How to simplify matters

Simon Peyton Jones and Andre Santos
Department of Computing Science, University of Glasgow, G12 8QQ
simonpj@dcs.gla.ac.uk

April 14, 2017

1 Motivation

Quite a few compilers use the *compilation by transformation* idiom. The idea is that as much of possible of the compilation process is expressed as correctness-preserving transformations, each of which transforms a program into a semantically-equivalent program that (hopefully) executes more quickly or in less space. Functional languages are particularly amenable to this approach because they have a particularly rich family of possible transformations. Examples of transformation-based compilers include the Orbit compiler,[.kranz orbit thesis.] Kelsey's compilers,[.kelsey thesis, hudak kelsey principles 1989.] the New Jersey SML compiler,[.appel compiling with continuations.] and the Glasgow Haskell compiler.[.ghc JFIT.] Of course many, perhaps most, other compilers also use transformation to some degree.

Compilation by transformation uses automatic transformations; that is, those which can safely be applied automatically by a compiler. There is also a whole approach to programming, which we might call *programming by transformation*, in which the programmer manually transforms an inefficient specification into an efficient program. This development process might be supported by a programming environment in which does the book keeping, but the key steps are guided by the programmer. We focus exclusively on automatic transformations in this paper.

Automatic program transformations seem to fall into two broad categories:

- **Glamorous transformations** are global, sophisticated, intellectually satisfying transformations, sometimes guided by some interesting kind of analysis. Examples include: lambda lifting,[.johnsson lambda lifting.] full laziness,[.hughes thesis, lester spe.] closure conversion,[.appel jim 1989.] deforestation,[.wadler 1990 deforestation, marlow wadler deforestation Glasgow92, chin phd 1990 march, gill launchbury.] transformations based on strictness analysis,[.peyton launchbury unboxed.] and so on. It is easy to write papers about these sorts of transformations.

- **Humble transformations** are small, simple, local transformations, which individually look pretty trivial. Here are two simple examples[1]

  We implicitly assume that no name-capture happens — it's just a short-hand, not an algorithm.

  ```
  let x = y in E[x]          ==>    E[y]

  case (x:xs) of             ==>    E1[x,xs]
    (y:ys) -> E1[y,ys]
    []     -> E2
  ```

  Transformations of this kind are almost embarrassingly simple. How could anyone write a paper about them?

This paper is about humble transformations, and how to implement them. Although each individual transformation is simple enough, there is a scaling issue: there are a large number of candidate transformations to consider, and there are a very large number of opportunities to apply them.

In the Glasgow Haskell compiler, all humble transformations are performed by the so-called *simplifier*. Our goal in this paper is to give an overview of how the simplifier works, what transformations it applies, and what issues arose in its design.

2 The language

Mutter mutter. Important points:

- Second order lambda calculus.
- Arguments are variables.
- Unboxed data types, and unboxed cases.

[1]The notation E[] stands for an arbitrary expression with zero or more holes. The notation E[e] denotes E[] with the holes filled in by the expression e.

Less important points:

- Constructors and primitives are saturated.
- if-then-else desugared to `case`

Give data type.

3 Transformations

This section lists all the transformations implemented by the simplifier. Because it is a complete list, it is a long one. We content ourselves with a brief statement of each transformation, augmented with forward references to Section 4 which gives examples of the ways in which the transformations can compose together.

3.1 Beta reduction

If a lambda abstraction is applied to an argument, we can simply beta-reduce. This applies equally to ordinary lambda abstractions and type abstractions:

```
(\x  -> E[x]) arg       ===>     E[arg]
(/\a -> E[a]) ty        ===>     E[ty]
```

There is no danger of duplicating work because the argument is guaranteed to be a simple variable or literal.

3.1.1 Floating applications inward

Applications can be floated inside a `let(rec)` or `case` expression. This is a good idea, because they might find a lambda abstraction inside to beta-reduce with:

```
(let(rec) Bind in E) arg       ===>      let(rec) Bind in (E arg)

(case E of {P1 -> E1;...; Pn -> En}) arg
        ===>
case E of {P1 -> E1 arg; ...; Pn -> En arg}
```

3.2 Transformations concerning `let(rec)`

3.2.1 Floating let out of let

It is sometimes useful to float a `let(rec)` out of a `let(rec)` right-hand side:

```
let x = let(rec) Bind in B1          ==>    let(rec) Bind in
in B2                                       let x = B1
                                            in B2
```

```
letrec x = let(rec) Bind in B1       ==>    let(rec) Bind
in B2                                                x = B1
                                            in B2
```

3.2.2 Floating case out of let

3.2.3 let to case

3.3 Transformations concerning case

3.3.1 Case of known constructor

If a `case` expression scrutinises a constructor, the `case` can be eliminated. This transformation is a real win: it eliminates a whole `case` expression.

```
case (C a1 .. an) of                ==>   E[a1..an]
        ...
        C b1 .. bn -> E[b1..bn]
        ...
```

If none of the constructors in the alternatives match, then the default is taken:

```
case (C a1 .. an) of                ==>   let y = C a1 .. an
        ...[no alt matches C]...          in E
        y -> E
```

There is an important variant of this transformation when the `case` expression scrutinises a *variable* which is known to be bound to a constructor. This situation can arise for two reasons:

- An enclosing `let(rec)` binding binds the variable to a constructor. For example:

  ```
  let x = C p q in ... (case x of ...) ...
  ```

- An enclosing `case` expression scrutinises the same variable. For example:

  ```
  case x of
          ...
          C p q -> ... (case x of ...) ...
          ...
  ```

  This situation is particularly common, as we discuss in Section 4.1.

In each of these examples, `x` is known to be bound to `C p q` at the inner `case`. The general rules are:

```
  case x of {...; C b1 .. bn -> E[b1..bn]; ...}
===> {x bound to C a1 .. an}
  E[a1..an]

  case x of {...[no alts match C]...; y -> E[y]}
===> {x bound to C a1 .. an}
  E[x]
```

3.3.2 Dead alternative elimination

```
  case x of
         C a .. z -> E
         ...[other alts]...
===>                                 x *not* bound to C
  case x of
         ...[other alts]...
```

We might know that `x` is not bound to a particular constructor because of an enclosing case:

```
         case x of
                C a .. z -> E1
                other -> E2
```

Inside `E1` we know that `x` is bound to `C`. However, if the type has more than two constructors, inside `E2` all we know is that `x` is *not* bound to `C`.

This applies to unboxed cases also, in the obvious way.

3.3.3 Case elimination

If we can prove that `x` is not bottom, then this rule applies.

```
  case x of            ==>  E[x]
        y -> E[y]
```

We might know that `x` is non-bottom because:

- `x` has an unboxed type.
- There's an enclosing case which scrutinises `x`.
- It is bound to an expression which provably terminates.

Since this transformation can only improve termination, even if we apply it when `x` is not provably non-bottom, we provide a compiler flag to enable it all the time.

3.3.4 Case of error

```
  case (error ty E) of Alts    ==>    error ty' E
                              where
               ty' is type of whole case expression
```

Mutter about types. Mutter about variables bound to error. Mutter about disguised forms of error.

3.3.5 Floating let(rec) out of case

A `let(rec)` binding can be floated out of a `case` scrutinee:

```
  case (let(rec) Bind in E) of Alts   ==>   let(rec) Bind in
                                            case E of Alts
```

This increases the likelihood of a case-of-known-constructor transformation, because `E` is not hidden from the `case` by the `let(rec)`.

3.3.6 Floating case out of case

Analogous to floating a `let(rec)` from a `case` scrutinee is floating a `case` from a `case` scrutinee. We have to be careful, though, about code size. If there's only one alternative in the inner case, things are easy:

```
  case (case E of {P -> R}) of   ==>   case E of {P -> case R of
     Q1 -> S1
Q1 -> S1
       ...
 ...
     Qm -> Sm
Qm -> Sm}
```

If there's more than one alternative there's a danger that we'll duplicate `S1`...`Sm`, which might be a lot of code. Our solution is to create a new local definition for each alternative:

```
  case (case E of {P1 -> R1; ...; Pn -> Rn}) of
    Q1 -> S1
    ...
    Qm -> Sm
===>
  let   s1 = \x1 ... z1 -> S1
        ...
        sm = \xm ... zm -> Sm
  in
  case E of
    P1 -> case R1 of {Q1 -> s1 x1 ... z1; ...; Qm -> sm xm ... zm}
    ...
    Pn -> case Rn of {Q1 -> s1 x1 ... z1; ...; Qm -> sm xm ... zm}
```

Here, `x1 ... z1` are that subset of variables bound by the pattern `Q1` which are free in `S1`, and similarly for the other `si`.

Is this transformation a win? After all, we have introduced `m` new functions! Section 4.3 discusses this point.

3.3.7 Case merging

```
  case x of
    ...[some alts]...
    other -> case x of
                ...[more alts]...
===>
  case x of
    ...[some alts]...
    ...[more alts]...
```

Any alternatives in `more alts` which are already covered by `some alts` should first be eliminated by the dead-alternative transformation.

3.4 Constructor reuse

3.5 Inlining

The inlining transformtion is simple enough: `let x = R in B[x] ===> B[R]` Inlining is more conventionally used to describe the instantiation of a function body at its call site, with arguments substituted for formal parameters. We treat this as a two-stage process: inlining followed by beta reduction. Since we are working with a higher-order language, not all the arguments may be available at every call site, so separating inlining from beta reduction allows us to concentrate on one problem at a time.

The choice of exactly *which* bindings to inline has a major impact on efficiency. Specifically, we need to consider the following factors:

- Inlining a function at its call site, followed by some beta reduction, very often exposes opportunities for further transformations. We inline many simple arithmetic and boolean operators for this reason.
- Inlining can increase code size.
- Inlining can duplicate work, for example if a redex is inlined at more than one site. Duplicating a single expensive redex can ruin a program's efficiency.

Our inlining strategy depends on the form of `R`:
Mutter mutter.

3.5.1 Dead code removal

If a `let`-bound variable is not used the binding can be dropped:

```
let x = E in B          ==>          B
                x not free in B
```

A similar transformation applies for `letrec`-bound variables. Programmers seldom write dead code, of course, but bindings often become dead when they are inlined.

4 Composing transformations

The really interesting thing about humble transformations is the way in which they compose together to carry out substantial and useful transformations. This section gives a collection of motivating examples, all of which have shown up in real application programs.

4.1 Repeated evals

Example: x+x, as in unboxed paper.

4.2 Lazy pattern matching

Lazy pattern matching is pretty inefficient. Consider:

```
let (x,y) = E in B
```

which desugars to:

```
let t = E
    x = case t of (x,y) -> x
    y = case t of (x,y) -> y
in B
```

This code allocates three thunks! However, if `B` is strict in *either* `x` *or* `y`, then the strictness analyser will easily spot that the binding for `t` is strict, so we can do a `let`-to-`case` transformation:

```
case E of
  (x,y) -> let t = (x,y) in
           let x = case t of (x,y) -> x
               y = case t of (x,y) -> y
           in B
```

whereupon the case-of-known-constructor transformation eliminates the `case` expressions in the right-hand side of `x` and `y`, and `t` is then spotted as being dead, so we get

```
case E of
  (x,y) -> B
```

4.3 Join points

One motivating example is this:

```
if (not x) then E1 else E2
```

After desugaring the conditional, and inlining the definition of not, we get

```
case (case x of True -> False; False -> True\}) of
      True  -> E1
      False -> E2
```

Now, if we apply our case-of-case transformation we get:

```
let e1 = E1
    e2 = E2
in
```

```
case x of
   True  -> case False of \{True -> e1; False -> e2\}
   False -> case True  of \{True -> e1; False -> e2\}
```

Now the case-of-known constructor transformation applies:

```
let e1 = E1
    e2 = E2
in
case x of
   True  -> e2
   False -> e1
```

Since there is now only one occurrence of e1 and e2 we can inline them, giving just what we hoped for:

```
case x of {True  -> E2; False -> E1}
```

The point is that the local definitions will often disappear again.

4.3.1 How join points occur

But what if they don't disappear? Then the definitions `s1` ... `sm` play the role of "join points"; they represent the places where execution joins up again, having forked at the `case x`. The "calls" to the `si` should really be just jumps. To see this more clearly consider the expression

```
if (x || y) then E1 else E2
```

A C compiler will "short-circuit" the evaluation of the condition if `x` turns out to be true generate code, something like this:

```
     if (x) goto l1;
     if (y) {...code for E2...}
l1:  ...code for E1...
```

In our setting, here's what will happen. First we desguar the conditional, and inline the definition of `||`:

```
case (case x of {True -> True; False -> y}) of
  True -> E1
  False -> E2
```

Now apply the case-of-case transformation:

```
let e1 = E1
    e2 = E2
```

```
in
case x of
   True  -> case True of {True -> e1; False -> e2}
   False -> case y    of {True -> e1; False -> e2}
```

Unlike the `not` example, only one of the two inner case simplifies, and we can therefore only inline `e2`, because `e1` is still mentioned twice[2]

```
let e1 = E1
in
case x of
   True  -> e1
   False -> case y of {True -> e1; False -> e2}
```

The code generator produces essentially the same code as the C code given above. The binding for `e1` turns into just a label, which is jumped to from the two occurrences of `e1`.

4.3.2 Case of error

The case-of-error transformation is often exposed by the case-of-case transformation. Consider

```
case (hd xs) of
      True  -> E1
      False -> E2
```

After inlining `hd`, we get

```
case (case xs of [] -> error "hd"; (x:_) -> x) of
      True  -> E1
      False -> E2
```

(I've omitted the type argument of `error` to save clutter.) Now doing case-of-case gives

```
let e1 = E1
    e2 = E2
in
case xs of
      []    -> case (error "hd") of { True -> e1; False -> e2 }
```

[2]Unless the inlining strategy decides that `E1` is small enough to duplicate; it is used in separate `case` branches so there's no concern about duplicating work. Here's another example of the way in which we make one part of the simplifier (the inlining strategy) help with the work of another (case-expression simplification.

```
        (x:_) -> case x             of { True -> e1; False -> e2 }
```

Now the case-of-error transformation springs to life, after which we can inline `e1` and `e2`:

```
case xs of
      []    -> error "hd"
      (x:_) -> case x of {True -> E1; False -> E2}
```

4.4 Nested conditionals combined

Sometimes programmers write something which should be done by a single `case` as a sequence of tests:

```
if x==0::Int then E0 else
if x==1      then E1 else
E2
```

After eliminating some redundant evals and doing the case-of-case transformation we get

```
case x of I# x# ->
case x# of
  0#    -> E0
  other -> case x# of
             1#    -> E1
             other -> E2
```

The case-merging transformation puts these together to get

```
case x of I# x# ->
case x# of
  0#    -> E0
  1#    -> E1
  other -> E2
```

Sometimes the sequence of tests cannot be eliminated from the source code because of overloading:

```
f :: Num a => a -> Bool
f 0 = True
f 3 = True
f n = False
```

If we specialise `f` to `Int` we'll get the previous example again.

4.5 Error tests eliminated

The elimination of redundant alternatives, and then of redundant cases, arises when we inline functions which do error checking. A typical example is this:

```
if (x `rem` y) == 0 then (x `div` y) else y
```

Here, both `rem` and `div` do an error-check for `y` being zero. The second check is eliminated by the transformations. After transformation the code becomes:

```
case x of I# x# ->
case y of I# y# ->
case y of
  0# -> error "rem: zero divisor"
  _  -> case x# rem# y# of
          0# -> case x# div# y# of
                  r# -> I# r#
          _  -> y
```

4.6 Atomic arguments

At this point it is possible to appreciate the usefulness of the Core-language syntax requirement that arguments are atomic. For example, suppose that arguments could be arbitrary expressions. Here is a possible transformation:

```
  f (case x of (p,q) -> p)
===>                                  f strict in its second argument
  case x of (p,q) -> f (p,p)
```

Doing this transformation would be useful, because now the argument to `f` is a simple variable rather than a thunk. However, if arguments are atomic, this transformation becomes just a special case of floating a `case` out of a strict `let`:

```
  let a = case x of (p,q) -> p
  in f a
===>                                  (f a) strict in a
  case x of (p,q) -> let a=p in f a
===>
  case x of (p,q) -> f p
```

There are many examples of this kind. For almost any transformation involving `let` there is a corresponding one involving a function argument. The same effect is achieved with much less complexity by restricting function arguments to be atomic.

5 Design

Dependency analysis Occurrence analysis

5.1 Renaming and cloning

Every program-transformation system has to worry about name capture. For example, here is an erroneous transformation:

```
   let y = E
   in
   (\x -> \y -> x + y) (y+3)
==>                                     WRONG!
   let y = E
   in
   (\y -> (y+3) + y)
```

The transformation fails because the originally free-occurrence of `y` in the argument `y+3` has been "captured" by the
`y`-abstraction. There are various sophisticated solutions to this difficulty, but we adopted a very simple one: we uniquely rename every locally-bound identifier on every pass of the simplifier. Since we are in any case producing an entirely new program (rather than side-effecting an existing one) it costs very little extra to rename the identifiers as we go.

So our example would become

```
   let y = E
   in
   (\x -> \y -> x + y) (y+3)
==>                                     WRONG!
   let y1 = E
   in
   (\y2 -> (y1+3) + y2)
```

The simplifier accepts as input a program which has arbitrary bound variable names, including "shadowing" (where a binding hides an outer binding for the same identifier), but it produces a program in which every bound identifier has a distinct name.

Both the “old” and “new” identifiers have type `Id`, but when writing type signatures for functions in the simplifier we use the types `InId`, for identifiers from the input program, and `OutId` for identifiers from the output program:

```
type InId  = Id
type OutId = Id
```

This nomenclature extends naturally to expressions: a value of type `InExpr` is an expression whose identifiers are from the input-program name-space, and similarly `OutExpr`.

6 The simplifier

The basic algorithm followed by the simplifier is:

1. Analyse: perform occurrence analysis and dependency analysis.
2. Simplify: apply as many transformations as possible.
3. Iterate: perform the above two steps repeatedly until no further transformations are possible. (A compiler flag allows the programmer to bound the maximum number of iterations.)

We make a effort to apply as many transformations as possible in Step 2. To see why this is a good idea, just consider a sequence of transformations in which each transformation enables the next. If each iteration of Step 2 only performs one transformation, then the entire program will to be re-analysed by Step 1, and re-traversed by Step 2, for each transformation of the sequence. Sometimes this is unavoidable, but it is often possible to perform a sequence of transformtions in a single pass.

The key function, which simplifies expressions, has the following type:

```
simplExpr :: SimplEnv
          -> InExpr -> [OutArg]
          -> SmplM OutExpr
```

The monad, `SmplM` can quickly be disposed of. It has only two purposes:

- It plumbs around a supply of unique names, so that the simplifier can easily invent new names.

- It gathers together counts of how many of each kind of transformation has been applied, for statistical purposes. These counts are also used in Step 3 to decide when the simplification process has terminated.

The signature can be understood like this:

- The environment, of type `SimplEnv`, provides information about identifiers bound by the enclosing context.
- The second and third arguments together specify the expression to be simplified.
- The result is the simplified expression, wrapped up by the monad.

The simplifier's invariant is this:

$$\texttt{simplExpr}\ \textit{env}\ \textit{expr}\ [a_1, \ldots, a_n] = \textit{expr}[\textit{env}]\ a_1\ \ldots\ a_n$$

That is, the expression returned by `simplExpr` *env expr* $[a_1, \ldots, a_n]$ is semantically equal (although hopefully more efficient than) *expr*, with the renamings in *env* applied to it, applied to the arguments $a_1, \ldots, a_n$.

6.1 Application and beta reduction

The arguments are carried "inwards" by `simplExpr`, as an accumulating parameter. This is a convenient way of implementing the transformations which float arguments inside a `let` and `case`. This list of pending arguments requires a new data type, `CoreArg`, along with its "in" and "out" synonyms, because an argument might be a type or an atom:

```
data CoreArg bindee = TypeArg UniType
                    | ValArg  (CoreAtom bindee)

type InArg  = CoreArg InId
type OutArg = CoreArg OutId
```

The equations for applications simply apply the environment to the argument (to handle renaming) and put the result on the argument stack, tagged to say whether it is a type argument or value argument:

```
simplExpr env (CoApp fun arg)  args
  = simplExpr env fun (ValArg  (simplAtom env arg) : args)
simplExpr env (CoTyApp fun ty) args
  = simplExpr env fun (TypeArg (simplTy env ty)    : args)
```