CHALMERS

Supercompiling Erlang

Master of Science Thesis

GÖRAN WEINHOLT

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, April 2013

Supercompiling Erlang

Göran Weinholt

Examiner: David Sands

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden, April 2013

Abstract

Erlang is a programming language with excellent support for parallel and distributed programming. The functional programming paradigm with its higher-order functions is used pervasively in Erlang, but code written in this style suffers in performance. In this work a supercompiler for Erlang is presented that can automatically eliminate higher-order functions and intermediate data structures.

Sammanfattning

Erlang är ett programmeringsspråk med utmärkt stöd för parallell och distribuerad programmering. Det är ett funktionellt språk där man ofta använder högre ordningens funktioner, men tyvärr leder dessa till försämrad prestanda. I det här arbetet presenteras en superkompilator för Erlang som automatiskt kan eliminera högre ordningens funktioner och temporära datastrukturer.

Preface

I am meant to write this report so that it is readable by people with a background similar to mine. This means identifying the concepts that are shared by me and the reader, identifying those that are new to the reader, and then explaining the new concepts in terms of the shared ones.

When I first started reading about supercompilation one of the problems I had was with the notation used in the formal definitions. This formality is of course excellent for its exactness and briefness (when used properly). But each new subject seems to bring out a new notation; one which is strangely similar to existing ones, yet different enough that the old rules do not apply.

They require a peculiar kind of thinking, a mixture between that of an automaton and a human. Yet by not being as well defined as they appear, often using natural language as an escape hatch or letting vital information be implicit, they confound automatons and humans alike. Add some intimidating Greek lettering and the picture is complete.

It is my hope that this report can serve to introduce a wider audience to supercompilation. I have tried to explain the new concepts with natural language and examples. If after reading this report you find the concept of supercompilation to be clear and simple, then I will have succeeded. Afterwards the formal notation used in the literature should be much more useful to you.

Acknowledgements

The first person I want to thank is Karol Ostrovsky, who in a discussion with David Sands brought up some issues he was having with Erlang. The second person I want to thank is David Sands, who suggested that I should do a thesis that addresses some issues Karol Ostrovsky was having with Erlang. Dave has been a great support towards the end of my time at Chalmers and in particular during this work. He is really the first person I should be thanking.

My thanks also to Peter A. Jonsson who created the supercompilation algorithm that I have based my work on. Peter also answered the questions I had about his research, for which I am grateful. Many thanks also to all those who made his algorithm possible. It is not possible to write down a complete list of the people who made this thesis possible. Thanks to Turchin, Aristotle, Newton and many other great men of the mind.

Contents

Chapter 1

Introduction

Not long ago the sentiment among programmers was that for most programming tasks a low-level language is appropriate. High-level languages were not considered serious, and the programs written in them were somehow less real than those written in a low-level language. Today we find high-level languages accepted in the mainstream.

The architecture of the computer has not changed much since the days when they were used primarily for computing, which is an activity that does not require much more abstraction than limited-precision numbers, arrays and perhaps functions. Low-level languages are designed to program the machine as it is. When a programmer requires more abstractions he must either express them at the level of the machine or choose a programming language that automates that process.

What exactly makes a programming language low-level or high-level may then be nothing but an historical accident, having more to do with the abstractions provided by the machine than with any attribute of the language itself. It is nevertheless true that programs written in high-level languages must be translated to the lower level of the machine, either by way of interpretation or compilation.

Interpretation necessarily introduces some overhead: the machine is now running both the interpreter and the program. The machine itself is a kind of interpreter, so there are now two interpreters reading the program. The overhead of the first interpreter can be removed by compilation, which translates the program into one that can be interpreted directly by the machine. Any high-level abstractions are translated into those provided by the machine and the result is called machine code.

At this point the program can be made to run with less overhead if the machine code is improved. The discipline of computer science that studies such low-level optimisations is quite mature. These transformations do not, however, deal with high-level abstractions. In translating high-level code into machine code the abstractions, the concepts, are lost. This is similar to dissolving a Nobel medal in aqua regia. All the gold will still be in the liquid, it will just no longer be a Nobel medal.

To optimise a high-level program it is necessary to work with the same concepts that the program uses. Since the compilation process discards the conceptual information it follows that such optimisations must be made before translation to machine code. Therefore they should be done on a representation of the program that is similar to the source code. The idea in basic terms is to change the program in ways that preserve its meaning but that makes it more suitable for the low level of the machine.

The programmer can do these optimisations himself, but it is tedious work. Programmers who have so often introduced automatisation into other fields should not be afraid of it in their own. With automatic optimisers the programmer who uses high-level languages need not sacrifice performance for comfort.

1.1 Source-level optimisation

In introductory courses to computer science the student is told that computer science has nothing to do with computers, that computer science is not a science, that computer science is about *how*. If computer science has nothing to do with computers, then who is it that is interested in *how* to make programs run faster? The efficiency of faster program is only relevant if run on an actual computer, and computer science allegedly does not deal with those.

While it is difficult to study program optimisations without reference to the fact that computers compute, it would be even more difficult to study them without reference to actual computer programs. Let us therefore look at an example of what is concretely meant by source-level optimisation.

```
four() ->
    X = 2,
    X + (1 + 1).
```

Let us start out with a simple example of an Erlang function (all examples in this thesis are written in Erlang). The first thing one notices is that the code obviously is silly and no person would ever write like that. But computer programs can also write code and they are usually not very careful to write code that looks good. Anyone who is familiar with programming languages can look at the example and see that it can be rewritten like this:

```
four() ->
    4.
```

There are actually three separate types of optimisation involved in this rewriting. One notices that the `X` in the last expression can be replaced by the constant `2`. This is called *constant propagation*:

```
four() ->
    X = 2,
    2 + (1 + 1).
```

Next one notices that `1 + 1` also can be replaced by a constant, and after that `2 + 2` can be replaced by a constant. This is called *constant folding* and is generally applicable when a primitive operation, such as addition, has constant arguments:

```
four() ->
    X = 2,
    4.
```

Lastly it is clear that `X` is no longer used and can be removed. When this is done the effect is called *dead code elimination* or *useless-variable elimination* [34]. After performing these optimisations the code certainly looks improved. But there is a very important distinction that must be made clear. The optimisations shown so far merely name what was done to the code.

Computers compute and computation is done by some specific means. If a computer program is to be devised that can do this rewriting automatically then one must first discover a method by which it can be done. Three kinds of optimisations have been demonstrated, but not yet a single method which can accomplish them.

The simplest of these optimisations is surely constant folding. One way of doing constant folding is as follows. First use a parser that translates the program into a data structure. Then walk over that data structure looking for opportunities for constant folding. When finding such an opportunity do the computation and replace that part of the code with the result.

The example contains two opportunities for constant folding, but an optimiser that only uses this method will only pick up the fact that $1 + 1 = 2$. If the optimiser is to recognise the second opportunity it must also be capable of constant propagation. Clearly such an optimiser is more powerful. The more powerful optimiser may be expected to produce better code.

1.1.1 Partial evaluation

Partial evaluation [17] is the computation performed by a particular type of program called a *partial evaluator*. For contrast one might say that a normal interpreter or compiler does full evaluation, i.e. it runs the whole program. Those parts of a program that can change from time to time (perhaps they depend on input from the user) are considered *dynamic*. Those that do not change are *static* and can in principle be run ahead of time. The technology of partial evaluation is well established and has been incorporated in commercial compilers [47].

The output of a partial evaluator is a new program, which is called a *residual program*, where only the dynamic computations remain. A partial evaluator can be used for optimisations, but there are also other interesting uses [4, 3, 29]. The partial evaluator described in [47] can accomplish constant propagation, constant folding, useless code elimination, copy propagation and procedure inlining. *Copy propagation* is similar to constant propagation, except that a variable has been bound to another variable instead of a constant. *Procedure inlining* replaces a procedure call with the body of the procedure.

Let us look at a more interesting example program and see what partial evaluation can do. Listing 1.1 on page 6 shows a program that translates a Unicode code point to its UTF-8 encoding. The optimisation opportunities in this program will only show themselves if procedure inlining is performed. Both `Len` and `I` are static in this program, so the expressions `Len >= I`, `6 * (Len - I)` and `element(Len + I, LengthCodes)` are also static. That means that they can all be evaluated by the partial evaluator.

Listing 1.2 shows what this program might look like after partial evaluation. There are two obvious inefficiencies in this residual program. It still calls the `lists:flatten` function and unfortunately there is not really that much a partial evaluator will do about that.[1] It could be taught to recognise this special case, but in general to remove such function calls requires more powerful transformations.

The other inefficiency is that a few uses of the bitwise operations (`bsr`, `bor` and `band`) are strictly not necessary and have no effect. It is certainly possible to teach the partial evaluator about these cases as well, but there is a surprising way of fixing it by modifying the original program. Normally one thinks that adding conditionals to a program will make it slower. This is not the case if the conditionals are static and a partial evaluator is used.

The idea is demonstrated in listing 1.3. This style may not be a great improvement on the original program[2], but the partial evaluator can avoid residualising the redundant bitwise operations in this version. The conditionals will be gone in the residual program because the guard expressions can be evaluated statically. Knowing what optimisations your compiler is capable of may change the way you write programs.

1.1.2 Deforestation

What can be done to remove the call to `lists:flatten` in listing 1.2? It would certainly be possible to modify the program so that the call is avoided, but it would be better if the optimiser could do this automatically. More powerful transformations are needed. Ideally one would like the code to end up like in listing 1.4.

The problem with the original code is really that it uses intermediate lists. One part of the program produces a list and another part immediately consumes it. This type of composition is quite common in functional programs. One type of transformation that can address this problem is called *deforestation* [48, 9].

The deforestation algorithm will fuse the producer and consumer in this example (assuming of course that it has access to the definition of `lists:flatten`). Somewhere inside the definition of `lists:flatten` is a pattern matching on the arguments. Because it is known in the example what the arguments will look like the pattern matching can be done ahead of time. It is not necessary to know exactly what the contents of the lists given to `lists:flatten` will be; the algorithm only needs to know the structure of the arguments.

The original deforestation algorithm did not turn out to be as practical as had been hoped, so in response a “short cut” was developed [6]. The new approach is often known as *shortcut fusion* and can perform deforestation on a subset of the cases the original algorithm could handle. The short cut consists of recognising special cases. In the original paper [6] the special function pairing *foldr* and *build* is recognised and [43] describes shortcut fusion for the functions *unfoldr* and *destroy*.

The short cut to deforestation algorithm would not be able to remove the intermediate lists in the UTF-8 encoder example. It would first be necessary to rewrite `lists:flatten` and the encoder in terms of a recognised function pairing. Short cut fusion does not have the generality of the original deforestation

[1]The list structure is static in this particular example and given as a direct argument to `lists:flatten`. A partial evaluator could conceivably, depending on how `lists:flatten` is written, eliminate the intermediate list. In practise this does not seem to be done.

[2]The style could be improved by hiding the conditionals in separate functions. If this is done then the partial evaluator will inline them and the same optimisations will be performed.

algorithm. Furthermore both algorithms are considering only part of the problem of how to go from listing 1.1 to listing 1.4. They do not supplant partial evaluation. There is a more general transformer that can replace both partial evaluation and deforestation.

1.1.3 Supercompilation

Supercompilation [45] is a transformation more powerful than deforestation and partial evaluation. They are similar enough that one can place them in a framework of transformations where different amounts of information is propagated [40].

Recall that the partial evaluator can evaluate those parts of a program that are static, while leaving intact those that are dynamic. The supercompiler will not stop when it encounters dynamic data. It will continue working on the program until some termination criteria is triggered.

Suppose that a program constructs a list. If the elements are static then a partial evaluator can in principle do constant folding and construct the list ahead of time. But if the elements of the list are dynamic this will not be possible. Whereas a supercompiler will not be able to do constant folding either in this case, it will do something better than give up. It will construct a model of the list and will use that model as if it were the actual list. If the program then does pattern matching on the list the supercompiler can look at the model and do the pattern matching ahead of time. As a result of this it would eliminate the call to `lists:flatten` in the UTF-8 encoder.

Supercompilation is more general than has been indicated here. Jonsson has recently worked on supercompilation for call-by-value semantics [19]. This is what is needed to make a supercompiler for Erlang that can take the program in listing 1.1 and find the program in 1.4 and this is what the rest of this thesis is about.

1.2 Delimitations

The goal of this work has been to develop an Erlang implementation of Jonsson's supercompiler. The thesis does not include a full treatment of the Erlang language: side-effects are explicitly outside the scope of the thesis, the pattern matching language is only handled partially. The implementation should work on examples from the literature.

The implementation is a first step towards a practical supercompiler that works with existing Erlang programs. It also replicates the work of Jonsson, showing that the algorithm works as expected, and for a language not previously tested. The report both explains the implementation and serves to introduce supercompilation to a wider audience.

Supercompilers have uses other than optimisation. They have e.g. been used to prove the equivalence of higher-order terms [26], perform inverse computation [8] and do theorem proving [44]. With the Futamura projections a supercompiler can be coupled with an interpreter and work as a compiler, or turn an interpreter into a compiler, etc [4, 41]. The intended use for this work is supercompilation applied to source-level program optimisation.

```
to_utf8(Code, Len, I, Set, Mask) when Len >= I ->
    [((Code bsr (6 * (Len - I))) bor Set) band Mask];
to_utf8(_, _, _, _, _) -> [].

to_utf8(Code, Len) ->
    LengthCodes = {16#00, 16#00, 16#C0, 16#E0, 16#F0},
    lists:flatten(
      [to_utf8(Code, Len, 1, element(Len+1, LengthCodes), 16#FF),
       to_utf8(Code, Len, 2, 16#80, 16#BF),
       to_utf8(Code, Len, 3, 16#80, 16#BF),
       to_utf8(Code, Len, 4, 16#80, 16#BF)]).

to_utf8(Code) when Code < 16#80 -> to_utf8(Code, 1);
to_utf8(Code) when Code < 16#800 -> to_utf8(Code, 2);
to_utf8(Code) when Code < 16#10000 -> to_utf8(Code, 3);
to_utf8(Code) -> to_utf8(Code, 4).
```

Listing 1.1: A UTF-8 encoder that benefits from partial evaluation.

```
to_utf8(Code) when Code < 16#80 ->
    lists:flatten([[((Code bsr 0) bor 0) band 16#FF],[],[],[]]);

to_utf8(Code) when Code < 16#800 ->
    lists:flatten([[((Code bsr 6) bor 16#C0) band 16#FF],
                  [((Code bsr 0) bor 16#80) band 16#BF],
                  [], []]);

to_utf8(Code) when Code < 16#10000 ->
    lists:flatten([[((Code bsr 12) bor 16#E0) band 16#FF],
                  [((Code bsr 6) bor 16#80) band 16#BF],
                  [((Code bsr 0) bor 16#80) band 16#BF],
                  []]);

to_utf8(Code) ->
    lists:flatten([[((Code bsr 18) bor 16#F0) band 16#FF],
                  [((Code bsr 12) bor 16#80) band 16#BF],
                  [((Code bsr 6) bor 16#80) band 16#BF],
                  [((Code bsr 0) bor 16#80) band 16#BF]]).
```

Listing 1.2: The UTF-8 encoder after partial evaluation.

```
to_utf8(Code, Len, I, Set, Mask) when Len >= I ->
    A = if Len == I -> Code; true -> Code bsr (6 * (Len - I)) end,
    B = if Set == 0 -> A; true -> A bor Set end,
    [if Mask == 16#FF -> B; true -> B band Mask end];
to_utf8(_, _, _, _, _) ->
    [].

to_utf8(Code, Len) ->
    LengthCodes = {16#00, 16#00, 16#C0, 16#E0, 16#F0},
    lists:flatten(
      [to_utf8(Code, Len, 1, element(Len+1, LengthCodes), 16#FF),
       to_utf8(Code, Len, 2, 16#80, 16#BF),
       to_utf8(Code, Len, 3, 16#80, 16#BF),
       to_utf8(Code, Len, 4, 16#80, 16#BF)]).

to_utf8(Code) when Code < 16#80 -> to_utf8(Code, 1);
to_utf8(Code) when Code < 16#800 -> to_utf8(Code, 2);
to_utf8(Code) when Code < 16#10000 -> to_utf8(Code, 3);
to_utf8(Code) -> to_utf8(Code, 4).
```

Listing 1.3: The UTF-8 encoder with static conditionals.

```
to_utf8(Code) when Code < 16#80 ->
    [Code];

to_utf8(Code) when Code < 16#800 ->
    [((Code bsr 6) bor 16#C0),
     (Code bor 16#80) band 16#BF];

to_utf8(Code) when Code < 16#10000 ->
    [((Code bsr 12) bor 16#E0),
     ((Code bsr 6) bor 16#80) band 16#BF,
     (Code bor 16#80) band 16#BF];

to_utf8(Code) ->
    [((Code bsr 18) bor 16#F0),
     ((Code bsr 12) bor 16#80) band 16#BF,
     ((Code bsr 6) bor 16#80) band 16#BF,
     (Code bor 16#80) band 16#BF].
```

Listing 1.4: The “ideal” version of the UTF-8 encoder.

Chapter 2

Supercompilation

The word supercompiler is combined from SUPERvise and COMPILE. A supercompiler evaluates (compiles) a program symbolically, much like a partial evaluator does, but it is more powerful. The residual program can have a structure that is different from the original program, with new recursive functions that were not in the original, where redundant computations and memory allocations have been removed, and where even redundant passes over data may have been removed [45].

The supervision in a supercompiler means that it models what the program is doing, and then generalises from the models. In the words of the creator of supercompilation, Valentin F. Turchin:

> The concept of a supercompiler is a product of cybernetic thinking. A program is seen as a machine. To make sense of it, one must observe its operation. *[. . .]* The supercompiler concept comes close to the way humans think and make science. We do not think in terms of rules of formal logic. We create mental and linguistic *models* of the reality we observe. How do we do that? We observe phenomena, generalize observations, and try to construct a self-sufficient model in terms of these generalizations.[45]

The way of thinking that Turchin wrote about is called induction[1] and it has been behind the amazing progress in the natural sciences [13]. Turchin then goes on to list a number of things a supercompiler can be used for, including: *metacomputation*, creation of new algorithms, problem solving and proving properties of software. In comparison program optimisation may seem like a parlour trick.

The full implications of supercompilation are yet to be realised. The sentiment in [8] was that "[d]espite these remarkable contributions, supercompilation has not found recognition outside a small circle of experts." This seems to hold true to this day.

[1]This is induction in the epistemological sense, not mathematical induction.

2.1 Overview of supercompilation

The basic questions that determine the design of a supercompiler are: how to *drive* the program, when to stop and what to do then. In order to explain where the different parts of supercompilation fit in the whole, let us do a thought experiment.

Imagine that you are to simplify a functional program by hand. You start by writing down the initial expression on a paper. Then you apply the appropriate rules of evaluation, one by one. In the end you will have computed the output of the program. But what if you do not know what values to give some of the variables (i.e. some of the inputs are missing)? The solution to the exercise must be a new program.

You might proceed by applying the rules of evaluation, also using every trick you know to make the program better. Eventually you get stuck on a variable. How do you proceed? Write down the expression you are stuck on and proceed with one of the subexpressions instead, making a note of what expression it came from. When you are done with a subexpression you proceed to the next one. What you are doing is a type of driving, and the derivation is called a *process tree* [7].

How will you stop? Compare the expression you are currently working on with all those you have already worked on. If you notice that you have worked on the same expression before (although perhaps with different variables), then it is a good time to stop. What you have found is a *renaming*. Since you have already seen it before, you have already simplified it once, and therefore you have the code that implements it. To reuse that code you can make a new function and replace the occurrences of the expression with calls to that function. This is called *folding*. The arguments to the function will be the variables used in the respective expressions.

Finding a renaming is the easy case, however. Some programs will have you write down an endless series of ever growing expressions, none of which is a renaming of an earlier expression. Since you are doing this by hand you will probably recognise the pattern and stop. The pattern may be obvious to a person, but a computer program needs a specific method that tells it when to stop. The method most commonly used is the *whistle*, which will be explained in detail later.

The pattern is that new expressions keep appearing that look like old expressions, except that the new expression has some extra code in it, which most likely comes from a loop in the program. Perhaps the program has a loop that adds an element to a list for every iteration. Then you will see that the expressions you are writing down are the same, except when comparing an old expression to a later one the later one has added more elements to the list. This means that the old expression is *embedded* in the new expression.

When you found a renaming you created a new function. With these growing expressions things are not as simple, but the goal is more or less the same. This time the expressions are different, so you need to create a function that can implement both of them. What you need is a more general function, where the differences between the expressions can be passed as arguments to the function. This is called *generalisation* and will also be explained in detail later.

Armed with a generalisation of the old and new expressions, you can create a new function. The generalisation will not only give you an expression that can implement either of the two expressions, it will also tell you the differences between them. These differences can be used as arguments to the new function.

For a more formal description of this type of supercompilation, see [38] and [8]. There are major variations between supercompilers, but the steps described above are common elements.

A distinction is made between positive and perfect supercompilation. In a *positive* supercompiler information about what is *true* in a program is propagated [35, 39]. A *perfect* supercompiler also maintains negative information, i.e. it keeps track of what is *false* [33, 7]. This lets it remove more unnecessary tests from the residual program. The implementation of a positive supercompiler is somewhat simpler because the propagation of truths can be done through substitutions (using e.g. information from *let* and *case* expressions), whereas negative information is managed by constraints.

The rest of this chapter will go into more detail on how the supercompiler knows when to stop, how generalisations work and what kind of supercompilation is appropriate for Erlang.

2.2 The whistle

The whistle is a nickname given to what is formally known as the homeomorphic embedding relation, written as $e_1 \trianglelefteq e_2$. The supercompiler uses it to see if an old expression e_1 is embedded in the new expression e_2. It is inductively defined as follows:

Variable	Constant	Coupling	Diving
$x \trianglelefteq y$	$n_1 \trianglelefteq n_2$	$\dfrac{s_1 \trianglelefteq t_1, \ldots, s_n \trianglelefteq t_n}{h(s_1, \ldots, s_n) \trianglelefteq h(t_1, \ldots, t_n)}$	$\dfrac{s \trianglelefteq t_i \text{ for some } i}{s \trianglelefteq h(t_1, \ldots, t_n)}$

Table 2.1: Homeomorphic embedding.

An outline for an implementation of the whistle can be found in listing 3.7 on page 24. This definition is more or less the one seen in the literature (e.g. [19, 37, 28, 36]). To the untrained eye this definition seems rather complicated, but it is actually much simpler than it appears.

First of all, the definition only contains variables, constants and function calls, but it is meant to be valid for all types of expressions. The parts that make up a function call can be thought of as a name and some subexpressions. Whenever two function names are compared here, that means the operation the expressions perform should be compared (e.g. if one arithmetic expression uses addition, then so should the other one). Where function arguments are used in the definition, just use whatever subexpressions the two expressions have.

The first part, $x \trianglelefteq y$, merely means that if the two expressions are variables, then that is a homeomorphic embedding. The next one says the same for constants. The rule for coupling means that two function calls are embedded in each other, but only as long as they are calling the same function (here named h) and all their arguments are pairwise embedded in each other.

If this was all there was, then the expressions would need to be completely the same, except maybe with different variables and constants. But the diving rule means that parts of the second expression can be ignored. Here s is an old expression and h is again a function name. In English the rule says: if s is embedded in one of the function arguments, then s is embedded in the function call. This means that only one of the arguments is relevant. Also note that h is not used anywhere else, so it does not matter what it is.

$$
\begin{aligned}
\texttt{f(X)} &\trianglelefteq \texttt{f(X)} \\
\texttt{f(X)} &\not\trianglelefteq \texttt{g(X)} \\
\texttt{f(X)} &\trianglelefteq \texttt{f(Y)} \\
\texttt{f(X)} &\trianglelefteq \texttt{[f(X)]} \\
\texttt{f(X+1)} &\trianglelefteq \texttt{g(f(X+1),Y)} \\
\texttt{f(X+1)} &\trianglelefteq \texttt{f(g(X+1,Y))} \\
\texttt{X+1} &\trianglelefteq \texttt{(X+1)+1} \\
\texttt{X+1} &\not\trianglelefteq \texttt{X-1} \\
\texttt{[X|Xs]} &\trianglelefteq \texttt{[A,X|Xs]} \\
\texttt{g(X*2,Y-1)} &\not\trianglelefteq \texttt{g(X,Y-1)}
\end{aligned}
$$

Table 2.2: Examples and non-examples of homeomorphic embedding.

When all of these rules are put together the result is a test that determines if an expression is embedded in another expression. The relevance to supercompilation comes from the diving part, that can basically ignore extra list items, extra function calls, extra additions from loop counters, and so on, in the second expression. So the whistle tests if two expressions are the same, but allows the second one to contain the kind of extra expressions that one typically sees in functional loop constructs.

These expressions will start to appear if the supercompiler is driving a loop that is in turn making the driving itself loop. Unfortunately the whistle is conservative [2] and sometimes the whistle will stop the driving algorithm even though it would have stopped on its own. When *the whistle blows* the supercompiler generalises the expression it is working on against the old expression, and as a result it loses some of the context of the expression. The result is that the residual program will be worse than it would otherwise have been, and getting a good residual program depends on deriving good generalisations.

2.3 Generalisation

What type of code triggers the whistle? Listing 3.6 on page 23 shows an implementation of list reversal that needs the whistle for driving to stop. One might say that the code is starting to repeat itself. The driving is stopped by:

$$\texttt{rev(Xs0, [X0|Ys])} \trianglelefteq \texttt{rev(Xs1, [X1,X0|Ys])}$$

The old expression to the left is embedded in the newer expression. This is not a renaming since the newer expression has an extra element `X1` in the list. The old expression does not have that element. If the driving is not stopped then new elements will keep being added. One way the supercompiler stops is by finding a renaming of an older expression, so how how can that be achieved in this case?

Suppose that the supercompiler already has created a function that implements the older expression. It does not matter what the implementation is, we only need to know that it exists. If the driving is to stop because of a renaming then the new expression should (at some point) be able to use that function.

This is a problem that should be familiar to most programmers. Think of the old expression as a library function, and for some reason or other, you have to write a new function that is expressed in terms of the library function. In the case of a supercompiler the library functions can rather strange. In this example you are given this library:

```
rev([],Ys) -> Ys;
rev([X|Xs],Ys) -> rev(Xs, [X|Ys]).
h(Xs0, X0, Ys) -> rev(Xs0, [X0|Ys]).
```

The actual implementation of `h` may be different, but the semantics must still be those of the old expression. How would you rewrite `rev(Xs1, [X1,X0|Ys])` in terms of `h`? This is the problem of generalisation, a difficult and very important problem in supercompilation [46].

The method of generalisation most commonly used today by positive supercompilers is based on *the most specific generalisation* (*msg*), which basically takes two expressions and finds differences between them [37]. This results in a more general expression and two *substitutions*. The more general expression contains new variables and the substitutions contain an entry for each variable. The expression given by the msg of the old and new expressions in our running example is:

```
rev(V0, [V1|V2])
```

Together with the substitutions:

```
V0 = Xs0,  V1 = X0,  V2 = Ys
V0 = Xs1,  V1 = X1,  V2 = [X|Ys]
```

By using the substitutions the more general expression can be turned back into one of the original expressions, but that is not the purpose of the substitutions in the case of supercompilation. Notice that the more general expression is a renaming of the old expression, the one implemented by `h`. If the supercompiler drives the more general expression it will now terminate.

In some programs the two expressions that triggered the whistle can not be generalised very well by the msg, e.g. because one of the expressions is a function call while the other one is a list. In this case the fallback strategy is to *split* the expression, which means to turn all its subexpressions into new variables.

The quality of the generalisations determine the quality of the residual program. When an expression is generalised some of the context of the expression is lost, and it is that context the supercompiler uses to perform its transformations. There are better generalisations available [30], but the msg is a very common one and illustrates a solution the problem rather well.

2.4 Supercompiling Erlang

There are a number of supercompilers described in the literature, but is there one that can be used to supercompile Erlang? Much of the recent research on supercompilation has been for lazy (call-by-need or call-by-name) languages [31, 2, 27]. Erlang has call-by-value semantics, so it is not obvious that these supercompilers can be used. One of the important differences between these evaluation orders is that they act differently with regards to termination. A program that goes into an infinite loop under call-by-value semantics might terminate under call-by-need semantics. Thus if the wrong supercompiler is used for Erlang the residual program may have a different meaning than that of the original program.

One supercompiler that handles call-by-value is the Java supercompiler [24, 25]. There are a few concerns that make it inappropriate for Erlang. One of its important characteristics is that it handles mutable objects, which Erlang does not have. Unlike most supercompilers it does not generate new recursive functions (or methods), and generally appears to be a poor fit for a functional higher-order language. The algorithm is also not presented fully, so it may be difficult to replicate, especially when targeting such a different language.

Another possibility is to take one of the supercompilers from the literature and adapt it to call-by-value semantics. In [2] the authors say that doing this with their supercompiler "should be straightforward". There is some uncertainty as to whether this would work. But in making their remark the authors point us in the direction of a supercompiler for a call-by-value language.

Jonsson and Nordlander have published a positive supercompiler that is meant for higher-order call-by-value language [21, 19]. The basic ideas used in their supercompiler are as those previously described in this chapter, but with some differences. The driving part of the algorithm and the building of the residual program is intermingled in such a way that no process tree is actually built. This is a little bit as if the supercompiler itself had been supercompiled beforehand: the driving produces a process tree and the building of the residual program consumes it, but now the intermediate data structure (the process tree) has been eliminated.

One possible problem is that the language their supercompiler was designed for is pure, i.e. free from side-effects. Erlang has a few side-effects, such as message passing, and it is difficult to say what this supercompiler algorithm will do with them (they might be e.g. removed or reordered). But if one keeps to a pure subset of Erlang, this problem can be eschewed. Now a supercompiler algorithm has been chosen and the next chapter will go into detail on its implementation.

2.5 Summary

Supercompilation is a powerful but relatively unknown program transformation technique capable of creating new recursive functions and removing many types of redundancies in programs. Its operation has been compared to inductive thinking.

When a supercompiler drives a program it creates what is more or less an potentially infinite tree of subexpressions, a process tree. When the tree has two expressions that are renamings of each other a new function is created and the expressions are replaced by function calls. This is often enough to make the tree finite, i.e. make the driving terminate properly.

For some programs renamings are not enough. The whistle tells you when an expression in the process tree is remarkably similar to one that came before it. When this happens it is most likely a good idea to stop, because those expressions are probably part of a loop, which the supercompiler otherwise might go on driving forever. The whistle is however conservative, and will sometimes stop driving too soon, and the residual program will not be as good as it should be.

When the whistle blows the current expression should be generalised, so that those parts that do not cause it to loop can be improved. The basic problem is to determine why a series of expressions is growing and factor out that problem from the current expression. A common method, the most specific generalisation, tries to extract all differences between the expressions. If that fails the fallback is to split, i.e. extract all subexpressions instead. A problem is that not all differences are the cause of the looping and splitting is even more approximative.

Chapter 3

Implementation

The positive supercompiler algorithm from [19] was chosen for this work. This chapter describes the implementation of the supercompiler. A general view of where the supercompiler fits into the existing Erlang system is given. Then the question of how the supercompiler algorithm can be adopted to Erlang is addressed. A few of the basic tools that are needed by the implementation are shown, followed by an incremental development of the main part of the supercompiler. It starts out with a simple structure and then more and more features are added.

3.1 Manipulating the program

Supercompilation is a form of partial evaluation and the algorithm chosen for this thesis is structured very much like a program evaluator. It will take as input an Erlang program and produce a new Erlang program, a *residual program*, as its output. As such it will need a way of parsing source code files and turning parsed code back into text.

There are many ways to do this. If one was starting from scratch then it would be necessary to write an Erlang parser. This parser would not need to be written in Erlang. Indeed, the whole supercompiler could be implemented in any language, but there are certain benefits to implementing a language in itself. The semantics of the language that is being evaluated are more readily available.

In the case of Erlang there are further benefits to writing the supercompiler in Erlang. There are library modules for parsing and working with parsed Erlang code. A particularly interesting feature of Erlang is that modules can specify another module to be used as a *parse transform.* When the compiler is compiling a module it first parses the source code and then lets the parse transform change the code in any way it likes.

The parse transform receives a list of *forms.* Each form represents a top-level construct of an Erlang module, e.g. an export attribute or a function declaration. The forms are tuples where the first element names what type of form it represents. The second element contains the line number for the form. The rest of the elements depend on what type the form has, e.g. function declarations have a name, an arity and a list of clauses.

When the parse transform is finished it returns a list of forms. This list is used by the compiler instead of the original forms. It is possible to view the new program by passing the *-P* flag to the *erlc* compiler. The net effect of making the supercompiler a parse transform is that it can be used as a new compiler pass.

The supercompiler has been written in a functional programming style, i.e. without the use of side-effects. *Environment records* are used to pass around information. These are used instead of global variables, processes that keep state, or something similar. The records contain information on what expressions have been seen by the supercompiler, what variables names have been used, the definitions of top-level functions, etc.

The supercompiler algorithm is itself defined in a functional style, so the environment records match the algorithm's information flow quite well. Most functions in the supercompiler have an environment record both as input and output. One reason to return a new environment record is if a new variable was used, and it needs to be different from every other variable in the program. Another reason has to do with Erlang's variable scoping rules, which allow variables declared inside expressions to be bound outside of them.

3.2 Adapting a supercompiler to Erlang

The supercompiler algorithm chosen for this thesis is the algorithm from Peter A. Jonsson's doctoral thesis [19]. Erlang is not exactly the same as the language used in the published algorithm. Looking beyond mere surface syntax, Erlang is a bigger language and offers different semantics for many constructs. In order to implement the supercompiler algorithm for Erlang it is necessary to understand the algorithm and adapt it to Erlang.

The language in the published algorithm has very simple scoping rules. A variable introduced in an expression has a scope that does not reach outside of the expression. This is true for some Erlang expressions, but not for others. The most visible case when this is true is for *match* expressions, which look like they might behave like variable assignment from C-like languages. A less obvious example is when a variable is defined by every clause of a *case* expression. After the *case* expression has been evaluated new variables may have become bound. The supercompiler needs to know the current set of bound variables, and therefore the implementation has to pass this information upwards, by returning a new environment record.

Variables can not be rebound in Erlang. If a bound variable occurs in a pattern of e.g. a *match* expression or a *case* clause then that part of the pattern must match the old value of the variable. If the value does not match then the pattern does not match, and if there are no other patterns to try then the result is a runtime error. The supercompiler should preserve these errors in the residual program, which it can do rather simply by keeping track of the bound variables, and preserving the original code when necessary. It should be noted that while variables can not be rebound, they can be *shadowed.* Variables used in the patterns of anonymous functions and list comprehensions are fresh variables and have nothing to do with other variables of the same names.

Much of the power of the supercompiler comes from rules that work with pattern matching. The pattern matching in Erlang is very powerful and can be difficult to handle. The patterns themselves can have complications such as repeated variables, matches or unusual data types. Alongside a pattern there may be a guard sequence. These are expressions in a limited side-effect free language. If the supercompiler is to produce improved code it must be able to statically determine if patterns and guard sequences will match a given expression.

Somewhat simpler to handle are some of the extra functional constructs that Erlang brings. The built-in append operator can be translated into a function call. List comprehensions are missing from the language used in the published algorithm, but they are easily translated into simpler terms. The technique used for this is the simplest one presented in [18]. There are better methods described, but they are basically manual optimisations of the simpler methods. The supercompiler should be able to automatically derive the better solutions.

The supercompiler algorithm uses *letrec* expressions. These expressions are used to define local recursive functions. There is no proper *letrec* in Erlang, as named functions can only exist at the top level of a module. The implementation strategy here has been to construct artificial *letrec*s that are later merely moved to the top level. Due to how the supercompiler algorithm works these local recursive functions never have any free variables. Therefore this strategy does not require lambda lifting [16].

Erlang has a number of side-effects, such as message passing. The supercompiler algorithm does not have anything in place to handle side-effects. They are outside the scope of this thesis. The problem is that the supercompiler may rearrange computations, so that the original evaluation order of the program is not preserved. Therefore side-effects may also be rearranged, e.g. the residual program might try to receive the reply to a message that has not yet been sent. This may not be as big of a problem as it seems. Armstrong recommends that Erlang programs be split into functional and non-functional parts [1]. If the modules of a program are divided in this way it will be easier to supercompile only the purely functional modules.

Erlang and the language in the published algorithm are of a very similar philosophy. Both are functional languages with call-by-value semantics (optional in the case of the research language). Both use pattern matching. Even though Erlang uses it much more extensively, the basic elements are the same. As will become apparent later, where Erlang's semantics become too difficult, the supercompiler can merely ignore the difficulty and go on with rest of the program. This means that the implementation can be developed incrementally and tested at each step of the process. These are the reasons why Jonsson's supercompiler algorithm was chosen for this thesis.

3.3 Tools for working with expressions

When attempting to solve a problem it is generally a good idea to first divide it into smaller problems. In the case of a program transformer some of those smaller problems are common enough that they are standard. The *erl_syntax* and *erl_syntax_lib* modules provide functions for walking over syntax trees in various ways, e.g. while building a new syntax tree or gathering data. While they provide some elementary tools, they are curiously missing others.

Most programming languages use named variables and Erlang is no different.

When the supercompiler creates a new variable it must be guaranteed to be different from every other variable in the program. The function that returns a new fresh variable is called *gensym*. The environment record contains the set of all previously used variable names and *gensym* merely picks a name that is not in the set. The variable names it creates are made up of two parts: a prefix and a unique id. The prefix is normally the name of an existing variable. Keeping the prefix separate makes it easy to ensure that variable names do not grow ever longer (fresh variables can be used as a prefix over and over again), and it makes it easier to reconstruct the original variable names.

Another common operation related to variables is *free variable analysis*. This analysis is used to determine which variables used inside an expression are bound outside of it. The implementation is based on the analysis in *erl_syntax_lib*. The only quirk is that it returns the variables in a stable order. One may think of the order as being defined by the order in which a non-random tree search algorithm will encounter each free variable, meaning that if the free variables of `f(X,Y)` are `[X,Y]`, then free variables of `f(Y,X)` are `[Y,X]`, and not the other way around.

When the supercompiler creates new functions it turns the free variables in the original expression into function arguments. If those variables were returned in a different order next time then the new function would be called with its arguments in the wrong order. Other implementations are certainly conceivable that would not require a stable order.

Alpha conversion rewrites code so that it uses fresh variable names. This operation is very common in program transformers and makes a lot of other analysis simpler. The benefits from alpha conversion come from the fact that if two variables are originally named the same, but are actually different according the scoping rules, they will be assigned different names. This is commonly used when "cutting and pasting" code, e.g. when inlining a function.

The implementation dispatches on the expression type. When an expression introduces a new variable the alpha converter generates a fresh variable and builds a substitution environment. Whenever it sees a variable reference it checks the substitution environment to see if there is a fresh variable that should be used in its place. The substitution environment is returned along with a new expression, which is the same as the original, except the names have been changed.

Alpha conversion for Erlang is a bit complicated due to the scoping rules. But for some of the later analysis alpha conversion means that the scoping rules have already been implemented and can be ignored. The information flow in the alpha conversion pass must match the scoping rules, and for Erlang some information is passed upwards, e.g. when a variable is defined by every clause in a *case* expression. The great number of expression types is another complication, especially since many of them can introduce new variable names.

3.4 How to build an Erlang supercompiler

The tools described in the previous section are going to be familiar to those who have worked on compilers before. There are some tools that one would normally not encounter outside of a supercompiler, and it is best to explain them in their proper context. Let us therefore write a supercompiler from basic principles, and explain the rest of the tools as they become needed.

```
drive(Env, Expr, R) ->
    build(Env, Expr, R).
build(Env, Expr, []) ->
    {Env,Expr}.
```

Listing 3.1: A supercompiler that stops immediately. These rules are default fallthrough rules that will be in the final supercompiler.

3.4.1 Driving the program

The main part of the supercompiler is the driving algorithm (see section 2.1). One might think of the supercompiler as a program interpreter that is allowed to stop at any point and return a new program (whereas a proper interpreter would attempt to return a value). If the supercompiler has access to all of the program's inputs it might actually run the program until it returns a value, but in principle it can stop anywhere along the way. Listing 3.1 shows that principle in action. The main entry point is the *drive* function and it implements the two fallthrough rules from Jonsson's supercompiler.

The relationship between the *drive* and *build* functions is similar to that of *eval* and *apply* in a tree interpreter. The *eval* function walks over the syntax tree of the code, just like *drive* walks the tree. When *eval* encounters a function call it uses *apply* to compute the value of that call. In a somewhat similar manner, the *drive* function calls *build* to compute a residual program. In order to do that, *build* may need to call *drive*, just as *apply* often needs to call *eval*.

3.4.2 Reduction contexts

The *drive* function takes an environment record *Env*, a parsed code expression *Expr* and a reduction context *R*. The job of the reduction context is to keep track of what code will consume the return value of the current expression. This is similar to, but not the same as, a return stack or a continuation.

A reduction context is an expression with a *hole* that indicates where the return value (or residual code) of the expression currently being driven should be placed. This is represented using the same technique as [19]. The contexts are implemented as lists of records, with one record type for each of the possible expression types a context can contain.

An alternative implementation would have been to merely use the symbolic form of the consuming expression, but it is easier to pattern match on a flattened list. The empty context is represented by the empty list, and its meaning is that the consumer of the return value is unknown or uninteresting. This has already been implemented in listing 3.1. The result is merely that when *build* is given an expression and an empty context it returns the expression as is.

The implementation uses five types of reduction contexts: binary operators, unary operators, function calls, case expressions and match expressions. They all include as much information as is necessary to rebuild the expression they represent (including source line information), except for the subexpression that goes in the hole.

Listing 3.2 shows two reduction contexts, each containing only one expression. The first context represents a call to some function taking two arguments.

```
<hole>(Xs, Ys).

case <hole> of
    [] -> [];
    [X|Xs] -> [X|ap(Xs,Ys)]
end.
```

Listing 3.2: Examples of reduction contexts. These are used for the third argument of *drive* and *build*. In the implementation they are represented by lists of records that contain parsed Erlang code.

```
drive(Env, {op,L,Op,E1,E2}, R) ->
    drive(Env, E1, [#op_ctxt{line=L, op=Op, e1=hole, e2=E2}|R]);
drive(Env, Expr, R) ->
    build(Env, Expr, R).
build(Env0, Expr, [#op_ctxt{line=L, op=Op, e1=hole, e2=E2}|R]) ->
    {Env,E} = drive(Env0, E2, []),
    build(Env, {op,L,Op,Expr,E}, R);
build(Env, Expr, []) ->
    {Env,Expr}.
```

Listing 3.3: Focusing on and building binary operator expressions.

The hole of this expression will usually be filled with the name of a function.

The second reduction context contains a *case* expression. Imagine if the expression that goes in the hole explicitly builds a list. It is a simple enough job to residualise the code that has already destructured the list into `X` and `Xs`. Because of the way in which the supercompiler passes around the reduction context this opportunity can arise even if the hole did not contain the list constructor explicitly in the source code text.

3.4.3 Focusing creates contexts

Reduction contexts are created by the subset of driving rules that *focus* on subexpressions. These are rules that, looking at an expression, pick one subexpression to continue driving on and place the rest of the expression in the reduction context. Listing 3.3 shows a new rule for focusing on the first operand of a binary operator (e.g. focusing on `E1` in `E1+E2`).

There is also a new *build* rule for when the context is a binary operator. There needs to be a build rule for every context type. This new rule is called by the fallthrough rule of *drive*, i.e. when nothing more interesting can be done with `E1`. The rule drives `E2`, which was saved in the reduction context, and finally builds a binary operator expression using the rest of the context `R`. Note that it drives `E2` in the empty context, meaning that it wants to get back an expression that can be used for `E2` in the residual program.

```
...
drive(Env, {'call',L,{atom,La,N},Args}, R) ->
    drive(Env, {'fun',La,{function,N,length(Args)}},
          [#call_ctxt{line=L, args=Args}|R]);
drive(Env, {'call',L,F,Args}, R) ->
    drive(Env, F, [#call_ctxt{line=L, args=Args}|R]);
...
build(Env0, Expr, [#call_ctxt{line=Line, args=Args0}|R]) ->
    {Env,Args} = drive_list(Env0, fun drive/3, Args0),
    build(Env, scp_expr:make_call(Line,Expr,Args), R);
...
```
Listing 3.4: Focusing on and building function calls.

3.4.4 Driving function calls

Any program that has been passed through the latest iteration of the supercompiler has come out unchanged. There are a few more rules of this sort that form the infrastructure of the supercompiler. Things start to get more interesting when the rules for function calls are in place. The support is provided in three parts, of which the first are the rules that focus on function calls and rebuild them. These are shown in listing 3.4. Erlang provides two ways of writing function calls, so there is an extra rule for that case. The new *build* rule uses a utility function that drives a list of expressions in the empty context, and another utility that residualises function calls[1].

The new *drive* rules focus on the expression in the operator position, which is usually the name of a global function. There is another *drive* rule that triggers when the current expression is the name of a global function. It looks up the definition of the function in the environment record. If successful it starts inlining the function, otherwise it calls *build* which residualises a function call.

The inlining happens in a few steps. The name of the global function is *plugged* into the context, meaning the context is turned into an expression without a hole. The free variables of the plugged context are extracted and a fresh function name is generated. The function definition is then alpha converted and driven in the current context.

There are now two versions of the same thing: the first has the function name in the hole, and the second has the function definition in the hole. The first expression (the plugged expression) is remembered and associated with the fresh function name. The second one is used for the definition of the fresh function. With some of the changes described later it will be possible for the fresh function to call itself, so it is placed inside a *letrec*. The body of the *letrec* is a call to the function and the free variables are used as arguments.

In order to do anything interesting when driving the function definition in the current context there must be a rule for driving function expressions in function call contexts. In [19] the equivalent rule expands into a primitive *let* form. The Erlang implementation needs to handle the fact that functions can do pattern matching on arguments. It does this by considering, for each of the formal arguments, which ones can be placed in *let* expressions and which

[1]The *make_call* function will later be extended to handle constructors.

ones must go inside a *case* expression. The *let* expressions are implemented as function calls: *let Lhs=Rhs in Body* becomes `(fun (Lhs) ->Body end)(Rhs)`.

3.4.5 Improved driving of function calls

Let us consider what the supercompiler has at this point. It has found a function call and has turned the context into an expression. It has then made a new function that implements that expression, and is equivalent to it, except that it contains the function definition instead of the function name. Then it placed that new function in a *letrec* and residualised a function call. The important point to remember is that the supercompiler now has an expression and the name of a function that implements that expression.

The supercompiler can now be improved by making a slight change to the driving of function calls. After plugging the function name into the context it will look for *renamings* of the expression. A renaming is an old expression which is equal to the expression, except that the variables may be named differently. If it finds a renaming that means it also knows the name of a function that implements the expression. Instead of going through the routine of making another fresh function it can merely residualise a call to the function that implements the old expression. The arguments to the function are the free variables of the new expression (the plugged context). The idea is shown in listing 3.5.

With this improvement the new functions that are created might become self-recursive. This happens if, when driving an expression, a renaming of that expression is found (see section 2.3 for a description of the benefits of this). Another improvement to the supercompiler becomes obvious on testing: sometimes the new functions are not self-recursive and the *letrec* is not needed. The fix is simple. Put a note in the environment record whenever a renaming is found, remembering the name of the function. Before building the *letrec* check if the current fresh function name was used in any renamings.

The implementation of the renaming relation $e_1 \equiv e_2$ is based on that of Jonsson. The algorithm keeps a work list that contains tuples of expressions (initially set to $[\{e_1, e_2\}]$). It takes tuples off the list, checking that they have the same type (e.g. both are function calls, or the same constant). If they do not have the same type then obviously there is no renaming. Subexpressions are added to the work list. If the two expressions are variables then a substitution from the first to the second variable is created and applied to the rest of the work list. The substitutions are accumulated and finally applied to the e_2. If the result is equal to e_2 then there is a renaming.

3.4.6 Ensuring termination

The improved supercompiler will terminate for many programs, but there are programs which it will keep driving forever. These are programs where a renaming never happens, in particular because some parameter in the program is growing. The result is an infinite stream of expressions that only change slightly, but enough so that there are no renamings. An example is shown in listing 3.6.

There are a few approaches one can take to this problem. The driving algorithm could be limited to a specific number of steps. Since the supercompiler always can stop safely an arbitrary limit can be set after which the supercompiler simply returns the program as it looks at that point.

```
drive_call(Env0, Funterm, Line, Name, Arity, Fun0, R) ->
    L = plug(Funterm, R),
    FV = free_variables(Env0#env.bound, L),
    case find_renaming(Env0, L) of
        {ok,Fname} ->
            Expr={'call',Line,{atom,Line,Fname},
                   [{var,Line,X} || X <- FV]},
            {Env0#env{found=[Fname|Env0#env.found]},Expr};
        _ ->
            {Env1,Fname} = gensym(Env0, Env0#env.name),
            {Env2,Fun} = alpha_convert(Env1, Fun0),
            Env3 = Env2#env{ls = [{Fname,L}|Env2#env.ls]},
            {Env4,E} = drive(Env3, Fun, R),
            {Env5,S} = fresh_variables(Env4, dict:new(), FV),
            Head = [subst(S, {var,Line,X}) || X <- FV],
            Body = subst(S, E),
            NewFun0 = {'fun',Line,
                        {clauses,[{clause,Line,Head,[],[Body]}]}},
            NewTerm = {'call',Line,{atom,Line,Fname},
                        [{var,Line,X} || X <- FV]},
            {Env6,NewFun} = alpha_convert(Env5, NewFun0),
            Letrec = make_letrec(Line,
                         [{Fname,fun_expr_arity(NewFun),NewFun}],
                         [NewTerm]),
            {Env6#env{ls=Env2#env.ls},Letrec}
    end.
```

Listing 3.5: Driving of named functions with a check for renamings.

```
%% Given this definition of rev/2:
rev([],Ys) -> Ys;
rev([X|Xs], Ys) -> rev(Xs, [X|Ys]).
%% During driving these expressions are generated:
rev(Xs0, [X0|Ys])
rev(Xs1, [X1,X0|Ys])
rev(Xs2, [X2,X1,X0|Ys])
rev(Xs3, [X3,X2,X1,X0|Ys])
%% Ad infinitum.
```

Listing 3.6: The reverse function gives a series of expressions that keep growing.

```
whistle(E1, E2) ->
    peel(E1, E2) orelse
        lists:any(fun (E) -> whistle(E1, E) end,
                  the subexpressions of E2).

peel(E1, E2) when E1 and E2 have different types -> false;
peel(E1, E2) when E1 and E2 have different number
  of subexpressions -> false;
peel(E1, E2) when E1 and E2 are different named
  functions -> false;
peel(E1, E2) when E1 and E2 are constants -> true;
peel(E1, E2) ->
    Es1 and Es2 = subexpressions of E1, E2,
    lists:all(fun ({E1,E2}) -> whistle(E1, E2) end,
              lists:zip(Es1, Es2)).
```

Listing 3.7: The whistle in Erlang pseudo code.

A more common technique used with supercompilers relies on the *homeomorphic embedding* relation $e_1 \trianglelefteq e_2$, also known as the *whistle*. This method uses the same list of expressions as used by the renaming check and is done after that check. Its job is to find old expressions that are the same as the new expression, except that the new expression may have additional code wrapped around some subexpression (in listing 3.6 the additional code is an extra element in the list).

The implementation of the whistle is similar to its definition (see section 2.2). An outline of an implementation is shown in listing 3.7. The *whistle* function first tries to match up the expressions with *peel*. If that fails it checks if the whistle blows on a subexpression of `E2` instead. This is the part of the algorithm where it can accept an additional element inside of a list, or some other growth.

The whistle is a conservative approximation to the problem of making the supercompiler always terminate [2]. In other words, it sometimes happens that the whistle finds an old expression embedded in a new expression, but the driving would have terminated on its own anyway. When this happens the quality of the residual program is not as good as it would otherwise have been.

3.4.7 Improved driving on early termination

The whistle is guaranteed to trigger on all programs that the supercompiler would otherwise have driven forever[2] and with the addition of the whistle the supercompiler will always terminate. At this point it is safe to merely return the new expression.

If the whistle triggers then the new expression (the context with the function name plugged into it) is the same as an old expression, except it contains an extra list item, or an extra function call, or perhaps an extra addition. This difference between the expressions is where the growth comes from and it is the

[2]The whistle may trigger even though the supercompiler would have terminated. See section 2.2.

reason why the supercompiler would not have terminated. Instead of returning the new expression verbatim, a better idea is to remove the growing part and keep driving. The supercompiler will still terminate and it may even find more improvements.

So the problem becomes how to remove the part of the code that is growing. The supercompiler has a new expression and an old one, and placed side-by-side you can see where the growth is (e.g. an additional item in a list, as in listing 3.6). The way supercompilers extract the differences between expressions is by using the *most specific generalisation* (msg). If the msg generalises the whole expression to a variable they do a *split* instead.

The *msg* as used in [19] takes two expressions and returns a new expression and a substitution, `{E3,S} = msg(E1,E2)`. In the new expression the differences between `E1` and `E2` have been replaced by fresh variables. The variables can all be found in `S`, and if `S` were to be applied to `E3` one would get back `E1`. This means the growing parts of `E1` are all gone from `E3` and can instead be found in `S`. If `E1` is the code `foo(X,bar(Y))` and `E2` is `foo(X,Y)`, then `E3` is `foo(X,V1)` and `S` maps `V1` to `bar(Y)`.

The idea is to drive `E3` and each of the expressions from the substitution `S`, and then put the whole thing back together by applying the new `S` to the new `E3`. This means that the differences between `E1` and `E2` are driven separately and therefore the supercompiler will not create forever growing code.

If `E1` and `E2` are different types of expressions (perhaps `E1` is an addition and `E2` is found somewhere in one of its operands), then the whole of `E3` will be a fresh variable. This is a problem because then `S` will contain `E1` verbatim, and that is the expression that triggered the whistle. The idea was to take out the growing parts from `E1`, but now the supercompiler got `E1` back. The alternative to *msg* is the *split*, which replaces all subexpressions of `E1` with fresh variables. If `E1` is `foo(X,bar(Y))` then `E3` would be `V1(V2,V3)` and `S` maps `V1` to `foo`, `V2` to `X` and `V3` to `bar(Y)`. These alternative versions of `E3` and `S` are then handled as those returned by *msg* were handled.

The fresh variables created by *msg* and *split* can refer to any kind of expression whatever. For this reason the environment record contains a set of all currently used such variables. This is then used in e.g. the function that returns true if an expression is guaranteed to terminate. Normally a variable reference would always terminate, but this is not necessarily true if the variable comes from *msg* or *split*.

The implementation of *split* is straightforward. This is a non-recursive function that dispatches on the expression type. For a function call it replaces the operator and the operands with fresh variables and creates the appropriate substitution. The implementation of *msg* is only slightly trickier. If the expressions are exactly the same then it returns the expression and an empty substitution. There is a default case for when the expressions are of different types, and here it replaces the whole expression with a variable. If the expressions do have the same type (e.g. both are a function call to the same function), then it calls *msg* recursively on the subexpressions and collects the differences between them while accumulating a substitution.

Both the *msg* and the *split* are approximations and they can be more aggressive than what is strictly necessary to get rid of the growing code. The *msg* compares two expressions, but it does not have any special knowledge of what differences are crucial to stopping the infinite growth. The *split* truly has no idea at all, it merely extract subexpressions. In our implementation we have, at the suggestion of Jonsson, added a special case to *split*. The special case is for append, which triggers the whistle unnecessarily, and it turns only the second argument into a fresh variable. This was necessary to get a nice result for the string to UTF-8 encoder.

3.4.8 Deforestation

In the introduction chapter it was claimed that a supercompiler can derive the ideal version of the UTF-8 encoder. There are a few pieces missing before the supercompiler can do that. The full supercompiler has rules for handling *case* expressions, rules that eliminate or improve them. It also has rules for focusing on the *case scrutinee*. In listing 1.3 the interesting *case* expression comes from the definition of `lists:flatten`.

Let us look at what is needed in order to eliminate the intermediate lists from the UTF-8 encoder in listing 1.3. Listing 3.8 starts with the focusing rule for *case* expressions. The *case* expression is added to the context and the supercompiler goes on to drive the scrutinee expression. After driving the scrutinee a while it will find a list constructor, which is handled by the second rule. The *cons* expression is handled like a function call, even though *cons* is primitive syntax in Erlang. The *make_call* function recognises the special constructor notation and creates a *cons* rather than a function call.

The third rule in listing 3.8 is triggered when the supercompiler has focused on a *cons* and the reduction context contains the arguments to the *cons* and a *case* expression. The rule tries to find the clause that matches the constructor (taking any guards and other complications into consideration). If it finds a matching clause then there are likely going to be a few variables in the pattern of that clause, and they need to be bound to the corresponding parts of the constructor expression (e.g. the pattern `[X|Xs]` would bind `X` to the `Head` expression and `Xs` to the `Tail` expression). It does this by constructing a *let* expression that it then drives.

The UTF-8 encoder in listing 1.3 only encodes a single codepoint. During the testing of the supercompiler the actual program used has been `flatten(map(fun to_utf8/1, S))`, using local definitions of `flatten` and `map`. The *case* expressions come from code inside `flatten` and `map`. The *case* expressions are added to the context, which is passed around until it reaches the *cons* expressions in `to_utf8/1`. This means that the *case* and the *cons* will meet and trigger the third rule of listing 3.8, even though they may not be in direct contact with each other in the source code of the program.

The supercompiler also has a rule for rebuilding *case* expressions. The code for this rule does not make for great viewing, but there is an important detail that deserves to be mentioned. The build rule for *case* expressions drives the bodies of the clauses in the surrounding reduction context. This means that it can move code that was previously on the outside of the *case* expression (i.e. code that was going to use the value returned by the *case*) to inside the clauses of the *case*. This lets the supercompiler push the consumer of a value (e.g. a *case* expressions) even closer to the producer (e.g. a *cons* expression).

If the third rule of listing 3.8 is triggered then the residual program contains one less constructor call. This is how the supercompiler can remove intermediate lists from a program.

```
drive(Env, {'case',Line,Expr,Clauses}, R) ->
    drive(Env, Expr, [#case_ctxt{line=Line, clauses=Clauses}|R]);
drive(Env, {cons,Line,Head,Tail}, R) ->
    drive(Env, {constructor,Line,cons},
          [#call_ctxt{line=Line, args=[Head,Tail]}|R]);
drive(Env, E0={constructor,L,Cons},
      Ctxt=[#call_ctxt{args=Args},
            #case_ctxt{clauses=Clauses}|R]) ->
    E = make_call(L, E0, Args),
    case find_constructor_clause(Env#env.bound, E, Clauses) of
        {ok,Lhss,Rhss,Body} ->
            drive(Env, make_let(L, Lhss, Rhss, Body), R);
        _ ->
            build(Env, E0, Ctxt)
    end;
```

Listing 3.8: Driving rules that work on *case* expressions.

3.4.9 Pattern matching

The rule that handles known constructors in *case* expressions is simple enough if the pattern matching in the language is simple. In Erlang however the pattern matching is in general not so straightforward. The current implementation handles pairs, tuples, constants and free non-repeated variables.

If a variable appears more than once in a pattern then there is an implicit assertion at runtime that forces the two occurrences to be equal. These assertions are similar to explicit matches ($X{=}Y$), which can also occur in patterns. Bound variables in patterns also result in an implicit assertion. A related difficulty is guard sequences that reference pattern variables.

Pattern matching in Erlang is pervasive. The syntax that looks like assignment ($X{=}Y$) is actually pattern matching. The supercompiler handles these by transforming them into *case* expressions (thus reusing the logic for *case* expressions). Where this is not possible it places them in the reduction context, since the pattern acts as a consumer of values. Another place where pattern matching shows up is in function arguments (the handling of which was discussed in subsection 3.4.4). Pattern matching also shows up in list comprehensions. These require no special logic in the driving algorithm because the supercompiler translates all list comprehensions into simpler code before it starts driving.

The implementation of pattern matching for *case* expressions works by repeated simplifications. The simplifications are more or less eliminating constants or extracting bindings. A simple partial evaluator processes any guard sequences. The algorithm has access to the scrutinee and the clauses, and tries to eliminate the scrutinee, select one clause and return a set of bindings. Those patterns which are difficult to handle are safely ignored, at the cost of worse residual code.

3.5 Tidying up

The code that comes back from the driving function may be an improvement over the original, but it is also likely to be more difficult to read. For a program transformer to be an effective tool the programmer must know how to use it, which means finding out what the tool has actually done. To this end the implementation has a module that tries to make the residual program more readable.

The fresh variables that *gensym* creates make the program more difficult to read, because they contain a unique identifier. All top-level functions in Erlang are closed, meaning the complete set of fresh variables used in a function are also defined inside that function. All that is necessary is to find the complete set of fresh variables in a function and give each one a better name.

Top-level functions in Erlang have multiple clauses, each one with its own patterns, guards and body. To make the implementation of the supercompiler simpler these are translated into *case* expressions. After the function has driven these *case* expressions should be lifted back into the top-level function. The pass that does this looks for top-level functions that only contain a *case* expression and does the necessary rewriting.

The bodies of function clauses and the bodies of *case* expression clauses are lists of expressions. These are easier to handle if they are rewritten into explicit sequencing expressions that only contain two expressions. After the supercompiler is finished with driving the sequencing expressions are flattened back out into lists of expressions.

With these little fixes the residual program is quite readable, but there is certainly more that could be done.

3.6 Summary

Supercompilation is achieved by a kind of partial evaluation. The supercompiler creates new functions for expressions and emits calls to those functions when it sees renamings of the expressions. This creates new functions and can eliminate redundant computations. Whenever the program contains a computation that consumes a value that computation is delayed and placed in a reduction context. The context is passed around until it meets the producer of a value. If the consumer and the producer match up then a redundant computation, such as a memory allocation, can be eliminated.

These elements of supercompilation are essential. After that comes the question of how to make the supercompile terminate every time. This is done by using an approximation referred to as the *whistle*. It detects expressions that are similar, except that one of them may have an extra list element or something else wrapped around some subexpression. This would indicate that the supercompiler has gone into a loop where it will start generating ever growing expressions. At this point the driving is stopped and the current expression is split into smaller expressions which are driven instead.

The way expressions are split is by taking the *most specific generalisation*, which extracts the differences between two expressions. If that fails then all subexpressions are extracted. Neither approach is always going to extract only those parts that are necessary to stop the infinite growth.

Most the functional parts of Erlang have been implemented in the supercompiler. Some of the pattern matching has been too difficult to implement fully using the chosen strategy, and when a program contains such patterns the implementation will not be able to improve that particular part of the program.

The current implementation gets good results on those examples it has been tested with, as will be shown in the next chapter.

Chapter 4

Results

The supercompiler has been implemented as described in the previous chapter. The question of how to evaluate a supercompiler is an interesting one. Ideally an optimisation should be tested on existing programs, but unfortunately the supercompiler is missing support for some key language features. In light of this a few small benchmark programs were used instead. The benchmarks have previously appeared in the literature [19, 48], except for the to_utf8 program. The inputs to the benchmark programs were not static (in the sense used in subsection 1.1.1), or there would have been an infinite speed-up. The program running the benchmarks was inspired by a similar program from [5].

Each benchmark is centred around a small expression that the supercompiler is tasked to improve. Table 4.1 shows the improvement in wall clock time over the non-supercompiled program (e.g., the append3 benchmark ran 1.43 times faster when supercompiled). Table 4.2 shows the reduction in memory allocations (where the number is 0.00, that means nearly all memory allocations were eliminated through supercompilation).

The performance increase was determined by measuring and comparing the time it took to run a benchmark for a large number of iterations. The benchmarks were timed repeatedly until they had settled near a minimum time, thus minimising the effect of variations caused by other software or by hardware. The benchmarks were compiled and run with Erlang R15B01 (erts-5.9.1) on an otherwise unloaded AMD FX$^{\text{TM}}$-8120 Eight-Core Processor with DDR3-1600 MHz CL9 memory. The programs were compiled both using the standard bytecode compiler BEAM [14] and the HiPE compiler [15, 42] that produces native machine code.

A manual inspection of the supercompiled programs showed that the expected transformations had occurred in all but the **sumsqs_lc** program. As shown in table 4.3, this program uses a list comprehension. The way the supercompiler desugars list comprehensions can cause the whistle to stop the driving too early. The result is a program that allocates just as much memory as the original program. The HiPE compiler interestingly manages to make the supercompiled program 12% faster than the original, whereas with BEAM the supercompiled program is slower.

The **append3** and **append3pp** benchmarks append three lists. Note that these programs first append `Xs` and `Ys`, and then append `Zs`. The better option is to first append `Ys` and `Zs`, and finally append `Xs` to the result. The benchmarks as they are written unnecessarily create a temporary list that the second append operation consumes. This is done quite intentionally, as one expects the supercompiler to correct this deficiency. Manual inspection of the optimised programs show that this transformation happens.

There are two variants of the append benchmark because Erlang also has a built-in append operator. The supercompiler translates this into a call to an append function it provides. This is done so that the supercompiler can transform the append in the expected manner. The numbers for **append3pp** in table 4.1 show that an append written in Erlang is unable to compete with the built-in append operator. The decrease in performance is not as bad when HiPE is used, but it is still far away from the performance increase one might hope for. Because the supercompiler removes one pass over the list `Xs` [19], increasing the size of that list may tip the balance in favour of the supercompiler, but this has not been explored.

The **vecdot** and **vecdot_int** programs differ in that the first uses floating point and the latter small integers. It appears that Erlang currently allocates memory for floating point computations, as one may note from the numbers in table 4.2. The vecdot_int program does not allocate garbage, whereas the vecdot program has had only a 46% reduction in allocations compared with the non-supercompiled program. This shows that the supercompiler has transformed the program as expected, but that the improvement had no effect on how floating point computations are performed.

The rest of the benchmarks behave as expected. The **flip** flips a tree twice and then sums the nodes. In the supercompiled program the tree flipping is gone and only a direct summation of the nodes remains. The **sumsqs** program is transformed into a (non tail-recursive) program that computes the result directly. Similarly the supercompiled **sumsqtr** program computes the squares of the nodes of a tree directly.

The **to_utf8** benchmark is UTF-8 encoder from the introduction chapter (listing 1.3) extended to handle strings. This was done by using the functions `map` and `flatten`, as one might do in a functional programming language. The supercompiler not only derives a version of the UTF-8 encoder equivalent to listing 1.4 (although it is less easy on the eyes), it also integrates the additional `map` and `flatten` functions. A cleaned up version of the residual program can be found in the appendix. The performance increase is respectable: the supercompiled program is 8-9 times faster. All intermediate lists have been removed.

Benchmark	*Bytecode (BEAM)*	*Native code (HiPE)*
append3	1.43	1.37
append3pp	0.60	0.75
flip	4.01	3.69
sumsqs	1.80	2.65
sumsqs_lc	0.93	1.12
sumsqtr	1.97	2.74
to_utf8	8.80	7.09
vecdot	1.91	1.42
vecdot_int	2.17	2.09

Table 4.1: Performance increase with supercompilation. Computed from the wall clock time for the non-supercompiled program divided by wall clock time for the supercompiled program.

Benchmark	*Remaining allocations*
append3	0.67
append3pp	0.66
flip	0.00
sumsqs	0.00
sumsqs_lc	1.00
sumsqtr	0.00
to_utf8	0.18
vecdot	0.54
vecdot_int	0.00

Table 4.2: Reduction in memory allocations. Computed from the amount of garbage generated by running the supercompiled program divided by the same for the non-supercompiled program.

Benchmark	*Core expression*
append3	`append(append(Xs, Ys), Zs)`
append3pp	`(Xs ++ Ys) ++ Zs`
flip	`sumtr(flip(flip(X)))`
sumsqs	`sum(map(fun (X) ->X * X end, Xs))`
sumsqs_lc	`sum([X * X || X <- Xs ])`
sumsqtr	`sumtr(squaretr(X))`
to_utf8	`flatten(map(fun to_utf8/1, S))`
vecdot	`sum(zipwith(fun (X, Y) ->X * Y end, Xs, Ys))`
vecdot_int	`sum(zipwith(fun (X, Y) ->X * Y end, Xs, Ys))`

Table 4.3: The core expressions of the benchmarks.

Chapter 5

Discussion

A supercompiler is now available for a widely used language, it is easy to invoke, it works as expected on examples from the literature and the residual programs are readily inspected. The supercompiler is not yet ready to take on existing Erlang programs, but the basics are all there. The availability of the supercompiler and the documentation provided by this report lowers the supercompilation learning curve.

The Erlang supercompiler is available online at `http://weinholt.se/`.

5.1 Related work

Others have worked on optimising Erlang and there are also a few supercompiler implementations.

- Optimisation of Erlang at a lower level has been treated by Stenman [42]. He describes HiPE, an optimising native code compiler for Erlang. He also shows some techniques for improving the performance of Erlang's concurrency features, including a transformation that can partially merge a sender and a receiver.
- The supercompilation algorithm used in this work has previously been implemented by Jonsson and Nordlander for the languages Haskell and Timber [19].
- Klyuchnikov has published HOSC, a supercompiler for a lazy higher-order functional language [27]. Its source code is available online and the paper contains a complete and formal description of the supercompiler.
- Mitchell's Supero is a supercompiler for a simplified core of Haskell [31]. Supero uses a new termination test and it has been designed to be simple.

5.2 Future work

As has been previously mentioned the supercompiler algorithm used does not handle side-effects. In an Erlang program side-effects can come from message passing or from calling other modules. It would be interesting to investigate just what happens to the side-effects in a supercompiled program.

Before support for side-effects can be considered the implementation needs to be extended to handle more of Erlang's constructs. Currently the implementation is missing support for *receive* expressions. These need to be added in a few functions that pattern match on the type of an expression, e.g. the alpha conversion function and the splitting function. Additionally support for records and other miscellaneous syntax should be implemented.

One of the problems with the current implementation is how it handles list comprehensions. The way they are currently translated causes the whistle to trigger prematurely. The sumsqs_lc benchmark shows that the resultant split makes the code worse than it could be. There are more intricate strategies for translating list comprehension [18], and it is possible that these will not cause any problems with the whistle. But the supercompiler should ideally be able to derive the code used by the more intricate strategy all by itself.

Another way to tackle the problem of list comprehensions is then to improve the whistle or the generalisations. Jonsson suggested a heuristic to improve the splitting of *append* as used by *flatmap* [20], but there is still room for improvement. It is possible that the generalisation in [30] is capable of handling the kind of code generated from list comprehensions.

The problems with list comprehensions was found by looking at what the supercompiler did to some small examples. The supercompiler should really be tested on existing programs. Before this is done it is difficult to say how common the problems with the whistle and generalisations will be.

The way pattern matching is handled by the implementation leaves a lot to be desired. Repeated simplifications works out well for small examples, but a quick look at existing Erlang code indicates that explicit matches in patterns is quite common. The current implementation is also incapable for handling the kind of code used in [4]. In retrospect a unification algorithm [32] should have been used.

The ability to run the supercompiler on existing programs would also provide a better way of benchmarking it. The benchmarks used are typical of the ones used in the literature on supercompilation. The intent is not really to show that the supercompiled program is much faster, it is rather to show that the desired transformations take place. It is questionable if a programmer would actually write code like that used in the benchmarks if performance was an issue.[1]

It would be quite interesting to look at some of the other uses for a supercompiler. The supercompiler has been implemented without using any side-effects, in the hope that it one day will be capable of *self-application*, i.e. capable of supercompiling itself. This would allow it to be used for the higher-level Futamura projections [4]. If unification is implemented then it should be possible to use the first Futamura projection, which could turn interpreters into compilers that generate Erlang code. It should already be capable of producing some inductive proofs as in [44], but this needs further investigation.

There are improvements to the supercompiler algorithm that should be investigated. There is a strengthened algorithm that is capable of removing more intermediate data [22] and there is a technique that stops code explosion [23],

[1]On the other hand, if a good supercompiler is widely available and its properties well known, programmers may be more likely to let the supercompiler take care of the performance.

which is when the supercompiled program becomes much larger than it should be.

Looking beyond supercompilation there is a stronger technique called distillation [10, 11, 12]. This transformation is capable of superlinear speedups, i.e. improvements that decrease the *Big-O* time complexity of a program. Distillation looks quite promising and is certainly one possible direction for future work.

Appendix A

Residual programs

String to UTF-8 encoder

This string to UTF-8 encoder is based on the one-character UTF-8 encoder in listing 1.4. The original expression is `flatten(map(fun to_utf8/1, S))`. The supercompiler derives a version that does not use intermediate lists. The code shown below has been cleaned up slightly to make it easier to read. A compiler will have no trouble generating the same code for the non-cleaned up version as for the version presented here.

```
string_to_utf8([]) ->
    [];
string_to_utf8([X|Xs]) ->
    case X of
        Code when Code < 16#80 ->
            B = Code,
            [B|string_to_utf8(Xs)];
        Code0 when Code0 < 16#800 ->
            B0 = (Code0 bsr 6) bor 16#C0,
            B1 = Code0 bor 16#80,
            [B0,B1 band 16#BF|string_to_utf8(Xs)];
        Code1 when Code1 < 16#10000 ->
            B2 = (Code1 bsr 12) bor 16#E0,
            B3 = (Code1 bsr 6) bor 16#80,
            B4 = Code1 bor 16#80,
            [B2,B3 band 16#BF,
             B4 band 16#BF|string_to_utf8(Xs)];
        Code2 ->
            B5 = (Code2 bsr 18) bor 16#F0,
            B6 = (Code2 bsr 12) bor 16#80,
            B7 = (Code2 bsr 6) bor 16#80,
            B8 = Code2 bor 16#80,
            [B5,B6 band 16#BF,B7 band 16#BF,
             B8 band 16#BF|string_to_utf8(Xs)]
    end.
```

Append three lists

The original expression is `(Xs ++ Ys) ++ Zs`. Note that the way the braces are placed means an intermediate list is constructed and then discarded. In the supercompiled version this redundancy is gone.

```
ap([], Ys, Zs) ->
    case Ys of
        [] ->
            Zs;
        [X|Xs] ->
            [X|ap2(Xs, Zs)]
    end;
ap([X|Xs], Ys, Zs) ->
    [X|ap(Xs, Ys, Zs)].

ap2([], Zs) ->
    Zs;
ap2([X|Xs], Zs) ->
    [X|ap2(Xs, Zs)].
```

Vector dot product

A common example is `sum(zipwith(fun (X, Y) -> X * Y end, Xs, Ys))`. Originally from Kort's thesis on deforestation [19, 31], this function has an intermediate list which supercompilation removes.

```
vecdot([X|Xs], Ys) ->
    case Ys of
        [] ->
            0;
        [Y|Ys0] ->
            X * Y + vecdot(Xs, Ys0)
    end;
vecdot(_, _) ->
    0.
```

Flip a tree twice and sum the nodes

In this example the original code is shown in the listing below.

```
sumtr({leaf,X}) ->
    X;
sumtr({branch,L,R}) ->
    sumtr(L) + sumtr(R).

flip({leaf,X}) ->
    {leaf,X};
flip({branch,L,R}) ->
    {branch,flip(R),flip(L)}.

flipsum(X) ->
    sumtr(flip(flip((X)))).
```

The supercompiler removes the double call to `flip()`, and the residual program does not allocate any structures at all:

```
flipsum({leaf,X}) ->
    X;
flipsum({branch,L,R}) ->
    flipsum(L) + flipsum(R).
```

Bibliography

[1] Joe Armstrong. *Making reliable distributed systems in the presence of software errors.* PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, 2003.

[2] Maximilian Bolingbroke and Simon Peyton Jones. Supercompilation by evaluation. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 135–146, New York, NY, USA, 2010. ACM.

[3] Edwin C. Brady and Kevin Hammond. Scrapping your inefficient engine: using partial evaluation to improve domain-specific language implementation. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 297–308, New York, NY, USA, 2010. ACM.

[4] Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12:381–391, 1999 (1971). Updated and revised.

[5] Abdulaziz Ghuloum. *Ikarus Scheme User's Guide.* 2009.

[6] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM.

[7] Robert Glück and Andrei Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In Patrick Cousot, Moreno Falaschi, Gilberto Filé, and Antoine Rauzy, editors, *Static Analysis*, volume 724 of *Lecture Notes in Computer Science*, pages 112–123. Springer Berlin / Heidelberg, 1993.

[8] Robert Glück and Morten Sørensen. A roadmap to metacomputation by supercompilation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 137–160. Springer Berlin / Heidelberg, 1996.

[9] Geoffrey William Hamilton. Higher order deforestation. In Herbert Kuchen and S. Doaitse Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 1140 of *Lecture Notes in Computer Science*, pages 213–227. Springer Berlin / Heidelberg, 1996.

[10] Geoffrey William Hamilton. Distillation: extracting the essence of programs. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '07, pages 61–70, New York, NY, USA, 2007. ACM.

[11] Geoffrey William Hamilton. Extracting the essence of distillation. In Amir Pnueli, Irina Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics*, volume 5947 of *Lecture Notes in Computer Science*, pages 151–164. Springer Berlin / Heidelberg, 2010.

[12] Geoffrey William Hamilton and Neil D. Jones. Distillation with labelled transition systems. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, PEPM '12, pages 15–24, New York, NY, USA, 2012. ACM.

[13] David Harriman. *The Logical Leap: Induction in Physics.* New American Library, 2010.

[14] Bogumil Hausman. Turbo Erlang: Approaching the speed of C. pages 119–135. Kluwer Academic Publishers, 1994.

[15] Erik Johansson, Mikael Pettersson, and Konstantinos Sagonas. A high performance Erlang system. In *Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '00, pages 32–43, New York, NY, USA, 2000. ACM.

[16] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 190–203. Springer Berlin Heidelberg, 1985.

[17] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, 1993.

[18] Simon Peter Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[19] Peter A. Jonsson. *Time- and Size-Efficient Supercompilation.* PhD thesis, Luleå University of Technology, Luleå, Sweden, 2011.

[20] Peter A. Jonsson. Private communication, 2012.

[21] Peter A. Jonsson and Johan Nordlander. Positive supercompilation for a higher order call-by-value language. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 277–288, New York, NY, USA, 2009. ACM.

[22] Peter A. Jonsson and Johan Nordlander. Strengthening supercompilation for call-by-value languages. In Andrei P. Nemytykh, editor, *Proceedings of the second International Valentin Turchin Memorial Workshop on Metacomputation in Russia*, pages 64–81. Ailamazyan University of Pereslavl, July 2010.

[23] Peter A. Jonsson and Johan Nordlander. Taming code explosion in supercompilation. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '11, pages 33–42, New York, NY, USA, 2011. ACM.

[24] Andrei V. Klimov. An approach to supercompilation for object-oriented languages: the Java supercompiler case study. In Andrei P. Nemytykh, editor, *Proceedings of the first International Workshop on Metacomputation in Russia*, pages 43–53. Ailamazyan University of Pereslavl, July 2008.

[25] Andrei V. Klimov. A Java supercompiler and its application to verification of cache-coherence protocols. In Amir Pnueli, Irina Virbitskaite, and Andrei Voronkov, editors, *Ershov Memorial Conference*, volume 5947 of *Lecture Notes in Computer Science*, pages 185–192. Springer, 2010.

[26] Ilya Klyuchnikov and Sergei Romanenko. Proving the equivalence of higher-order terms by means of supercompilation. In Amir Pnueli, Irina Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics*, volume 5947 of *Lecture Notes in Computer Science*, pages 193–205. Springer Berlin / Heidelberg, 2010.

[27] Ilya G. Klyuchnikov. Supercompiler HOSC 1.0: under the hood. Technical Report Preprint 63, Keldysh Institute of Applied Mathematics, 2009.

[28] Michael Leuschel. Homeomorphic embedding for online termination of symbolic methods. In Torben Mogensen, David Schmidt, and I. Sudborough, editors, *The Essence of Computation*, volume 2566 of *Lecture Notes in Computer Science*, pages 379–403. Springer Berlin / Heidelberg, 2002.

[29] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services.* PhD thesis, Columbia University, 1992.

[30] Neil Mitchell. *Transformation and Analysis of Functional Programs.* PhD thesis, University of York, 2008.

[31] Neil Mitchell. Rethinking supercompilation. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 309–320, New York, NY, USA, 2010. ACM.

[32] Stuart Russell and Peter Norvig. *Artificial Intelligence: A modern approach.* Prentice Hall, 2009.

[33] Jens P. Secher and Morten Heine Sørensen. On perfect supercompilation. In *Proceedings of the Third International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, PSI '99, pages 113–127, London, UK, UK, 2000. Springer-Verlag.

[34] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University, May 1991.

[35] Morten Heine Sørensen. Turchin's supercompiler revisited - an operational theory of positive information propagation. Master's thesis, University of Copenhagen, Copenhagen, Denmark, 1996. Revised edition.

[36] Morten Heine Sørensen. Convergence of program transformers in the metric space of trees. *Sci. Comput. Program.*, 37(1-3):163–205, May 2000.

[37] Morten Heine Sørensen and Robert Glück. An algorithm of generalization in positive supercompilation. In *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 465–479. MIT Press, 1995.

[38] Morten Heine Sørensen and Robert Glück. Introduction to supercompilation. In *Partial Evaluation – Practice and Theory, DIKU 1998 International Summer School*, pages 246–270, London, UK, 1999. Springer-Verlag.

[39] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6:811–838, 1993.

[40] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In *Proceedings of the 5th European Symposium on Programming: Programming Languages and Systems*, ESOP '94, pages 485–500, London, UK, 1994. Springer-Verlag.

[41] Michael Sperber. Self-applicable online partial evaluation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 465–480. Springer Berlin / Heidelberg, 1996.

[42] Erik Stenman. *Efficient Implementation of Concurrent Programming Languages.* PhD thesis, Uppsala Universitet, 2002.

[43] Josef Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, ICFP '02, pages 124–132, New York, NY, USA, 2002. ACM.

[44] Valentin F. Turchin. The use of metasystem transition in theorem proving and program optimization. In Jaco Bakker and Jan Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 645–657. Springer Berlin Heidelberg, 1980.

[45] Valentin F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, June 1986.

[46] Valentin F. Turchin. The algorithm of generalization in the supercompiler. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland, 1988.

[47] Oscar Waddell. *Extending the Scope of Syntactic Abstraction.* PhD thesis, Indiana University, 1999.

[48] Philip Wadler. Deforestation: transforming programs to eliminate trees. In *Proceedings of the Second European Symposium on Programming*, pages 231–248, Amsterdam, The Netherlands, 1988. North-Holland Publishing Co.