representation of the environment is *private* to the closure. This affords considerable flexibility in the representation of environments and is thus exploited to good advantage by Shao and Appel [30] and Wand and Steckler [39].

Previous accounts consider closure conversion as a transformation to *untyped* terms, even if the source language is typed [34, 17, 1, 39, 9]. This is adequate for compilers that make little or no use of types in the back end or at run time. However, when compiling typed languages, it is often advantageous to propagate type information through each stage of the compiler, and to make use of types at link- or even run time. For example, Leroy's representation analysis [18, 31] uses types to determine procedure calling conventions, and Ohori's record compilation [25] uses a representation of types at run time to access components of a record. Compilation strategies for polymorphic languages, such as those proposed by Morrison *et al.* [24] and Harper and Morrisett [14], rely on analyzing types at run time to support unboxed representations and non-parametric operators, including printing and structural equality. Tag-free garbage collection [3, 37, 23] for both monomorphic and polymorphic programming languages relies on analyzing types at run time to determine the size and layout of objects in the heap. To support any of these implementation strategies, it is necessary to propagate type information through closure conversion and into the generated code. The purpose of this paper is to demonstrate how this can be done in both a simply-typed and a polymorphic setting.

We present closure conversion as an example of a *type-directed* and *type-preserving* translation. In general, such

*This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050, and in part by the National Science Foundation under Grant No. CCR-9502674, and in part by the Isaac Newton Institute for Mathematical Sciences, Cambridge, England. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of ARPA or the U.S. Government. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

†This research was performed while the first author was visiting the Fox Project at Carnegie Mellon University.

"*Closure conversion* is a program transformation used by compilers to separate code from data.

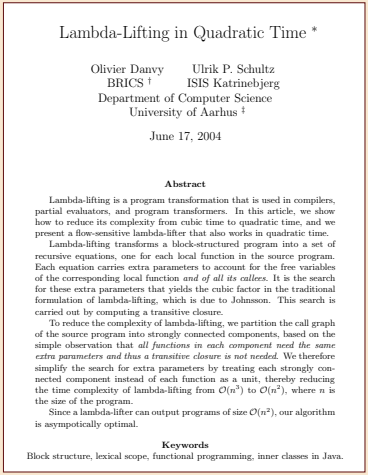
[...]

translating to typed target languages is a useful methodology for building compilers

[...]

we use existential type abstraction to ensure the privacy of environment representation in much the same way that Pierce and Turner hide the representation types of instance variables"

Paper 3: *Lambda-Lifting in Quadratic Time*



Lambda-Lifting in Quadratic Time *

Olivier Danvy Ulrik P. Schultz
BRICS † ISIS Katrinebjerg
Department of Computer Science
University of Aarhus ‡

June 17, 2004

Abstract

Lambda-lifting is a program transformation that is used in compilers, partial evaluators, and program transformers. In this article, we show how to reduce its complexity from cubic time to quadratic time, and we present a flow-sensitive lambda-lifter that also works in quadratic time.

Lambda-lifting transforms a block-structured program into a set of recursive equations, one for each local function in the source program. Each equation carries extra parameters to account for the free variables of the corresponding local function *and of all its callees*. It is the search for these extra parameters that yields the cubic factor in the traditional formulation of lambda-lifting, which is due to Johnsson. This search is carried out by computing a transitive closure.

To reduce the complexity of lambda-lifting, we partition the call graph of the source program into strongly connected components, based on the simple observation that *all functions in each component need the same extra parameters and thus a transitive closure is not needed*. We therefore simplify the search for extra parameters by treating each strongly connected component instead of each function as a unit, thereby reducing the time complexity of lambda-lifting from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$, where n is the size of the program.

Since a lambda-lifter can output programs of size $\mathcal{O}(n^2)$, our algorithm is asympotically optimal.

Keywords

Block structure, lexical scope, functional programming, inner classes in Java.

"Lambda-lifting is a program transformation that is used in compilers, partial evaluators, and program transformers.

[...]

Lambda-lifting transforms a block-structured program into a set of recursive equations, one for each local function in the source program. Each equation carries extra parameters to account for the free variables of the corresponding local function *and of all its callees*.

[...]

Since a lambda-lifter can output programs of size $O(n^2)$, our algorithm is asympotically optimal."

Writing suggestions

Relations

How are closure conversion and lambda lifting related?

How are closure conversion and defunctionalization related?

Preservation

Can the result of each translation always be given a type?

Do the translations extend to languages with more sophisticated type systems?

Variations

Can the translations be applied selectively? Do they support separate compilation?

Facilitation

What analyses and subsequent translations do the translations facilitate?