Statically Unrolling Recursion to Improve Opportunities for Parallelism

Neil Deshpande Stephen A. Edwards
Department of Computer Science, Columbia University, New York
Technical Report CUCS-011-12

July 2012*

Abstract

We present an algorithm for unrolling recursion in the Haskell functional language. Adapted from a similar algorithm proposed by Rugina and Rinard for imperative languages, it essentially inlines a function in itself as many times as requested. This algorithm aims to increase the available parallelism in recursive functions, with an eye toward its eventual application in a Haskell-to-hardware compiler. We first illustrate the technique on a series of examples, then describe the algorithm, and finally show its Haskell source, which operates as a plug-in for the Glasgow Haskell Compiler.

1 Introduction

Ongoing work at Columbia by Stephen A. Edwards and Martha A. Kim aims to produce a compiler capable of compiling functional code, such as programs written in the Haskell language [8]. For this compiler, exposing and exploiting parallelism will be crucial for producing high-performance circuits.

Recursive functions are a particular focus of this compiler since its goal is to enable the implementation of software-like algorithm, such as those employing recursive, abstract data types, and recursive algorithms are a natural fit for manipulating such data structures.

Our plan for implementing unbounded recursion in hardware relies on storing activation records in stack memory, an approach mimicking how it is usually implemented on sequential processors[1]. While this is conceptually simple, it is inherently sequential since only one call of the function can make progress at any point in time.

*This document's unusual page size is intended to aid on-screen viewing and printing two pages per sheet. Furthermore, it is written as literate Haskell (an .lhs file); its source can be compiled both with LaTeX and the Haskell compiler; see Section 5.2.

[1]Ghica, Smith, and Singh [7] present an alternative but comparable approach that replaces each register in a recursive function with a stack of such registers.

The goal of the work in this paper is to improve the available parallelism of recursive functions by "widening" them into the equivalent of multiple recursive calls. We closely follow Rugina and Rinard [12], who implemented a similar algorithm for imperative code (in C). Their objectives were a little different since they assumed execution on a sequential processor: they aimed to reduce control overhead (e.g., stack management) by increasing the size of functions, which also facilitated traditional compiler optimizations such as register allocation.

In our setting, unrolling recursion provides two advantages: it can increase the number of operations, ranging from simple addition to external (non-recursive) function calls, that can be executed in parallel per recursive invocation; and, if a function makes multiple recursive calls (i.e., the unrolled call graph is a tree), unrolling the top function makes it possible to execute more subtrees in parallel.

Our basic algorithm amounts to function inlining, a standard technique in compilers for functional languages [11], with a refinement that avoids unhelpful duplication of local helper functions (see Section 2.2).

We implement this algorithm as a plugin for the Glasgow Haskell Compiler (GHC). As such, it restructures the GHC "Core" functional IR and allows us to run benchmarks to identify gains resulting from this transformation.

We include the complete source code of our implementation in Section 4 and describe how to compile and benchmark it in Section 5.

2 Examples

2.1 Inlining Recursive Functions

Figures 1–4 illustrate our procedure for inlining recursive functions to increase the available parallelism. The fib function implements a naïve, recursive algorithm to calculate the *n*th Fibonacci number. While this inefficient algorithm would never be used to compute Fibonacci numbers, its structure is representative of other, more realistic functions, so we consider it here.

Our procedure operates in three steps. We begin by making a copy of the fib function and making each call the other. This produces the mutually recursive functions in Figure 2. Next, we inline the body of the second function, a lambda expression, at its two call sites and alpha-rename the arguments for clarity. This produces Figure 3. Finally, we beta-reduce the two lambda expressions (formerly the two recursive calls), producing fib4 (Figure 4).

Similarly, Figures 5–8 illustrate our algorithm applied to a Huffman decoder, which uses a Huffman tree to transform a list of Booleans to a list of characters.

As before, we create two mutually recursive functions, dec and dec2, that have similar bodies but call each other instead of recursing on themselves (Figure 6). Inlining dec2 gives us dec3, which once more recurses on itself (Figure 7). Finally, we beta reduce the lambda expressions in the body of dec3 to get dec4 as shown in Figure 8.

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Figure 1: A recursive function for calculating the *n*th Fibonacci number, which we will use to illustrate our algorithm.

```
fib2 0 = 1
fib2 1 = 1
fib2 n = fib2' (n-1) + fib2' (n-2)

fib2' 0 = 1
fib2' 1 = 1
fib2' n = fib2 (n-1) + fib2 (n-2)
```

Figure 2: The first step: a mutually recursive variant obtained by duplicating the fib function and redirecting the recursive calls.

```
fib3 0 = 1
fib3 1 = 1
fib3 n = ((λn1 →
            case n1 of
            0 → 1
            1 → 1
            _ → fib3 (n1-1) +
                fib3 (n1-2)
          ) (n-1)) +
          ((λn2 →
            case n2 of
            0 → 1
            1 → 1
            _ → fib3 (n2-1) +
                fib3 (n2-2)
           ) (n-2))
```

Figure 3: After inlining the two calls of fib2 and alpha-renaming the arguments.

```
fib4 0 = 1
fib4 1 = 1
fib4 n = (case (n-1) of
            0 → 1
            1 → 1
            _ → fib4 ((n-1)-1) +
                fib4 ((n-1)-2)) +
          (case (n-2) of
            0 → 1
            1 → 1
            _ → fib4 ((n-2)-1) +
                fib4 ((n-2)-2))
```

Figure 4: After beta-reducing the two lambda terms in Figure 3.

```
data Htree = Leaf Char
           | Branch Htree Htree

decoder tree i = dec tree i where
  dec state i =
    case state of
      Leaf x      → x : (dec tree i)
      Branch t f →
        case i of
          []          → []
          True  : ys → dec t ys
          False : ys → dec f ys
```

Figure 5: A simple Huffman decoder

```
decoder tree i = dec tree i where
  dec state i =
    case state of
      Leaf x      → x : (dec2 tree i)
      Branch t f → case i of
          []          → []
          True  : ys → dec2 t ys
          False : ys → dec2 f ys
  dec2 state i =
    case state of
      Leaf x      → x : (dec tree i)
      Branch t f →
       case i of
          []          → []
          True  : ys → dec t ys
          False : ys → dec f ys
```

Figure 6: After expanding dec to mutually recursive functions

2.2 *Lifting local helper functions*

If the function being inlined has any local functions, it would be natural to make a separate copy every time the function is inlined. For example, consider the version of Quicksort presented in Figure 9, where *part* is a local function. Figure 10 shows a single, naïve unrolling of the *qsort* function produces three copies of the *part* function.

Such copies are redundant since they would be invoked sequentially anyway and do not reduce overhead such as from function calls, so we would like to avoid them.

To avoid such redundancy, we want to lift the local function out of the recursive function being inlined. In addition, such lifting can expose opportunities for other optimizations. For example, for Quicksort, lifting *part* out of its scope makes it easier to run our unrolling algorithm in it as well.

Lifting a function out of the scope in which it is defined can be done using Johnsson's lambda lifting [10], which typically adds additional arguments for variables that are otherwise captured by the lambda expression (e.g., not arguments), although this is not necessary in the case of *part*. Figure 11 shows the result of lifting out *part*, which, when unrolled, gives Figure 12, which has not duplicated the body of *part*.

```
decoder tree i = dec3 tree i where
  dec3 state i = case state of
    Leaf x → x : ((λstate1 i1 →
     case state1 of
       Leaf x1 → x1 : (dec3 tree i1)
       Branch t1 f1 → case i1 of
         []          → []
         True : ys1  → dec3 t1 ys1
         False : ys1 → dec3 f1 ys1
     ) tree i)
    Branch t f → case i of
      []          → []
      True : ys   → (λstate2 i2 →
        case state2 of
          Leaf x2 → x2:(dec3 tree i2)
          Branch t2 f2 → case i2 of
            []          → []
            True : ys2  → dec3 t2 ys2
            False : ys2 → dec3 f2 ys2
        ) t ys
      False : ys →
        (λstate3 i3 → case state3 of
          Leaf x3 → x3:(dec3 tree i3)
          Branch t3 f3 → case i3 of
            []          → []
            True : ys3  → dec3 t3 ys3
            False : ys3 → dec3 f3 ys3
        ) f ys
```

Figure 7: dec2 inlined, which effectively makes three copies of it

```
decoder tree i = dec4 tree i where
  dec4 state i = case state of
    Leaf x → x : (case tree of
      Leaf x1      →
        x1 : (dec4 tree i)
      Branch t1 f1 → case i of
        []          → []
        True : ys1  → dec4 t1 ys1
        False : ys1 → dec4 f1 ys1)

    Branch t f → case i of
      []          → []
      True : ys   → case t of
        Leaf x2      →
          x2 : (dec4 tree ys)
        Branch t2 f2 → case ys of
          []          → []
          True : ys2  → dec4 t2 ys2
          False : ys2 → dec4 f2 ys2

      False : ys → case f of
        Leaf x3      →
          x3 : (dec4 tree ys)
        Branch t3 f3 → case ys of
          []          → []
          True : ys3  → dec4 t3 ys3
          False : ys3 → dec4 f3 ys3
```

Figure 8: After beta-reducing the lambda expressions in Figure 7. Each invocation of dec4 processes now processes two incoming bits rather than one.

```
qsort []     = []
qsort (y:ys) = let (ll, rr) = part y ys [] [] in
               (qsort ll) ++[y] ++(qsort rr)
  where part p [] l r     = (l, r)
        part p (x:xs) l r = if x <p then part p xs (x:l) r
                            else         part p xs l     (x:r)
```

Figure 9: Quicksort implemented with a local partition function

```
qsort []     → []
qsort (y:ys) → let (ll, rr) = part y ys [] [] in
    case ll of
      []       → []
      (y1:ys1) → let (ll1, rr1) = part1 y1 ys1 [] []
        in (qsort ll1) ++[y1] ++(qsort rr1)
          where part1 p1 []       l1 r1 = (l1, r1)
                part1 p1 (x1:xs1) l1 r1 =
                  if x1 <p1 then part1 p1 xs1 (x1:l1) r1
                  else           part1 p1 xs1 l1     (x1:r1)
    ++[y] ++
    case rr of
      []       → []
      (y2:ys2) → let (ll2, rr2) = part2 y2 ys2 [] []
        in (qsort ll2) ++[y2] ++(qsort rr2)
          where part2 p2 []       l2 r2 = (l2, r2)
                part2 p2 (x2:xs2) l2 r2 =
                  if x2 <p2 then part2 p2 xs2 (x2:l2) r2
                  else           part2 p2 xs2 l2     (x2:r2)
  where part p []     l r = (l,r)
        part p (x:xs) l r = if x <p then part p xs (x:l) r
                            else         part p xs l     (x:r)
```

Figure 10: Quicksort after one iteration of naïve unrolling, which made three duplicates of the *part* function.

```
part p []     l r = (l, r)
part p (x:xs) l r = if x < p then part p xs (x:l) r
                    else          part p xs l     (x:r)

qsort []     = []
qsort (y:ys) = let (ll, rr) = part y ys [] [] in
               (qsort ll) ++[y] ++(qsort rr)
```

Figure 11: Quicksort after lifting *part* outside the scope of *qsort*

```
part p z l r = case z of
  []     → (l, r)
  (x:xs) → if x < p then part p xs (x:l) r
           else          part p xs l     (x:r)

qsort w = case w of
  []     → []
  (y:ys) → let (ll, rr) = part y ys [] [] in
    case ll of
      []       → []
      (y1:ys1) → let (ll1, rr1) = part y1 ys1 [] []
                 in (qsort ll1) ++[y1] ++(qsort rr1)
    ++[y] ++
    case rr of
      []       → []
      (y2:ys2) → let (ll2, rr2) = part y2 ys2 [] []
                 in (qsort ll2) ++[y2] ++(qsort rr2)
```

Figure 12: The Quicksort of Figure 11 unrolled once: *part* was not duplicated

2.3 Case Transformations

The Fibonacci example is typical of many recursive functions: a top-level pattern match that distinguishes the base case followed by some expressions that do work and finally recursive calls (Figure 1). When we apply our inlining procedure, the structure becomes a pattern match followed by expressions followed by another pattern match and more expressions and finally a recursive call (Figure 4).

In many cases, it may be possible hoist the inner cases up past the first level of expressions to decrease the number of decisions to be made and potentially increase parallelism.

```
fib5 n =
  case (n, n−1, n−2) of
    (0, _, _) → 1
    (1, _, _) → 1
    (_, 0, 0) → 1 +1
    (_, 0, 1) → 1 +1
    (_, 0, _) → 1 + fib5 ((n−2)−1) +fib5 ((n−2)−2)
    (_, 1, 0) → 1 +1
    (_, 1, 1) → 1 +1
    (_, 1, _) → 1 + fib5 ((n−2)−1) +fib5 ((n−2)−2)
    (_, _, 0) → fib5 ((n−1)−1) +fib5 ((n−1)−2) +1
    (_, _, 1) → fib5 ((n−1)−1) +fib5 ((n−1)−2) +1
    (_, _, _) → fib5 ((n−1)−1) +fib5 ((n−1)−2) +
                fib5 ((n−2)−1) +fib5 ((n−2)−2)
```

Figure 13: Hoisting up the inner cases of Figure 4

```
fib6 0 =1
fib6 1 =1
fib6 2 =2
fib6 3 =3
fib6 n = fib6 (n−2) +2 ∗ fib6 (n−3) + fib6 (n−4)
```

Figure 14: After performing arithmetic and other simplifications on Figure 13

This requires the inner cases not depend on things computed by the first level of expressions and for these expressions never to diverge, although useful recursive functions rarely diverge anyway.

Such a transformation also improve the opportunities for more optimizations such as common subexpression elimination. Figures 13 and 14 illustrate how such optimizations could improve the generated code, going so far as to reduce the number of recursive calls and in this case, begin to illustrate the relationships among the Fibonacci sequence, Pascal's triangle, and the binomial theorem.

$$
\begin{array}{l}
\mathit{simplInline}(f,d) \\
\quad f = \lambda x_1.\lambda x_2 \ldots \lambda x_n.E_f \\
\quad E_{f^{(0)}} = \mathit{subst}(E_f)[f^{(0)}/f] \\
\quad f^{(0)} = \lambda x_1.\lambda x_2 \ldots \lambda x_n.E_{f^{(0)}} \\
\quad \text{for } i = 0,1,\ldots,d-1 \\
\qquad E_{f'} = \mathit{subst}(E_{f^{(i)}})[f'/f^{(i)}] \\
\qquad E_{f''} = \mathit{subst}(E_{f^{(i)}})[f''/f^{(i)}] \\
\qquad f' = \lambda x_1.\lambda x_2 \ldots \lambda x_n.E_{f''} \\
\qquad f'' = \lambda x_1.\lambda x_2 \ldots \lambda x_n.E_{f'} \\
\qquad f''' = \mathit{subst}\left(\lambda x_1.\lambda x_2 \ldots \lambda x_n.\left(\mathit{subst}(E_{f''})[(\lambda x_1.\lambda x_2 \ldots \lambda x_n.E_{f'})/f'']\right)\right)[f'''/f'] \\
\qquad f''' \rightarrow_\beta f'''' \rightarrow_\alpha f^{(i+1)} \\
\quad \mathit{return} \quad f^{(d)}
\end{array}
$$

Figure 15: Our inlining algorithm

3 A Simple Inlining Algorithm

We present our algorithm in Figure 15. Here, we argue for its correctness. Let g be a recursive function that contains no free variables and is not part of a group of mutually recursive functions. Our algorithm is correct even if the function is part of such a group, but we will not consider that case for now. We can represent g as

$$g = \lambda x_1.\lambda x_2.\ldots.\lambda x_n.E_g, \tag{1}$$

where E_g is some expression containing recursive calls to the function g. It follows that

$$g\ a_1\ a_2\ \cdots\ a_m \equiv (\lambda x_1.\lambda x_2.\ldots.\lambda x_n.E_g)\ a_1\ a_2\ \cdots\ a_m. \tag{2}$$

Let $\mathit{subst}\ (M)\ [E/x]$ represent the result of substituting the literal E for the literal x in the expression M[2] and let

$$E_f = \mathit{subst}\ (E_g)\ [f/g] \quad \text{and} \quad f = \lambda x_1.\lambda x_2.\ldots.\lambda x_n.E_f.$$

Changing an application of g to an application of f with the *same* arguments is meaning preserving. That is,

$$g \quad a_1\ a_2\ \cdots\ a_m \quad \equiv \quad f \quad a_1\ a_2\ \ldots\ a_m.$$

Thus, renaming a function is safe as long as we rename all the recursive calls within its body consistently and the new name is unique. Now, let

$$E_{f'} = \mathit{subst}(E_g)[f'/g] \quad \text{and} \quad E_{f''} = \mathit{subst}(E_g)[f''/g],$$

$$\text{Also, let} \quad f' = \lambda x_1.\lambda x_2.\ldots.\lambda x_n.E_{f''} \quad \text{and} \quad f'' = \lambda x_1.\lambda x_2.\ldots.\lambda x_n.E_{f'}$$

[2]Following, e.g., Peyton Jones and Marlow [11].

be two mutually recursive functions. Based on the argument given above, we claim that f', f'', and g are all equivalent since all that we have changed is the name bound to the lambda expression. Now, let

$$f''' = subst\left(\lambda x_1.\lambda x_2 \ldots \lambda x_n.\left(subst\left(E_{f''}\right)\left[\lambda x_1.\lambda x_2 \ldots \lambda x_n.E_{f'}/f''\right]\right)\right)\left[f'''/f'\right]. \quad (3)$$

That is, we derive f''' from f' by substituting the expression bound to f'' at its call sites within $E_{f''}$, followed by renaming. Thus, f''' is equivalent to g.

Finally, let $f''' \rightarrow_\beta f^{(1)}$, that is, $f^{(1)}$ is obtained from f''' by beta reducing the lambdas that were introduced in (3). Since beta reduction is safe, $f^{(1)}$ is equivalent to f''', and by transitivity, to g.

GHC uses unique integers to distinguish variables. Thus, when we inline the lambda expression according to (3), we need to rename all the bound variables. This is equivalent to an alpha conversion which is safe if the new names are unique. To ensure this, we use the *UniqSupply* utility provided by the GHC API. Furthermore, we assume f contains no free variables, ensuring the series of transformations outlined above are meaning preserving.

```
data Expr b
  = Var    Id
  | Lit    Literal
  | App    (Expr b) (Arg b)
  | Lam    b (Expr b)
  | Let    (Bind b) (Expr b)
  | Case   (Expr b) b Type [Alt b]
  | Cast   (Expr b) Coercion
  | Tick   (Tickish Id) (Expr b)
  | Type   Type
  | Coercion Coercion
  deriving (Data, Typeable)

-- Type synonym for expressions that occur in function argument positions.
-- Only Arg should contain a Type at top level, general Expr should not
type Arg b = Expr b
-- A case split alternative. Consists of the constructor leading to the alternative,
-- the variables bound from the constructor,
-- and the expression to be executed given that binding.
type Alt b = (AltCon, [b], Expr b)
-- Binding, used for top level bindings in a module and local bindings in a Let
data Bind b = NonRec b (Expr b)
            | Rec [(b, (Expr b))]
  deriving (Data, Typeable)

type CoreProgram = [CoreBind]

-- The common case for the type of binders and variables when
-- we are manipulating the Core language within GHC
type CoreBndr = Var
type CoreExpr = Expr CoreBndr -- Expressions where binders are CoreBndrs
type CoreArg  = Arg  CoreBndr -- Argument expressions where binders are CoreBndrs
type CoreBind = Bind CoreBndr -- Binding groups where binders are CoreBndrs
type CoreAlt  = Alt  CoreBndr -- Case alternatives where binders are CoreBndrs
```

Figure 16: The GHC Core language, adapted from CoreSyn.lhs [1]

4 Implementation

We implemented our algorithm as a plugin that transforms GHC's Core (Figure 16).

4.1 The SimplInline *plugin*

Our plugin is executed as one of the Core-to-Core passes after the desugaring phase. GHC exposes most of the functions in its source through the GhcPlugins package. Thus, we can use much of GHC's infrastructure.

Our module exports a single function, *plugin*, which GHC will pick up and run as one of the Core-to-Core passes. We extend *defaultPlugin* (which does nothing) by providing an implementation for the *installCoreToDos* function. The GHC User Guide [9] has a more thorough discussion of what each of these types mean. Our *install* function simply adds our custom *pass*, called *Recursion Inline*, to the list of passes executed by the GHC.

During the optimization phase, GHC invokes the *pass* function by passing an instance of *ModGuts* to it. The *ModGuts* type represents a module being compiled by the GHC [9]. The *pass* function retrieves all the top level bindings from the *ModGuts* and runs the *unrollTop* function on each binding. *unrollTop* is a simple tail recursive function that calls $unroll^{(i)}$ (i.e. the *i*th iteration of *unroll*) on the binding that has been passed to it. *unroll* uses the *recHelper* function to recurse down the SAST that has been passed to it until it finds a simple recursive binding. Once a simple recursive binding has been found, it calls *mkPartner* to create a pair of mutually recursive functions. Then, *betaReduce* and other helper functions are used to inline one of the two mutually recursive functions into the other. Alpha renaming is carried out using *renameLocals* during the inlining phase to avoid name collisions and to ensure that the transformation is meaning preserving. Finally, *pass* returns the transformed instance of *ModGuts* to the GHC for running other passes from the pipeline.

```
module SimplInline (plugin) where
import GhcPlugins
import Unique
import Debug.Trace
import System.IO.Unsafe

plugin :: Plugin
plugin = defaultPlugin {
  installCoreToDos = install
}

install :: [CommandLineOption] →[CoreToDo] →CoreM [CoreToDo]
install _ todo =do
    reinitializeGlobals
  return (CoreDoPluginPass "Recursion␣Inline" pass : todo)
```

The function below, *pass*, prints debugging information and generates the arguments for the top level call to *unrollTop*. We use the *UniqSupply* module for generating *Unique*'s, which we use to create unique names for new local variables. GHC uses this mechanism to disambiguate variables. The *mkSplitUniqSupply* function returns a tree of *UniqSupply* values. Each *UniqSupply* can give us one *Unique* value. It can also be split to give us two distinct *UniqSupply* values. The *mkSplitUniqSupply* function takes a single character as a seed. We chose to use 'Z' as the seed since it is currently one of the characters that GHC itself does not use as a seed for its own *UniqSupply* values.

```
pass :: ModGuts →CoreM ModGuts
pass guts = let binds = mg_binds guts
                us    = unsafePerformIO $ mkSplitUniqSupply 'Z'
  in let binds' = altHelperU us f unrollTop binds []
     in return (guts { mg_binds =binds' })
        where
          f u g b = let {
          b' =g u 1 $
             trace ("Received␣SAST      ::::::::::::::::::::::::::::::      " ++sh b) b;
           }
             in trace ("New␣SAST      ::::::::::::::::::::::::::::::      " ++sh b') b'

unrollTop :: UniqSupply → Int → Bind CoreBndr → Bind CoreBndr
unrollTop us i bndr = if i ==0 then bndr
  else let (us1, us2) = splitUniqSupply us
    in unrollTop us1 (i−1) (unroll us2 bndr)
```

unroll takes a binding and calls *mkPartner* to generate a pair of mutually recursive functions as described in section 3. Finally, it calls *betaReduce* to actually inline *e* into *e'* at the call sites for *b'*. Since only function applications are reduced by *betaReduce*, we use *substitute* to replace all other occurences of *b'* with *b* within *e'*. Future work: rather than indiscriminately inlining the first recursive function that we encounter, we should use annotations or a predicate that tells us which functions should be inlined.

```
unroll :: UniqSupply → Bind CoreBndr → Bind CoreBndr
unroll us (NonRec b e) =NonRec b $ recHelper us e
unroll us (Rec [bndr]) = let (us1, us2) = splitUniqSupply us
  in let [(b, e'), (b', e)] = mkPartner us1 bndr
     in Rec [(b, substitute varToCoreExpr b' b $ betaReduce b' e us2 e')]

−− We do not handle mutually recursive groups currently
unroll us m              =m
```

recHelper descends to the first simple recursive function in the given SAST and calls *unroll* on it.

```
recHelper :: UniqSupply → Expr CoreBndr → Expr CoreBndr
recHelper us e = let (us1, us2) = splitUniqSupply us
  in case e of
    Let bnd expr          → Let (unroll us1 bnd) (recHelper us2 expr)
    Lam var expr          → Lam var (recHelper us expr)
    App expr arg          → App (recHelper us1 expr) (recHelper us2 arg)
    Case expr bnd t alts  → Case (recHelper us1 expr) bnd t $
                              altHelperU us2 altTransformU recHelper alts []
    Cast expr co          → Cast (recHelper us expr) co
    Tick id expr          → Tick id (recHelper us expr)
    x                     → x
```

mkPartner takes a simple recursive function and generates two mutually recursive functions as described in section 3. This function calls *renameLocals* to ensure that the local variables in the two mutually recursive functions appear as distinct variables to GHC.

```
mkPartner :: UniqSupply → (CoreBndr, Expr CoreBndr)
                        → [(CoreBndr, Expr CoreBndr)]
mkPartner us (b, e) = let (us1, us2) = splitUniqSupply us
  in let b' = mkPBnd us1 b
    in let e' = renameLocals us2 $ substitute varToCoreExpr b b' e
      in [(b, e'), (b', e)]
```

mkPBnd creates a *Unique* from the *UniqSupply* that it receives and creates a local variable having the same type as the binding passed to it by calling *mkSysLocal*, which is exposed by GHC. Currently, an unfortunate side effect is that wild card variables lose their wild card behavior, since we don't play around with the *IdInfo*. We need to fix this so further passes of the compiler have more opportunities for optimization.

```
mkPBnd :: UniqSupply → CoreBndr → CoreBndr
mkPBnd us var = let uniq = uniqFromSupply us
  in mkSysLocal (fsLit "r2d2") uniq (varType var)
```

These are a bunch of utility functions used to iterate over lists etc.

```
altHelperU us f g (alt : alts) res = let (us1, us2) = splitUniqSupply us
  in altHelperU us2 f g alts $ (f us1 g alt) : res
altHelperU us f g [] res           = reverse res

altTransformU us f (altCon, bnds, ex) = (altCon, bnds, f us ex)
```

```
altHelper f alts = map (λ(altCon, bnds, ex) → (altCon, bnds, f ex)) alts

isVarBound u bnd = case bnd of
  NonRec v expr          → u ==v
  Rec [(v, expr)]        → u ==v
  Rec ((v, expr):bnds)   → u ==v || isVarBound u (Rec bnds)

sh x = showSDoc $ ppr x
```

substitute takes a function *f*, a variable to replaced *b*, an expression *e* bound to *b* and the tree (SAST) in which the replacement is supposed to take place *e'*. It recursively descends the tree and replaces all occurences of *b* by the result of *f e*, except when the name *b* is bound to a lambda within *e'*.

```
substitute f b e e' = case e' of
  var@(Var id)                → if id ==b then f e else var
  App expr arg                → App (substitute f b e expr)
                                    (substitute f b e arg)
  lam@(Lam bnd expr)          → if b ==bnd then lam
    else Lam bnd (substitute f b e expr)
  lett@(Let bnd expr)         → if isVarBound b bnd then lett
    else Let (bndHelper f b e bnd) (substitute f b e expr)
  cas@(Case expr bnd t alts ) → if bnd ==b then cas
    else Case (substitute f b e expr) bnd t (altHelper (substitute f b e) alts)
  Tick t expr                 → Tick t (substitute f b e expr)
  Cast expr co                → Cast (substitute f b e expr) co
  x                           → x
```

substituteU performs the same task as *substitute*, but we use this version when the function *f* needs a *UniqSupply* argument in addition to *e*.

```
substituteU f b e us e' = let (us1, us2) = splitUniqSupply us
  in case e' of
    var@(Var id)                  → if id ==b then f us e else var
    App expr arg                  → App (substituteU f b e us1 expr)
                                        (substituteU f b e us2 arg)
    lam@(Lam bnd expr)            → if b ==bnd then lam
      else Lam bnd (substituteU f b e us expr)
    lett@(Let bnd expr)           → if isVarBound b bnd then lett
      else Let (bndHelperU us1 f b e bnd) (substituteU f b e us2 expr)
    cas@(Case expr bnd t alts )   → if bnd ==b then cas
      else Case (substituteU f b e us1 expr) bnd t $
             altHelperU us2 altTransformU (substituteU f b e) alts []
    Tick t expr                   → Tick t (substituteU f b e us expr)
    Cast expr co                  → Cast (substituteU f b e us expr) co
    x                             → x
```

renameLocals takes a *UniqSupply* (*us*) and an SAST *e* and recursively descends the SAST, renaming all the local variables within the tree and substituting all occurences of the old names with the corresponding newly generated names.

```
renameLocals us e = let (us1, us2) = splitUniqSupply us
  in case e of
    App expr (Type t)     → App (renameLocals us expr) (Type t)
    App expr arg          → App (renameLocals us1 expr) (renameLocals us2 arg)
    Lam var expr          → lamHelper us var expr
    Let bnd expr          → letHelper us bnd expr
    Case expr var t alts  → let var'          = mkPBnd us1 var
                                (us3, us4) = splitUniqSupply us2
        in let alts' = altHelper (substitute varToCoreExpr var var') alts
          in Case (renameLocals us3 expr) var' t $
               altHelperU us4 renAltU id alts' []
    Tick t expr           → Tick t (renameLocals us expr)
    Cast expr co          → Cast (renameLocals us expr) co
    x                     → x
```

betaReduce takes a variable *b* bound to a function *e* and the SAST to be modified which is *e'*. It replaces all applications of *b* in the SAST *e'* by *e*. Furthermore, it beta-reduces all the applications of *e* with the arguments specified in the SAST. This is slightly different from classical beta reduction which takes a single lambda and reduces it using a single

argument. This method supports reducing an arbitrary number of arguments as well as curried functions.

```
betaReduce :: CoreBndr → Expr CoreBndr → UniqSupply → Expr CoreBndr
  → Expr CoreBndr
betaReduce b e@(Lam v ex) us e' = let (us1, us2) = splitUniqSupply us
  in case e' of
    App (Var var) arg → let arg' = betaReduce b e us1 arg
      in if var ==b then renameLocals us2 $ substitute id v arg' ex
        else App (Var var) arg'
    App app@(App expr a) arg → let arg' = betaReduce b e us1 arg
      in case betaReduce b e us2 app of
        Lam v' ex' → substitute id v' arg' ex'
        app' → App app' arg'
    App expr arg → App (betaReduce b e us1 expr) (betaReduce b e us2 arg)
    lam@(Lam bnd expr) → if b ==bnd then lam
      else Lam bnd (betaReduce b e us expr)
    lett@(Let bnd expr) → if isVarBound b bnd then lett
      else let expr' = betaReduce b e us1 expr
        in case bnd of
          NonRec vr expre → Let (NonRec vr (betaReduce b e us expre)) expr'
          Rec bnds → Let (Rec $
            altHelperU us (substBndU v) (betaReduce b e) bnds []
            ) expr'
    cas@(Case expr bnd t alts) → if bnd ==b then cas
      else Case (betaReduce b e us1 expr) bnd t $
        altHelperU us2 altTransformU (betaReduce b e) alts []
    Tick t expr → Tick t (betaReduce b e us expr)
    Cast expr co → Cast (betaReduce b e us expr) co
    x → x

-- Don't beta reduce things that are not functions
-- instead just go in and inline them
betaReduce b e us e' = substituteU renameLocals b e us e'

substBndU v us g (var, expr) = if var ≠ v then (var, g us expr)
  else (var, expr)
```

Finally, we have a bunch of helper functions for doing some heavy lifting:

lamHelper renames locals within a lambda expression. We rename the bound variable of the lambda and substitute all occurences with of the bound variable with the new name.

This is alpha conversion. Finally, *renameLocals* is called recursively on the body of the new lambda. However, we have to be careful that we don't play with the type variables for type lambdas or with the bound variable of a dictionary.

```
lamHelper us var expr
  | isTyVar var      = Lam var $ renameLocals us expr
  | isDictId var     = Lam var $ renameLocals us expr
  | otherwise        = let (us1, us2) = splitUniqSupply us
    in let var' = mkPBnd us1 var
      in Lam var' $ renameLocals us2 $ substitute varToCoreExpr var var' expr
```

```
varRenameU us _ var = mkPBnd us var
```

renAltU renames locals within a case alternative. This entails renaming all the locals bound from the constructor used for the pattern match (*altCon*) and substituting all occurences with the new names. Finally, we recurse on the body of case alternative.

```
renAltU us _ (altCon, vars, expr) = let (us1, us2) = splitUniqSupply us
  in let nvars = altHelperU us1 varRenameU id vars []
    in let subs          = zip vars nvars
           f x (b, b') = substitute varToCoreExpr b b' x
       in (altCon, nvars, renameLocals us2 $ foldl f expr subs)
```

letHelper renames the locals within a *let* expression defined by a binding *bnd* and an expression *expr*.

```
letHelper us bnd expr = let (us1, us2) = splitUniqSupply us
  in case bnd of
    NonRec v e → let v' = mkPBnd us1 v
      in let (us3, us4) = splitUniqSupply us2
        in Let (NonRec v' $ renameLocals us3 e)
          (renameLocals us4 $ substitute varToCoreExpr v v' expr)
    Rec bnds   → let vars  = map fst bnds
                     exprs = map snd bnds
      in let nvars = altHelperU us1 varRenameU id vars []
        in let subs          = zip vars nvars
               f x (v, v') = substitute varToCoreExpr v v' x
          in let exprs'      = map (λx → foldl f x subs) exprs
                 (us3, us4) = splitUniqSupply us2
            in let bnd' = Rec $ zip nvars $
                              altHelperU us3 (λu g e→ g u e) renameLocals exprs' []
              in Let bnd' $ renameLocals us4 $ foldl f expr subs
```

bndHelper replaces *v* by *f e* in *bnd* if *v* has not already been name captured within *bnd*.

```
bndHelper f v e bnd = case bnd of
  NonRec var expr → if var ≠ v then NonRec var $ substitute f v e expr
                    else NonRec var expr
  Rec bnds        → Rec $ map (λ(var, expr) → if var ≠ v
                                then (var, substitute f v e expr)
                                else (var, expr)) bnds

bndHelperU us f v e bnd = case bnd of
  NonRec var expr → NonRec var $ substituteU f v e us expr
  Rec bnds        → Rec $ altHelperU us (substBndU v) (substituteU f v e) bnds []
```

5 Performance Analysis

We compiled and installed the *SimplInline* plugin and ran it on the *nofib* benchmark to measure its performance. The run was conducted on a Lenovo ThinkPad E420 running Ubuntu 11.10. The plugin requires the GHC infrastructure in order to build and deploy it.

5.1 Getting the ghc-7.4.1 infrastructure

The GHC source is required in order to run the *nofib* benchmark, since the benchmark uses some of the build infrastructure of the GHC source. We need to install the haskell platform before we can build the GHC. That is, we need ghc to build the ghc source. This can be done as follows:

1. Install a ghc binary (version 6 or higher) either from the GHC download site [2] or by running (on Ubuntu):

   ```
   sudo apt-get install ghc
   ```

2. Get the ghc-7.4.1 source from the GHC download site [2]

3. Build ghc-7.4.1 from source by the running following sequence of commands in the unpacked directory [3]:

   ```
   ./configure
   make
   make install
   ```

4. Make the newly built ghc-7.4.1 binary the default ghc on your PATH.

5. Get the haskell-platform-2011.4.0.0 source from the GHC download site [6].

6. Build the haskell platform from source by the running following sequence of commands in the unpacked directory [6]

```
./configure
make
make install
```

5.2 *Compiling and Installing the* SimplInline *plugin*

Compiling and installing this plugin requires ghc 7.4.1 and cabal 1.10 (or higher). The plugin can be compiled by the following command:

```
ghc -c SimplInline.lhs -package ghc
```

We use the cabal utility to install the *SimplInline* plugin. The SimplInline.cabal file can be found in Figure 17. The plugin is installed by executing the following command in the directory where SimplInline.lhs file has been compiled. The cabal file should also be present in the same directory.

```
cabal install
```

Once the plugin has been installed, we can use it to optimize programs by employing the *-fplugin=SimplInline* option with ghc. For instance, we can compile and run a program foo.hs with the *SimplInline* optimization by using the following sequence of commands:

```
ghc -c -fplugin=SimplInline foo.hs
ghc -o foo foo.hs
./foo
```

5.3 *Running the* nofib *benchmark*

1. Get the *nofib* source from the git repository [4] as a tarball
2. Unpack the *nofib* tarball under the root of the ghc source tree on the same level as compiler and libraries.
3. In ghc-7.4.1/mk/build.mk, set

   ```
   WithNofibHc =ghc
   ```

4. We are now set to run the benchmark. Run the following commands in the directory ghc-7.4.1/nofib [5]

   ```
   make clean && make boot && make -k >& vanillaLog
   make clean && make boot && make -k \
    EXTRA_HC_OPTS="-fplugin=SimplInline" >& simplInlineLog
   nofib-analyse/nofib-analyse vanillaLog simplInlineLog \
    >analysis.txt
   ```

The file `analysis.txt` now contains the results of the benchmark run.

```
-- The name of the package.
Name:                SimplInline

-- The package version. See the Haskell package versioning policy
-- standards guiding when and how versions should be incremented.
Version:             0.1

-- A short (one-line) description of the package.
Synopsis:            Simple Recursion Unrolling

-- URL for the project homepage or repository.
Homepage:            http://patch-tag.com/r/neil/simplInline

-- The license under which the package is released.
License:             BSD3

-- The file containing the license text.
License-file:        LICENSE

-- The package author(s).
Author:              Neil Deshpande

-- An email address to which users can send suggestions, bug reports,
-- and patches.
Maintainer:          neil.deshpa@gmail.com

Category:            System

Build-type:          Simple

-- Constraint on the version of Cabal needed to build this package.
Cabal-version:       >=1.10

Library
  -- Modules exported by the library.
  Exposed-modules:     SimplInline

  -- Packages needed in order to build this package.
  Build-depends:       base,
                       ghc
```

Figure 17: The cabal (package manager) specification for our plug-in

References

[1] http://hackage.haskell.org/trac/ghc/browser/compiler/coreSyn/CoreSyn.lhs.

[2] http://www.haskell.org/ghc/download_ghc_7_4_1.

[3] http://hackage.haskell.org/trac/ghc/wiki/Building/QuickStart.

[4] https://github.com/ghc/nofib.

[5] http://hackage.haskell.org/trac/ghc/wiki/Building/RunningNoFib.

[6] http://hackage.haskell.org/platform/linux.html.

[7] Dan R. Ghica, Alex Smith, and Satnam Singh. Geometry of synthesis IV: Compiling affine recursion into static hardware. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 221–233, Tokyo, Japan, September 2011.

[8] http://www.haskell.org/.

[9] http://www.haskell.org/ghc/docs/7.4.1/html/users_guide/compiler-plugins.html#writing-compiler-plugins.

[10] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proceedings of Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 190–203, Nancy, France, 1985. Springer.

[11] Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12:393–434, September 2002.

[12] Radu Rugina and Martin Rinard. Recursion unrolling for divide and conquer programs. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC)*, volume 2017 of *Lecture Notes in Computer Science*, pages 34–48, Yorktown Heights, New York, August 2000.