Supercompilation Experiment

Jarno Le Conté

Utrecht University

Abstract. Minimal reimplementation of Bolingbroke's supercompiler for call-by-need languages, making use of attribute grammars instead of monadic structures. Additionally we formalized some rules and explain the difficult parts on the basis of pictures.

1 Introduction

We implement a supercompiler from scratch for a call-by-need language following the algorithm provided by Bolingbroke [2], but we only focus on a minimal subset that only contains the required parts to make the supercompiler work. We implement most parts using the UU Attribute Grammar System [6]. Also we point out difficult parts we have encountered when looking at Bolingbroke's implementation. Furthermore we try to visualize the supercompilation process with the help of examples and figures.

We implement the supercompiler for an extended lambda calculus with data types and recursive let. Because Bolingbroke's algorithm uses Sestoft-style operational semantics as basis for evaluation, we will see that we stay close the the standard evaluation rules.

Important to note is that we not fully succeed making a compiler which compiles all programs correctly. But we give some examples of failing programs and try to point out which code is responsible for giving wrong results.

This paper have the following structure. In section 3 we first discuss the idea of supercompilation. In section 4 we give an overview of the implementation and discuss each component in detail. With this knowledge we will look in section 5 at some examples, and visualize the steps that are performed by the supercompiler. At each example we point out the parts that are difficult to understand and we try to indicate the implementation errors. We conclude in section 6.

2 Contributions

The main goal of this project is to implement a supercompiler as described in the literature by Bolingbroke. The experience by doing this have lead to the following contributions.

Main contributions

- We encounter some difficulties implementing Bolingbroke's approach. Therefore we point out these parts and explain it in more detail, using figures to visualize the compilation process.
- We focus on a minimal supercompiler, containing only the parts that are strictly necessary to correctly supercompile, thus leave out optimizations and not obtain work sharing, making the final supercompiler more easy to understand.
- Furthermore we use Attribute Grammars (AG) [6] as replacement for the monadic structure in order to have a stateful supercompiler. Now programming becomes easier because you only have to deal with attributes. They are even simpler to reason with, because we can use tree figures to visualize the flow.

Additional contributions

- Bolingbroke extend the operational semantics by Sestoft but do not specify how to deal with renaming and computing free variables. We try to make this explicit in the semantic rules.
- We created a single multi-step reducer, which fully reduce a state until no reduction could happen anymore. This differs from bolingbroke's implementation where it requires separate calls to *reduce* and *normalize* and where only one reduction step at the time happens.
- We make a pure functional implementation not making use of unsafe IO operations.

3 Supercompilatation

Supercompilation is the process of program transformation whereby the resulting program is optimized and should run faster. It combines together different optimization strategies that already exists such as partial evaluation, specialization and deforestation, which lead to an stronger optimization than when these strategies applied individually.

The idea of supercompilation is quite old, introduced by Turchin [5]. The research from last years brings these ideas to call-by-need languages as well. The approach by Mitchell [3] is more less ad-hoc, while Bolingbroke [2] came up with an implementation that closely follow the standard operational semantics for evaluation and also allow recursive let bindings. Bolingbroke showed that supercompilation is an abstraction that could be used for different lazy languages, as long you make necessary changes to adopt the operational semantics.

For our experiment we use a extended untyped lambda calculus, including *variables*, lambda *abstractions*, function *applications*, *recursive let*, datatype *constructors* and pattern matches through *case expressions*. Let us first take a look at two examples to become familiar with the language and see which kind of optimization can be reached with this supercompiler.

```
let                                          let
  inc = λx → x + 1                             h0 xs =
  map = λf xs →                                  case xs of
    case xs of                                     [] → []
      [] → []                          ⇝           (y : ys) → (y + 1) : h0 ys
      (y : ys) → f y : map f ys                in
in                                               h0 zs
  map inc zs
```

On the left you see the original program, mapping the function *inc* over all numbers in the list *zs*, which increments each item by one. You can see in the supercompiled version on the right the evaluation proceeds by specializing the call to *map* on its arguments. Then it will evaluate both branches despite of the fact *xs* is unknown yet. Then it will also specialize the function *inc* by inlining it at *f y*. Now the supercompiler will also proceed on the tail of the list despite of the fact the head isn't a value yet. When supercompiling the tail it encounters a expression that have the same pattern as the input program and therefore tiebacks on the term just supercompiled, which is bounded to *h0*. The result is a program where the function *inc* is fully inlined.

The following example shows that the supercompiling can do deforestation as well.

```
let                                          let
  map = λf xs →                                h0 f g xs =
    case xs of                                   case xs of
      [] → []                          ⇝           [] → []
      (y : ys) → f y : map f xs                  (y : ys) → f (g y) : h0 f g ys
in                                           in
  map f (map g ys)                             h0 f g xs
```

What you see here is that fusion happens on the two calls to *map* to get rid off the intermediate data structure.

3.1 Requirements

In order to do this kind of program transformations, the supercompiler is built of several components that work together. First of all we need an **evaluator** that partially evaluate expressions. For example, the evaluator will optimize an expression as follow.

```
let id = λx → x                 ⇝    let id = λx → x
in id not y                          in not y
```

We also perform **dead code removal** to get rid off the unnessecary bindings.

```
let id = λx → x                 ⇝    not y
in not y
```

In the case evaluation gets stuck on a free variable, the **splitter** provides a way to individually evaluate sub parts of the expression.

```
let id = λx → x                      case x of
in case x of                    ⇝      True → False
   True → id False                     False → True
   False → id id True
```

There is also a component **drive** which will alternate the evaluation and splitting steps in order to obtain the following result.

```
let                                                 case x of
   id = λx → x                                        True → y
   not = λx →                                         False →
     case x of                                          case y of
        True → False                                      True → False
        False → True                    ⇝                 False → True
   xnor = λx y →
     case x of
        True → id y
        False → id not y
in
   xnor x y
```

The drive function furthermore take care of **memoization** in order to tieback on expression patterns that are already supercompiled. We already have seen this in an earlier example where we tieback on a binding *h0*. In order to determine equal expression patterns we require a **match** function that can check for equality of terms. The matcher is discussed in more detail in section 4.4.

Furthermore a supercompiler requires a **termination checker**. This due the fact that we may have to deal with infinite recursive function calls and infinite data types. In the first case the evaluation will never stop specializing, while in the second case the splitting will go on forever. This checker is conservative, and make sure the compiler always terminates, but at the cost of obtaining an fully optimized program. We will look at the termination checker in section 4.5.

Now we have already point out the components of the supercompiler. Each component will be discussed in detail in section 4.

3.2 Working on states rather than terms

Basically what a supercompiler does is alternating evaluation and splitting steps. For evaluation we need evaluation machinery which comes with a stack and heap. Because we have to split when evaluation gets stuck, we need splitting the state of that machine, which means that we have to split 3 aspects, namely the stack, the heap and the term currently in focus. Conclusion is that the supercompiler always works on states rather then terms. We denote a state as a 3-tuple with a heap H, a stack K and a term e in focus.

$$\langle H | e | K \rangle$$

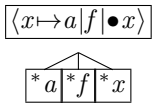
We use the following notation as shorthand if we have to deal with an empty stack and empty heap.

$$\langle e \rangle$$

We write down evaluation by a serie of reduction steps on states. For example, to evaluate the expression $\mathbf{let}\ f = \lambda x \to x\ \mathbf{in}\ f\ y$ we write

$$\begin{gathered}\langle \mathbf{let}\ x = a\ \mathbf{in}\ f\ x \rangle \rightsquigarrow \\ \langle x \mapsto a | f\ x | \epsilon \rangle \rightsquigarrow \\ \langle x \mapsto a | f | \bullet x \rangle\end{gathered}$$

The evaluation rules are discussed in more detail in section 4.1. After the reduction gets stuck on a free variable it will perform a split, which we denote as follow.



In the example above the input state $\langle x \mapsto a | f | \bullet x \rangle$ will be split and should result in substates. Each such substate could be individually reduced further and may split again. The substates denoted with a *star** are not really substates in the sense that they can't be processed further, these states can be seen as terminal terms and will be used to build the optimized output term by folding the tree using a rebuild function. So in this example there are no substates other than terminal terms, which means that supercompilation stops here and rebuild a term $\mathbf{let}\ x = a\ \mathbf{in}\ f\ x$, which actually means that we do not retrieve any optimization at all in this case. We discuss splitting and rebuilding in more detail in section 4.2.

4 Components

Now we will look at all components in more detail and also refer to implementation decisions. It could be useful to take a look at the examples in section 5 already to keep overview and see final results.

4.1 Evaluate

We implemented the evaluator using an AG, where each production matches a rule in the operational semantics from figure 1. It will reduce a state by applying a serie of reduction steps, each performing a rule that matches the input state, until no further reduction could happen anymore. The output of the reducer is always a state that contains a term in focus that is either a variable or a value. No other term can be a valid output. The only values in our language are *data types* and *lambda abstractions*.

$$
\begin{array}{lrcl}
\text{VAR} & \langle H, x\theta \mapsto (e^{t_e}, \theta_e) | x^t | K | \theta \rangle & \rightsquigarrow & \langle H | e^{t_e} | \texttt{update}^t x\theta, K | \theta_e \rangle \\
\text{UPDATE-val} & \langle H | v^t | \texttt{update}^{t_x} x, K | \theta \rangle & \rightsquigarrow & \langle H, x \mapsto (v^{t_x}, \theta) | x^{t_x} | K | \theta \rangle \\
\text{UPDATE-var} & \langle H[x\theta \mapsto v^{t_v}] | x^t | \texttt{update}^{t_y} y, K | \theta \rangle & \rightsquigarrow & \langle H, y \mapsto (x^{t_y}, \theta) | x^t | K | \theta \rangle \\
\text{APP} & \langle H | (e^{t_e}\ x)^t | K | \theta \rangle & \rightsquigarrow & \langle H | e^{t_e} | \bullet^t x\theta, K | \theta \rangle \\
\text{BETA-val} & \langle H | (\lambda y.e^{t_e})^t | \bullet^{t_x} x, K | \theta \rangle & \rightsquigarrow & \langle H | e^{t_e} | K | \theta[y \mapsto x] \rangle \\
\text{BETA-var} & \langle H[f\theta \mapsto (\lambda y.e^{t_e})^{t_\lambda}] | f^t | \bullet^{t_x} x, K | \theta \rangle & \rightsquigarrow & \langle H | e^{t_e} | K | \theta[y \mapsto x] \rangle \\
\text{CASE} & \langle H | (\texttt{case}\ e^{t_e}\ \texttt{of}\ \{\overline{\texttt{C}\ \overline{x} \to e_C^{t_C}}\})^t | K | \theta \rangle & \rightsquigarrow & \langle H | e^{t_e} | \{\overline{\texttt{C}\ \overline{x} \to e_C^{t_C}}\}, K | \theta \rangle \\
\text{DATA-val} & \langle H | (\texttt{C}\ \overline{x})^t | (\{\overline{\texttt{C}\ \overline{y} \to e^{t_e}}\}^{t_k}, \theta_k), K | \theta \rangle & \rightsquigarrow & \langle H | e^{t_e} | K | \theta_k[y \mapsto x\theta] \rangle \\
\text{DATA-var} & \langle H[z\theta \mapsto (\texttt{C}\ \overline{x})^{t_C}] | z^t | (\{\overline{\texttt{C}\ \overline{y} \to e^{t_e}}\}^{t_k}, \theta_k), K | \theta \rangle & \rightsquigarrow & \langle H | e^{t_e} | K | \theta_k[y \mapsto x\theta] \rangle \\
\text{LETREC} & \langle H | (\texttt{let}\ \overline{x = e_x^{t_x}}\ \texttt{in}\ e^{t_e})^t | K | \theta \rangle & \rightsquigarrow & \texttt{let}\ \overline{x' = \texttt{fresh}} \\
 & & \rightsquigarrow & \texttt{in}\ \langle H, \overline{x' \mapsto (e_x^{t_x}, \theta[x \mapsto x'])} | e^{t_e} | K | \theta[x \mapsto x'] \rangle
\end{array}
$$

Fig. 1. Operational Semantics Evaluator

These rules are an extended specification of the rules given by Bolingbroke. We added a fourth component to our tuple that will store the rename map. These renaming is essential in this semantics but not described by Bolingbroke. In general it follows the Sestoft-style semantics [4], what means we build a rename map along the way and apply it at the moment a variable is used.

Notice that the rules *beta*, *data* and *update* comes in two variants, namely a rule for variables and another rule that work on values. The variable rule could have been ommitted because we could choose to perform the *var* rule first followed by the rule for *beta-val*, *data-val* or *update-val*. This will make sense to reach our goal to minimize the implementation. Although we have chosen to still include both variants, because it minifies the steps taken by the compiler and therefore make the examples where we show the compilers process much easier.

The reason why we implement the evaluator in an AG is just because it is simple to program in this way. In many rules you have to modify the heap, stack or term, but not all at the same time. In AG you have only to specify the changes, which makes it very handy for this purpose. Further it is just working like a classic function call, where the inherit attributes corresponds with arguments of the function call and a single synthesized attribute corresponds with a single (tupled) return value. In order to drive the AG, we uses a dummy data type named Step, which is instantiated as higher-order AG with the rule (or operation) we like to perform next. Each Step instantiates another Step, what

means that we chain Steps together until we reach a fully reduced term or when the termination checker will prevent us from going on.

A termination checker is involved to make sure no infinite recursion could happen. The only rule that could be responsible for infinite recursion is the *beta* rule. Therefore Bolingbroke split the evaluator in two parts, named the functions *reduce* and *normalize*, where *reduce* is a evaluator which includes the *beta* rule and therefore needs a termination check, while *normalize* is the same evaluator without *beta* rule and therefore guarantees to stop. The calls to *reduce* and *normalize* are interleaved to make sure the result of *reduce* is normalized. We just integrate these two functions into a single AG, and make use of AG's driving mechanism that will continuously apply rules until we reach a normalized state that cannot be reduced further. Therefore we call this a multi-step reducer. We integrate the termination check in the beta rule. If the check fails, it continues applying only normalize rules, that means all rules except *beta*, and that makes sure it finally returns a normalized state. We haven't proven that the approaches are indeed the same, but we assume based on the arguments given above.

4.2 Splitter

If a state cannot be reduced, it will be split in zero or more substates where each state is optimized individually. These substates are stored in a field called *holes* from the datatype *Bracket*. Along these substates we store a rebuild function that describes how it can be build an output term which make use of the optimized terms obtained from the substates. Therefore each substate corresponds with an argument passed to the rebuild function. A bracket for the state $\langle$**case** *xs* **of** *True* $\rightarrow\langle ...\rangle$; *False* $\rightarrow\langle ...\rangle\rangle$ can looks as follows, where the **case** expression contains two branches and therefore have two substates, called s_1 and s_2.

```
Bracket {
  holes = [s1, s2],
  build = λ[e1, e2] →
    case xs of
      True → e1
      False → e2
}
```

We will create individual brackets for the three components of the state $\langle H|e|K\rangle$ that finally will be combined together into a single bracket. The heap and stack brackets even composed of smaller brackets for every entry it contains. Combining the brackets can be done with a helper function *plusBrackets* which is the same thing as the rebuild function, except that this function requires brackets as input instead of states.

How the build function will looks like depends on the thing where we create a bracket for. For example, the heap will have a rebuild function that creates a *Let* term containing all heap bindings. The stack is composed of different brackets

depending on the type of stack frame. Application frames generates an *App* term, update frames a *Var* term and case frames results in a *Case* term. As last we need a bracket for terms. We only have to deal with 3 type of terms, namely variables, datatypes and lambdas, this due the fact that the splitter receives normalized input, thanks to the normalization process described in section 4.1, which means that we only have to deal with variables or values. In the case of a lambda value, we will first optimize the body. That means that we have a single substate and a rebuild function that wraps the optimized term into a *Lam* again.

$$
\begin{aligned}
&\mathit{Bracket}\ \{\\
&\quad \mathit{holes} = [s],\\
&\quad \mathit{build} = \lambda[e] \rightarrow\\
&\qquad \lambda x \rightarrow e\\
&\}
\end{aligned}
$$

The bracket for a variable or datatype value is even simpler, because there are no substates involved and therefore needs a rebuild function that just return a constant term and not depends on any subterm. For example, a bracket for a variable x look like

$$
\begin{aligned}
&\mathit{Bracket}\ \{\\
&\quad \mathit{holes} = [\,],\\
&\quad \mathit{build} = \lambda[\,] \rightarrow x\\
&\}
\end{aligned}
$$

We call this a *terminal bracket* and denote it with a star like ${}^*x$ or ${}^*\mathit{Cons}\ x\ xs$. This notation come in handy when we draw splitting figures. Note that what we call terminal brackets is **not** the same concept as a *termBracket* in Bolingbroke's paper, because a *termBracket* e is just wrapping a term into a bracket, by first wrapping it into state $\langle \mathrm{e} \rangle$, lets call that state s, and produces a bracket that immediately returns the optimized subterm without modifying it, so that bracket looks like this

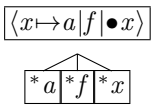
$$
\begin{aligned}
&\mathit{Bracket}\ \{\\
&\quad \mathit{holes} = [s],\\
&\quad \mathit{build} = \lambda[e] \rightarrow e\\
&\}
\end{aligned}
$$

Now we recall the example from the introduction and see which splitting steps are involved. We would like to split the following state

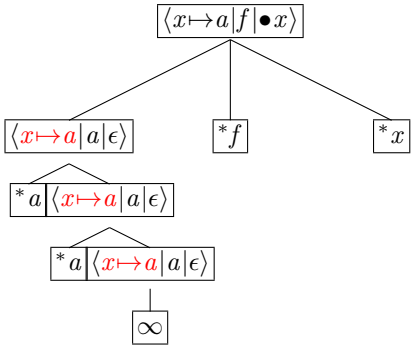
$$\langle x \mapsto a | f | \bullet x \rangle$$

Here $x \mapsto a$ is a heap binding, f the term in focus, and $\bullet x$ the stack frame that applies f to x. First the term f will result in a terminal bracket ${}^*f$. Then the stack creates a *termBracket* for the expression x, which results in a terminal bracket ${}^*x$. After that the heap will create a *termBracket* for every binding,

meaning that we obtain a terminal bracket $^*a$ too. The split look as follows



Although, most splits are not that easy, because in many cases we need to push down heap bindings to substates. If the substate refers to such binder, we need to push down that heap binding. Some analysis is required in order to know which heap bindings should be passed down. We leave out this analysis and just pass down all heap bindings to all states in order to keep the implementation simple, at the cost of additional unnecessary compilation steps. That means that our splitting process will looks as follows.



Notice that this compilation will go on and on forever if we would not have a termination checker, but we would see in the following section that memoization can take care of this too in some cases, which means that we only compile $\langle x \mapsto a | a | \epsilon \rangle$ once. We use the red marks as indication of heap bindings that are unnecessary passed down.

4.3 Drive

This drive component contains a function *sc* which drives the whole supercompilation, taking care of memoization, termination, splitting and building the output program.

Attribute Grammar We implemented the drive component using an Attribute Grammar as a replacement for Bolingbroke's monadic approach. What it does is nothing more than depth-first traversal over a tree that dynamically grows when a state is split, making use of the higher-order functionality of AG to dynamically add child states to the split state. Promises will be generated along the way in order to do memoization, which means that later on in the traversal we can tieback on previously compiled states.

We use the following example program in order to draw nice AG figures, although the program itself is not very useful.

```
case x of
  None →
    let
      f = id
      g = not
    in
      f g a
  True → False
  False → True
```

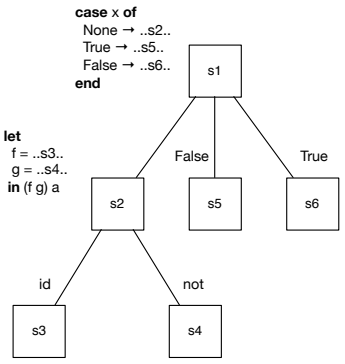


Fig. 2.

In figure 2 the program is visualized in an AG diagram, which contains all states that are involved in the supercompilation. These states are normally produced while performing splits and will be instantiated on the fly using higher-order AG's, but for simplicity they are now all visible from the beginning. The states are labeled by $s_1 ... s_n$, the order depends on the order which they will be encountered. At each state we have written an abstract term representation that corresponds to that state along with the subterms that individually are supercompiled.

An overview of the requirements of the drive component is given here. These requirements closely relates to the attributes specified in the AG, so we mention them here and visualize them, along with their flow, in figures 2-6.

- Assigning unique identifiers to states using attribute **freshProm**, and performing splits that instantiates substates using *higher-order* AG's. (fig. 2)
- Memoization is done by *chaining* **promises**. (fig. 3)
- States rebuild to terms. *Synthesized* in the attribute **sc**. (fig. 4)
- These terms fulfill promises, leading to *synthesized* **bindings**. (fig. 5)
- To perform termination checking, states are *inherited* in **history**. (fig. 6)

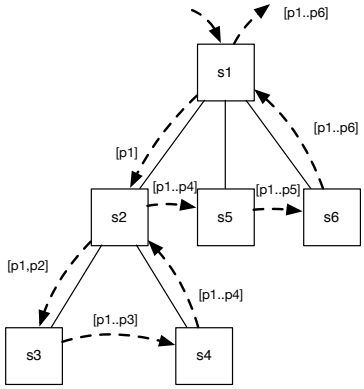


Fig. 3. Promises are chained, used for memoization. A promise p_i is an abstraction over the the state s_i.

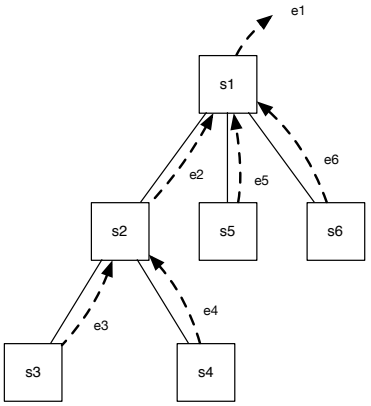


Fig. 4. Optimized terms (expressions) will travel up. A term e_i is build from their synthesized terms.

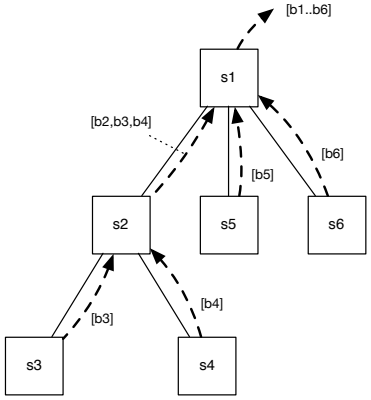


Fig. 5. Synthesize bindings that will be used in output program. A binding b_i consist of the term e_i bound to a variable called h_i.

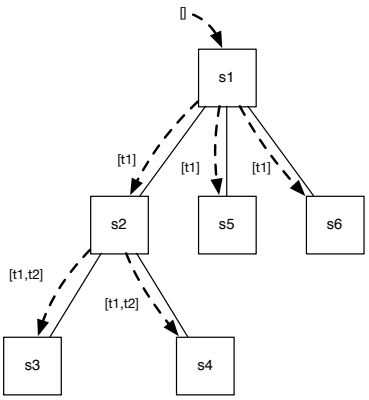


Fig. 6. History is passed down as a list of previously seen states, where t_i is the tagbag representation of state s_i.

The relation between these attributes are visualized in figure 7. It requires some explanation why so many arrows are involved. This have to do with the fact that we uses memoization and need to stop compiling when we find a memo.

Because normally an AG automatically goes in recursion, we must insert a check everywhere, in order to prevent recursion when encountering a memo. So all synthesized attributes need to implement this check in the *lhs* rule of the AG. This explains why *promises* has arrows to *sc* and *bindings*. Furthermore at every split, which is responsible for dynamically creating children, a termination check required. A termination check needs the attribute *history*. So every chained or synthesized attribute that uses the children also depends on the inherit attribute *history*.

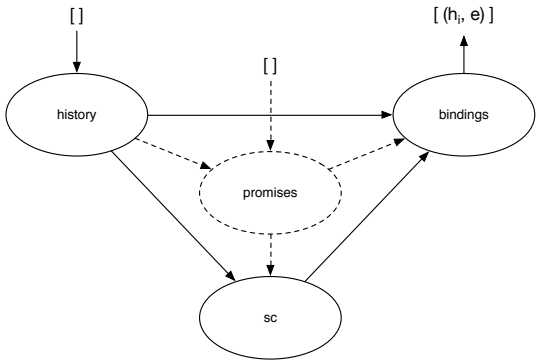


Fig. 7. Dependencies between attributes. The dashed component should be optional.

We draw the *promises* component with dashed lines because we think this component should be optional in a supercompiler, at least in order to create a minimal implementation. We do not succeed in leaving out this component, because our drive AG only terminates when all leaves are *terminal brackets*, not having sub states, or when encountering a memo. In order to leave out memoization we have to deal with the infinite unfolding as mentioned in the splitter (section 4.2). Maybe the termination checker should solve this problem in that case, but right now it only prevents from doing multiple *beta* reduction steps. This gap between memoization and termination checking needs additional research.

Walkthrough The process of driving starts with a single state that needs to be supercompiled. In order to memoize, a *promise* is created for this state. We know that at some moment this promise will be fulfilled with an optimized term. This also means that from now on, we can refer to this promise if we encounter a state equal to this one. Note that we do not check syntactic equivalence of states, but rather check for equal meaning as we discuss in the section Matching. Therefore each time encountering a state, we check if it matches a memoized state. If no such match exists then we create a new promise labeled by a unique identifier.

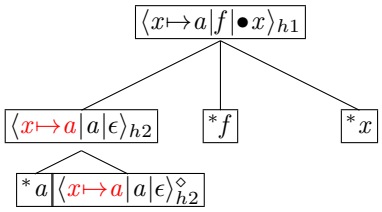
$$\langle \mathrm{x} \rangle_{h1}$$

Otherwise, if we do find a match, we can tieback on a previously compiled state, indicated by the diamond symbol and the identifier of the state it refers to.

$$\langle \mathrm{x} \rangle^{\diamond}_{h1}$$

After creating a promises it calls the functions *reduce*, *normalize* and *split*. Reducing the state means that we partially evaluate it, what is described already in section Evaluate. Note that we only call *reduce* in the case this state passes the *termination check*, otherwise it could start to unfold infinitely. The reduced term need to be normalized again before passed to the splitter. The splitter delivers new states that are instantiated as higher-order AG's for the current state, what means we can recursively continue supercompiling on these states.

Sometimes memoization can also play the role as termination checker, because as we have seen in section 4.2 that we repeatedly try to supercompile the same state. In that case, we only need to supercompile the first occurrence and tieback on this result in following occurrences. Therefore the AG from previous section now looks like this.



Now the memoization makes sure the supercompiler terminates. This only holds for splits that results in states that we have seen before. The following examples do need a termination checker because memoization not plays a role here

let *count* $n = n : count\ (n + 1)$
in *count* 0

This due the fact that every call to count leads to a function specialization for every unique integer. So every state is unique and cannot be matched by the matcher. Note that our language do not have to deal with integers, but the same can happen in the following example.

let *listsOf* $x\ xs = xs : listsOf\ (x : xs)$
in *listsOf* $y\ [\,]$

In order to perform termination checks, the drive component will keep track of *history*. It stores that seen states and passes it to the termination when performing a termination check.

Results We have shown that the drive function can be implemented within an AG, which make use of higher-order and automatic driving capabilities to traverse the tree and implicit pass on data like *promises* and *history* by making them explicit through attributes. We still need to do further research in order to omit memoization and still satisfy the termination property.

4.4 Matching

The matcher is required to let memoization work. The purpose of these module is to match two states and check if they are equal. This does not test for syntactic equality but rather test for equal meaning, this means equality modulo alpha-renaming. That means that in order to identify two states s_1 and s_2 equal, there should exists two substitutions θ_1 and θ_2, so that

$$s_1\theta_1 = s_2\theta_2 = s$$

where s is a state that is the most-specific generalization of this two states. Notice that θ contains variable-to-variable mappings, so only performs alpha-renaming. The matcher is responsible for finding this substitutions θ_1 and θ_2, and furthermore finds the common state s.

Although our goal is to create a minimal implementation of the supercompiler, we choose to fully implement the matcher as described in Bolingbroke's paper. Mainly because we could not figure out which parts are actually required, due the complexity that is involved to deal with work sharing. This is the reason why the matcher also delivers the common state s even we are only interested in θ to make memoization work.

This time we could not make use of AG to implement this component. Because what we do here is try to match two data structures, while AG is intended to work on a single datatype. So we use a monadic structure, the same as what Bolingbroke did.

The matcher starts comparing two states, by comparing the state's heap, term and stack, using the functions *matchHeap*, *matchTerm* and *matchStack*. Furthermore *matchTerm* delegates work to *matchAlt*, *matchBind* and *matchVar* to respectively match case alternatives, let-bindings and variables. The main concept is to substitute each free variable in both states to a single unique variable that the states then will have in common. In order to supply unique identifiers we need the monadic structure. Also lambda- and let-binders need to be renamed in order to make terms equal.

Bolingbroke also specify a function *fixup* that will run before returning θ and s. He suggests that the purpose of this function is to obtain work sharing. But just leave out this function can not be done. In that case compilation may not terminate because of direct recursive promises. Not sure why that is the case, but just include the fixup function solves the problem.

It could be that there are still some errors in the matcher, because in our opinion it is the most complex component of the supercompiler. Bolingbroke discussed the matcher but leave details out and refers to general knowledge

about matching in lambda calculus. This makes it hard to reimplement and therefore our implementation needs verification.

4.5 Termination

A termination checker is required to make sure the compilation process is finite. If this checker determines that compilation will loop forever, it stops further evaluation of the current state. Termination checks are used at two places. One in the drive function and one in the evaluation.

1. The termination check within drive component is performed because otherwise it will possibly unfold a data structure forever when one of the substates repeatedly tiebacks on the same promise, but not triggers memoization. An example is

   let *count* $n = n : \mathit{count}\ (n + 1)$
   in *count* 0

2. The termination check in the evaluator is required to prevent non-termination in the case of recursive functions that diverge.

   let *count* $n = \mathit{count}\ (n + 1)$
   in *count* 0

   The compiler is also not smart enough to determine non-diverge functions as like the example below. It just stops when encounter the recursive function call.

   let *count* $n =$
     case n **of**
       $0 \to 0$
       $\_ \to \mathit{count}\ (n - 1)$
   in *count* 10

The termination check fails in the case it recognizes equal states. Equality is determined by the tagbag representation of states, which have nothing to do with equality checks in the matcher. The concept of tagbags is explained by Bolingbroke, so we do not repeat it here. Although we can make some small notes, that were maybe not directly clear from his paper.

– The main idea is that it checks for equal states both containing the same terms, but only may differs term occurrences. This due unfolding where a term from the heap is inserted into the focus term. If it starts to occur multiple times, but for the rest stays the same state, then we stop compiling. For example this will happen when start to unfold an addition like x, $(x+x)$ and $(x + x + x)$ where all expressions have the same tagbag representation because outer terms of composed expressions $(x + x)$ have just the same tag as one of its arguments. Another example is that from above *count* $(n - 1)$ where you see a recursive call, where $(n-1)$ have the same tag each call and therefore encountered equally.

- A tagbag can be viewed as a key-value mapping. The keys are integers, each representing a unique term. The values describe how many times they are present. For example the variable x tagged as x^1 can be in a state for two times like $\langle f \mapsto x^1 | x^1 | \epsilon \rangle$, which gives us a tagbag $1 \mapsto 2$.
- The tagbag for a single term is just a singleton set containing the tag of the outer term, so the inner terms are not encountered.
- The tagbag of the heap is just the union of the tagbag representation of all bindings, not including the binder variable itself.

4.6 Free Variables

Getting the free variables is trivial and Bolingbroke therefore do not explain it in his paper. Although the semantics of the supercompiler is not standard, therefore I think there is the need to explain it in more detail, especially how to deal with renaming.

$$fvs\ \langle H|e|K|\theta_e\rangle = (fvs_H(H)\theta_e \cup fvs_e(e) \cup fvs_K(K)) \setminus (bvs_H(H) \cup bvs_K(K))$$

$$fvs_H(\emptyset) = \emptyset$$
$$fvs_H((x \mapsto e, \theta_e), H) = fvs_e(e)\theta_e \cup fvs_H(H)$$

$$fvs_K(\emptyset) = \emptyset$$
$$fvs_K(\bullet x, K) = \{x\} \cup fvs_K(K)$$
$$fvs_K((\mathbf{case}\{\overline{alt}\}, \theta_c), K) = \bigcup_{\overline{alt}} fvs_{alt}(alt)\theta_c \cup fvs_K(K)$$
$$fvs_K(\_, K) = fvs_K(K)$$

$$fvs_{alt}(\mathtt{C}\,\overline{y} \to e) = fvs_e(e) \setminus \overline{y}$$

$$fvs_e(x) = \{x\}$$
$$fvs_e(\lambda x\,.\,e) = fvs_e(e) \setminus \{x\}$$
$$fvs_e(f\ x) = fvs_e(f) \cup \{x\}$$
$$fvs_e(C\,\overline{x}) = \overline{x}$$
$$fvs_e(\mathtt{let}\ \overline{x = e_x}\ \mathtt{in}\ e) = (\bigcup_{\overline{e_x}} fvs_e(e_x) \cup fvs_e(e)) \setminus \overline{x}$$

Fig. 8. free variables

The most interesting part is to how calculate free variables from a state tuple $\langle$H|e|K$\rangle$. The formal rules are written in figure 8. We first calculate the free variables of every component. Then we compute the free variables for the state as whole, which is not just the union of these components because some of the variables become bound by the binders from the heap or the update frames from the stack. Therefore we need to omit some of these variables and we do that by subtracting a set of bound variables, which rules are written down in figure 9. The result will be the variables that are free in this state.

$bvs_H(\emptyset) = \emptyset$
$bvs_H(x \mapsto e, H) = \{x\} \cup bvs_H(H)$

$bvs_K(\emptyset) = \emptyset$
$bvs_K(\texttt{update}x, K) = \{x\} \cup bvs_K(K)$
$bvs_K(\_, K) = bvs_K(K)$

Fig. 9. bound variables

Additionally we make explicit how to deal with renaming, because within a state there are also rename maps stored. In the first place as fourth component in a state tuple $\langle \mathrm{H}|\mathrm{e}|\mathrm{K}|\theta\rangle$, but also in all heap bindings and the stack frames for case expressions. The fourth component renaming must be applied on the term in focus, you can see that happens in the *fvs* rule. The renaming stored in the heap bindings and the stack frames for case expressions must be applied on the corresponding heap binding and case alternatives respectively, what you can see happen in the rules fvs_H and fvs_K.

4.7 Renaming

The rename component provides three different functions that are all related to renaming. One is the uniqueness renaming, used to initial rename the program to make all binders uniquely named. The second one is a helper function to rename terms including binders, which is used in the matcher. This one have some overlap with the uniqueness renaming, except that the renaming is given instead of uniquely supplied. The third one is the renaming of variables that is performed while supercompiling, following the Sestoft-style semantics, which keeps track of a rename map and apply the renaming on variables at the time of their usage.

1. The initial uniqueness renaming does a single traversal over a term and renames all binders of **let**, *lambda* and **case** expression patterns. The implementation requires no further explanation.
2. The second renaming function renames a complete term, including the binders. This follows the same approach as uniqueness naming, but instead of introducing unique names, we provide a renaming map. The renaming of whole terms is used in the matcher in order to match two terms, by trying to make them equal, which also requires the same naming of binders.
3. The renaming that is performed while supercompiling is more complex. Therefore we take a closer look at it. Given a rename map, it can easily rename some variable using the function *rename*. Although the question is, do we need transitive lookups? For example when the rename map contains the following mapping:

$\{x \to y, y \to z\}$

If we like to know x and not do transitive lookups, we only get the result y. If we do transitive lookups we get z. But we must ask ourself if such transitive mapping can exists in the first place. The answer is no, these transitive mapping *cannot* exists. For example, encounter this example that at first sight will give the transitive bindings from above:

$(\lambda y \,.\, (\lambda x \,.\, \_)\ y)\ z$

But if we look at the operational semantics and apply the *beta* rule on the outer lambda, it results in a renaming from y to z. Then it encounters the inner lambda where it inserts the renaming x to z, instead of x to y. This is because we first apply the renaming $y\theta$ before the inner lambda is applied. The final mapping looks like this:

$\{x \to z, y \to z\}$

In general what is does is apply the renaming before inserting a new mapping. This will happen in the following three evaluation rules *beta*, *data* and *let*.

On second thoughts our initial uniqueness renaming (1) is somewhat superfluous. We just renamed all binders in order to prevent from creating transitive binding. But now we have shown in (3) that such bindings will not exists for lambda abstractions, we do not have to rename the lambda binders. But we still need to rename let bindings and case patterns, because they are stored in the heap and otherwise would be overwritten if we encounter nested bindings using the same name.

Bolingbroke not does initial renaming at all, but just does it at run time. We cannot figure out if this is a strict requirement of the semantics. Our approach will maybe fail in recursive structures, at the moment heap bindings will be instantiated, because then the same name is used while Bolingbroke generates a unique name at that moment. We still need to verify if this is indeed the case.

4.8 Dead Code Removal

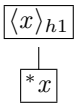
Dead code removal is introduced to make the output program more readable by omitting bindings that aren't used. It's not sufficient to only look at the body to see which bindings are used, because bindings can also call each other due to use of recursive let. On the other hand we can not simply look at all bindings that are somewhere called, because it's not useful to look at the calls within dead bindings. Therefore we uses a iterative approach to follow the bindings that are called from the body, resulting in list of bindings that are not-dead.

The function *deadCodeRemoval* is called as last step in the pipeline to clean up to output program. But right now, we need to call it within the compiling process as well. This due our goal to do not all optimizations and push down all

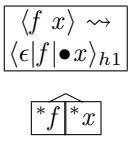
heap bindings in the splitter, including dead bindings. These results in additional free variables. Because promises are abstracted over these free variables, they become abstracted over dead code. By performing dead code analysis before we calculate the free variables we prevent from the superfluous abstraction. In the future we need to implement this analysis in the splitter itself to prevent the work duplication from the beginning.

5 Examples

Now we show some examples to visualize the supercompilation process. Given an input program we show the evaluation-, reduction- and splitting steps that are taken. The supercompilation output is an optimized program specified in memoized bindings $h_1 \ldots h_n$. This program can be feed to a normal evaluator to simplify the output. The final output is an optimized version of our input program.

input program	**supercompilation steps**	**output supercompilation**	**output program**
VAR			
x		**let** $h1 = \lambda x \,.\, x$ **in** $h1\ x$	x

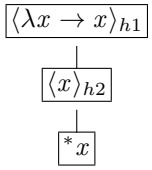
- Start with the state $\langle \epsilon | x | \epsilon \rangle$.
- Because x isn't in the heap, we can't look it up and can't reduce this term.
- So we split $\langle \epsilon | x | \epsilon \rangle$, which creates a memoized binding h_1. There is no heap or stack to be split, but we try to split x. Because x can't be split, we will receive ${}^*x$, which is a terminal bracket who rebuilds to the term x.
- This term fulfill the promise h_1. It is abstracted over the free variables $\lambda x \,.\, x$, resulting in a binding $h1 = \lambda x \,.\, x$ that travels up in th tree.
- A final program is created by wrapping all bindings in a **let** construct and make an initial call to the first (and only) binding $h1$.

APP			
$f\ x$		**let** $h1 = \lambda f \lambda x \,.\, f\ x$ **in** $h1\ f\ x$	$f\ x$

- $\langle \epsilon | f x | \epsilon \rangle$ reduces to $\langle \epsilon | f | \bullet x \rangle$ when the APP rule is applied.
- Because f isn't in the heap, we can't apply the $BETA$ rule, so no further reduction is performed.
- This state starts splitting after creating a promise h_1. Because the variable f can't be split it wil return a terminal bracket ${}^*f$. The stack frame $\bullet x$ also returns a terminal bracket ${}^*x$.
- The promise h_1 is fulfilled, abstracted over the free variables, resulting in $h1 = \lambda f \lambda x \,.\, f\ x$.

LAM

$\lambda x \to x$



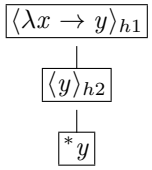
let
  $h1 = \lambda x \to x$
  $[h2 = \lambda x \,.\, x]$
in $h1$

$\lambda x \to x$

- The state $\langle \epsilon | \lambda x \to x | \epsilon \rangle$ can't be reduced because the lambda expression is already a value and no stack frame can be applied.
- Split this term results in a promise h_1, where the state $\langle \epsilon | x | \epsilon \rangle$ (body of the lambda expression) will be further reduced.
- Because $\langle \epsilon | x | \epsilon \rangle$ can't be reduced we split again, resulting in a promise h_2, and the terminal bracket x^*.
- **Note** that the origin of the similar looking promises h_1 and h_2 are not the same. They both results in identity functions, the only difference is that h_1 is a promise for the expression $\lambda x \to x$ while h_2 is a promise only for the variable x. Beause h_2 needs to be abstracted over the free variables it results in $\lambda x \,.\, x$. We use different lambda notation to distinguish between the binders that origins from the expression itself, and the ones that are created due the abstraction of the promise.
- The reason why the promise h_2 is written in brackets is because it isn't used in the output program, therefore we can safely eliminate this binding. Although h_2 was required in the process of supercompilation in order to rebuild h_1.

LAM-2

$\lambda x \to y$



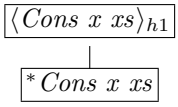
let
  $h1 = \lambda y \,.\, \lambda x \to y$
  $[h2 = \lambda y \,.\, y]$
in $h1\ y$

$\lambda x \to y$

- Almost the same example as LAM in the way of compilation. Only now, h_1 is abstracted over the free variable y. Notice that the binder x is still part of the expression $\lambda x \to y$ and has nothing to do with the abstraction over the promise.

CON

$Cons\ x\ xs$



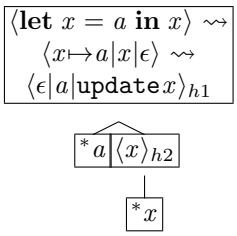
let $h1 = \lambda x \lambda xs \,.$
  $Cons\ x\ xs$
in $h1\ x\ xs$

$Cons\ x\ xs$

- Datatypes are values in our language, so these term cannot reduce or split, resulting in a terminal brack $^*Cons\ x\ xs$.
- Note that the arguments of constructors are always variables. For that reason we do not split the arguments. If these variables points to existing heap bindings, then these bindings are optmized due the fact the splitter optimizes all heap bindings anyway.

LET

let $x = a$ **in** x



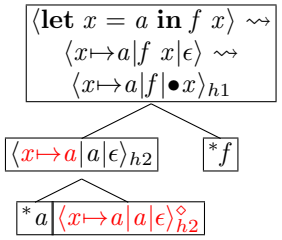
let
  $h1 = \lambda a\,.$
    let $x = a$ **in** x
  $[h2 = \lambda x\,.\,x]$
in $h1\ a$

a

- The state $\langle$**let** $x = a$ **in** $x\rangle$ reduces by the LET and VAR rules.
- Then the state $\langle\epsilon|a|\texttt{update}x\rangle$ is split.
- The focus term cannot be split and directly results in the terminal bracket $^*a$.
- Splitting a the stack frame $\texttt{update}x$ is now defined in terms of a call to *termBracket*. This means means that we first obtain an identity bracket and get the terminal bracket *x afterwards. The implementation can be probably simplified in order to return the terminal bracket directly. We did that also for $\bullet x$ in the APP example.
- Furthermore the split of $\texttt{update}x$ make sure that it passes a binding $\lambda x \to a$ to the function *splitHeap* in order to finally rebuild to a **let** expression that include the binding again.

LET-2

let $x = a$ **in** $f\ x$



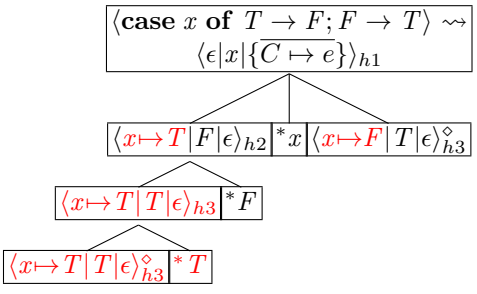
let
  $h1 = \lambda a \lambda f\,.$
    let $x = a$ **in** $f\ x$
  $[h2 = \lambda a\,.\,a]$
in $h1\ a\ f$

$f\ a$

- It reduces $\langle$**let** $x = a$ **in** $f\ x\rangle$ to a state $\langle x{\mapsto}a|f|\bullet x\rangle$ by apply the LET and APP rules.
- Then it becomes stuck on the variable f, that is not present in the heap, so we split.
- The variables f and x directly results in terminal brackets $^*f$ and $^*x$.
- Furthermore there will be a substate for every heap binding, in this case the substate $\langle a\rangle$.
- Because we leave out smart analysis to determine which heap bindings are alive, we just push down all bindings. What actually means we create a bracket for the state $\langle x{\mapsto}a|a|\epsilon\rangle$. Because this could have been omitted we mark it red.
- Splitting that state, results in a terminal bracket $^*a$ and a substate that we already have seen before, therefore we just tieback on the promise h_2, indicated by the diamond symbol.

CASE

case x **of**
  $T \to F$
  $F \to T$



let
  $h1 = \lambda x\,.$
    case x **of**
      $T \to F$
      $F \to h3$
  $[h2 = F]$
  $h3 = T$
in $h1\ x$

case x **of**
  $T \to F$
  $F \to T$

- The scrutine of a case expression becomes the focus term, the alternatives are stored in the stack.

- Splitting the focus term results in a terminal bracket *x. Splitting the alternatives each results in a substate.
- Notice that the case alternative patterns contain knowledge that is useful when optimize the branches. Therefore this knowledge is included through an additional heap binding.
- Furtermore we pass down all heap bindings, which results in many unecessary compilation steps. This also means that when encounter the second branch, we have already compiled all heap bindings in the compilation of the first branch, so we can just tieback on the results obtained in h_3. This also explains why the second branch in the output program contains a reference to a promise, while the first branch directly reached a value.

S-APP

$(\lambda x \to x)\ y$

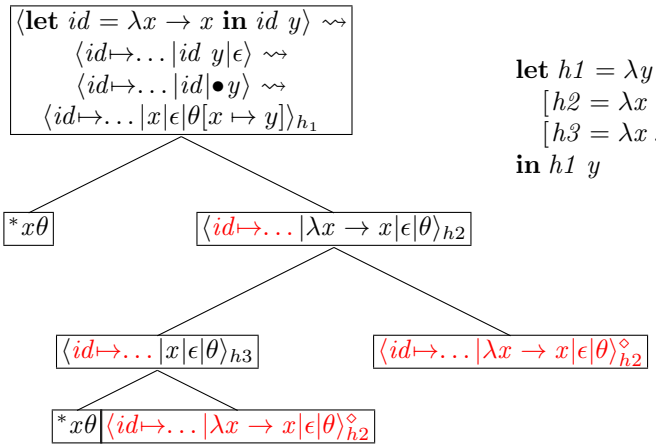
$$\begin{array}{c}\boxed{\begin{array}{l}\langle(\lambda x \to x)\ y\rangle \rightsquigarrow \\ \langle\epsilon|\lambda x \to x|\bullet y\rangle \rightsquigarrow \\ \langle\epsilon|x|\epsilon|\theta[x \mapsto y]\rangle_{h_1}\end{array}} \\ \boxed{{}^*x\theta}\end{array}$$

let $h1 = \lambda y\,.\,y$ $\quad y$
in $h1\ y$

- First apply the APP rule, then the rule for BETA. Now we deal with the renaming introduced by the beta reduction. Therefore we use the 4-tuple notation to store the substitution (rename map) in the state explicit.
- Splitting this state results in a terminal bracket $^*x$. We always apply the substitution on a terminal bracket $^*x\theta$ resulting in $^*y$ before returning it.
- Rename the terminal brackets should have been done in the previous examples as well. We do not mentioned it because the rename map is still empty there.

S-ID

let $id =$
$\quad \lambda x\,.\,\to x$
in $id\ y$



let $h1 = \lambda y\,.\,y$ $\quad y$
$\quad [h2 = \lambda x \to x]$
$\quad [h3 = \lambda x\,.\,x]$
in $h1\ y$

- Reduction follows the steps LET, APP and BETA-VAR. The BETA-VAR rule does the lookup in the heap and then apply that expression to the variable stored in $\bullet y$.
- The beta reduction generates a substitution that must be passed down to the substates.
- The substitution is applied when rebuilding the term.

Until now we only listed the basic examples above. In the next version of this paper we like to add complex examples and also show failing cases.

6 Conclusion

We reimplement Bolingbroke's supercompiler in a minimal form, making use of attribute grammars instead of monadic structures. By doing this we encountered several difficulties, sometimes due the lack of explanation in the existing papers. Therefore we use visuals to show the compilation process and make splitting steps explicit. We also formalized the operational semantics of the evaluator and give rules how to compute *fvs* and *bvs*. Especially we focussed on making explicit how to deal with renaming. We not really succeed in making a minimal compiler because leaving out components will quickly results in non-termination. For example, memoization is required otherwise the AG will recurse forever. Besides that fact, the AG turns out to be useful in order to create a nice implementation for evaluation and driving. We not succeed in making the compiler fully correct. There is still work to do in order to fix bugs.

References

1. Bolingbroke, Maximilian C. Call-by-need supercompilation. *University of Cambridge, Computer Laboratory, UCAM-CL-TR-835*, May 2013.
2. M. Bolingbroke and S. Peyton Jones. Supercompilation by evaluation. In *Proceedings of the 2010 ACM SIGPLAN Haskell Symposium*, September 2010.
3. Neil Mitchell. Rethinking supercompilation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Program- ming, ICFP 2010*. ACM, 2010.
4. P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming, 7(03):231–264*, 1997.
5. Valentin F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst., 8(3):292–325*, 1986.
6. UU Attribute Grammar Compiler (UUAGC), Software Technology, Utrecht University, http://foswiki.cs.uu.nl/foswiki/HUT/AttributeGrammarSystem